```
12      color_flip(root);
13    root->size = size(root->lc) + size(root->rc) + 1;
14    return root;
15  }
16
17  template <class Key, class Compare>
18  typename Set<Key, Compare>::Node_Set<Key, Compare>::insert(
19      Set::Node_root, const Key &key) const {
20    if (root == nullptr) return new Node(key, kRed, 1);
21    if (root->key == key)
22      ;
23    else if (cmp\_(key, root->key))  // if (key < root->key)
24      root->lc = insert(root->lc, key);
25    else
26      root->rc = insert(root->rc, key);
27    return fix_up(root);
28  }
```

删除

删除操作基于这样的思想：我们不能删除黑色的节点，因为这样会破坏黑高. 所以我们需要保证我们最后删除的节点是红色的.

删除最小值节点

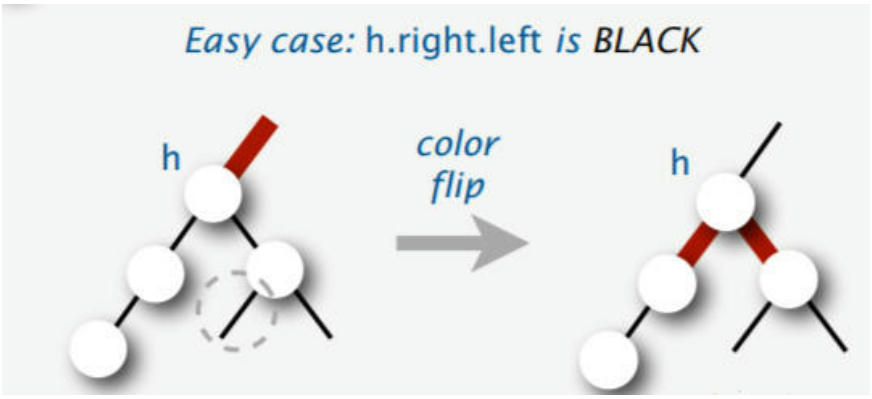
首先来试一下删除整棵树里的最小值.

怎么才能保证最后删除的节点是红色的呢? 我们需要在向下递归的过程中保证一个性质: 如果当前节点是 `h`，那么需要保证 `h` 是红色，或者 `h->lc` 是红色.

考虑这样做的正确性, 如果我们能够通过各种旋转和反转颜色操作成功维护这个性质, 那么当我们到达最小的节点 `h_min` 的时候，有 `h_min` 是红色，或者 `h_min` 的左子树——但是 `h_min` 根本没有左子树! 所以这就保证了最小值节点一定是红的，既然它是红色的，我们就可以大胆的删除它，然后用与插入操作相同的调整思路对树进行调整.

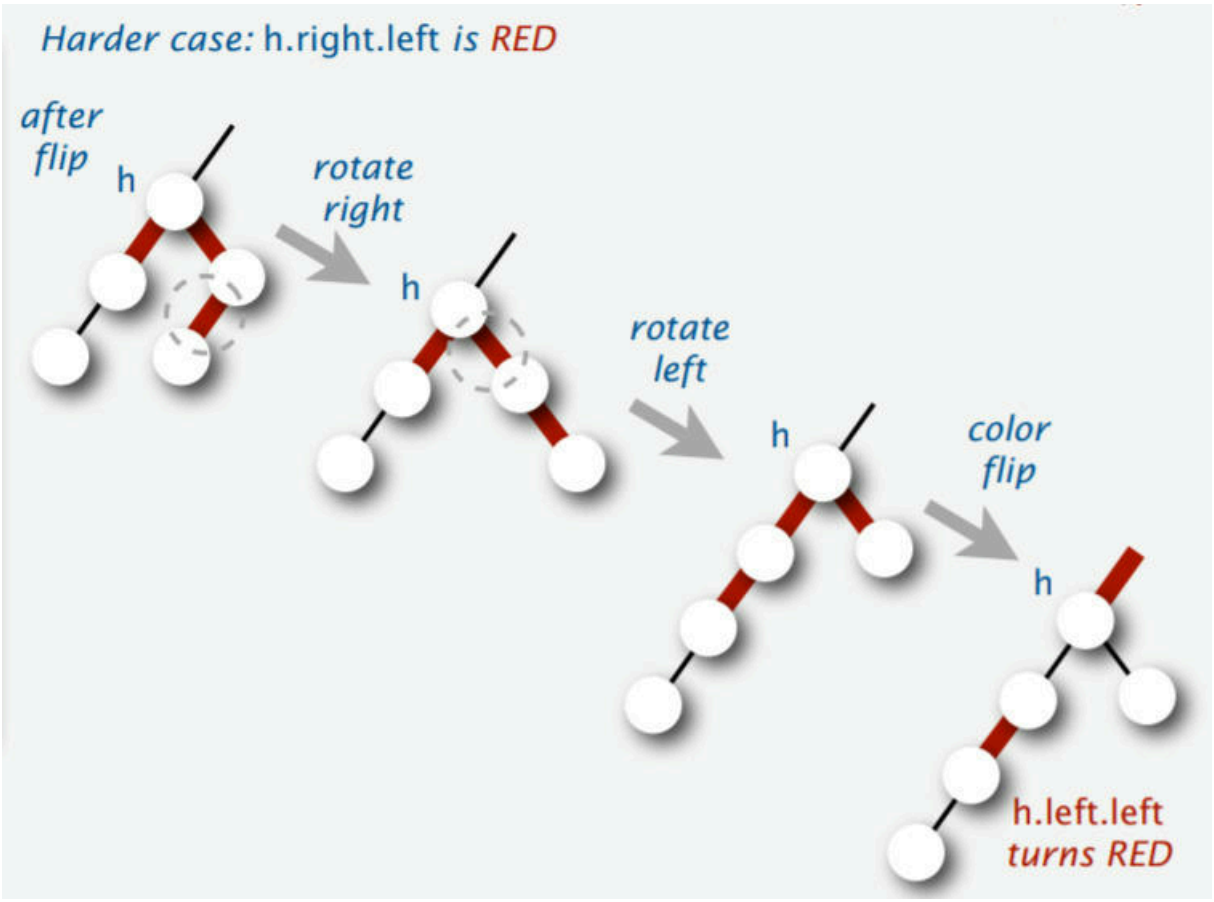
下面我们来考虑怎么满足这个性质，注意，我们会在向下递归的时候 **临时地** 破坏左偏红黑树的若干性质，但是当我们从递归中返回时还会将其恢复.

如下图所描述的，是一种较为简单的情况，此时 `h->rc->lc` 为黑色，我们只需要一次翻转颜色即可:



并且，在如上所示的翻转之后，不会使 `h->rc` 与 `h->rc->lc` 形成连续的红边;

但如果 `h->rc->lc` 是红色，情况会比较复杂:



如果只进行翻转颜色，会产生连续的红边，而考虑我们递归返回的时候，是无法修复这样的情况的，因此需要进行处理.

然后就可以进行删除了：

参考代码（部分）

```
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_left(
    Set::Node *root) const {
  color_flip(root);
  if (is_red(root->rc->lc)) {
    // assume that root->rc != nullptr when calling this function
    root->rc = rotate_right(root->rc);
    root = rotate_left(root);
    color_flip(root);
  }
  return root;
}

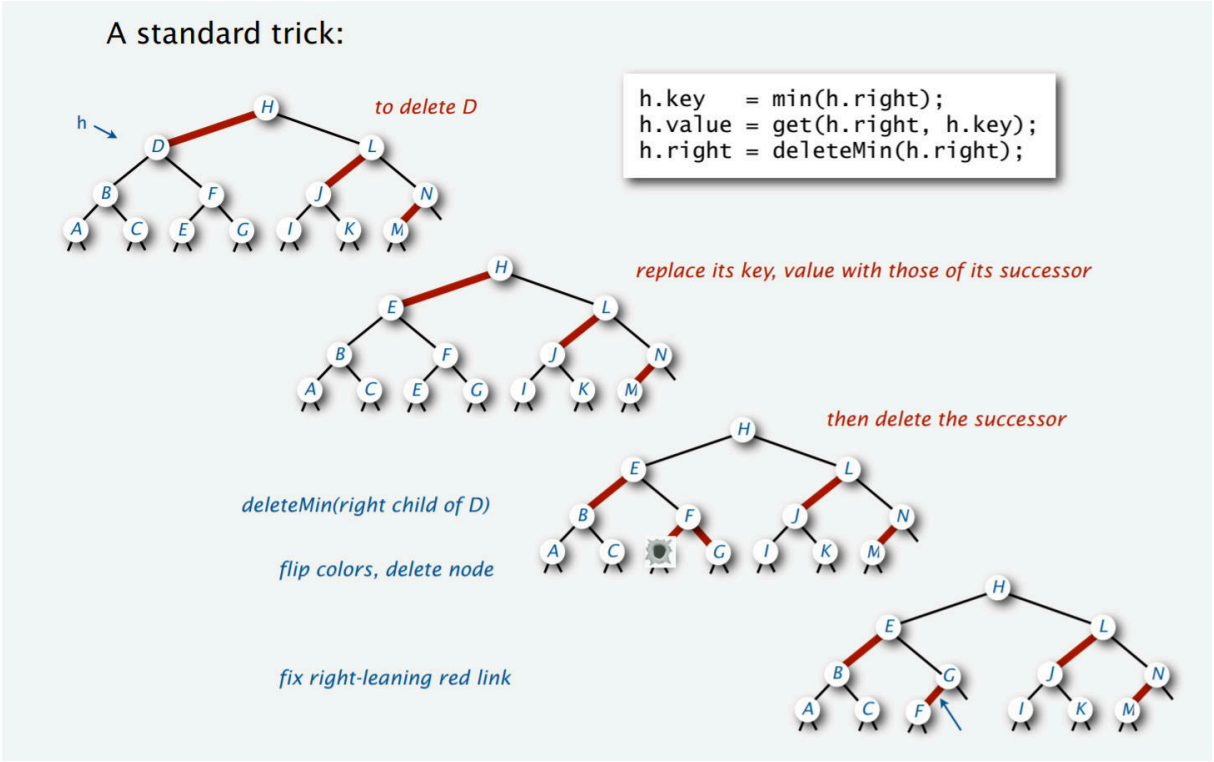
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_min(
    Set::Node *root) const {
  if (root->lc == nullptr) {
    delete root;
    return nullptr;
  }
```

```
  if (!is_red(root->lc) && !is_red(root->lc->lc)) {
    // make sure either root->lc or root->lc->lc is red
    // thus make sure we will delete a red node in the end
    root = move_red_left(root);
  }
  root->lc = delete_min(root->lc);
  return fix_up(root);
}
```

删除任意节点

我们首先考虑删除叶子：与删最小值类似，我们在删除任意值的过程中也要维护一个性质，不过这次比较特殊，因为我们不是只向左边走，而是可以向左右两个方向走，因此在删除过程中维护的性质是这样的：如果往左走，当前节点是 `h`，那么需要保证 `h` 是红色，或者 `h->lc` 是红色；如果往右走，当前节点是 `h`，那么需要保证 `h` 是红色，或者 `h->rc` 是红色．这样可以保证我们最后总会删掉一个红色节点．

下面考虑删除非叶子节点，我们只需要找到其右子树（如果有）里的最小节点，然后用右子树的最小节点的值代替该节点的值，最后删除右子树里的最小节点．



那如果没有右子树怎么办？我们需要把左子树旋转过来，这样就不会出现这个问题了．

参考代码（部分）

```
template <class Key, class Compare>
typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_arbitrary(
    Set::Node *root, Key key) const {
  if (cmp_(key, root->key)) {
    // key < root->key
    if (!is_red(root->lc) && !(is_red(root->lc->lc)))
```

```
      root = move_red_left(root);
    // ensure the invariant: either root->lc or root->lc->lc (or root and
    // root->lc after dive into the function) is red, to ensure we will
    // eventually delete a red node. therefore we will not break the black
    // height balance
    root->lc = delete_arbitrary(root->lc, key);
  } else {
    // key >= root->key
    if (is_red(root->lc)) root = rotate_right(root);
    if (key == root->key && root->rc == nullptr) {
      delete root;
      return nullptr;
    }
    if (!is_red(root->rc) && !is_red(root->rc->lc)) root =
move_red_right(root);
    if (key == root->key) {
      root->key = get_min(root->rc);
      root->rc = delete_min(root->rc);
    } else {
      root->rc = delete_arbitrary(root->rc, key);
    }
  }
  return fix_up(root);
}
```

实现

下面的代码是用左偏红黑树实现的 `Set`，即有序不可重集合：

参考代码

```
#include <algorithm>
#include <memory>
#include <vector>

template <class Key, class Compare = std::less<Key>>
class Set {
 private:
  enum NodeColor { kBlack = 0, kRed = 1 };

  struct Node {
    Key key;
    Node *lc{nullptr}, *rc{nullptr};
    size_t size{0};
    NodeColor color;  // the color of the parent link

    Node(Key key, NodeColor color, size_t size)
        : key(key), color(color), size(size) {}

    Node() = default;
  };

```

```
  void destroyTree(Node *root) const {
    if (root != nullptr) {
      destroyTree(root->lc);
      destroyTree(root->rc);
      root->lc = root->rc = nullptr;
      delete root;
    }
  }

  bool is_red(const Node *nd) const {
    return nd == nullptr ? false : nd->color;  // kRed == 1, kBlack == 0
  }

  size_t size(const Node *nd) const { return nd == nullptr ? 0 : nd->size; }

  Node *rotate_left(Node *node) const {
    // left rotate a red link
    //          <1>                  <2>
    //        /    \\               //    \
    //       *      <2>    ==>     <1>     *
    //             /   \          /   \
    //            *     *        *     *
    Node *res = node->rc;
    node->rc = res->lc;
    res->lc = node;
    res->color = node->color;
    node->color = kRed;
    res->size = node->size;
    node->size = size(node->lc) + size(node->rc) + 1;
    return res;
  }

  Node *rotate_right(Node *node) const {
    // right rotate a red link
    //            <1>               <2>
    //          //    \           /    \\
    //         <2>     *   ==>   *      <1>
    //        /   \                    /   \
    //       *     *                  *     *
    Node *res = node->lc;
    node->lc = res->rc;
    res->rc = node;
    res->color = node->color;
    node->color = kRed;
    res->size = node->size;
    node->size = size(node->lc) + size(node->rc) + 1;
    return res;
  }

  NodeColor neg_color(NodeColor n) const { return n == kBlack ? kRed :
kBlack; }

```

```
  void color_flip(Node *node) const {
    node->color = neg_color(node->color);
    node->lc->color = neg_color(node->lc->color);
    node->rc->color = neg_color(node->rc->color);
  }

  Node *insert(Node *root, const Key &key) const;
  Node *delete_arbitrary(Node *root, Key key) const;
  Node *delete_min(Node *root) const;
  Node *move_red_right(Node *root) const;
  Node *move_red_left(Node *root) const;
  Node *fix_up(Node *root) const;
  const Key &get_min(Node *root) const;
  void serialize(Node *root, std::vector<Key> *) const;
  void print_tree(Set::Node *root, int indent) const;
  Compare cmp_ = Compare();
  Node *root_{nullptr};

public:
  using KeyType = Key;
  using ValueType = Key;
  using SizeType = std::size_t;
  using DifferenceType = std::ptrdiff_t;
  using KeyCompare = Compare;
  using ValueCompare = Compare;
  using Reference = Key &;
  using ConstReference = const Key &;

  Set() = default;

  Set(Set &) = default;

  Set(Set &&) noexcept = default;

  ~Set() { destroyTree(root_); }

  SizeType size() const;

  SizeType count(const KeyType &key) const;

  SizeType erase(const KeyType &key);

  void clear();

  void insert(const KeyType &key);

  bool empty() const;

  std::vector<Key> serialize() const;

  void print_tree() const;
};
```

```
125
126 template <class Key, class Compare>
127 typename Set<Key, Compare>::SizeType Set<Key, Compare>::count(
128     ConstReference key) const {
129   Node *x = root_;
130   while (x != nullptr) {
131     if (key == x->key) return 1;
132     if (cmp_(key, x->key))  // if (key < x->key)
133       x = x->lc;
134     else
135       x = x->rc;
136   }
137   return 0;
138 }
139
140 template <class Key, class Compare>
141 typename Set<Key, Compare>::SizeType Set<Key, Compare>::erase(
142     const KeyType &key) {
143   if (count(key) > 0) {
144     if (!is_red(root_->lc) && !(is_red(root_->rc))) root_->color = kRed;
145     root_ = delete_arbitrary(root_, key);
146     if (root_ != nullptr) root_->color = kBlack;
147     return 1;
148   } else {
149     return 0;
150   }
151 }
152
153 template <class Key, class Compare>
154 void Set<Key, Compare>::clear() {
155   destroyTree(root_);
156   root_ = nullptr;
157 }
158
159 template <class Key, class Compare>
160 void Set<Key, Compare>::insert(const KeyType &key) {
161   root_ = insert(root_, key);
162   root_->color = kBlack;
163 }
164
165 template <class Key, class Compare>
166 bool Set<Key, Compare>::empty() const {
167   return size(root_) == 0;
168 }
169
170 template <class Key, class Compare>
171 typename Set<Key, Compare>::Node *Set<Key, Compare>::insert(
172     Set::Node *root, const Key &key) const {
173   if (root == nullptr) return new Node(key, kRed, 1);
174   if (root->key == key)
175     ;
176   else if (cmp_(key, root->key))  // if (key < root->key)
```

```
177     root->lc = insert(root->lc, key);
178   else
179     root->rc = insert(root->rc, key);
180   return fix_up(root);
181 }
182
183 template <class Key, class Compare>
184 typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_min(
185     Set::Node *root) const {
186   if (root->lc == nullptr) {
187     delete root;
188     return nullptr;
189   }
190   if (!is_red(root->lc) && !is_red(root->lc->lc)) {
191     // make sure either root->lc or root->lc->lc is red
192     // thus make sure we will delete a red node in the end
193     root = move_red_left(root);
194   }
195   root->lc = delete_min(root->lc);
196   return fix_up(root);
197 }
198
199 template <class Key, class Compare>
200 typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_right(
201     Set::Node *root) const {
202   color_flip(root);
203   if (is_red(root->lc->lc)) {  // assume that root->lc != nullptr when
    calling
204                                // this function
205     root = rotate_right(root);
206     color_flip(root);
207   }
208   return root;
209 }
210
211 template <class Key, class Compare>
212 typename Set<Key, Compare>::Node *Set<Key, Compare>::move_red_left(
213     Set::Node *root) const {
214   color_flip(root);
215   if (is_red(root->rc->lc)) {
216     // assume that root->rc != nullptr when calling this function
217     root->rc = rotate_right(root->rc);
218     root = rotate_left(root);
219     color_flip(root);
220   }
221   return root;
222 }
223
224 template <class Key, class Compare>
225 typename Set<Key, Compare>::Node *Set<Key, Compare>::fix_up(
226     Set::Node *root) const {
227   if (is_red(root->rc) && !is_red(root->lc))  // fix right leaned red link
```

```
228     root = rotate_left(root);
229   if (is_red(root->lc) &&
230       is_red(root->lc->lc))  // fix doubly linked left leaned red link
231     // if (root->lc == nullptr), then the second expr won't be evaluated
232     root = rotate_right(root);
233   if (is_red(root->lc) && is_red(root->rc))
234     // break up 4 node
235     color_flip(root);
236   root->size = size(root->lc) + size(root->rc) + 1;
237   return root;
238 }
239
240 template <class Key, class Compare>
241 const Key &Set<Key, Compare>::get_min(Set::Node *root) const {
242   Node *x = root;
243   // will crash as intended when root == nullptr
244   for (; x->lc != nullptr; x = x->lc);
245   return x->key;
246 }
247
248 template <class Key, class Compare>
249 typename Set<Key, Compare>::SizeType Set<Key, Compare>::size() const {
250   return size(root_);
251 }
252
253 template <class Key, class Compare>
254 typename Set<Key, Compare>::Node *Set<Key, Compare>::delete_arbitrary(
255     Set::Node *root, Key key) const {
256   if (cmp_(key, root->key)) {
257     // key < root->key
258     if (!is_red(root->lc) && !(is_red(root->lc->lc)))
259       root = move_red_left(root);
260     // ensure the invariant: either root->lc or root->lc->lc (or root and
261     // root->lc after dive into the function) is red, to ensure we will
262     // eventually delete a red node. therefore we will not break the black
263     // height balance
264     root->lc = delete_arbitrary(root->lc, key);
265   } else {
266     // key >= root->key
267     if (is_red(root->lc)) root = rotate_right(root);
268     if (key == root->key && root->rc == nullptr) {
269       delete root;
270       return nullptr;
271     }
272     if (!is_red(root->rc) && !is_red(root->rc->lc)) root =
    move_red_right(root);
273     if (key == root->key) {
274       root->key = get_min(root->rc);
275       root->rc = delete_min(root->rc);
276     } else {
277       root->rc = delete_arbitrary(root->rc, key);
278     }
```

```
  }
  return fix_up(root);
}

template <class Key, class Compare>
std::vector<Key> Set<Key, Compare>::serialize() const {
  std::vector<int> v;
  serialize(root_, &v);
  return v;
}

template <class Key, class Compare>
void Set<Key, Compare>::serialize(Set::Node *root,
                                  std::vector<Key> *res) const {
  if (root == nullptr) return;
  serialize(root->lc, res);
  res->push_back(root->key);
  serialize(root->rc, res);
}

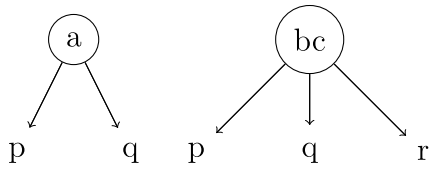
template <class Key, class Compare>
void Set<Key, Compare>::print_tree(Set::Node *root, int indent) const {
  if (root == nullptr) return;
  print_tree(root->lc, indent + 4);
  std::cout << std::string(indent, '-') << root->key << std::endl;
  print_tree(root->rc, indent + 4);
}

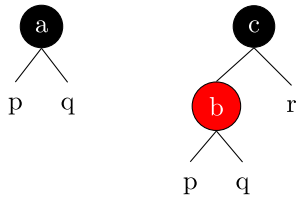
template <class Key, class Compare>
void Set<Key, Compare>::print_tree() const {
  print_tree(root_, 0);
}
```

与 2-3 树的关系

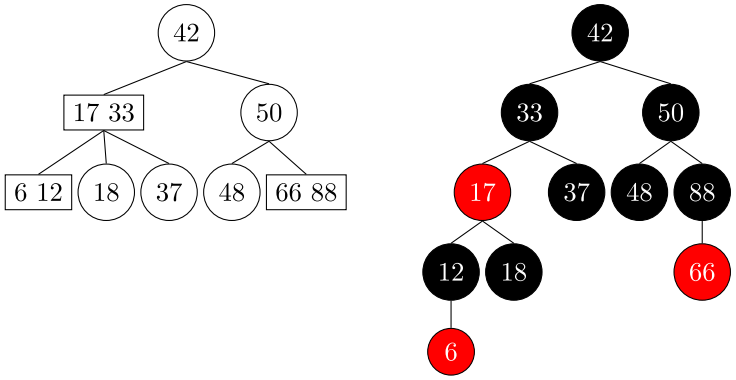
2-3 树是 3 阶 B 树，每个结点都是 2 结点或 3 结点，存储一个或两个数据元素．非叶结点的 2 结点和 3 结点分别只能有两个或三个孩子．而且，2-3 树中存储的所有数据都是有序的．

2-3 树和左偏红黑树实质是等价的．2-3 树中一个节点可以存储 1 个元素或 2 个元素，而红黑树的一个节点只能存储一个元素．如下图所示，2-3 树的 2 节点对应一个黑色节点，3 节点对应一个红色节点和一个黑色节点（可以将 bc 视作平行）．





下图是一棵 2-3 树对应的左偏红黑树.



2-3 树和左偏红黑树的插入与删除操作是一一对应的. [21]

参考资料与拓展阅读

- Left-Leaning Red-Black Trees[3]- Robert Sedgewick Princeton University
- Balanced Search Trees[4]-_Algorithms_Robert Sedgewick | Kevin Wayne

[1] 2-3 树

[2] 这篇博文

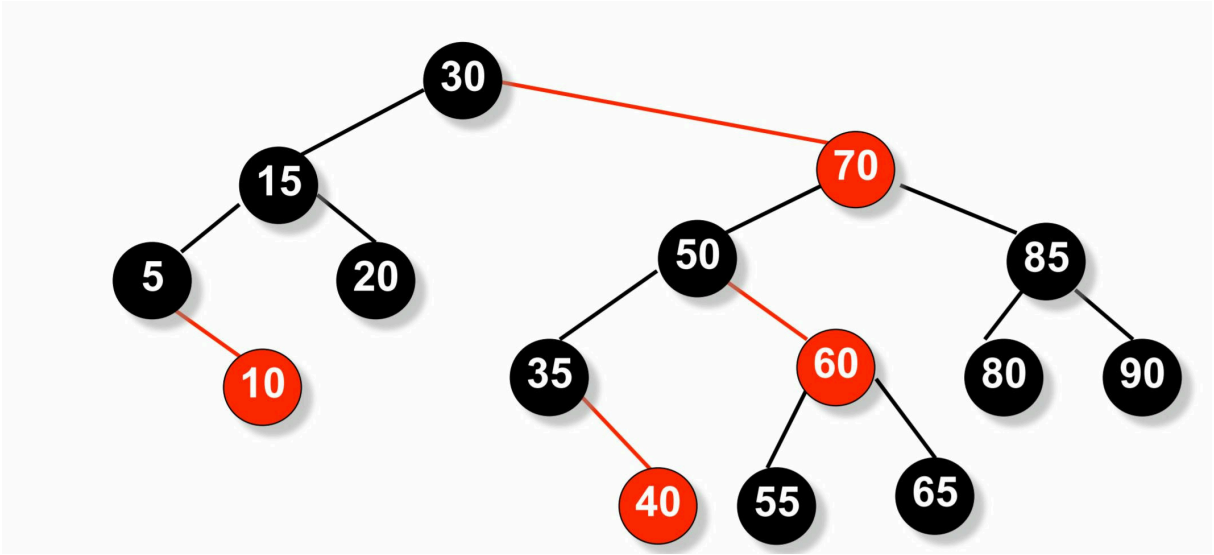
[3] Left-Leaning Red-Black Trees

[4] Balanced Search Trees

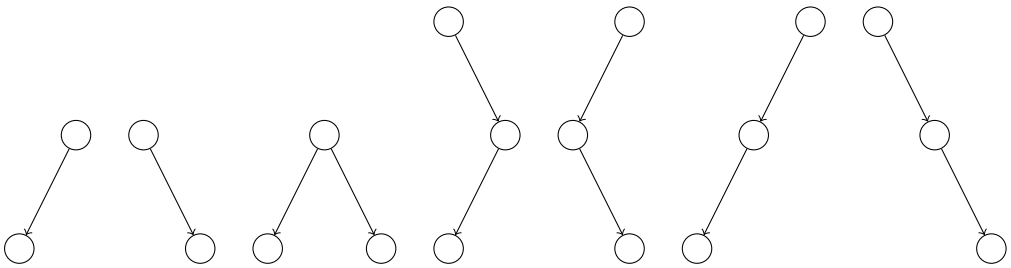
10.15.11 AA 树

AA 树是一种用于高效存储和检索有序数据的平衡树形结构，Arne Andersson 教授于 1993 年在他的论文 "Balanced search trees made simple" 中介绍，设计的目的是减少红黑树考虑的不同情况. AA 树可以在 $O(\log N)$ 的时间内做查找，插入和删除. 下面是一个 AA 树的例子.

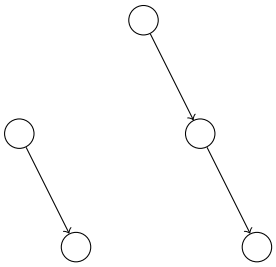
[21]这篇博文[2] 提供了详细的描述. 文中的「红黑树」实际上指的是「左偏红黑树」.



AA 树是红黑树的一种变体，与红黑树不同，AA 树上的红色节点只能作为右子节点．这导致 AA 树模拟了 2-3 树而不是 2-3-4 树，从而极大地简化了维护操作．红黑树的维护算法需要考虑七种不同的情况来正确平衡树．



因为红色节点只能作为右子节点，AA 树只需要考虑两种情况．



定义

AA 树遵循与红黑树相同的规则，但添加了一条新规则，**即红色节点不能作为左孩子出现**．

1. 每个节点都可以是红色或黑色．
2. 根节点总是黑色．
3. 叶节点（NULL）总是黑色．
4. 红色节点的两个子节点必须都是黑色，即没有两个相邻的红色节点．
5. 从根节点到 NULL 节点的每条路径都有相同数量的黑色节点．
6. 红色节点只能作为右子节点．

平衡维护

AA 树的每个节点维护一个 **level** 字段，类似红黑树的每个节点维护一个 color 字段 ("RED" or "BLACK")．level 的规定满足以下 5 个条件：

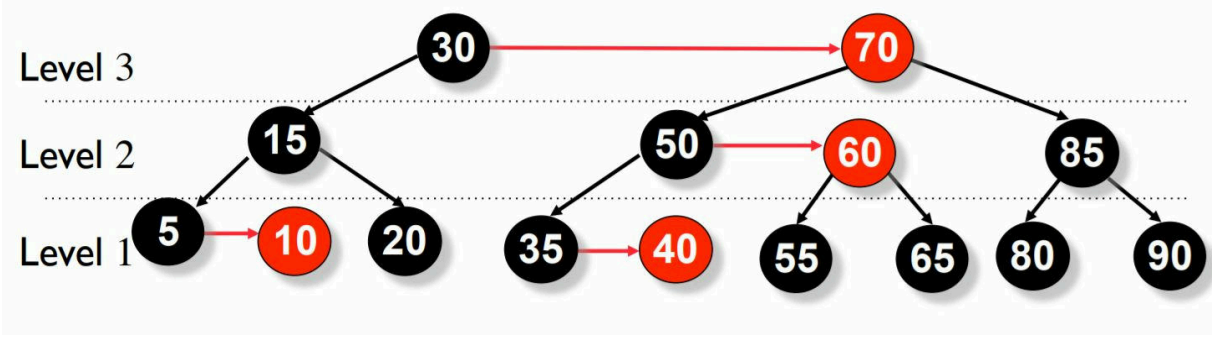
1、每个叶节点的 level 是 1.

2、每个左孩子的 level 是其父节点的 level 减 1.

3、每个右孩子的 level 等于其父节点的 level 或等于其父节点的 level 减 1.

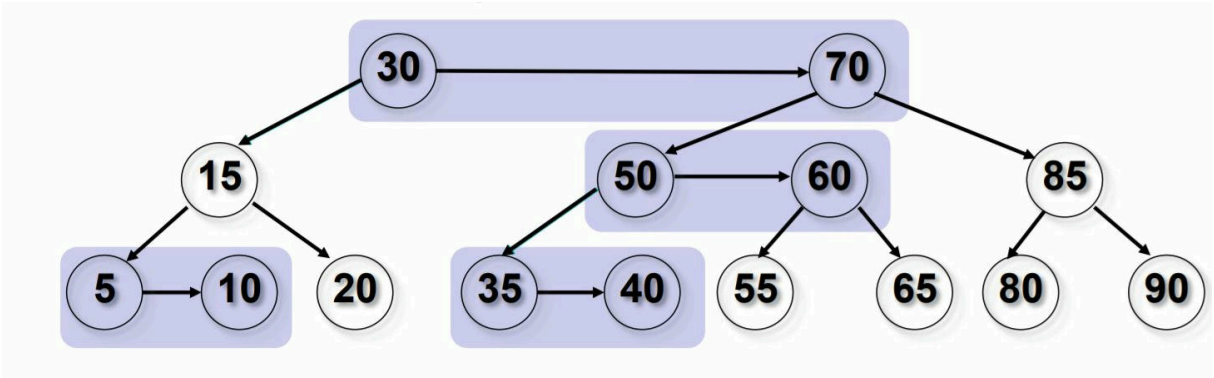
4、每个右孙子的 level 严格小于其祖父节点的 level.

5、每个 level 大于 1 的节点有两个孩子.



水平链接（Horizontal Link）

子节点的 level 等于父节点的 level 的链接被称为 **水平链接**，类似于红黑树中的红链接. 允许单独的右水平链接，但不允许连续的右水平链接；不允许左水平链接. 这些限制比红黑树的限制更加严格，因此 AA 树的平衡过程比红黑树的平衡过程在程序上要简单得多.



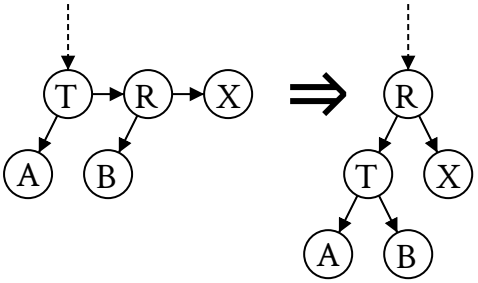
插入和删除操作可能会暂时导致 AA 树失去平衡（即违反 AA 树的不变性）. 恢复平衡只需要两种不同的操作："**skew**"（斜化）和"**split**"（分裂）. "Skew"是将一个包含左水平链接的子树进行右旋转，以替换为一个包含右水平链接的子树. "Split"是进行左旋转并增加 level，以替换一个包含两个或更多连续的右水平链接的子树，使其变为一个包含两个较少连续的右水平链接的子树. 保持平衡的插入和删除的实现通过依赖"skew"和"split"操作来仅在需要时修改树，而不是由调用者决定是否进行"skew"或"split"，从而变得更加简化.

split（左旋）

出现连续向右的水平方向链（连续三个向右的孩子属于同一 level， 节点 R 和节点 X 都是红色节点）.

此时向左旋转节点 *T*，把小于等于此 level 的节点看做一个子树.

1. 子树的根的右孩子变为新的子树根；
2. 原来的子树根变为新子树根的左孩子；
3. 新的子树根 level+1.



伪代码实现

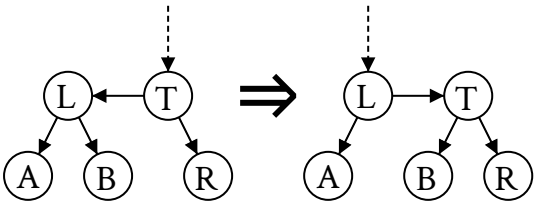
```
function split(root)
    if root → right → right → level == root → level
        rotate_ left(root)
end function
```

skew（右旋）

出现向左的水平方向链（连续两个向左的孩子属于同一 level）

向右旋转节点 T，把小于等于此 level 的节点看做一个子树.

1. 子树的根的左孩子变为新的子树根；
2. 原来的子树根变为新子树根的右孩子.



伪代码实现

```
function skew(root)
    if root → left → level == root → level
        rotate_ right(root)
end function
```

AA 树的操作

AA 树本身是一棵二叉搜索树，所以搜索操作与其他二叉搜索树相同. 插入和删除操作与 *AVL* 树相同，首先在树中将 key 插入或删除，然后沿着搜索路径回退到根，并在此过程中重构树.

插入

伪代码实现

```
1  function insert(root, add)
2      if root == NULL
3          root ← add
4      else if add → key < root → key      //如果允许重复 <=
5          insert(root → left, add)
6      else if add → key > root → key
7          insert(root → right, add)
8      end if
9      //如果不允许重复，在每一 level 上进行 skew 和 split
10     skew(root);
11     split(root);
12 end function
```

删除

删除过程与其他二叉平衡树类似，首先将内部节点的删除转换为叶子节点的删除. 具体方法是将内部节点与它最接近的前驱或后继节点替换. 由于 AA 树的所有 level 大于 1 的节点都有两个子节点，前驱或后继节点将位于 level 1，删除 level 1 的节点较为简单.

伪代码实现

```
1  //To rebalance the tree
2  if root->left->level < root->level − 1 or root->right->level < root->level − 1
3  {
4      if root->right->level >− root->level
5      {
6          root->right->level ← root->level
7      }
8      skew(root)
9      skew(root->right)
10     skew(root->right->right)
11     split(root)
12     split(root->right)
13 }
```

性能

AA 树的性能与红黑树的性能相当. 尽管 AA 树进行的旋转操作比红黑树多，但 AA 树的算法更简单，最终导致相近的性能. 红黑树的性能在各种情况下更加一致，而 AA 树往往更扁平，这使 AA 树有稍快的搜索速度.

参考资料

1. AA tree - Wikipedia[1]
2. Introduction to AA trees[2]
3. AA tree - Visualization[3]
4. CMSC 420 Lecture 6: 2-3, Red-black, and AA trees[4]

外部参考资料

[1] AA tree - Wikipedia

[2] Introduction to AA trees

[3] AA tree - Visualization

[4] CMSC 420 Lecture 6: 2-3, Red-black, and AA trees 

10.16 跳表

跳表 (Skip List) 是由 William Pugh 发明的一种查找数据结构，支持对数据的快速查找，插入和删除.

跳表的期望空间复杂度为 $O(n)$，跳表的查询，插入和删除操作的期望时间复杂度都为 $O(\log n)$.

基本思想

顾名思义，跳表是一种类似于链表的数据结构. 更加准确地说，跳表是对有序链表的改进.

为方便讨论，后续所有有序链表默认为 **升序** 排序.

一个有序链表的查找操作，就是从头部开始逐个比较，直到当前节点的值大于或者等于目标节点的值. 很明显，这个操作的复杂度是 $O(n)$.

跳表在有序链表的基础上，引入了 **分层** 的概念. 首先，跳表的每一层都是一个有序链表，特别地，最底层是初始的有序链表. 每个位于第 i 层的节点有 p 的概率出现在第 $i+1$ 层，p 为常数.

记在 n 个节点的跳表中，期望包含 $\frac{1}{p}$ 个元素的层为第 $L(n)$ 层，易得 $L(n) = \log_{\frac{1}{p}} n$.

在跳表中查找，就是从第 $L(n)$ 层开始，水平地逐个比较直至当前节点的下一个节点大于等于目标节点，然后移动至下一层. 重复这个过程直至到达第一层且无法继续进行操作. 此时，若下一个节点是目标节点，则成功查找；反之，则元素不存在. 这样一来，查找的过程中会跳过一些没有必要的比较，所以相比于有序链表的查询，跳表的查询更快. 可以证明，跳表查询的平均复杂度为 $O(\log n)$.

复杂度证明

空间复杂度

对于一个节点而言，节点的最高层数为 i 的概率为 $p^{i-1}(1-p)$. 所以，跳表的期望层数为 $\sum_{i\geq 1} ip^{i-1}(1-p) = \frac{1}{1-p}$，且因为 p 为常数，所以跳表的 **期望空间复杂度** 为 $O(n)$.

在最坏的情况下，每一层有序链表等于初始有序链表，即跳表的 **最差空间复杂度** 为 $O(n \log n)$.

时间复杂度

从后向前分析查找路径，这个过程可以分为从最底层爬到第 $L(n)$ 层和后续操作两个部分. 在分析时，假设一个节点的具体信息在它被访问之前是未知的.

假设当前我们处于一个第 i 层的节点 x，我们并不知道 x 的最大层数和 x 左侧节点的最大层数，只知道 x 的最大层数至少为 i. 如果 x 的最大层数大于 i，那么下一步应该是向上走，这种情况的概率为 p；如果 x 的最大层数等于 i，那么下一步应该是向左走，这种情况概率为 $1-p$.

令 $C(i)$ 为在一个无限长度的跳表中向上爬 i 层的期望代价，那么有：

$$C(0) = 0$$

$$C(i) = (1-p)(1+C(i)) + p(1+C(i-1))$$

解得 $C(i) = \frac{i}{p}$.

由此可以得出：在长度为 n 的跳表中，从最底层爬到第 $L(n)$ 层的期望步数存在上界 $\frac{L(n)-1}{p}$.

现在只需要分析爬到第 $L(n)$ 层后还要再走多少步. 易得，到了第 $L(n)$ 层后，向左走的步数不会超过第 $L(n)$ 层及更高层的节点数总和，而这个总和的期望为 $\frac{1}{p}$. 所以到了第 $L(n)$ 层后向左走的期望步数存在上界 $\frac{1}{p}$. 同理，到了第 $L(n)$ 层后向上走的期望步数存在上界 $\frac{1}{p}$.

所以，跳表查询的期望查找步数为 $\frac{L(n)-1}{p} + \frac{2}{p}$，又因为 $L(n) = \log_{\frac{1}{p}} n$，所以跳表查询的 **期望时间复杂度** 为 $O(\log n)$.

在最坏的情况下，每一层有序链表等于初始有序链表，查找过程相当于对最高层的有序链表进行查询，即跳表查询操作的 **最差时间复杂度** 为 $O(n)$.

插入操作和删除操作就是进行一遍查询的过程，途中记录需要修改的节点，最后完成修改. 易得每一层至多只需要修改一个节点，又因为跳表期望层数为 $\log_{\frac{1}{p}} n$，所以插入和修改的 **期望时间复杂度** 也为 $O(\log n)$.

具体实现

获取节点的最大层数

模拟以 p 的概率往上加一层，最后和上限值取最小.

```
int randomLevel() {
  int lv = 1;
  // MAXL = 32, S = 0xFFFF, PS = S * P, P = 1 / 4
  while ((rand() & S) < PS) ++lv;
  return min(MAXL, lv);
}
```

查询

查询跳表中是否存在键值为 `key` 的节点. 具体实现时，可以设置两个哨兵节点以减少边界条件的讨论.

```
V& find(const K& key) {
  SkipListNode<K, V>* p = head;

  // 找到该层最后一个键值小于 key 的节点，然后走向下一层
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
      p = p->forward[i];
    }
  }
  // 现在是小于，所以还需要再往后走一步
  p = p->forward[0];

  // 成功找到节点
  if (p->key == key) return p->value;

  // 节点不存在，返回 INVALID
  return tail->value;
}
```

插入

插入节点 `(key, value)`. 插入节点的过程就是先执行一遍查询的过程，中途记录新节点是要插入哪一些节点的后面，最后再执行插入. 每一层最后一个键值小于 `key` 的节点，就是需要进行修改的节点.

```
void insert(const K &key, const V &value) {
  // 用于记录需要修改的节点
  SkipListNode<K, V> *update[MAXL + 1];

  SkipListNode<K, V> *p = head;
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
```

```
      p = p->forward[i];
    }
    // 第 i 层需要修改的节点为 p
    update[i] = p;
  }
  p = p->forward[0];

  // 若已存在则修改
  if (p->key == key) {
    p->value = value;
    return;
  }

  // 获取新节点的最大层数
  int lv = randomLevel();
  if (lv > level) {
    lv = ++level;
    update[lv] = head;
  }

  // 新建节点
  SkipListNode<K, V> *newNode = new SkipListNode<K, V>(key, value, lv);
  // 在第 0~lv 层插入新节点
  for (int i = lv; i >= 0; --i) {
    p = update[i];
    newNode->forward[i] = p->forward[i];
    p->forward[i] = newNode;
  }

  ++length;
}
```

删除

删除键值为 `key` 的节点. 删除节点的过程就是先执行一遍查询的过程, 中途记录要删的节点是在哪一些节点的后面，最后再执行删除. 每一层最后一个键值小于 `key` 的节点，就是需要进行修改的节点.

```
bool erase(const K &key) {
  // 用于记录需要修改的节点
  SkipListNode<K, V> *update[MAXL + 1];

  SkipListNode<K, V> *p = head;
  for (int i = level; i >= 0; --i) {
    while (p->forward[i]->key < key) {
      p = p->forward[i];
    }
    // 第 i 层需要修改的节点为 p
    update[i] = p;
  }
  p = p->forward[0];

  // 节点不存在
  if (p->key != key) return false;

```

```
  // 从最底层开始删除
  for (int i = 0; i <= level; ++i) {
    // 如果这层没有 p 删除就完成了
    if (update[i]->forward[i] != p) {
      break;
    }
    // 断开 p 的连接
    update[i]->forward[i] = p->forward[i];
  }

  // 回收空间
  delete p;

  // 删除节点可能导致最大层数减少
  while (level > 0 && head->forward[level] == tail) --level;

  // 跳表长度
  --length;
  return true;
}
```

完整代码

下列代码是用跳表实现的 map. 未经正经测试, 仅供参考.

参考代码

```
#include <cassert>
#include <climits>
#include <ctime>
#include <iostream>
#include <map>
using namespace std;

template <typename K, typename V>
struct SkipListNode {
  int level;
  K key;
  V value;
  SkipListNode **forward;

  SkipListNode() {}

  SkipListNode(K k, V v, int l, SkipListNode *nxt = NULL) {
    key = k;
    value = v;
    level = l;
    forward = new SkipListNode *[l + 1];
    for (int i = 0; i <= l; ++i) forward[i] = nxt;
  }

  ~SkipListNode() {
    if (forward != NULL) delete[] forward;
```

```
  }
};

template <typename K, typename V>
struct SkipList {
  static constexpr int MAXL = 32;
  static constexpr int P = 4;
  static constexpr int S = 0xFFFF;
  static constexpr int PS = S / P;
  static constexpr int INVALID = INT_MAX;

  SkipListNode<K, V> *head, *tail;
  int length;
  int level;

  SkipList() {
    srand(time(nullptr));

    level = length = 0;
    tail = new SkipListNode<K, V>(INVALID, 0, 0);
    head = new SkipListNode<K, V>(INVALID, 0, MAXL, tail);
  }

  ~SkipList() {
    delete head;
    delete tail;
  }

  int randomLevel() {
    int lv = 1;
    while ((rand() & S) < PS) ++lv;
    return MAXL > lv ? lv : MAXL;
  }

  void insert(const K &key, const V &value) {
    SkipListNode<K, V> *update[MAXL + 1];

    SkipListNode<K, V> *p = head;
    for (int i = level; i >= 0; --i) {
      while (p->forward[i]->key < key) {
        p = p->forward[i];
      }
      update[i] = p;
    }
    p = p->forward[0];

    if (p->key == key) {
      p->value = value;
      return;
    }

    int lv = randomLevel();
```

```
79      if (lv > level) {
80        lv = ++level;
81        update[lv] = head;
82      }
83
84      SkipListNode<K, V> *newNode = new SkipListNode<K, V>(key, value, lv);
85      for (int i = lv; i >= 0; --i) {
86        p = update[i];
87        newNode->forward[i] = p->forward[i];
88        p->forward[i] = newNode;
89      }
90
91      ++length;
92    }
93
94    bool erase(const K &key) {
95      SkipListNode<K, V> *update[MAXL + 1];
96      SkipListNode<K, V> *p = head;
97
98      for (int i = level; i >= 0; --i) {
99        while (p->forward[i]->key < key) {
100         p = p->forward[i];
101       }
102       update[i] = p;
103     }
104     p = p->forward[0];
105
106     if (p->key != key) return false;
107
108     for (int i = 0; i <= level; ++i) {
109       if (update[i]->forward[i] != p) {
110         break;
111       }
112       update[i]->forward[i] = p->forward[i];
113     }
114
115     delete p;
116
117     while (level > 0 && head->forward[level] == tail) --level;
118     --length;
119     return true;
120   }
121
122   V &operator[](const K &key) {
123     V v = find(key);
124     if (v == tail->value) insert(key, 0);
125     return find(key);
126   }
127
128   V &find(const K &key) {
129     SkipListNode<K, V> *p = head;
130     for (int i = level; i >= 0; --i) {
```

```
131       while (p->forward[i]->key < key) {
132         p = p->forward[i];
133       }
134     }
135     p = p->forward[0];
136     if (p->key == key) return p->value;
137     return tail->value;
138   }
139
140   bool count(const K &key) { return find(key) != tail->value; }
141 };
142
143 int main() {
144   SkipList<int, int> L;
145   map<int, int> M;
146
147   clock_t s = clock();
148
149   for (int i = 0; i < 1e5; ++i) {
150     int key = rand(), value = rand();
151     L[key] = value;
152     M[key] = value;
153   }
154
155   for (int i = 0; i < 1e5; ++i) {
156     int key = rand();
157     if (i & 1) {
158       L.erase(key);
159       M.erase(key);
160     } else {
161       int r1 = L.count(key) ? L[key] : 0;
162       int r2 = M.count(key) ? M[key] : 0;
163       assert(r1 == r2);
164     }
165   }
166
167   clock_t e = clock();
168   cout << "Time elapsed: " << (double)(e - s) / CLOCKS_PER_SEC << endl;
169   // about 0.2s
170
171   return 0;
172 }
```

跳表的随机访问优化

访问跳表中第 k 个节点，相当于访问初始有序链表中的第 k 个节点，很明显这个操作的时间复杂度是 $O(n)$ 的，并不足够优秀.

跳表的随机访问优化就是对每一个前向指针，再多维护这个前向指针的长度. 假设 A 和 B 都是跳表中的节点，其中 A 为跳表的第 a 个节点，B 为跳表的第 b 个节点 $(a < b)$，且在跳表的某一层中 A 的前向指针指向 B，那么这个前向指针的长度为 $b - a$.

现在访问跳表中的第 k 个节点，就可以从顶层开始，水平地遍历该层的链表，直到当前节点的位置加上当前节点在该层的前向指针长度大于等于 k，然后移动至下一层．重复这个过程直至到达第一层且无法继续行操作．此时，当前节点就是跳表中第 k 个节点．

这样，就可以快速地访问到跳表的第 k 个元素．可以证明，这个操作的时间复杂度为 $O(\log n)$．

参考资料

1. Skip Lists: A Probabilistic Alternative to Balanced Trees[1]
2. Skip List[2]
3. A Skip List Cookbook[3]

外部参考资料

[1] Skip Lists: A Probabilistic Alternative to Balanced Trees

[2] Skip List

[3] A Skip List Cookbook

10.17 可持久化数据结构

10.17.1 可持久化数据结构简介

Authors: morris821028

简介

可持久化数据结构 (Persistent data structure) 总是可以保留每一个历史版本，并且支持操作的不可变特性 (immutable).

可持久化分类

部分可持久化 (Partially Persistent)

所有版本都可以访问，但是只有最新版本可以修改.

完全可持久化 (Fully Persistent)

所有版本都既可以访问又可以修改.

若支持将两个历史版本合并，则又称为 Confluently Persistent

实际应用

几何计算

在几何计算中有许多离线算法, 如扫描线算法一次扫过去回答所有询问, 在时间复杂度分析上相当优异．但强迫在线的情况下，每一次都扫描一次，询问操作的时间复杂度就从对数时间降成线性．为了解决这一种情况, 持久化技术给了另一种思维, 我们将扫描线的时间轴作为一个变动依据, 持久化相关的结构, 只要我们能将询问在对数时间内穿梭于这个时间轴，必能动态解决先前的问题.

字串处理

为了达到非常高效率的合并操作, 防止大量重复性字串的生成伴随的效能退化, 使得各方面的操作都能远低于线性操作．如 C++ rope 就是一个持久化的数据结构．不只是字串操作，若处理类型有大量重复的情况，持久化的概念便能派上用场.

版本回溯

实际上就是对应大部分的应用软体中的 redo/undo. 如果资料库/操作变动为了高效率操作而会配上复杂的结构（并不像 hash, set 反转操作只需要常数或对数时间），那么为了快速回推变动结果，持久化结构就是要减少 redo/undo 的花费.

资料库本身可以常数回推，纪录变动的部分情况即可. 而应用层的计算，大部分实作都是砍掉快取，并且重新计算出一份新的结构，有时候回推的变动大小为 m，为了重新计算结构而消耗了 n+m，如果 n 和 m 的差距非常大，那连续回推的体感就很糟糕.

函数式编程

函数式编程需要特别的数据结构以符合语言特性，其中不可变的性质更为重要，以利于并行环境与除错. 如面向对象编程的 Java 8 后引入 stream 类，支援写出函数式的语法设计，可提供惰性求值、无限值域等的特殊功能.

参考

- https://en.wikipedia.org/wiki/Persistent_data_structure[1]
- MIT 课程 https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2005/lecture-notes/persistent.pdf[2]

外部参考资料

[1] https://en.wikipedia.org/wiki/Persistent_data_structure

[2] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-854j-advanced-algorithms-fall-2005/lecture-notes/persistent.pdf

10.17.2 可持久化线段树

主席树

主席树全称是可持久化权值线段树，参见 知乎讨论[1].

> ⚠ **关于函数式线段树**
>
> **函数式线段树** 是指使用函数式编程思想的线段树. 在函数式编程思想中，将计算机运算视为数学函数，并避免可改变的状态或变量. 不难发现，函数式线段树是 完全可持久化→10.17.1@p2423 的.

引入

先引入一道题目：给定 n 个整数构成的序列 a，将对于指定的闭区间 $[l,r]$ 查询其区间内的第 k 小值.

你该如何解决?

一种可行的方案是：使用主席树. 主席树的主要思想就是：保存每次插入操作时的历史版本，以便查询区间第 k 小.

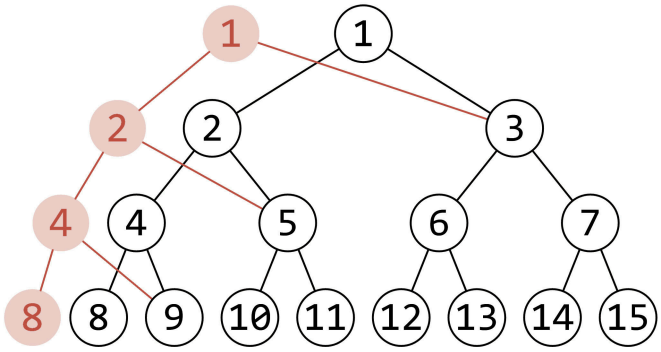
怎么保存呢? 简单暴力一点，每次开一棵线段树呗.

那空间还不爆掉?

解释

我们分析一下，发现每次修改操作修改的点的个数是一样的.

（例如下图，修改了 $[1,8]$ 中对应权值为 1 的结点，红色的点即为更改的点）



只更改了 $O(\log n)$ 个结点，形成一条链，也就是说每次更改的结点数 = 树的高度.

注意主席树不能使用堆式存储法，就是说不能用 $x \times 2$，$x \times 2 + 1$ 来表示左右儿子，而是应该动态开点，并保存每个节点的左右儿子编号.

所以我们只要在记录左右儿子的基础上，保存插入每个数的时候的根节点就可以实现持久化了.

我们把问题简化一下：每次求 $[1, r]$ 区间内的 k 小值.

怎么做呢？只需要找到插入 r 时的根节点版本，然后用普通权值线段树（有的叫键值线段树/值域线段树）做就行了.

这个相信大家都能理解，回到原问题——求 $[l, r]$ 区间 k 小值.

这里我们再联系另外一个知识：**前缀和**.

这个小东西巧妙运用了区间减法的性质，通过预处理从而达到 $O(1)$ 回答每个询问.

我们可以发现，主席树统计的信息也满足这个性质.

所以……如果需要得到 $[l, r]$ 的统计信息，只需要用 $[1, r]$ 的信息减去 $[1, l-1]$ 的信息就行了.

至此，该问题解决！

关于空间问题，我们分析一下：由于我们是动态开点的，所以一棵线段树只会出现 $2n - 1$ 个结点.

然后，有 n 次修改，每次至多增加 $\lceil \log_2 n \rceil + 1$ 个结点. 因此，最坏情况下 n 次修改后的结点总数会达到 $2n - 1 + n(\lceil \log_2 n \rceil + 1)$. 此题的 $n \le 10^5$，单次修改至多增加 $\lceil \log_2 10^5 \rceil + 1 = 18$ 个结点，故 n 次修改后的结点总数为 $2 \times 10^5 - 1 + 18 \times 10^5$，忽略掉 -1，大概就是 20×10^5.

最后给一个忠告：千万不要吝啬空间（大多数题目中空间限制都较为宽松，因此一般不用担心空间超限的问题）！大胆一点，直接上个 $2^5 \times 10^5$，接近原空间的两倍（即 `n << 5`）.

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
constexpr int MAXN = 1e5;  // 数据范围
int tot, n, m;
int sum[(MAXN << 5) + 10], rt[MAXN + 10], ls[(MAXN << 5) + 10],
    rs[(MAXN << 5) + 10];
int a[MAXN + 10], ind[MAXN + 10], len;

int getid(const int &val) {  // 离散化
  return lower_bound(ind + 1, ind + len + 1, val) - ind;
}

```

```
int build(int l, int r) {  // 建树
  int root = ++tot;
  if (l == r) return root;
  int mid = l + r >> 1;
  ls[root] = build(l, mid);
  rs[root] = build(mid + 1, r);
  return root;  // 返回该子树的根节点
}

int update(int k, int l, int r, int root) {  // 插入操作
  int dir = ++tot;
  ls[dir] = ls[root], rs[dir] = rs[root], sum[dir] = sum[root] + 1;
  if (l == r) return dir;
  int mid = l + r >> 1;
  if (k <= mid)
    ls[dir] = update(k, l, mid, ls[dir]);
  else
    rs[dir] = update(k, mid + 1, r, rs[dir]);
  return dir;
}

int query(int u, int v, int l, int r, int k) {  // 查询操作
  int mid = l + r >> 1,
      x = sum[ls[v]] - sum[ls[u]];  // 通过区间减法得到左儿子中所存储的数值个数
  if (l == r) return l;
  if (k <= x)  // 若 k 小于等于 x ，则说明第 k 小的数字存储在左儿子中
    return query(ls[u], ls[v], l, mid, k);
  else  // 否则说明在右儿子中
    return query(rs[u], rs[v], mid + 1, r, k - x);
}

void init() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; ++i) scanf("%d", a + i);
  memcpy(ind, a, sizeof ind);
  sort(ind + 1, ind + n + 1);
  len = unique(ind + 1, ind + n + 1) - ind - 1;
  rt[0] = build(1, len);
  for (int i = 1; i <= n; ++i) rt[i] = update(getid(a[i]), 1, len, rt[i - 1]);
}

int l, r, k;

void work() {
  while (m--) {
    scanf("%d%d%d", &l, &r, &k);
    printf("%d\n", ind[query(rt[l - 1], rt[r], 1, len, k)]);  // 回答询问
  }
}

int main() {
  init();
  work();
  return 0;
```

```
}
```

拓展：基于主席树的可持久化并查集

主席树是实现可持久化并查集的便捷方式，在此也提供一个基于主席树的可持久化并查集实现示例.

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

struct SegmentTree {
  int lc, rc, val, rnk;
};

constexpr int MAXN = 100000 + 5;
constexpr int MAXM = 200000 + 5;

SegmentTree
    t[MAXN * 2 +
      MAXM * 40];  // 每次操作1会修改两次，一次修改父节点，一次修改父节点的秩
int rt[MAXM];
int n, m, tot;

int build(int l, int r) {
  int p = ++tot;
  if (l == r) {
    t[p].val = l;
    t[p].rnk = 1;
    return p;
  }
  int mid = (l + r) / 2;
  t[p].lc = build(l, mid);
  t[p].rc = build(mid + 1, r);
  return p;
}

int getRnk(int p, int l, int r, int pos) {  // 查询秩
  if (l == r) {
    return t[p].rnk;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return getRnk(t[p].lc, l, mid, pos);
  } else {
    return getRnk(t[p].rc, mid + 1, r, pos);
  }
}

int modifyRnk(int now, int l, int r, int pos, int val) {  // 修改秩（高度）
  int p = ++tot;
  t[p] = t[now];
  if (l == r) {
    t[p].rnk = max(t[p].rnk, val);
    return p;
  }
```

```
  int mid = (l + r) / 2;
  if (pos <= mid) {
    t[p].lc = modifyRnk(t[now].lc, l, mid, pos, val);
  } else {
    t[p].rc = modifyRnk(t[now].rc, mid + 1, r, pos, val);
  }
  return p;
}

int query(int p, int l, int r, int pos) {  // 查询父节点（序列中的值）
  if (l == r) {
    return t[p].val;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    return query(t[p].lc, l, mid, pos);
  } else {
    return query(t[p].rc, mid + 1, r, pos);
  }
}

int findRoot(int p, int pos) {  // 查询根节点
  int f = query(p, 1, n, pos);
  if (pos == f) {
    return pos;
  }
  return findRoot(p, f);
}

int modify(int now, int l, int r, int pos, int fa) {  // 修改父节点（合并）
  int p = ++tot;
  t[p] = t[now];
  if (l == r) {
    t[p].val = fa;
    return p;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    t[p].lc = modify(t[now].lc, l, mid, pos, fa);
  } else {
    t[p].rc = modify(t[now].rc, mid + 1, r, pos, fa);
  }
  return p;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  rt[0] = build(1, n);
  for (int i = 1; i <= m; i++) {
    int op, a, b;

    cin >> op;
    if (op == 1) {
```

```
104       cin >> a >> b;
105       int fa = findRoot(rt[i - 1], a), fb = findRoot(rt[i - 1], b);
106       if (fa != fb) {
107         if (getRnk(rt[i - 1], 1, n, fa) >
108             getRnk(rt[i - 1], 1, n, fb)) {  // 按秩合并
109           swap(fa, fb);
110         }
111         int tmp = modify(rt[i - 1], 1, n, fa, fb);
112         rt[i] = modifyRnk(tmp, 1, n, fb, getRnk(rt[i - 1], 1, n, fa) + 1);
113       } else {
114         rt[i] = rt[i - 1];
115       }
116     } else if (op == 2) {
117       cin >> a;
118       rt[i] = rt[a];
119     } else {
120       cin >> a >> b;
121       rt[i] = rt[i - 1];
122       cout << (findRoot(rt[i], a) == findRoot(rt[i], b)) << '\n';
123     }
124   }
125
126   return 0;
127 }
128
```

参考

https://en.wikipedia.org/wiki/Persistent_data_structure[2]

https://www.cnblogs.com/zinthos/p/3899565.html[3]

外部参考资料

[1] 知乎讨论



[2] https://en.wikipedia.org/wiki/Persistent_data_structure



[3] https://www.cnblogs.com/zinthos/p/3899565.html



10.17.3 可持久化块状数组

10.17.4 可持久化平衡树

可持久化无旋转 Treap

前置知识

OI 常用的可持久化平衡树 一般就是 **可持久化无旋转 Treap** 所以推荐首先学习 **无旋转 Treap**→10.15.2@p2221.

思想/做法

对于非旋转 Treap，可通过 **Merge** 和 **Split** 操作过程中复制路径上经过的节点（一般在 **Split** 操作中复制，确保不影响以前的版本）就可完成可持久化.

对于旋转 Treap，在复制路径上经过的节点同时，还需复制受旋转影响的节点（若其已为这次操作中复制的节点，则无需再复制），对于一次旋转一般只影响两个节点，那么不会增加其时间复杂度.

上述方法一般被称为 path copying.

「一切可支持操作都可以通过 **Merge Split Newnode Build** 完成」，而 **Build** 操作只用于建造无需理会，**Newnode**（新建节点）就是用来可持久化的工具.

我们来观察一下 **Merge** 和 **Split**，我们会发现它们都是由上而下的操作！

因此我们完全可以 **参考线段树的可持久化操作** 对它进行可持久化.

可持久化操作

可持久化 是对 **数据结构** 的一种操作，即保留历史信息，使得在后面可以调用之前的历史版本.

对于 **可持久化线段树** 来说，每一次新建历史版本就是把 **沿途的修改路径** 复制出来

那么对可持久化 Treap（目前国内 OI 常用的版本）来说：

在复制一个节点 X_a（X 节点的第 a 个版本）的新版本 X_{a+1}（X 节点的第 $a+1$ 个版本）以后：

- 如果某个儿子节点 Y 不用修改信息，那么就把 X_{a+1} 的指针直接指向 Y_a（Y 节点的第 a 个版本）即可.
- 反之，如果要修改 Y，那么就在 **递归到下层** 时 **新建** Y_{a+1}（Y 节点的第 $a+1$ 个版本）这个新节点用于 **存储新的信息**，同时把 X_{a+1} 的指针指向 Y_{a+1}（Y 节点的第 $a+1$ 个版本）.

可持久化

需要的东西：

- 一个 `struct` 数组 存 **每个节点** 的信息（一般叫做 `tree` 数组）；（当然写 **指针版** 平衡树的大佬就可以考虑不用这个数组了）
- 一个 **根节点数组**，存每个版本的树根，每次查询版本信息时就从 **根数组存的节点** 开始；
- `split()` 分裂 **从树中分裂出两棵树**
- `merge()` 合并 **把两棵树按照随机权值合并**
- `newNode()` 新建一个节点
- `build()` 建树

Split

对于 **分裂操作**，每次分裂路径时 **新建节点** 指向分出来的路径，用 `std::pair` 存新分裂出来的两棵树的根.

`split(x,k)` 返回一个 `std::pair`

表示把 $_x$ 为根的树的前 k 个元素放在 **一棵树** 中，剩下的节点构成在另一棵树中，返回这两棵树的根（first 是第一棵树的根，second 是第二棵树的）.

- 如果 x 的 **左子树** 的 $key \geq k$，那么 **直接递归进左子树**，把左子树分出来的第二颗树和当前的 x **右子树** 合并.
- 否则递归 **右子树**.

```
1  static std::pair<int, int> _split(int _x, int k) {
2    if (_x == 0)
3      return std::make_pair(0, 0);
4    else {
5      int _vs = ++_cnt;  // 新建节点（可持久化的精髓）
6      _trp[_vs] = _trp[_x];
7      std::pair<int, int> _y;
8      if (_trp[_vs].key <= k) {
9        _y = _split(_trp[_vs].leaf[1], k);
10       _trp[_vs].leaf[1] = _y.first;
11       _y.first = _vs;
12     } else {
13       _y = _split(_trp[_vs].leaf[0], k);
```

```
      _trp[_vs].leaf[0] = _y.second;
      _y.second = _vs;
    }
    _trp[_vs]._update();
    return _y;
  }
}
```

Merge

`merge(x,y)` 返回 merge 出的树的根.

同样递归实现. 如果 **x 的随机权值>y 的随机权值**, 则 `merge(x_{rc},y)`, 否则 `merge(x,y_{lc})`.

```
static int _merge(int _x, int _y) {
  if (_x == 0 || _y == 0)
    return _x ^ _y;
  else {
    if (_trp[_x].fix < _trp[_y].fix) {
      _trp[_x].leaf[1] = _merge(_trp[_x].leaf[1], _y);
      _trp[_x]._update();
      return _x;
    } else {
      _trp[_y].leaf[0] = _merge(_x, _trp[_y].leaf[0]);
      _trp[_y]._update();
      return _y;
    }
  }
}
```

可持久化 WBLT

前置知识

可持久化 WBLT 由 WBLT 改动而来, 所以首先学习 WBLT→10.15.4@p2272.

思想/做法

使用 **路径复制** 的方法, 将一次操作中 **修改过** 的节点复制下来, 不能影响之前的节点.

处理懒标记

为了处理懒标记, 我们这样考虑: 在一棵持久化的 WBLT 上, 一个点可能有多个父亲, 但是儿子数量只能是 0 或 2 个. pushdown 的下放懒标记的操作, 只会影响它的儿子, 我们对一个点进行 pushdown, 是没有影响的; 反而是它的儿子, 它的儿子可能不止它一个父亲, 将它的标记下放到儿子, 可能导致在别的父亲的版本上, 多了一个不属于那个版本的懒标记, 这就错了; 除非它的儿子只有它一个父亲. 所以我们应该在 pushdown 的时候, 复制一遍儿子, 把懒标记打到新的儿子上.

实现路径复制

在进行路径复制的时候, 我们可以定义一个 refresh 函数, 它接受一个节点 p 的引用, 表示把节点 p 复制一下, 产生一个新的节点, 重新赋值给 p. 使用 refresh 函数的原则是, 如果它将要被修改, 或者它拥有的儿子即将发生变动 (而不是它的儿子的信息将要被修改), 那么就 refresh 它, 否则不需要.

对于静态的查询, 除了 pushdown 之外都不用 refresh. 如果保证什么操作都做路径复制, 那么 pushdown 和 refresh 的顺序是无所谓的.

针对持久化 WBLT 的小优化

这里有一个优化．观察到 pushdown 的时候要复制两个节点，可以写标记永久化，但是刚才说了，如果它的儿子只有它一个父亲，可以不用复制．针对这一个性质，可以进行优化，以减少复制多余的节点．

考虑记录每个节点有多少个父亲（认为每个版本的根都有一个父亲），记为 use．每次 refresh 的时候，如果 $use \le 1$ 则不需要重新复制节点，否则新建节点，并且 use 自减 1，表示父亲带着这个儿子跑了，这样父亲就可以随意修改新的节点而不影响其它版本．另外每次复制节点的时候，如果节点有儿子，那么两个儿子的 use 自增 1；合并两个子树时，返回的节点对两个儿子也有一个父亲的 use；删除节点时，两个子节点都丢失一个父亲：这样能优化一些时空．

代码实现

完整代码（可持久化文艺平衡树）

```cpp
#include <cstring>
#include <iostream>
#include <utility>
using namespace std;
using LL = long long;

template <int N>
struct WBLT {
  static constexpr double alpha = 0.292;
  int ch[N << 1][2], siz[N << 1], tot, root, tsh[N << 1], tct;
  int val[N << 1];
  LL sum[N << 1];
  bool rev[N << 1];
  int use[N << 1];

  WBLT() { root = newnode(-(int)((1u << 31) - 1)); }

  bool isleaf(int p) { return !ch[p][0]; }

  void destroy(int p) { tsh[++tct] = p; }

  void clone(int p, int q) {
    memcpy(ch[p], ch[q], sizeof ch[0]);
    val[p] = val[q];
    siz[p] = siz[q];
    sum[p] = sum[q];
    rev[p] = rev[q];
    if (!isleaf(p)) {
      use[ch[p][0]] += 1;
      use[ch[p][1]] += 1;
    }
  }

  int newnode(LL v) {
    int p = tct ? tsh[tct--] : ++tot;
    memset(ch[p], 0, sizeof ch[p]);
    val[p] = v;
    siz[p] = 1;
    sum[p] = v;
```

```
    rev[p] = false;
    use[p] = 1;
    return p;
  }

  void refresh(int &p) {
    if (use[p] <= 1) return;
    use[p] -= 1;
    int q = exchange(p, newnode(0));
    clone(p, q);
  }

  void maintain(int p) {  // also known as: pushup
    if (isleaf(p)) return;
    val[p] = val[ch[p][1]];
    sum[p] = sum[ch[p][0]] + sum[ch[p][1]];
    siz[p] = siz[ch[p][0]] + siz[ch[p][1]];
  }

  void spread(int &p) {
    if (isleaf(p)) return;
    refresh(p);
    rev[p] ^= 1;
  }

  void pushdown(int p) {
    if (!rev[p] || isleaf(p)) return;
    spread(ch[p][0]), spread(ch[p][1]);
    swap(ch[p][0], ch[p][1]);
    rev[p] = false;
  }

  void rotate(int p, int r) {
    if (isleaf(p) || isleaf(ch[p][r])) return;
    refresh(ch[p][r]);
    pushdown(ch[p][r]);
    int q = ch[p][r];
    swap(ch[p][0], ch[p][1]);
    swap(ch[p][r], ch[q][r]);
    swap(ch[q][0], ch[q][1]);
    maintain(q);
    maintain(p);
  }

  void update(int p) {  // also known as: maintain
    if (isleaf(p)) return;
    int r = siz[ch[p][0]] < siz[ch[p][1]];
    if (siz[ch[p][!r]] >= siz[p] * alpha) return;
    refresh(ch[p][r]);
    pushdown(ch[p][r]);
    if (siz[ch[ch[p][r]][!r]] >= siz[ch[p][r]] * (1 - alpha * 2) / (1 -
alpha))
```

```
91        rotate(ch[p][r], !r);
92      rotate(p, r);
93    }
94
95    void insert(int &p, int v, int k) {
96      refresh(p);
97      pushdown(p);
98      int r = siz[ch[p][0]] < k;
99      if (isleaf(p)) {
100       ch[p][0] = newnode(val[p]);
101       ch[p][1] = newnode(v);
102     } else {
103       if (r) k -= siz[ch[p][0]];
104       insert(ch[p][r], v, k);
105     }
106     maintain(p);
107     update(p);
108   }
109
110   void erase(int &p, int k) {
111     refresh(p);
112     pushdown(p);
113     int r = siz[ch[p][0]] < k;
114     if (isleaf(ch[p][r])) {
115       use[ch[p][0]] -= 1;
116       use[ch[p][1]] -= 1;
117       clone(p, ch[p][!r]);
118     } else {
119       if (r) k -= siz[ch[p][0]];
120       erase(ch[p][r], k);
121     }
122     maintain(p);
123     update(p);
124   }
125
126   int merge(int p, int q) {
127     if (!p || !q) return p + q;
128     if (min(siz[p], siz[q]) >= alpha * (siz[p] + siz[q])) {
129       int t = newnode(0);
130       ch[t][0] = p, use[p] += 1;
131       ch[t][1] = q, use[q] += 1;
132       maintain(t);
133       return t;
134     }
135     if (siz[p] >= siz[q]) {
136       pushdown(p);
137       if (siz[ch[p][0]] >= alpha * (siz[p] + siz[q])) {
138         return merge(ch[p][0], merge(ch[p][1], q));
139       } else {
140         pushdown(ch[p][1]);
141         return merge(merge(ch[p][0], ch[ch[p][1]][0]),
142                      merge(ch[ch[p][1]][1], q));
```

```
      }
    } else {
      pushdown(q);
      if (siz[ch[q][1]] >= alpha * (siz[p] + siz[q])) {
        return merge(merge(p, ch[q][0]), ch[q][1]);
      } else {
        pushdown(ch[q][0]);
        return merge(merge(p, ch[ch[q][0]][0]),
                     merge(ch[ch[q][0]][1], ch[q][1]));
      }
    }
  }

  void split(int p, int k, int &x, int &y) {
    if (!k) return x = 0, y = p, void();
    if (isleaf(p)) return x = p, y = 0, void();
    pushdown(p);
    if (k <= siz[ch[p][0]]) {
      split(ch[p][0], k, x, y);
      y = merge(y, ch[p][1]);
    } else {
      split(ch[p][1], k - siz[ch[p][0]], x, y);
      x = merge(ch[p][0], x);
    }
  }

  LL getsum(int L, int R, int &p, int l, int r) {
    if (L <= l && r <= R) return sum[p];
    pushdown(p);
    int mid = l + siz[ch[p][0]] - 1;
    LL ret = 0;
    if (L <= mid) ret += getsum(L, R, ch[p][0], l, mid);
    if (mid < R) ret += getsum(L, R, ch[p][1], mid + 1, r);
    return ret;
  }

  LL getsum(int &p, int L, int R) { return getsum(L + 1, R + 1, p, 1,
siz[p]); }
};

WBLT<6400010> t;
int m;
int root[500010];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> m;
  root[0] = t.root;
  LL lastans = 0;
  for (int i = 1; i <= m; i++) {
    LL op, l, r;
    int v;
```

```
    cin >> v >> op >> l;
    t.use[root[i] = root[v]] += 1;
    if (op != 2) cin >> r;
    l ^= lastans, r ^= lastans;
    int x, y, z;
    switch (op) {
      case 1:
        t.insert(root[i], r, l + 1);
        break;
      case 2:
        t.erase(root[i], l + 1);
        break;
      case 3:
        t.split(root[i], l, x, y);
        t.split(y, r - l + 1, y, z);
        t.spread(y);
        root[i] = t.merge(x, t.merge(y, z));
        break;
      case 4:
        cout << (lastans = t.getsum(root[i], l, r)) << endl;
        break;
    }
  }
  return 0;
}

```

例题

洛谷 P3835【模版】可持久化平衡树[1]

你需要实现一个数据结构，要求提供如下操作（最开始时数据结构内无数据）：

1. 插入 x 数；
2. 删除 x 数（若有多个相同的数，应只删除一个，如果没有请忽略该操作）；
3. 查询 x 数的排名（排名定义为比当前数小的数的个数 $+1$）；
4. 查询排名为 x 的数；
5. 求 x 的前驱（前驱定义为小于 x，且最大的数，如不存在输出 $-2\,147\,483\,647$）；
6. 求 x 的后继（后继定义为大于 x，且最小的数，如不存在输出 $2\,147\,483\,647$）.

以上操作均基于某一个历史版本，同时生成一个新的版本（操作 $3, 4, 5, 6$ 即保持原版本无变化）. 而每个版本的编号则为操作的序号. 特别地，最初的版本编号为 0.

就是 **普通平衡树** 一题的可持久化版，操作和该题类似.

只是使用了可持久化的 merge 和 split 操作.

推荐的练手题

1. 「Luogu P3919」可持久化数组（模板题）[2]
2. 「Codeforces 702F」T-shirt[3]
3. 「Luogu P5055」可持久化文艺平衡树[4]
4. 「Luogu P5350」序列[5]

外部参考资料

[1] 洛谷 P3835【模版】可持久化平衡树 

[2] 「Luogu P3919」可持久化数组（模板题） 

[3] 「Codeforces 702F」T-shirt 

[4] 「Luogu P5055」可持久化文艺平衡树 

[5] 「Luogu P5350」序列 

10.17.5 可持久化字典树

引入

可持久化 Trie 的方式和可持久化线段树的方式是相似的，即每次只修改被添加或值被修改的节点，而保留没有被改动的节点，在上一个版本的基础上连边，使最后每个版本的 Trie 树的根遍历所能分离出的 Trie 树都是完整且包含全部信息的.

大部分的可持久化 Trie 题中，Trie 都是以 01-Trie→8.6@p903 的形式出现的.

例题 最大异或和[1]

对一个长度为 n 的数组 a 维护以下操作：

1. 在数组的末尾添加一个数 x，数组的长度 n 自增 1.
2. 给出查询区间 $[l,r]$ 和一个值 k，求当 $l \le p \le r$ 时，$k \oplus \bigoplus_{i=p}^{n} a_i$ 的最大值.

过程

这个求的值可能有些麻烦，利用常用的处理连续异或的方法，记 $s_x = \bigoplus_{i=1}^{x} a_i$，则原式等价于 $s_{p-1} \oplus s_n \oplus k$，观察到 $s_n \oplus k$ 在查询的过程中是固定的，题目的查询变化为查询在区间 $[l-1, r-1]$ 中异或定值（$s_n \oplus k$）的最大值.

继续按类似于可持久化线段树的思路，考虑每次的查询都查询整个区间. 我们只需把这个区间建一棵 Trie 树，将这个区间中的每个树都加入这棵 Trie 中，查询的时候，尽量往与当前位不相同的地方跳.

查询区间，只需要利用前缀和和差分的思想，用两棵前缀 Trie 树（也就是按顺序添加数的两个历史版本）相减即得到该区间的 Trie 树. 再利用动态开点的思想，不添加没有计算过的点，以减少空间占用.

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
constexpr int MAXN = 600010;
int n, q, a[MAXN], s[MAXN], l, r, x;
char op;

struct Trie {
  int cnt, rt[MAXN], ch[MAXN * 33][2], val[MAXN * 33];

  void insert(int o, int lst, int v) {
    for (int i = 28; i >= 0; i--) {
      val[o] = val[lst] + 1;  // 在原版本的基础上更新
      if ((v & (1 << i)) == 0) {
        if (!ch[o][0]) ch[o][0] = ++cnt;
        ch[o][1] = ch[lst][1];
        o = ch[o][0];
        lst = ch[lst][0];
      } else {
```

```
21          if (!ch[o][1]) ch[o][1] = ++cnt;
22          ch[o][0] = ch[lst][0];
23          o = ch[o][1];
24          lst = ch[lst][1];
25        }
26      }
27      val[o] = val[lst] + 1;
28    }
29
30    int query(int o1, int o2, int v) {
31      int ret = 0;
32      for (int i = 28; i >= 0; i--) {
33        int t = ((v & (1 << i)) ? 1 : 0);
34        if (val[ch[o1][!t]] - val[ch[o2][!t]])
35          ret += (1 << i), o1 = ch[o1][!t],
36                           o2 = ch[o2][!t];  // 尽量向不同的地方跳
37        else
38          o1 = ch[o1][t], o2 = ch[o2][t];
39      }
40      return ret;
41    }
42  } st;
43
44  int main() {
45    cin.tie(nullptr)->sync_with_stdio(false);
46    cin >> n >> q;
47    for (int i = 1; i <= n; i++) cin >> a[i], s[i] = s[i - 1] ^ a[i];
48    for (int i = 1; i <= n; i++)
49      st.rt[i] = ++st.cnt, st.insert(st.rt[i], st.rt[i - 1], s[i]);
50    while (q--) {
51      cin >> op;
52      if (op == 'A') {
53        n++;
54        cin >> a[n];
55        s[n] = s[n - 1] ^ a[n];
56        st.rt[n] = ++st.cnt;
57        st.insert(st.rt[n], st.rt[n - 1], s[n]);
58      }
59      if (op == 'Q') {
60        cin >> l >> r >> x;
61        l--;
62        r--;
63        if (l == 0)
64          cout << max(s[n] ^ x, st.query(st.rt[r], st.rt[0], s[n] ^ x)) << '\n';
65        else
66          cout << st.query(st.rt[r], st.rt[l - 1], s[n] ^ x) << '\n';
67      }
68    }
69    return 0;
70  }
71
```

外部参考资料

[1] 最大异或和 

10.17.6 可持久化可并堆

可持久化可并堆一般用于求解 k 短路问题.

如果一种可并堆的时间复杂度不是均摊的，那么它在可持久化后单次操作的时间复杂度就保证是 $O(\log n)$ 的，即不会因为特殊数据而使复杂度退化.

可持久化左偏树

在学习本内容前，请先了解 左偏树→10.7.4@p2035 的相关内容.

过程

回顾左偏树的合并过程，假设我们要合并分别以 x, y 为根节点的两棵左偏树，且维护的左偏树满足小根堆的性质：

1. 如果 x, y 中有结点为空，返回 $x + y$.
2. 选择 x, y 两结点中权值更小的结点，作为合并后左偏树的根.
3. 递归合并 x 的右子树与 y，将合并后的根节点作为 x 的右儿子.
4. 维护当前合并后左偏树的左偏性质，维护 `dist` 值，返回选择的根节点.

由于每次递归都会使 `dist[x]+dist[y]` 减少一，而 `dist[x]` 是 $O(\log n)$ 的，一次最多只会修改 $O(\log n)$ 个结点，所以这样做的时间复杂度是 $O(\log n)$ 的.

可持久化要求保留历史信息，使得之后能够访问之前的版本. 要将左偏树可持久化，就要将其沿途修改的路径复制一遍.

所以可持久化左偏树的合并过程是这样的：

1. 如果 x, y 中有结点为空，返回 $x + y$.
2. 选择 x, y 两结点中权值更小的结点，新建该结点的一个复制 p，作为合并后左偏树的根.
3. 递归合并 p 的右子树与 y，将合并后的根节点作为 p 的右儿子.
4. 维护以 p 为根的左偏树的左偏性质，维护其 `dist` 值，返回 p.

由于左偏树一次最多只会修改并新建 $O(\log n)$ 个结点，设操作次数为 m，则可持久化左偏树的时间复杂度和空间复杂度均为 $O(m \log n)$.

参考实现

```
int merge(int x, int y) {
  if (!x || !y) return x + y;
  if (v[x] > v[y]) swap(x, y);
  int p = ++cnt;
  lc[p] = lc[x];
  v[p] = v[x];
  rc[p] = merge(rc[x], y);
  if (dist[lc[p]] < dist[rc[p]]) swap(lc[p], rc[p]);
  dist[p] = dist[rc[p]] + 1;
  return p;
}
```

10.18 树套树

10.18.1 线段树套线段树

Authors: Chrogeek, HeRaNO, Dev-XYS, Dev-jqe

常见用途

在算法竞赛中，我们有时需要维护多维度信息．在这种时候，我们经常需要树套树来记录信息．

实现原理

我们考虑用树套树如何实现在二维平面上进行单点修改，区域查询．我们考虑外层的线段树，最底层的 1 到 n 个节点的子树，分别代表第 1 到第 n 行的线段树．那么这些底层的节点对应的父节点，就代表其两个子节点的子树所在的一片区域．

性质

空间复杂度

通常情况下，我们不可能对于外层线段树的每一个结点都建立一颗子线段树，空间需求过大．树套树一般采取动态开点的策略．单次修改，我们会涉及到外层线段树的 $\log n$ 个节点，且对于每个节点的子树涉及 $\log n$ 个节点，所以单次修改产生的空间最多为 $\log^2 n$．

时间复杂度

对于询问操作，我们考虑我们在外层线段树上进行 $\log n$ 次操作，每次操作会在一个内层线段树上进行 $\log n$ 次操作，所以时间复杂度为 $\log^2 n$． 修改操作，与询问操作复杂度相同，也为 $\log^2 n$．

经典例题

陌上花开[1] 将第一维排序处理，然后用树套树维护第二维和第三维．

示例代码

第二维查询

```
int tree_query(int k, int l, int r, int x) {
  if (k == 0) return 0;
  if (1 <= l && r <= sec[x].y) return vec_query(ou_root[k], 1, p, 1, sec[x].z);
  int mid = l + r >> 1, res = 0;
  if (1 <= mid) res += tree_query(ou_ch[k][0], l, mid, x);
  if (sec[x].y > mid) res += tree_query(ou_ch[k][1], mid + 1, r, x);
  return res;
}
```

第二维修改

```
void tree_insert(int &k, int l, int r, int x) {
  if (k == 0) k = ++ou_tot;
  vec_insert(ou_root[k], 1, p, sec[x].z);
  if (l == r) return;
  int mid = l + r >> 1;
  if (sec[x].y <= mid)
    tree_insert(ou_ch[k][0], l, mid, x);
  else
    tree_insert(ou_ch[k][1], mid + 1, r, x);
}
```

第三维查询

```
int vec_query(int k, int l, int r, int x, int y) {
  if (k == 0) return 0;
  if (x <= l && r <= y) return data[k];
  int mid = l + r >> 1, res = 0;
  if (x <= mid) res += vec_query(ch[k][0], l, mid, x, y);
  if (y > mid) res += vec_query(ch[k][1], mid + 1, r, x, y);
  return res;
}
```

第三维修改

```
void vec_insert(int &k, int l, int r, int loc) {
  if (k == 0) k = ++tot;
  data[k]++;
  if (l == r) return;
  int mid = l + r >> 1;
  if (loc <= mid) vec_insert(ch[k][0], l, mid, loc);
  if (loc > mid) vec_insert(ch[k][1], mid + 1, r, loc);
}
```

相关算法

面对多维度信息的题目时，如果题目没有要求强制在线，我们还可以考虑 CDQ **分治**，或者 **整体二分** 等分治算法，来避免使用高级数据结构，减少代码实现难度.

外部参考资料

[1] 陌上花开 

10.18.2 平衡树套线段树

10.18.3 线段树套平衡树

Authors: Dev-jqe, HeRaNO, huaruoji

常见用途

在算法竞赛中，我们有时需要维护多维度信息. 在这种时候，我们经常需要树套树来记录信息. 当需要维护前驱，后继，第 k 大，某个数的排名，或者插入删除的时候，我们通常需要使用平衡树来满足我们的需求，即线段树套平衡树.

过程

我们以 **二逼平衡树** 为例，来解释实现原理.

关于树套树的构建，我们对于外层线段树正常建树，对于线段树上的某一个节点，建立一棵平衡树，包含该节点所覆盖的序列. 具体操作时我们可以将序列元素一个个插入，每经过一个线段树节点，就将该元素加入到该节点的平衡树中.

操作一，求某区间中某值的排名：我们对于外层线段树正常操作，对于在某区间中的节点的平衡树，我们返回平衡树中比该值小的元素个数，合并区间时，我们将小的元素个数求和即可. 最后将返回值 $+1$，即为某值在某区间中的排名.

操作二，求某区间中排名为 k 的值：我们可以采用二分策略. 因为一个元素可能存在多个，其排名为一区间，且有些元素原序列不存在. 所以我们采取和操作一类似的思路，我们用小于该值的元素个数作为参考进行二分，即可得解.

操作三，将某个数替换为另外一个数：我们只要在所有包含某数的平衡树中删除某数，然后再插入另外一个数即可．外层依旧正常线段树操作．

操作四，求某区间中某值的前驱：我们对于外层线段树正常操作，对于在某区间中的节点的平衡树，我们返回某值在该平衡树中的前驱，线段树的区间结果合并时，我们取最大值即可．

性质

空间复杂度

我们每个元素加入 $O(\log n)$ 个平衡树，所以空间复杂度为 $O((n+q)\log n)$．

时间复杂度

- 对于 1，3，4 操作，我们考虑我们在外层线段树上进行 $O(\log n)$ 次操作，每次操作会在一个内层平衡树树上进行 $O(\log n)$ 次操作，所以时间复杂度为 $O(\log^2 n)$．
- 对于 2 操作，多一个二分过程，为 $O(\log^3 n)$．

经典例题

二逼平衡树[1] 外层线段树，内层平衡树．

实现

平衡树部分代码请参考 Splay→10.15.3@p2254 等其他条目．

操作一：

```
int vec_rank(int k, int l, int r, int x, int y, int t) {
  if (x <= l && r <= y) {
    return spy[k].chk_rank(t);
  }
  int mid = l + r >> 1;
  int res = 0;
  if (x <= mid) res += vec_rank(k << 1, l, mid, x, y, t);
  if (y > mid) res += vec_rank(k << 1 | 1, mid + 1, r, x, y, t);
  if (x <= mid && y > mid) res--;
  return res;
}
```

操作二：

```
int el = 0, er = 100000001, emid;
while (el != er) {
  emid = el + er >> 1;
  if (vec_rank(1, 1, n, tl, tr, emid) - 1 < tk)
    el = emid + 1;
  else
    er = emid;
}
printf("%d\n", el - 1);
```

操作三：

```
void vec_chg(int k, int l, int r, int loc, int x) {
  int t = spy[k].find(dat[loc]);
  spy[k].dele(t);
  spy[k].insert(x);
  if (l == r) return;
```

```
  int mid = l + r >> 1;
  if (loc <= mid) vec_chg(k << 1, l, mid, loc, x);
  if (loc > mid) vec_chg(k << 1 | 1, mid + 1, r, loc, x);
}
```

操作四：

```
int vec_front(int k, int l, int r, int x, int y, int t) {
  if (x <= l && r <= y) return spy[k].chk_front(t);
  int mid = l + r >> 1;
  int res = 0;
  if (x <= mid) res = max(res, vec_front(k << 1, l, mid, x, y, t));
  if (y > mid) res = max(res, vec_front(k << 1 | 1, mid + 1, r, x, y, t));
  return res;
}
```

相关算法

面对多维度信息的题目时，如果题目没有要求强制在线，我们还可以考虑 CDQ 分治→13.4.2@p3176，或者 整体二分→13.4.3@p3199 等分治算法，来避免使用高级数据结构，减少代码实现难度.

外部参考资料

[1] 二逼平衡树

10.18.4 树状数组套权值线段树

Authors: Ir1d, sshwy, Enter-tainer, H-J-Granger, ouuan, GavinZhengOI, hsfzLZH1, xyf007

静态区间 k 小值（POJ 2104 K-th Number）[1] 的问题可以用 权值线段树→10.17.2@p2424 在 $O(n \log n)$ 的时间复杂度内解决.

如果区间变成动态的呢？即，如果还要求支持一种操作：单点修改某一位上的值，又该怎么办呢？

例题 二逼平衡树（树套树）[2]

维护一个有序数列，其中需要提供以下操作：

- 查询 x 在区间内的排名；
- 查询区间内排名为 k 的值；
- 修改某一位置上的数值；
- 查询 x 在区间内的前驱（前驱定义为小于 x，且最大的数）；
- 查询 x 在区间内的后继（后继定义为大于 x，且最小的数）.

例题 洛谷 P2617 Dynamic Rankings[3]

给定一个含有 n 个数的序列 $a_1, a_2 \ldots a_n$，需要支持两种操作：

- `Q l r k` 表示查询下标在区间 $[l, r]$ 中的第 k 小的数
- `C x y` 表示将 a_x 改为 y

如果用 线段树套平衡树→10.18.3@p2441 中所论述的，用线段树套平衡树，即对于线段树的每一个节点，对于其所表示的区间维护一个平衡树，然后用二分来查找 k 小值. 由于每次查询操作都要覆盖多个区间，即有多个节点，但是平衡树并不能多个值一起查找，所以时间复杂度是 $O(n \log^3 n)$，并不是最优的.

优化的思路是把二分答案的操作和查询小于一个值的数的数量两种操作结合起来，使用 **线段树套动态开点权值线段树**，由于所有线段树的结构是相同的，可以在多棵树上同时进行线段树上二分.

在修改操作进行时，先在线段树上从上往下跳到被修改的点，删除所经过的点所指向的动态开点权值线段树上的原来的值，然后插入新的值，要经过 $O(\log n)$ 个线段树上的节点，在动态开点权值线段树上一次修改操作是 $O(\log n)$ 的，所以修改操作的时间复杂度为 $O(\log^2 n)$.

在查询答案时，先取出该区间覆盖在线段树上的所有点，然后用类似于静态区间 k 小值的方法，将这些点一起向左儿子或向右儿子跳. 如果所有这些点左儿子存储的值大于等于 k，则往左跳，否则往右跳. 由于最多只能覆盖 $O(\log n)$ 个节点，所以最多一次只有这么多个节点向下跳，时间复杂度为 $O(\log^2 n)$.

由于线段树的常数较大，在实现中往往使用常数更小且更方便处理前缀和的 **树状数组** 实现. 另外空间复杂度是 $O(n \log^2 n)$ 的，使用时 **注意空间限制**.

给出一种代码实现：

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
#include <set>
#define LC o << 1
#define RC o << 1 | 1
using namespace std;
constexpr int MAXN = 1000010;
int n, m, a[MAXN], u[MAXN], x[MAXN], l[MAXN], r[MAXN], k[MAXN], cur, cur1, cur2,
    q1[MAXN], q2[MAXN], v[MAXN];
char op[MAXN];
set<int> ST;
map<int, int> mp;

struct segment_tree  // 封装的动态开点权值线段树
{
  int cur, rt[MAXN * 4], sum[MAXN * 60], lc[MAXN * 60], rc[MAXN * 60];

  void build(int& o) { o = ++cur; }

  void print(int o, int l, int r) {
    if (!o) return;
    if (l == r && sum[o]) printf("%d ", l);
    int mid = (l + r) >> 1;
    print(lc[o], l, mid);
    print(rc[o], mid + 1, r);
  }

  void update(int& o, int l, int r, int x, int v) {
    if (!o) o = ++cur;
    sum[o] += v;
    if (l == r) return;
    int mid = (l + r) >> 1;
    if (x <= mid)
```

```
36        update(lc[o], l, mid, x, v);
37      else
38        update(rc[o], mid + 1, r, x, v);
39    }
40  } st;
41
42  // 树状数组实现
43  namepace fenwick_impl {
44    int lowbit(int o) { return (o & (-o)); }
45
46    void upd(int o, int x, int v) {
47      for (; o <= n; o += lowbit(o)) st.update(st.rt[o], 1, n, x, v);
48    }
49
50    void gtv(int o, int* A, int& p) {
51      p = 0;
52      for (; o; o -= lowbit(o)) A[++p] = st.rt[o];
53    }
54
55    int qry(int l, int r, int k) {
56      if (l == r) return l;
57      int mid = (l + r) >> 1, siz = 0;
58      for (int i = 1; i <= cur1; i++) siz += st.sum[st.lc[q1[i]]];
59      for (int i = 1; i <= cur2; i++) siz -= st.sum[st.lc[q2[i]]];
60      // printf("j %d %d %d %d\n",cur1,cur2,siz,k);
61      if (siz >= k) {
62        for (int i = 1; i <= cur1; i++) q1[i] = st.lc[q1[i]];
63        for (int i = 1; i <= cur2; i++) q2[i] = st.lc[q2[i]];
64        return qry(l, mid, k);
65      } else {
66        for (int i = 1; i <= cur1; i++) q1[i] = st.rc[q1[i]];
67        for (int i = 1; i <= cur2; i++) q2[i] = st.rc[q2[i]];
68        return qry(mid + 1, r, k - siz);
69      }
70    }
71  }
72  using namespace fenwick_impl;
73
74  // 线段树实现
75  namespace segtree_impl {
76  void build(int o, int l, int r) {
77    st.build(st.rt[o]);
78    if (l == r) return;
79    int mid = (l + r) >> 1;
80    build(LC, l, mid);
81    build(RC, mid + 1, r);
82  }
83
84  void print(int o, int l, int r) {
85    printf("%d %d:", l, r);
86    st.print(st.rt[o], 1, n);
87    printf("\n");
```

```
  if (l == r) return;
  int mid = (l + r) >> 1;
  print(LC, l, mid);
  print(RC, mid + 1, r);
}

void update(int o, int l, int r, int q, int x, int v) {
  st.update(st.rt[o], 1, n, x, v);
  if (l == r) return;
  int mid = (l + r) >> 1;
  if (q <= mid)
    update(LC, l, mid, q, x, v);
  else
    update(RC, mid + 1, r, q, x, v);
}

void getval(int o, int l, int r, int ql, int qr) {
  if (l > qr || r < ql) return;
  if (ql <= l && r <= qr) {
    q[++cur] = st.rt[o];
    return;
  }
  int mid = (l + r) >> 1;
  getval(LC, l, mid, ql, qr);
  getval(RC, mid + 1, r, ql, qr);
}

int query(int l, int r, int k) {
  if (l == r) return l;
  int mid = (l + r) >> 1, siz = 0;
  for (int i = 1; i <= cur; i++) siz += st.sum[st.lc[q[i]]];
  if (siz >= k) {
    for (int i = 1; i <= cur; i++) q[i] = st.lc[q[i]];
    return query(l, mid, k);
  } else {
    for (int i = 1; i <= cur; i++) q[i] = st.rc[q[i]];
    return query(mid + 1, r, k - siz);
  }
}
}  // namespace segtree_impl

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) scanf("%d", a + i), ST.insert(a[i]);
  for (int i = 1; i <= m; i++) {
    scanf(" %c", op + i);
    if (op[i] == 'C')
      scanf("%d%d", u + i, x + i), ST.insert(x[i]);
    else
      scanf("%d%d%d", l + i, r + i, k + i);
  }
  for (set<int>::iterator it = ST.begin(); it != ST.end(); it++)
```

```
140     mp[*it] = ++cur, v[cur] = *it;
141   for (int i = 1; i <= n; i++) a[i] = mp[a[i]];
142   for (int i = 1; i <= m; i++)
143     if (op[i] == 'C') x[i] = mp[x[i]];
144   n += m;
145   // build(1,1,n);
146   for (int i = 1; i <= n; i++) upd(i, a[i], 1);
147   // print(1,1,n);
148   for (int i = 1; i <= m; i++) {
149     if (op[i] == 'C') {
150       upd(u[i], a[u[i]], -1);
151       upd(u[i], x[i], 1);
152       a[u[i]] = x[i];
153     } else {
154       gtv(r[i], q1, cur1);
155       gtv(l[i] - 1, q2, cur2);
156       printf("%d\n", v[qry(1, n, k[i])]);
157     }
158   }
159   return 0;
160 }
```

外部参考资料

[1] 静态区间 k 小值（POJ 2104 K-th Number）

[2] 二逼平衡树（树套树）

[3] 洛谷 P2617 Dynamic Rankings

10.18.5 分块套树状数组

Authors: Backl1ght, Tiphereth-A, Enter-tainer, Ir1d, ksyx, leoleoasd, Xeonacid, aaron20100919

简介

分块套树状数组在特定条件下可以用来做一些树套树可以做的事情，但是相比起树套树，分块套树状数组代码编写更加简短，更加容易实现.

简单的例子

一个简单的例子就是二维平面中矩阵区域内点数的查询.

矩形区域查询

给出 n 个二维平面中的点 (x_i, y_i)，其中 $1 \le i \le n, 1 \le x_i, y_i \le n, 1 \le n \le 10^5$，要求实现以下中操作：

1. 给出 a, b, c, d，询问以 (a, b) 为左上角，c, d 为右下角的矩形区域内点的个数.
2. 给出 x, y，将横坐标为 x 的点的纵坐标改为 y.

题目 **强制在线**，保证 $x_i \neq x_j (1 \le i, j \le n, i \neq j)$.

对于操作 1，可以通过矩形容斥将其转化为 4 个二维偏序的查询去解决，然后因为强制在线，CDQ 分治之类的离线算法就解决不了，于是想到了树套树，比如树状数组套 Treap. 这确实可以解决这个问题，但是代码太长了，也不是特别好实现.

注意到，题目还额外保证了 $x_i \neq x_j (1 \le i, j \le n, i \neq j)$，这个时候就可以用分块套树状数组解决.

初始化

首先，一个 x 只对应一个 y，所以可以用一个数组记录这个映射关系，比如令 Y_i 表示横坐标为 i 的点的纵坐标.

然后，以 $\sqrt{n}$ 为块大小对横坐标进行分块. 为每个块建一棵权值树状数组. 记 T_i 为第 i 个块对应的树状数组，$T_{i,j}$ 表示块 i 里纵坐标在 $(j-lowbit(j),j]$ 内的点的个数.

查询

对于操作 1，将其转化为 4 个二维偏序的查询. 现在只需要解决给出 a,b，询问有多少个点满足 $1\le x_i\le a,1\le y_i\le b$.

现在要查询横坐标的范围为 $[1,a]$. 因为查询范围最右边可能有一段不是完整的块, 所以暴力扫一遍这个段，看是否满足 $Y_i\le b$，统计出这个段满足要求的点的个数.

现在就只需要处理完整的块. 暴力扫一遍前面的块，查询每个块对应的树状数组中值小于 b 的个数，累加到答案上.

这就完事了? 不，注意到处理完整的块的时候，其实相当于查询 T 的前缀和，如果修改时也使用树状数组的技巧处理 T，那么查询时复杂度会更低.

修改

普通的做法就先找到点 x 所在的块，然后一减一加两个权值树状数组单点修改，再将 Y_x 置为 y.

如果用了上面说的优化, 那就是对 T 也走一个树状数组修改的流程, 每次修改也是一减一加两个权值树状数组单点修改.

对上述步骤进行一定的改变，比如将一减一加改成只减，就是删点；改成只加，就是加点. 但是必须要注意一个 x 只能对应一个 y.

空间复杂度

分块分了 $\sqrt{n}$ 个块，每个块一个树状数组 $O(n)$ 的空间，所以空间复杂度为 $O(n\sqrt{n})$.

时间复杂度

查询的话，遍历非完整块的段 $O(\sqrt{n})$. 然后，对 T 走树状数组查询，每个经历到的 T_i 也走树状数组查询，这一步是 $O(\log(\sqrt{n})\log n)$ 的复杂度. 所以查询的时间复杂度为 $O(\sqrt{n}+\log(\sqrt{n})\log n)$.

修改和查询一样，复杂度为 $O(\sqrt{n}+\log(\sqrt{n})\log n)$.

例题 1

Intersection of Permutations[1]

给出两个排列 a 和 b，要求实现以下两种操作：

1. 给出 l_a,r_a,l_b,r_b，要求查询既出现在 $a[l_a...r_a]$ 又出现在 $b[l_b...r_b]$ 中的元素的个数.
2. 给出 x,y，$swap(b_x,b_y)$.

序列长度 n 满足 $2\le n\le 2\cdot 10^5$，操作个数 q 满足 $1\le q\le 2\cdot 10^5$.

对于每个值 i，记 x_i 是它在排列 b 中的下标，y_i 是它在排列 a 中的下标. 这样，操作一就变成了一个矩形区域内点的个数的询问，操作 2 可以看成两个修改操作. 而且因为是排列，所以满足一个 x 对应一个 y，所以这题可以用分块套树状数组来写.

参考代码（分块套树状数组 - 1s）

```
1 #include <cmath>
```

```cpp
#include <cstdio>
using namespace std;
constexpr int N = 2e5 + 5;
constexpr int M = 447 + 5;  // sqrt(N) + 5

int n, m, pa[N], pb[N];

int nn, block_size, block_cnt, block_id[N], L[N], R[N], T[M][N];

void build(int n) {
  nn = n;
  block_size = sqrt(nn);
  block_cnt = nn / block_size;
  for (int i = 1; i <= block_cnt; ++i) {
    L[i] = R[i - 1] + 1;
    R[i] = i * block_size;
  }
  if (R[block_cnt] < nn) {
    ++block_cnt;
    L[block_cnt] = R[block_cnt - 1] + 1;
    R[block_cnt] = nn;
  }
  for (int j = 1; j <= block_cnt; ++j)
    for (int i = L[j]; i <= R[j]; ++i) block_id[i] = j;
}

int lb(int x) { return x & -x; }

void add(int p, int v, int d) {
  for (int i = block_id[p]; i <= block_cnt; i += lb(i))
    for (int j = v; j <= nn; j += lb(j)) T[i][j] += d;
}

int getsum(int p, int v) {
  if (!p) return 0;
  int res = 0;
  int id = block_id[p];
  for (int i = L[id]; i <= p; ++i)
    if (pb[i] <= v) ++res;
  for (int i = id - 1; i; i -= lb(i))
    for (int j = v; j; j -= lb(j)) res += T[i][j];
  return res;
}

void update(int x, int y) {
  add(x, pb[x], -1);
  add(y, pb[y], -1);
  swap(pb[x], pb[y]);
  add(x, pb[x], 1);
  add(y, pb[y], 1);
}

```

```
int query(int la, int ra, int lb, int rb) {
  int res = getsum(rb, ra) - getsum(rb, la - 1) - getsum(lb - 1, ra) +
            getsum(lb - 1, la - 1);
  return res;
}

int main() {
  scanf("%d %d", &n, &m);
  int v;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pa[v] = i;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pb[i] = pa[v];

  build(n);
  for (int i = 1; i <= n; ++i) add(i, pb[i], 1);

  int op, la, lb, ra, rb, x, y;
  for (int i = 1; i <= m; ++i) {
    scanf("%d", &op);
    if (op == 1) {
      scanf("%d %d %d %d", &la, &ra, &lb, &rb);
      printf("%d\n", query(la, ra, lb, rb));
    } else if (op == 2) {
      scanf("%d %d", &x, &y);
      update(x, y);
    }
  }
  return 0;
}
```

参考代码（树状数组套 Treap—TLE）

```
#include <cstdio>
#include <random>
using namespace std;
constexpr int N = 2e5 + 5;
mt19937 rng(random_device{}());

int n, m, pa[N], pb[N];

// Treap
struct Treap {
  struct node {
    node *l, *r;
    int sz, rnd, v;

    node(int _v) : l(NULL), r(NULL), sz(1), rnd(rng()), v(_v) {}
  };

  int get_size(node*& p) { return p ? p->sz : 0; }

  void push_up(node*& p) {
```

```
21    if (!p) return;
22    p->sz = get_size(p->l) + get_size(p->r) + 1;
23  }
24
25  node* root;
26
27  node* merge(node* a, node* b) {
28    if (!a) return b;
29    if (!b) return a;
30    if (a->rnd < b->rnd) {
31      a->r = merge(a->r, b);
32      push_up(a);
33      return a;
34    } else {
35      b->l = merge(a, b->l);
36      push_up(b);
37      return b;
38    }
39  }
40
41  void split_val(node* p, const int& k, node*& a, node*& b) {
42    if (!p)
43      a = b = NULL;
44    else {
45      if (p->v <= k) {
46        a = p;
47        split_val(p->r, k, a->r, b);
48        push_up(a);
49      } else {
50        b = p;
51        split_val(p->l, k, a, b->l);
52        push_up(b);
53      }
54    }
55  }
56
57  void split_size(node* p, int k, node*& a, node*& b) {
58    if (!p)
59      a = b = NULL;
60    else {
61      if (get_size(p->l) <= k) {
62        a = p;
63        split_size(p->r, k - get_size(p->l), a->r, b);
64        push_up(a);
65      } else {
66        b = p;
67        split_size(p->l, k, a, b->l);
68        push_up(b);
69      }
70    }
71  }
72
```

```
  void ins(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    a = merge(a, new node(val));
    root = merge(a, b);
  }

  void del(int val) {
    node *a, *b, *c, *d;
    split_val(root, val, a, b);
    split_val(a, val - 1, c, d);
    delete d;
    root = merge(c, b);
  }

  int qry(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    int res = get_size(a);
    root = merge(a, b);
    return res;
  }

  int qry(int l, int r) { return qry(r) - qry(l - 1); }
};

// Fenwick Tree
Treap T[N];

int lb(int x) { return x & -x; }

void ins(int x, int v) {
  for (; x <= n; x += lb(x)) T[x].ins(v);
}

void del(int x, int v) {
  for (; x <= n; x += lb(x)) T[x].del(v);
}

int qry(int x, int mi, int ma) {
  int res = 0;
  for (; x; x -= lb(x)) res += T[x].qry(mi, ma);
  return res;
}

int main() {
  scanf("%d %d", &n, &m);
  int v;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pa[v] = i;
  for (int i = 1; i <= n; ++i) scanf("%d", &v), pb[i] = pa[v];
  for (int i = 1; i <= n; ++i) ins(i, pb[i]);

```

```
  int op, la, lb, ra, rb, x, y;
  for (int i = 1; i <= m; ++i) {
    scanf("%d", &op);
    if (op == 1) {
      scanf("%d %d %d %d", &la, &ra, &lb, &rb);
      printf("%d\n", qry(rb, la, ra) - qry(lb - 1, la, ra));
    } else if (op == 2) {
      scanf("%d %d", &x, &y);
      del(x, pb[x]);
      del(y, pb[y]);
      swap(pb[x], pb[y]);
      ins(x, pb[x]);
      ins(y, pb[y]);
    }
  }
  return 0;
}
```

例题 2

Complicated Computations[2]

给出一个序列 a，将 a 所有连续子序列的 MEX 构成的数组作为 b，问 b 的 MEX. 一个序列的 MEX 是序列中最小的没出现过的 **正整数**.

序列的长度 n 满足 $1 \le n \le 10^5$.

观察：一个序列的 MEX 为 mex，当且仅当这个序列包含 1 至 $mex-1$，但不包含 mex.

依次判断是否存在 MEX 为 1 至 $n+1$ 的连续子序列. 如果没有 MEX 为 i 的连续子序列，那么答案即为 i. 如果都存在，那么答案为 $n+2$.

在判断 i 时，将序列视为由零或多个 i 分隔的多个段. 如果存在一个段，这个段中包含 1 至 $i-1$，但不包含 i，那么就说明存在值为 i 的连续子序列.

用一个数组 Y_j 记录上一个值为 a_j 的元素的位置，以 j 作为 x，Y_j 作为 y，a_j 作为 z. 这样，计算段内是否包含 1 至 $i-1$ 就是一个三维偏序的问题. 形式化的说，判断段 $[l,r]$ 的 MEX 值是否为 i，就是看满足 $l \le j \le r, Y_j \le l-1, a_j \le i-1$ 的点的个数是否为 $i-1$.

如果在判断完值为 i 的元素之后再将对应的点插入，这时因为 $[l,r]$ 内只存在 $a_j \le i-1$ 的元素，所以上述三维偏序问题就可以转换为二维偏序的问题.

参考代码（分块套树状数组 - 78ms）

```
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;
constexpr int N = 1e5 + 5;
constexpr int M = 316 + 5;  // sqrt(N) + 5

// 分块
int nn, b[N], block_size, block_cnt, block_id[N], L[N], R[N], T[M][N];
```

```
10
11 void build(int n) {
12   nn = n;
13   block_size = sqrt(nn);
14   block_cnt = nn / block_size;
15   for (int i = 1; i <= block_cnt; ++i) {
16     L[i] = R[i - 1] + 1;
17     R[i] = i * block_size;
18   }
19   if (R[block_cnt] < nn) {
20     ++block_cnt;
21     L[block_cnt] = R[block_cnt - 1] + 1;
22     R[block_cnt] = nn;
23   }
24   for (int j = 1; j <= block_cnt; ++j)
25     for (int i = L[j]; i <= R[j]; ++i) block_id[i] = j;
26 }
27
28 int lb(int x) { return x & -x; }
29
30 // d = 1: 加点(p, v)
31 // d = -1: 删点(p, v)
32 void add(int p, int v, int d) {
33   for (int i = block_id[p]; i <= block_cnt; i += lb(i))
34     for (int j = v; j <= nn; j += lb(j)) T[i][j] += d;
35 }
36
37 // 询问[1, r]内，纵坐标小于等于val的点有多少个
38 int getsum(int p, int v) {
39   if (!p) return 0;
40   int res = 0;
41   int id = block_id[p];
42   for (int i = L[id]; i <= p; ++i)
43     if (b[i] && b[i] <= v) ++res;
44   for (int i = id - 1; i; i -= lb(i))
45     for (int j = v; j; j -= lb(j)) res += T[i][j];
46   return res;
47 }
48
49 // 询问[l, r]内，纵坐标小于等于val的点有多少个
50 int query(int l, int r, int val) {
51   if (l > r) return -1;
52   int res = getsum(r, val) - getsum(l - 1, val);
53   return res;
54 }
55
56 // 加点(p, v)
57 void update(int p, int v) {
58   b[p] = v;
59   add(p, v, 1);
60 }
61
```

```cpp
int n, a[N];
vector<int> g[N];

int main() {
  scanf("%d", &n);

  // 为了减少讨论，加了哨兵节点
  // 因为树状数组添加的时候，为0可能会死循环，所以整体往右偏移一位
  // a_1和a_{n+2}为哨兵节点
  for (int i = 2; i <= n + 1; ++i) scanf("%d", &a[i]);
  for (int i = 2; i <= n + 1; ++i) g[a[i]].push_back(i);

  // 分块
  build(n + 2);

  int ans = n + 2, lst, ok;
  for (int i = 1; i <= n + 1; ++i) {
    g[i].push_back(n + 2);

    lst = 1;
    ok = 0;
    for (int pos : g[i]) {
      if (query(lst + 1, pos - 1, lst) == i - 1) {
        ok = 1;
        break;
      }
      lst = pos;
    }

    if (!ok) {
      ans = i;
      break;
    }

    lst = 1;
    g[i].pop_back();
    for (int pos : g[i]) {
      update(pos, lst);
      lst = pos;
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

参考代码（线段树套 Treap-468ms）

```cpp
#include <cstdio>
#include <random>
#include <vector>
using namespace std;
```

```
constexpr int N = 1e5 + 5;

vector<int> g[N];
int n, a[N];

mt19937 rng(random_device{}());

struct Treap {
  struct node {
    node *l, *r;
    unsigned rnd;
    int sz, v;

    node(int _v) : l(NULL), r(NULL), rnd(rng()), sz(1), v(_v) {}
  };

  int get_size(node*& p) { return p ? p->sz : 0; }

  void push_up(node*& p) {
    if (!p) return;
    p->sz = get_size(p->l) + get_size(p->r) + 1;
  }

  node* root;

  node* merge(node* a, node* b) {
    if (!a) return b;
    if (!b) return a;
    if (a->rnd < b->rnd) {
      a->r = merge(a->r, b);
      push_up(a);
      return a;
    } else {
      b->l = merge(a, b->l);
      push_up(b);
      return b;
    }
  }

  void split_val(node* p, const int& k, node*& a, node*& b) {
    if (!p)
      a = b = NULL;
    else {
      if (p->v <= k) {
        a = p;
        split_val(p->r, k, a->r, b);
        push_up(a);
      } else {
        b = p;
        split_val(p->l, k, a, b->l);
        push_up(b);
      }
```

```
    }
  }

  void split_size(node* p, int k, node*& a, node*& b) {
    if (!p)
      a = b = NULL;
    else {
      if (get_size(p->l) <= k) {
        a = p;
        split_size(p->r, k - get_size(p->l), a->r, b);
        push_up(a);
      } else {
        b = p;
        split_size(p->l, k, a, b->l);
        push_up(b);
      }
    }
  }

  void insert(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    a = merge(a, new node(val));
    root = merge(a, b);
  }

  int query(int val) {
    node *a, *b;
    split_val(root, val, a, b);
    int res = get_size(a);
    root = merge(a, b);
    return res;
  }

  int qry(int l, int r) { return query(r) - query(l - 1); }
};

// Segment Tree
Treap T[N << 2];

void insert(int x, int l, int r, int p, int val) {
  T[x].insert(val);
  if (l == r) return;
  int mid = (l + r) >> 1;
  if (p <= mid)
    insert(x << 1, l, mid, p, val);
  else
    insert(x << 1 | 1, mid + 1, r, p, val);
}

int query(int x, int l, int r, int L, int R, int val) {
  if (l == L && r == R) return T[x].query(val);
```

```
  int mid = (l + r) >> 1;
  if (R <= mid) return query(x << 1, l, mid, L, R, val);
  if (L > mid) return query(x << 1 | 1, mid + 1, r, L, R, val);
  return query(x << 1, l, mid, L, mid, val) +
         query(x << 1 | 1, mid + 1, r, mid + 1, R, val);
}

int query(int l, int r, int val) {
  if (l > r) return -1;
  return query(1, 1, n, l, r, val);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
  for (int i = 1; i <= n; ++i) g[a[i]].push_back(i);

  // a_0 和 a_{n+1}为哨兵节点
  int ans = n + 2, lst, ok;
  for (int i = 1; i <= n + 1; ++i) {
    g[i].push_back(n + 1);

    lst = 0;
    ok = 0;
    for (int pos : g[i]) {
      if (query(lst + 1, pos - 1, lst) == i - 1) {
        ok = 1;
        break;
      }
      lst = pos;
    }

    if (!ok) {
      ans = i;
      break;
    }

    lst = 0;
    g[i].pop_back();
    for (int pos : g[i]) {
      insert(1, 1, n, pos, lst);
      lst = pos;
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

外部参考资料

[1] Intersection of Permutations  [2] Complicated Computations 

10.19 K-D Tree

Authors: hsfzLZH1, Ir1d, JosephusW

k-D Tree(KDT , k-Dimension Tree) 是一种可以 **高效处理 k 维空间信息** 的数据结构.

在结点数 n 远大于 2^k 时，应用 k-D Tree 的时间效率很好.

在算法竞赛的题目中，一般有 $k = 2$. 在本页面分析时间复杂度时，将认为 k 是常数.

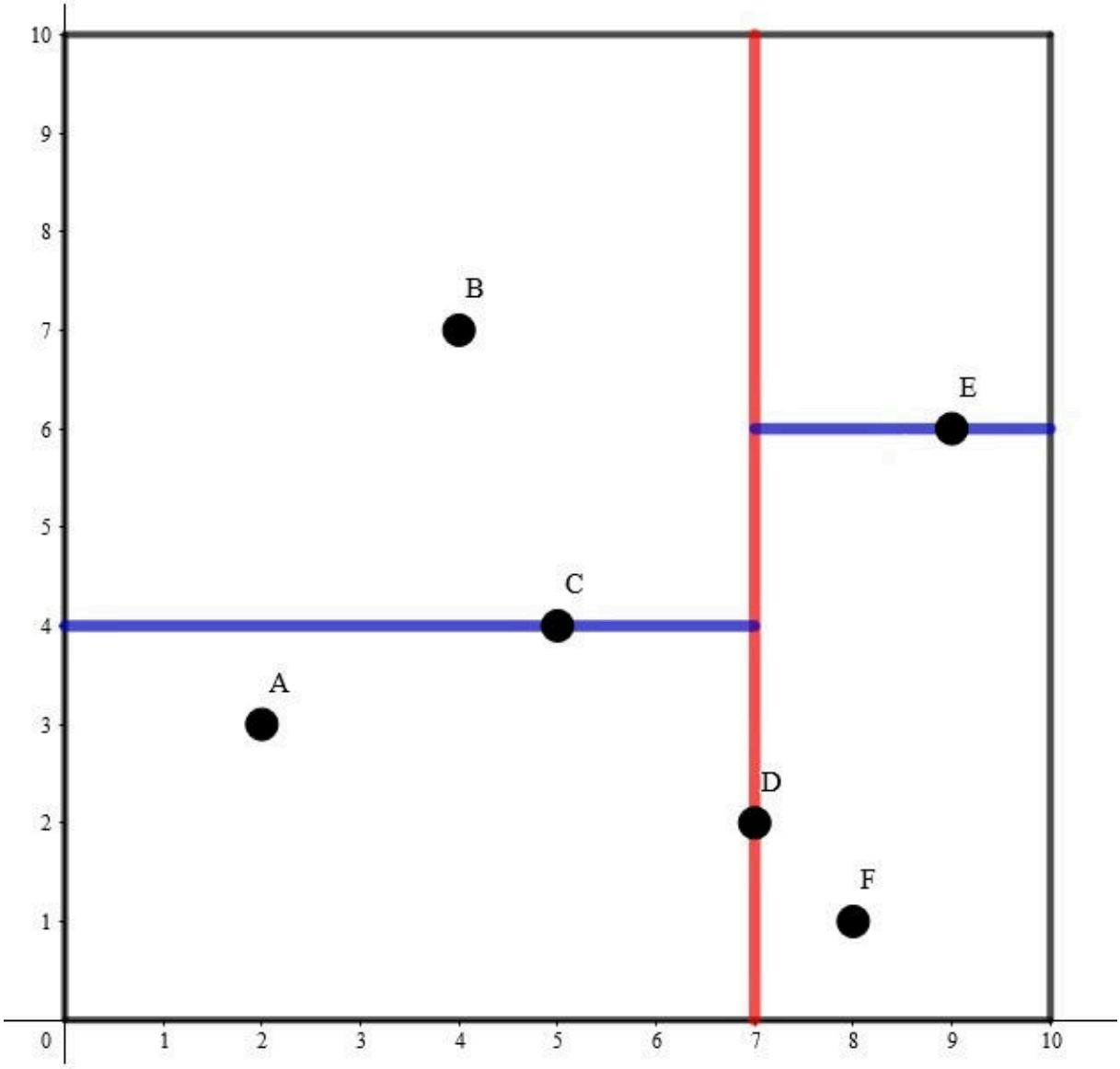
建树

k-D Tree 具有二叉搜索树的形态，二叉搜索树上的每个结点都对应 k 维空间内的一个点. 其每个子树中的点都在一个 k 维的超长方体内，这个超长方体内的所有点也都在这个子树中.

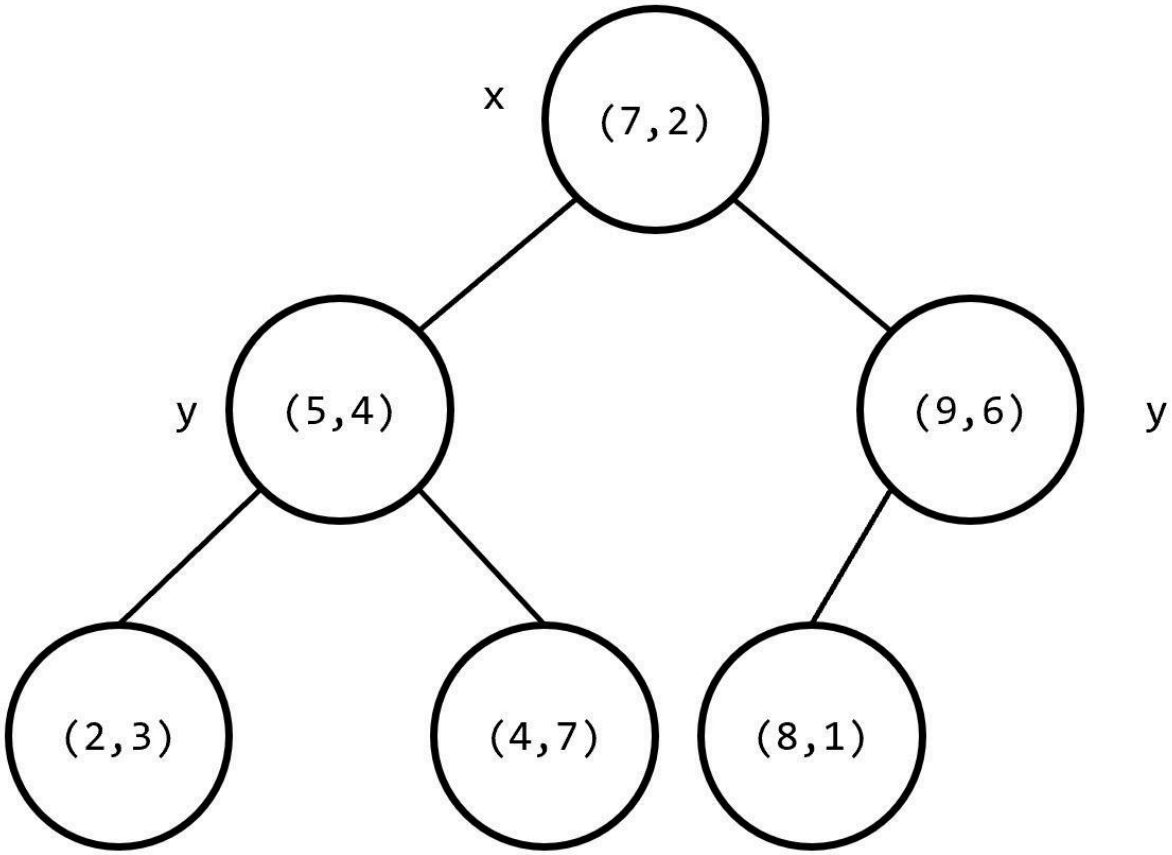
假设我们已经知道了 k 维空间内的 n 个不同的点的坐标，要将其构建成一棵 k-D Tree，步骤如下:

1. 若当前超长方体中只有一个点，返回这个点.
2. 选择一个维度，将当前超长方体按照这个维度分成两个超长方体.
3. 选择切割点: 在选择的维度上选择一个点, 这一维度上的值小于这个点的归入一个超长方体（左子树），其余的归入另一个超长方体（右子树）.
4. 将选择的点作为这棵子树的根节点，递归对分出的两个超长方体构建左右子树，维护子树的信息.

为了方便理解，我们举一个 $k = 2$ 时的例子.



其构建出 k-D Tree 的形态可能是这样的：



其中树上每个结点上的坐标是选择的分割点的坐标，非叶子结点旁的 x 或 y 是选择的切割维度.

这样的复杂度无法保证. 对于 2, 3 两步，我们提出两个优化：

1. 轮流选择 k 个维度，以保证在任意连续 k 层里每个维度都被切割到.
2. 每次在维度上选择切割点时选择该维度上的 **中位数**，这样可以保证每次分成的左右子树大小尽量相等.

可以发现，使用优化 2 后，构建出的 k-D Tree 的树高最多为 $\log n + O(1)$.

现在，构建 k-D Tree 时间复杂度的瓶颈在于快速选出一个维度上的中位数，并将在该维度上的值小于该中位数的置于中位数的左边，其余置于右边. 如果每次都使用 `sort` 函数对该维度进行排序，时间复杂度是 $O(n \log^2 n)$ 的. 事实上，单次找出 n 个元素中的中位数并将中位数置于排序后正确的位置的复杂度可以达到 $O(n)$.

我们来回顾一下快速排序的思想. 每次我们选出一个数，将小于该数的置于该数的左边，大于该数的置于该数的右边，保证该数在排好序后正确的位置上，然后递归排序左侧和右侧的值. 这样的期望复杂度是 $O(n \log n)$ 的. 但是由于 k-D Tree 只要求要中位数在排序后正确的位置上，所以我们只需要递归排序包含中位数的 **一侧**. 可以证明，这样的期望复杂度是 $O(n)$ 的. 在 `algorithm` 库中，有一个实现相同功能的函数 `nth_element()`，要找到 `s[l]` 和 `s[r]` 之间的值按照排序规则 `cmp` 排序后在 `s[mid]` 位置上的值，并保证 `s[mid]` 左边的值小于 `s[mid]`，右边的值大于 `s[mid]`，只需写 `nth_element(s+l,s+mid,s+r+1,cmp)`.

借助这种思想，构建 k-D Tree 时间复杂度是 $O(n \log n)$ 的.

高维空间上的操作

在查询高维矩形区域内的所有点的一些信息时，记录每个结点子树内每一维度上的坐标的最大值和最小值. 如果当前子树对应的矩形与所求矩形没有交点，则不继续搜索其子树；如果当前子树对应的矩形完全包含在所求矩形内，返回当前子树内所有点的权值和；否则，判断当前点是否在所求矩形内，更新答案并递归在左右子树中查找答案.

实现

```
int query(int p) {
  if (!p) return 0;
  bool flag{false};
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].L[k] && t[p].R[k] <=
h.x[k]));
  if (!flag) return t[p].sum;
  for (int k : {0, 1})
    if (t[p].R[k] < l.x[k] || h.x[k] < t[p].L[k]) return 0;
  int ans{0};
  flag = false;
  for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].x[k] && t[p].x[k] <=
h.x[k]));
  if (!flag) ans = t[p].v;
  return ans += query(t[p].l) + query(t[p].r);
}
```

复杂度分析

先考虑二维的，在查询矩形 R 时，我们将 k-D Tree 上的结点分为三类：

1. 与 R 无交.
2. 完全被 R 包含.
3. 部分被 R 包含.

显然单次查询的复杂度是第 3 类点的个数. 注意到第三类点的矩形要么完全包含 R，要么互不包含，而前者显然只有 $O(h) = O(\log n)$ 个，现在我们来分析后者的个数.

首先，我们不妨令矩形的所有边偏移 ϵ，使得查询矩形不穿过已经有的任何点. 这样显然是不影响矩形的查询所涵盖的点集的.

注意到互不包含的第 3 类点所对应的矩形，一定有 R 的一条边穿过之. 所以我们只需要计算 R 的每条边穿过的矩形个数，即任意一条线段最多经过多少个点对应的矩形.

考虑对于某一个结点 u, 它有四个孙子, 且它到每一个孙子都在两个维度上各进行了一次划分. 经过观察可以发现，按照这种方法将一个矩形划分成四个子矩形，一条与坐标轴平行的线段最多经过两个区域，即从 u 出发的查询，最多向下进入两个孙子仍有第 3 类点（如果线段刚好与分割边界重合则不一定，但是我们偏移查询矩形边界的操作使得这种情况不存在）.

而因为建树的时候, 每个点是其整个子树在当前划分维度上的中位数, 所以子树大小必定减半. 于是，设 u 的子树大小为 n，我们能写出如下递归式：

$$T(n) = 2T(n/4) + O(1)$$

由主定理得 $T(n) = O(\sqrt{n})$.

将递归式推广到 k 维，即 $T(n) = 2^{k-1}T(n/2^k) + O(1)$，于是 $T(n) = O(n^{1-\frac{1}{k}})$（将 k 视为常数）.

插入/删除

如果维护的这个 k 维点集是可变的，即可能会插入或删除一些点，此时 k-D Tree 的平衡性无法保证. 由于 k-D Tree 的构造，不能支持旋转，类似与 FHQ Treap 的随机优先级也不能保证其复杂度. 对此，有两种比较常见的维护方法.

Note

很多选手会使用替罪羊树结构来维护. 但是注意到在刚才的复杂度分析中, 要求儿子的子树大小严格减半，即树高必须为严格的 $\log n + O(1)$，而替罪羊树只满足树高 $O(\log n)$，故查询复杂度无法保证.

根号重构

插入的时候，先存下来要插入的点，每 B 次插入进行一次重构.

删除打个标记即可. 如果要求较为严格，可以维护树内有多少个被删除了，达到 B 则重构.

修改复杂度均摊 $O(n \log n/B)$，查询 $O(B + n^{1-\frac{1}{k}})$，若二者数量同阶则 $B = O(\sqrt{n \log n})$ 最优（修改 $O(\sqrt{n \log n})$，查询 $O(\sqrt{n \log n} + n^{1-\frac{1}{k}})$）.

二进制分组

考虑维护若干棵 2 的自然数次幂的 k-D Tree，满足这些树的大小之和为 n.

插入的时候，新增一棵大小为 1 的 k-D Tree，然后不断将相同大小的树合并（直接拍扁重构）. 实现的时候可以只重构一次.

容易发现需要合并的树的大小一定从 2^0 开始且指数连续. 复杂度类似二进制加法, 是均摊 $O(n \log^2 n)$ 的，因为重构本身带 log.

查询的时候，直接分别在每棵树上查询，复杂度为 $O\left(\sum_{i\geq 0}(\frac{n}{2^i})^{1-\frac{1}{k}}\right) = O(n^{1-\frac{1}{k}})$.

例题

洛谷 P4148 简单题[1]

在一个初始值全为 0 的 $n \times n$ 的二维矩阵上，进行 q 次操作，每次操作为以下两种之一：

1. `1 x y A`：将坐标 (x, y) 上的数加上 A.
2. `2 x1 y1 x2 y2`：输出以 $(x1, y1)$ 为左下角，$(x2, y2)$ 为右上角的矩形内（包括矩形边界）的数字和.

强制在线. 内存限制 `20M`. 保证答案及所有过程量在 `int` 范围内.

$1 \leq n \leq 500000, 1 \leq q \leq 200000$

20M 的空间卡掉了所有树套树，强制在线卡掉了 CDQ 分治，只能使用 k-D Tree.

以下是二进制分组的参考代码.

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N(2e5), LG{18};

struct pt {
  int x[2];
  int v, sum;
  int l, r;
  int L[2], R[2];
} t[N + 5], l, h;
```

```
12
13 int rt[LG];
14 int b[N + 5], cnt;
15
16 void upd(int p) {
17   t[p].sum = t[t[p].l].sum + t[t[p].r].sum + t[p].v;
18   for (int k : {0, 1}) {
19     t[p].L[k] = t[p].R[k] = t[p].x[k];
20     if (t[p].l) {
21       t[p].L[k] = min(t[p].L[k], t[t[p].l].L[k]);
22       t[p].R[k] = max(t[p].R[k], t[t[p].l].R[k]);
23     }
24     if (t[p].r) {
25       t[p].L[k] = min(t[p].L[k], t[t[p].r].L[k]);
26       t[p].R[k] = max(t[p].R[k], t[t[p].r].R[k]);
27     }
28   }
29 }
30
31 int build(int l, int r, int dep = 0) {
32   int p{(l + r) >> 1};
33   nth_element(b + l, b + p, b + r + 1,
34               [dep](int x, int y) { return t[x].x[dep] < t[y].x[dep]; });
35   int x{b[p]};
36   if (l < p) t[x].l = build(l, p - 1, dep ^ 1);
37   if (p < r) t[x].r = build(p + 1, r, dep ^ 1);
38   upd(x);
39   return x;
40 }
41
42 void append(int &p) {
43   if (!p) return;
44   b[++cnt] = p;
45   append(t[p].l);
46   append(t[p].r);
47   p = 0;
48 }
49
50 int query(int p) {
51   if (!p) return 0;
52   bool flag{false};
53   for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].L[k] && t[p].R[k] <=
h.x[k]));
54   if (!flag) return t[p].sum;
55   for (int k : {0, 1})
56     if (t[p].R[k] < l.x[k] || h.x[k] < t[p].L[k]) return 0;
57   int ans{0};
58   flag = false;
59   for (int k : {0, 1}) flag |= (!(l.x[k] <= t[p].x[k] && t[p].x[k] <=
h.x[k]));
60   if (!flag) ans = t[p].v;
61   return ans += query(t[p].l) + query(t[p].r);
```

```
}

int main() {
  int n;
  cin >> n;
  int lst{0};
  n = 0;
  while (true) {
    int op;
    cin >> op;
    if (op == 1) {
      int x, y, A;
      cin >> x >> y >> A;
      x ^= lst;
      y ^= lst;
      A ^= lst;
      t[++n] = {{x, y}, A};
      b[cnt = 1] = n;
      for (int sz{0};; ++sz)
        if (!rt[sz]) {
          rt[sz] = build(1, cnt);
          break;
        } else
          append(rt[sz]);
    } else if (op == 2) {
      cin >> l.x[0] >> l.x[1] >> h.x[0] >> h.x[1];
      l.x[0] ^= lst;
      l.x[1] ^= lst;
      h.x[0] ^= lst;
      h.x[1] ^= lst;
      lst = 0;
      for (int i{0}; i < LG; ++i) lst += query(rt[i]);
      cout << lst << "\n";
    } else
      break;
  }
  return 0;
}
```

邻域查询

Warning

使用 k-D Tree 单次查询最近点的时间复杂度最坏还是 $O(n)$ 的，但不失为一种优秀的骗分算法，使用时请注意．在这里对邻域查询的讲解仅限于加强对 k-D Tree 结构的认识．

例题 luogu P1429 平面最近点对（加强版） [2]

给定平面上的 n 个点 (x_i, y_i)，找出平面上最近两个点对之间的 欧几里得距离→12.4@p3089.

$2 \le n \le 200000, 0 \le x_i, y_i \le 10^9$

首先建出关于这 n 个点的 2-D Tree.

枚举每个结点，对于每个结点找到不等于该结点且距离最小的点，即可求出答案. 每次暴力遍历 2-D Tree 上的每个结点的时间复杂度是 $O(n)$ 的，需要剪枝. 我们可以维护一个子树中的所有结点在每一维上的坐标的最小值和最大值. 假设当前已经找到的最近点对的距离是 ans，如果查询点到子树内所有点都包含在内的长方形的 **最近** 距离大于等于 ans，则在这个子树内一定没有答案，搜索时不进入这个子树.

此外，还可以使用一种启发式搜索的方法，即若一个结点的两个子树都有可能包含答案，先在与查询点距离最近的一个子树中搜索答案. 可以认为，**查询点到子树对应的长方形的最近距离就是此题的估价函数**.

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <iomanip>
#include <iostream>
using namespace std;
constexpr int MAXN = 200010;
int n, d[MAXN], lc[MAXN], rc[MAXN];
double ans = 2e18;

struct node {
  double x, y;
} s[MAXN];

double L[MAXN], R[MAXN], D[MAXN], U[MAXN];

double dist(int a, int b) {
  return (s[a].x - s[b].x) * (s[a].x - s[b].x) +
         (s[a].y - s[b].y) * (s[a].y - s[b].y);
}

bool cmp1(node a, node b) { return a.x < b.x; }

bool cmp2(node a, node b) { return a.y < b.y; }

void maintain(int x) {
  L[x] = R[x] = s[x].x;
  D[x] = U[x] = s[x].y;
  if (lc[x])
    L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
    D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
  if (rc[x])
    L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
    D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
}

int build(int l, int r) {
  if (l > r) return 0;
  if (l == r) {
    maintain(l);
```

```
    return l;
  }
  int mid = (l + r) >> 1;
  double avx = 0, avy = 0, vax = 0, vay = 0;  // average variance
  for (int i = l; i <= r; i++) avx += s[i].x, avy += s[i].y;
  avx /= (double)(r - l + 1);
  avy /= (double)(r - l + 1);
  for (int i = l; i <= r; i++)
    vax += (s[i].x - avx) * (s[i].x - avx),
        vay += (s[i].y - avy) * (s[i].y - avy);
  if (vax >= vay)
    d[mid] = 1, nth_element(s + l, s + mid, s + r + 1, cmp1);
  else
    d[mid] = 2, nth_element(s + l, s + mid, s + r + 1, cmp2);
  lc[mid] = build(l, mid - 1), rc[mid] = build(mid + 1, r);
  maintain(mid);
  return mid;
}

double f(int a, int b) {
  double ret = 0;
  if (L[b] > s[a].x) ret += (L[b] - s[a].x) * (L[b] - s[a].x);
  if (R[b] < s[a].x) ret += (s[a].x - R[b]) * (s[a].x - R[b]);
  if (D[b] > s[a].y) ret += (D[b] - s[a].y) * (D[b] - s[a].y);
  if (U[b] < s[a].y) ret += (s[a].y - U[b]) * (s[a].y - U[b]);
  return ret;
}

void query(int l, int r, int x) {
  if (l > r) return;
  int mid = (l + r) >> 1;
  if (mid != x) ans = min(ans, dist(x, mid));
  if (l == r) return;
  double distl = f(x, lc[mid]), distr = f(x, rc[mid]);
  if (distl < ans && distr < ans) {
    if (distl < distr) {
      query(l, mid - 1, x);
      if (distr < ans) query(mid + 1, r, x);
    } else {
      query(mid + 1, r, x);
      if (distl < ans) query(l, mid - 1, x);
    }
  } else {
    if (distl < ans) query(l, mid - 1, x);
    if (distr < ans) query(mid + 1, r, x);
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i <= n; i++) cin >> s[i].x >> s[i].y;
```

```
  build(1, n);
  for (int i = 1; i <= n; i++) query(1, n, i);
  cout << fixed << setprecision(4) << sqrt(ans) << '\n';
  return 0;
}

```

例题 「CQOI2016」K 远点对[3]

给定平面上的 n 个点 (x_i, y_i)，求欧几里得距离下的第 k 远无序点对之间的距离.

$n \leq 100000, 1 \leq k \leq 100, 0 \leq x_i, y_i < 2^{31}$

和上一道例题类似，从最近点对变成了 k 远点对，估价函数改成了查询点到子树对应的长方形区域的最远距离. 用一个小根堆来维护当前找到的前 k 远点对之间的距离，如果当前找到的点对距离大于堆顶，则弹出堆顶并插入这个距离，同样的，使用堆顶的距离来剪枝.

由于题目中强调的是无序点对，即交换前后两点的顺序后仍是相同的点对，则每个有序点对会被计算两次，那么读入的 k 要乘以 2.

参考代码

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
constexpr int MAXN = 100010;
long long n, k;
priority_queue<long long, vector<long long>, greater<long long>> q;

struct node {
  long long x, y;
} s[MAXN];

bool cmp1(node a, node b) { return a.x < b.x; }

bool cmp2(node a, node b) { return a.y < b.y; }

long long lc[MAXN], rc[MAXN], L[MAXN], R[MAXN], D[MAXN], U[MAXN];

void maintain(int x) {
  L[x] = R[x] = s[x].x;
  D[x] = U[x] = s[x].y;
  if (lc[x])
    L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
    D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
  if (rc[x])
    L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
    D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
}

```

```
int build(int l, int r) {
  if (l > r) return 0;
  int mid = (l + r) >> 1;
  double av1 = 0, av2 = 0, va1 = 0, va2 = 0;  // average variance
  for (int i = l; i <= r; i++) av1 += s[i].x, av2 += s[i].y;
  av1 /= (r - l + 1);
  av2 /= (r - l + 1);
  for (int i = l; i <= r; i++)
    va1 += (av1 - s[i].x) * (av1 - s[i].x),
        va2 += (av2 - s[i].y) * (av2 - s[i].y);
  if (va1 > va2)
    nth_element(s + l, s + mid, s + r + 1, cmp1);
  else
    nth_element(s + l, s + mid, s + r + 1, cmp2);
  lc[mid] = build(l, mid - 1);
  rc[mid] = build(mid + 1, r);
  maintain(mid);
  return mid;
}

long long sq(long long x) { return x * x; }

long long dist(int a, int b) {
  return max(sq(s[a].x - L[b]), sq(s[a].x - R[b])) +
         max(sq(s[a].y - D[b]), sq(s[a].y - U[b]));
}

void query(int l, int r, int x) {
  if (l > r) return;
  int mid = (l + r) >> 1;
  long long t = sq(s[mid].x - s[x].x) + sq(s[mid].y - s[x].y);
  if (t > q.top()) q.pop(), q.push(t);
  long long distl = dist(x, lc[mid]), distr = dist(x, rc[mid]);
  if (distl > q.top() && distr > q.top()) {
    if (distl > distr) {
      query(l, mid - 1, x);
      if (distr > q.top()) query(mid + 1, r, x);
    } else {
      query(mid + 1, r, x);
      if (distl > q.top()) query(l, mid - 1, x);
    }
  } else {
    if (distl > q.top()) query(l, mid - 1, x);
    if (distr > q.top()) query(mid + 1, r, x);
  }
}

int main() {
  cin >> n >> k;
  k *= 2;
  for (int i = 1; i <= k; i++) q.push(0);
  for (int i = 1; i <= n; i++) cin >> s[i].x >> s[i].y;
```

```
  build(1, n);
  for (int i = 1; i <= n; i++) query(1, n, i);
  cout << q.top() << endl;
  return 0;
}

```

习题

「SDOI2010」捉迷藏[4]

「Violet」天使玩偶/SJY 摆棋子[5]

「国家集训队」JZPFAR[6]

「BOI2007」Mokia 摩基亚[7]

luogu P4475 巧克力王国[8]

「CH 弱省胡策 R2」TATT[9]

外部参考资料

[1] 洛谷 P4148 简单题 

[2] luogu P1429 平面最近点对（加强版） 

[3] 「CQOI2016」K 远点对 

[4] 「SDOI2010」捉迷藏 

[5] 「Violet」天使玩偶/SJY 摆棋子 

[6] 「国家集训队」JZPFAR 

[7] 「BOI2007」Mokia 摩基亚 

[8] luogu P4475 巧克力王国 

[9] 「CH 弱省胡策 R2」TATT 

10.20 动态树

10.20.1 Link Cut Tree

简介

Link/Cut Tree 是一种数据结构，我们用它来解决 **动态树问题**.

Link/Cut Tree 又称 Link-Cut Tree，简称 LCT，但它不叫动态树，动态树是指一类问题.

Splay Tree 是 LCT 的基础，但是 LCT 用的 Splay Tree 和普通的 Splay 在细节处不太一样（进行了一些扩展）.

问题引入

维护一棵树，支持如下操作：

- 修改两点间路径权值.
- 查询两点间路径权值和.
- 修改某点子树权值.
- 查询某点子树权值和.

这是一道树剖模版题.

但是再加一个操作：

- 断开并连接一些边，保证仍是一棵树.

要求在线求出上面的答案.

这就成了动态树问题，可以使用 LCT 求解.

动态树问题

维护一个 **森林**，支持删除某条边，加入某条边，并保证加边，删边之后仍是森林. 我们要维护这个森林的一些信息.

一般的操作有两点连通性，两点路径权值和，连接两点和切断某条边、修改信息等.

从 LCT 的角度回顾一下树链剖分

- 对整棵树按子树大小进行剖分，并重新标号.
- 我们发现重新标号之后，在树上形成了一些以链为单位的连续区间，并且可以用线段树进行区间操作.

转向动态树问题

我们发现我们刚刚讲的树剖是以子树大小作为划分条件. 那我们能不能重定义一种剖分，使它更适应我们的动态树问题呢?

考虑动态树问题需要什么链.

由于动态维护一个森林，显然我们希望这个链是我们指定的链，以便利用来求解.

实链剖分

对于一个点连向它所有儿子的边，我们自己选择一条边进行剖分，我们称被选择的边为实边，其他边则为虚边. 对于实边，我们称它所连接的儿子为实儿子. 对于一条由实边组成的链，我们同样称之为实链. 请记住我们选择实链剖分的最重要的原因：它是我们选择的，灵活且可变. 正是它的这种灵活可变性，我们采用 Splay Tree 来维护这些实链.

LCT

我们可以简单的把 LCT 理解成用一些 Splay 来维护动态的树链剖分，以期实现动态树上的区间操作. 对于每条实链，我们建一个 Splay 来维护整个链区间的信息.

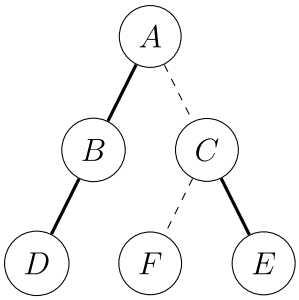
辅助树

我们先来看一看辅助树的一些性质，再通过一张图实际了解一下辅助树的具体结构.

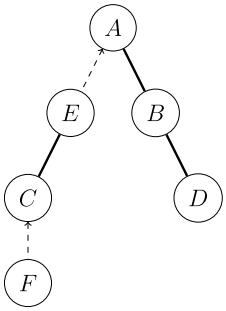
在本文里，你可以认为一些 Splay 构成了一个辅助树，每棵辅助树维护的是一棵树，一些辅助树构成了 LCT，其维护的是整个森林.

1. 辅助树由多棵 Splay 组成，每棵 Splay 维护原树中的一条路径，且中序遍历这棵 Splay 得到的点序列，从前到后对应原树「从上到下」的一条路径.
2. 原树每个节点与辅助树的 Splay 节点一一对应.
3. 辅助树的各棵 Splay 之间并不是独立的. 每棵 Splay 的根节点的父亲节点本应是空，但在 LCT 中每棵 Splay 的根节点的父亲节点指向原树中 **这条链** 的父亲节点（即链最顶端的点的父亲节点）. 这类父亲链接与通常 Splay 的父亲链接区别在于儿子认父亲，而父亲不认儿子，对应原树的一条 **虚边**. 因此，每个连通块恰好有一个点的父亲节点为空.
4. 由于辅助树的以上性质，我们维护任何操作都不需要维护原树，辅助树可以在任何情况下拿出一个唯一的原树，我们只需要维护辅助树即可.

现在我们有一棵原树，如图所示.（加粗边是实边，虚线边是虚边.）



由刚刚的定义，辅助树的结构如图所示.



考虑原树和辅助树的结构关系

- 原树中的实链 : 在辅助树中节点都在一棵 Splay 中.
- 原树中的虚链 : 在辅助树中，子节点所在 Splay 的 Father 指向父节点，但是父节点的两个儿子都不指向子节点.
- 注意：原树的根不等于辅助树的根.
- 原树的 Father 指向不等于辅助树的 Father 指向.
- 辅助树是可以在满足辅助树、Splay 的性质下任意换根的.
- 虚实链变换可以轻松在辅助树上完成，这也就是实现了动态维护树链剖分.

接下来要用到的变量声明

- `ch[N][2]` 左右儿子
- `f[N]` 父亲指向
- `sum[N]` 路径权值和
- `val[N]` 点权
- `tag[N]` 翻转标记
- `laz[N]` 权值标记
- `siz[N]` 辅助树上子树大小
- Other_Vars

函数声明

一般数据结构函数（字面意思）

1. `PushUp(x)`
2. `PushDown(x)`

Splay 树的函数

下面是 Splay 树中用到的函数，具体可以查阅 Splay 树→10.15.3@p2254.

1. `Get(x)` 获取 x 是父亲的哪个儿子.
2. `Splay(x)` 通过和 Rotate 操作联动实现把 x 旋转到 **当前 Splay 的根**.
3. `Rotate(x)` 将 x 向上旋转一层的操作.

新操作

1. `Access(x)` 把从根到 x 的所有点放在一条实链里，使根到 x 成为一条实路径，并且在同一棵 Splay 里. **只有此操作是必须实现的，其他操作视题目而实现.**
2. `IsRoot(x)` 判断 x 是否是所在树的根.
3. `Update(x)` 在 `Access` 操作之后，递归地从上到下 `PushDown` 更新信息.
4. `MakeRoot(x)` 使 x 点成为其所在树的根.
5. `Link(x, y)` 在 x, y 两点间连一条边.
6. `Cut(x, y)` 把 x, y 两点间边删掉.
7. `Find(x)` 找到 x 所在树的根节点编号.
8. `Fix(x, v)` 修改 x 的点权为 v.
9. `Split(x, y)` 提取出 x, y 间的路径，方便做区间操作.

宏定义

- `#define ls ch[p][0]`
- `#define rs ch[p][1]`

函数讲解

`PushUp()`

```
void PushUp(int p) {
  // maintain other variables
  siz[p] = siz[ls] + siz[rs] + 1;
}
```

`PushDown()`

```
void PushDown(int p) {
  if (tag[p] != std_tag) {
    // pushdown the tag
    tag[p] = std_tag;
  }
}
```

`Splay() && Rotate()`

这里 `Splay()` 和 `Rotate()` 与 Splay 树的实现有些区别.

```
#define Get(x) (ch[f[x]][1] == x)

void Rotate(int x) {
  int y = f[x], z = f[y], k = Get(x);
  if (!isRoot(y)) ch[z][ch[z][1] == y] = x;
  // 上面这句一定要写在前面，普通的 Splay 是不用的，因为 isRoot （后面会讲）
  ch[y][k] = ch[x][!k], f[ch[x][!k]] = y;
  ch[x][!k] = y, f[y] = x, f[x] = z;
  PushUp(y), PushUp(x);
}

void Splay(int x) {
  Update(
      x);  // 马上就能看到啦. 在 Splay 之前要把旋转会经过的路径上的点都 PushDown
  for (int fa; fa = f[x], !isRoot(x); Rotate(x)) {
```

```
    if (!isRoot(fa)) Rotate(Get(fa) == Get(x) ? fa : x);
  }
}
```

以上函数可以查阅 Splay 树→10.15.3@p2254.

下面是 LCT 独有的函数.

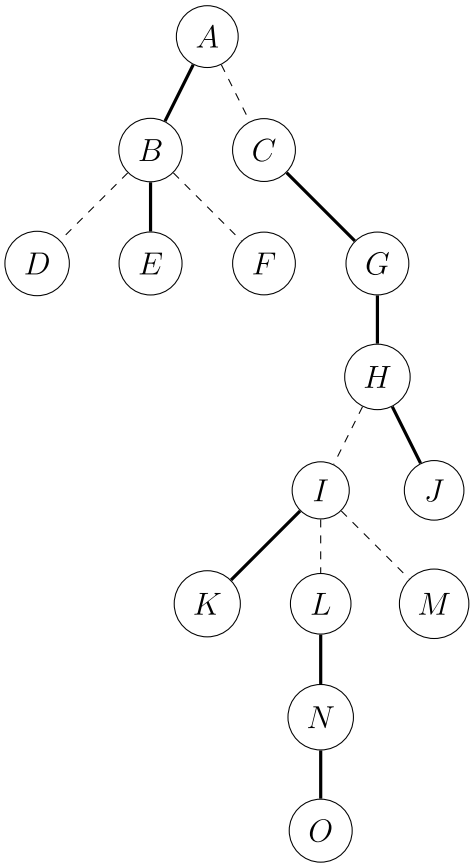
isRoot()

```
// 在前面我们已经说过，LCT 具有 如果一个儿子不是实儿子，他的父亲找不到它的性质
// 所以当一个点既不是它父亲的左儿子，又不是它父亲的右儿子，它就是当前 Splay 的根
#define isRoot(x) (ch[f[x]][0] != x && ch[f[x]][1] != x)
```

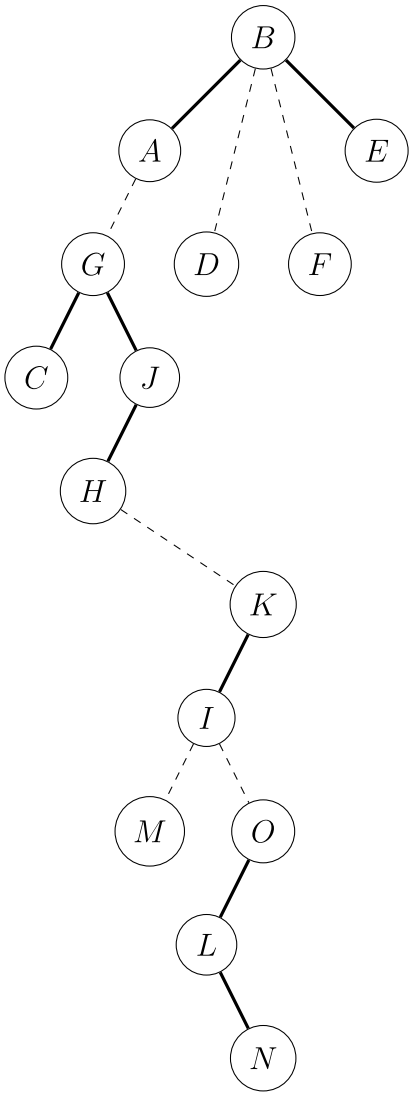
Access()

```
// Access 是 LCT
// 的核心操作，试想我们想求解一条路径，而这条路径恰好就是我们当前的一棵 Splay，
// 直接调用其信息即可. 先来看一下代码，再结合图来看看过程
int Access(int x) {
  int p;
  for (p = 0; x; p = x, x = f[x]) {
    Splay(x), ch[x][1] = p, PushUp(x);
  }
  return p;
}
```

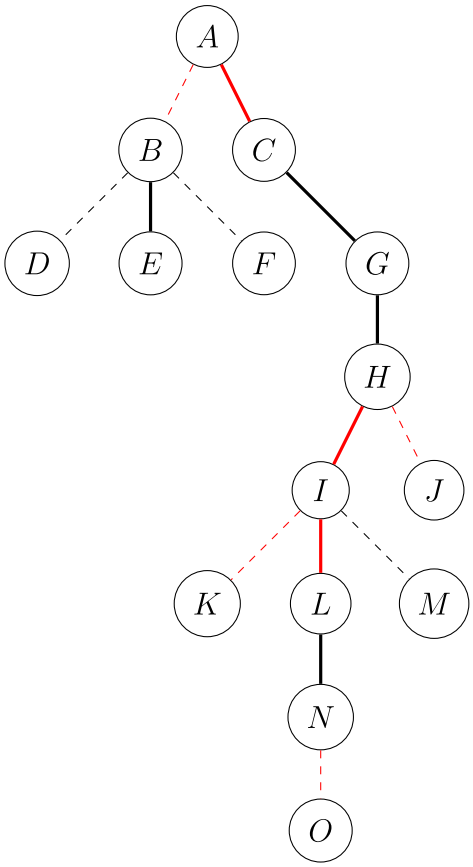
- 我们有这样一棵树，实线为实边，虚线为虚边.



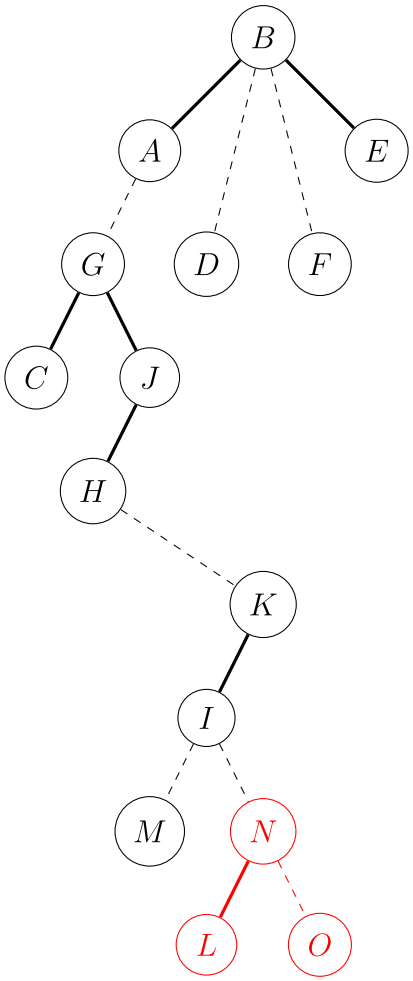
- 它的辅助树可能长成这样（构图方式不同可能 LCT 的结构也不同）.



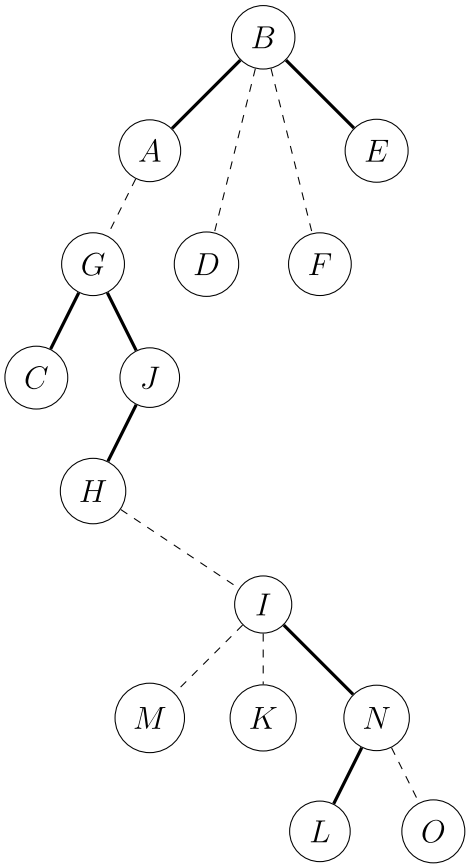
- 现在我们要 `Access(N)`，把 A 到 N 路径上的边都变为实边，拉成一棵 Splay.



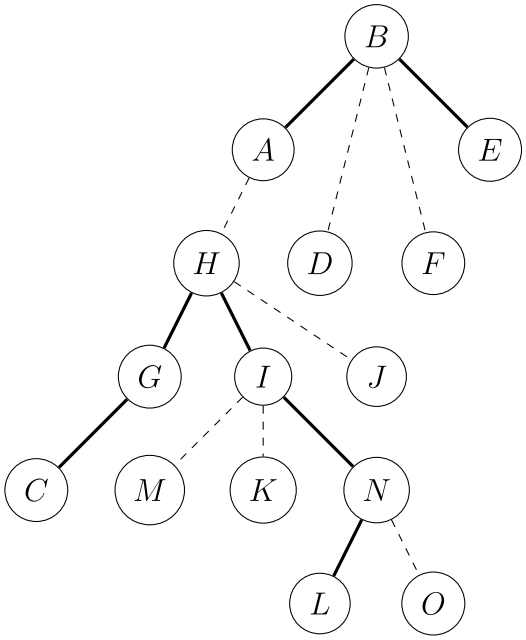
- 实现的方法是从下到上逐步更新 Splay.
- 首先我们要把 N 旋至当前 Splay 的根.
- 为了保证 AuxTree（辅助树）的性质，原来 N 到 O 的实边要更改为虚边.
- 由于认父不认子的性质，我们可以单方面的把 N 的儿子改为 `NULL`.
- 于是原来的 AuxTree 就从下图变成了下下图.



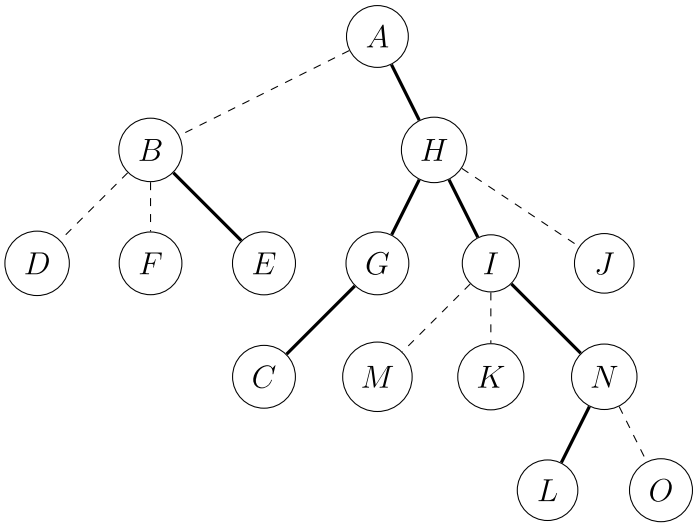
- 下一步，我们把 N 指向的 Father I 也旋转到 I 的 Splay 树根.
- 原来的实边 I—K 要去掉，这时候我们把 I 的右儿子指向 N，就得到了 I—L 这样一棵 Splay.



- 接下来，按照刚刚的操作步骤，由于 I 的 Father 指向 H，我们把 H 旋转到他所在 Splay Tree 的根，然后把 H 的 rs 设为 I.
- 之后的树是这样的.



- 同理我们 `Splay(A)`，并把 A 的右儿子指向 H.
- 于是我们得到了这样一棵 AuxTree. 并且发现 A—N 的整个路径已经在同一棵 Splay 中了.




```
// 回顾一下代码
int Access(int x) {
  int p;
  for (p = 0; x; p = x, x = f[x]) {
    Splay(x), ch[x][1] = p, PushUp(x);
  }
  return p;
}
```

我们发现 `Access()` 其实很容易，只有如下四步操作：

1. 把当前节点转到根.
2. 把儿子换成之前的节点.
3. 更新当前点的信息.
4. 把当前点换成当前点的父亲，继续操作.

这里提供的 Access 还有一个返回值．这个返回值相当于最后一次虚实链变换时虚边父亲节点的编号．该值有两个含义：

- 连续两次 Access 操作时，第二次 Access 操作的返回值等于这两个节点的 LCA.
- 表示 x 到根的链所在的 Splay 树的根．这个节点一定已经被旋转到了根节点，且父亲一定为空.

`Update()`

```
// 从上到下一层一层 pushDown 即可
void Update(int p) {
  if (!isRoot(p)) Update(f[p]);
  pushDown(p);
}
```

`makeRoot()`

- `Make_Root()` 的重要性丝毫不亚于 `Access()`．我们在需要维护路径信息的时候，一定会出现路径深度无法严格递增的情况，根据 AuxTree 的性质，这种路径是不能出现在一棵 Splay 中的.
- 这时候我们需要用到 `Make_Root()`.
- `Make_Root()` 的作用是使指定的点成为原树的根，考虑如何实现这种操作.
- 设 `Access(x)` 的返回值为 y，则此时 x 到当前根的路径恰好构成一个 Splay，且该 Splay 的根为 y.
- 考虑将树用有向图表示出来，给每条边定一个方向，表示从儿子到父亲的方向．容易发现换根相当于将 x 到根的路径的所有边反向（请仔细思考）.
- 因此将 x 到当前根的路径翻转即可.

- 由于 y 是 x 到当前根的路径所代表的 Splay 的根，因此将以 y 为根的 Splay 树进行区间翻转即可.

```
void makeRoot(int p) {
  p = Access(p);
  swap(ch[p][0], ch[p][1]);
  tag[p] ^= 1;
}
```

Link()

- Link 两个点其实很简单，先 Make_Root(x)，然后把 x 的父亲指向 y 即可. 显然，这个操作肯定不能发生在同一棵树内，所以记得先判一下.

```
void Link(int x, int p) {
  makeRoot(x);
  splay(x);
  f[x] = p;
}
```

Split()

- Split 操作意义很简单，就是拿出一棵 Splay，维护的是 x 到 y 的路径.
- 先 MakeRoot(x)，然后 Access(y). 如果要 y 做根，再 Splay(y).
- 另外 Split 这三个操作可以直接把需要的路径拿出到 y 的子树上，可以进行其他操作.

Cut()

- Cut 有两种情况，保证合法和不一定保证合法.
- 如果保证合法，直接 Split(x, y)，这时候 y 是根，x 一定是它的儿子，双向断开即可. 就像这样:

```
void Cut(int x, int p) { makeRoot(x), Access(p), Splay(p), ls = f[x] = 0; }
```

如果是不保证合法，我们需要判断一下是否有，这里选择使用 map 存一下，但是这里有一个利用性质的方法:

想要删边，必须要满足如下三个条件:

1. x, y 连通.
2. x, y 的路径上没有其他的链.
3. x 没有右儿子.

总结一下，上面三句话的意思就一个: x, y 之间有边.

具体实现就留作一个思考题给大家. 判断连通需要用到后面的 Find，其他两点稍作思考分析一下结构就知道该怎么判断了.

Find()

- Find() 查找的是 x 所在的 **原树** 的根，请不要把原树根和辅助树根弄混. 在 Access(p) 后，再 Splay(p). 这样根就是树里深度最小的那个，一直往左儿子走，沿途 PushDown 即可.
- 一直走到没有 ls，非常简单.
- 注意，每次查询之后需要把查询到的答案对应的结点 Splay 上去以保证复杂度.

```
int Find(int p) {
  Access(p);
  Splay(p);
  pushDown(p);
  while (ls) p = ls, pushDown(p);
  Splay(p);
```

```
  return p;
}
```

注意事项

- 操作前一定要想一想需不需要 `PushUp` 或者 `PushDown`，LCT 由于特别灵活的原因，少 `Pushdown` 或者 `Pushup` 一次就可能把修改改到不该改的点上！
- LCT 的 `Rotate` 和 Splay 的不太一样，`if (z)` 一定要放在前面.
- LCT 的 `Splay` 操作就是旋转到根，没有旋转到谁儿子的操作，因为不需要.

时间复杂度

LCT 中的大部分操作都基于 `Access`，其余操作的时间复杂度都为常数，因此我们只需要分析 `Access` 操作的时间复杂度.

其中，`Access` 的时间复杂度主要来自于多次 splay 操作和对路径中虚边的访问，接下来分别分析这两部分的时间复杂度.

1. splay
   - 定义 $w(x)=\log size(x)$，其中 $size(x)$ 表示以 x 为根的所有虚边和实边的数量之和.
   - 定义势能函数 $\Phi=\sum_{x\in T}w(x)$，其中 T 表示所有节点的集合.

   由 Splay 的时间复杂度→10.15.3@p2254 分析易知，splay 操作的均摊时间复杂度为 $O(\log n)$.
2. 访问虚边

   参考 重链剖分→11.6.6@p2658，定义两种虚边：
   - **重虚边**：从节点 v 到其父节点的虚边，其中 $size(v)>\frac{1}{2}size(parent(v))$.
   - **轻虚边**：从节点 v 到其父节点的虚边，其中 $size(v)\leq\frac{1}{2}size(parent(v))$.

   对于虚边的处理，可以使用势能分析，定义势能函数 Φ 为所有重虚边的数量，定义均摊成本 $c_i=t_i+\Delta\Phi_i$，其中 t_i 为实际操作的成本，$\Delta\Phi_i$ 为势能的变化.
   - 走过重虚边后，会将重虚边转换为实边，该操作会减少 1 的势能，因为它通过加强重要连接来优化树的结构. 且由于其实际操作成本为 $O(1)$，抵消了势能的增加，故不会增加均摊成本，所有的均摊成本集中在轻虚边的处理上.
   - 每次 `Access` 操作最多遍历 $O(\log n)$ 条轻虚边，因此至多消耗 $O(\log n)$ 的实际操作成本，转化得到 $O(\log n)$ 条重虚边，即势能以 $O(\log n)$ 的代价增加.

   由此，最终访问虚边的均摊复杂度为实际操作成本和势能变化的和，即 $O(\log n)$.

综上所述，LCT 中 `Access` 操作的时间复杂度是 splay 和 虚边访问的复杂度之和，因此最后的均摊复杂度为 $O(\log n)$，即 n 个节点的 LCT，做 m 次 `Access` 操作的时间复杂度为 $O(n\log n+m\log n)$，从而基于 `Access` 操作的 `Cut`,`Link`,`Findroot` 等操作的均摊复杂度也为 $O(\log n)$.

习题

- 「BZOJ 3282」Tree[1]
- 「HNOI2010」弹飞绵羊[2]

维护树链信息

LCT 通过 `Split(x,y)` 操作，可以将树上从点 x 到点 y 的路径提取到以 y 为根的 Splay 内，树链信息的修改和统计转化为平衡树上的操作，这使得 LCT 在维护树链信息上具有优势. 此外，借助 LCT 实现的在树链上二分比树链剖分少一个 $O(\log n)$ 的复杂度.

例题 「国家集训队」Tree II[3]

给出一棵有 n 个结点的树，每个点的初始权值为 1. q 次操作，每次操作均为以下四种之一：

1. `- u1 v1 u2 v2`：将树上 u_1, v_1 两点之间的边删除，连接 u_2, v_2 两点，保证操作合法且连边后仍是一棵树.
2. `+ u v c`：将树上 u, v 两点之间的路径上的点权都增加 c.
3. `* u v c`：将树上 u, v 两点之间的路径上的点权都乘以 c.
4. `/ u v`：输出树上 u, v 两点之间的路径上的点权之和对 51061 取模后的值.

$1 \leq n, q \leq 10^5, 0 \leq c \leq 10^4$

- 操作可以直接 `Cut(u1,v1),Link(u2,v2)`.

对树上 u, v 两点之间的路径进行修改时，先 `Split(u,v)`.

此题要求进行在辅助树上的子树加，子树乘，子树求和操作，所以我们除了一般 LCT 需要维护的子树翻转标记，还要维护子树加法标记和子树乘法标记. 处理标记的方法和在 Splay 上是一样的.

在打上和下传加法标记时，子树权值和的变化量和子树中的结点数有关，所以我们还要维护子树的大小 `siz`.

在下传标记时，需要注意顺序，先下传乘法标记再下传加法标记. 子树翻转和子树加乘两种标记没有冲突.

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
constexpr long long MAXN = 100010;
constexpr long long mod = 51061;
long long n, q, u, v, c;
char op;

struct Splay {
  long long ch[MAXN][2], fa[MAXN], siz[MAXN], val[MAXN], sum[MAXN],
rev[MAXN],
      add[MAXN], mul[MAXN];

  void clear(long long x) {
    ch[x][0] = ch[x][1] = fa[x] = siz[x] = val[x] = sum[x] = rev[x] =
add[x] =
        0;
    mul[x] = 1;
  }

  long long getch(long long x) { return (ch[fa[x]][1] == x); }

  long long isroot(long long x) {
    clear(0);
    return ch[fa[x]][0] != x && ch[fa[x]][1] != x;
  }

  void maintain(long long x) {
    clear(0);
    siz[x] = (siz[ch[x][0]] + 1 + siz[ch[x][1]]) % mod;
```

```
30      sum[x] = (sum[ch[x][0]] + val[x] + sum[ch[x][1]]) % mod;
31    }
32
33    void pushdown(long long x) {
34      clear(0);
35      if (mul[x] != 1) {
36        if (ch[x][0])
37          mul[ch[x][0]] = (mul[x] * mul[ch[x][0]]) % mod,
38          val[ch[x][0]] = (val[ch[x][0]] * mul[x]) % mod,
39          sum[ch[x][0]] = (sum[ch[x][0]] * mul[x]) % mod,
40          add[ch[x][0]] = (add[ch[x][0]] * mul[x]) % mod;
41        if (ch[x][1])
42          mul[ch[x][1]] = (mul[x] * mul[ch[x][1]]) % mod,
43          val[ch[x][1]] = (val[ch[x][1]] * mul[x]) % mod,
44          sum[ch[x][1]] = (sum[ch[x][1]] * mul[x]) % mod,
45          add[ch[x][1]] = (add[ch[x][1]] * mul[x]) % mod;
46        mul[x] = 1;
47      }
48      if (add[x]) {
49        if (ch[x][0])
50          add[ch[x][0]] = (add[ch[x][0]] + add[x]) % mod,
51          val[ch[x][0]] = (val[ch[x][0]] + add[x]) % mod,
52          sum[ch[x][0]] = (sum[ch[x][0]] + add[x] * siz[ch[x][0]] % mod)
% mod;
53        if (ch[x][1])
54          add[ch[x][1]] = (add[ch[x][1]] + add[x]) % mod,
55          val[ch[x][1]] = (val[ch[x][1]] + add[x]) % mod,
56          sum[ch[x][1]] = (sum[ch[x][1]] + add[x] * siz[ch[x][1]] % mod)
% mod;
57        add[x] = 0;
58      }
59      if (rev[x]) {
60        if (ch[x][0]) rev[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]]
[1]);
61        if (ch[x][1]) rev[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]]
[1]);
62        rev[x] = 0;
63      }
64    }
65
66    void update(long long x) {
67      if (!isroot(x)) update(fa[x]);
68      pushdown(x);
69    }
70
71    void print(long long x) {
72      if (!x) return;
73      pushdown(x);
74      print(ch[x][0]);
75      printf("%lld ", x);
76      print(ch[x][1]);
77    }
```

```
78
79    void rotate(long long x) {
80      long long y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
81      fa[x] = z;
82      if (!isroot(y)) ch[z][chy] = x;
83      ch[y][chx] = ch[x][chx ^ 1];
84      fa[ch[x][chx ^ 1]] = y;
85      ch[x][chx ^ 1] = y;
86      fa[y] = x;
87      maintain(y);
88      maintain(x);
89      maintain(z);
90    }
91
92    void splay(long long x) {
93      update(x);
94      for (long long f = fa[x]; f = fa[x], !isroot(x); rotate(x))
95        if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
96    }
97
98    void access(long long x) {
99      for (long long f = 0; x; f = x, x = fa[x])
100       splay(x), ch[x][1] = f, maintain(x);
101   }
102
103   void makeroot(long long x) {
104     access(x);
105     splay(x);
106     swap(ch[x][0], ch[x][1]);
107     rev[x] ^= 1;
108   }
109
110   long long find(long long x) {
111     access(x);
112     splay(x);
113     while (ch[x][0]) x = ch[x][0];
114     splay(x);
115     return x;
116   }
117 } st;
118
119 main() {
120   scanf("%lld%lld", &n, &q);
121   for (long long i = 1; i <= n; i++) st.val[i] = 1, st.maintain(i);
122   for (long long i = 1; i < n; i++) {
123     scanf("%lld%lld", &u, &v);
124     if (st.find(u) != st.find(v)) st.makeroot(u), st.fa[u] = v;
125   }
126   while (q--) {
127     scanf(" %c%lld%lld", &op, &u, &v);
128     if (op == '+') {
129       scanf("%lld", &c);
```

```
130       st.makeroot(u), st.access(v), st.splay(v);
131       st.val[v] = (st.val[v] + c) % mod;
132       st.sum[v] = (st.sum[v] + st.siz[v] * c % mod) % mod;
133       st.add[v] = (st.add[v] + c) % mod;
134     }
135     if (op == '-') {
136       st.makeroot(u);
137       st.access(v);
138       st.splay(v);
139       if (st.ch[v][0] == u && !st.ch[u][1]) st.ch[v][0] = st.fa[u] = 0;
140       scanf("%lld%lld", &u, &v);
141       if (st.find(u) != st.find(v)) st.makeroot(u), st.fa[u] = v;
142     }
143     if (op == '*') {
144       scanf("%lld", &c);
145       st.makeroot(u), st.access(v), st.splay(v);
146       st.val[v] = st.val[v] * c % mod;
147       st.sum[v] = st.sum[v] * c % mod;
148       st.mul[v] = st.mul[v] * c % mod;
149     }
150     if (op == '/')
151       st.makeroot(u), st.access(v), st.splay(v), printf("%lld\n",
    st.sum[v]);
152   }
153   return 0;
154 }
```

习题

- luogu P3690【模板】Link Cut Tree（动态树）[4]
- 「SDOI2011」染色[5]
- 「SHOI2014」三叉神经树[6]

维护连通性质

判断是否连通

借助 LCT 的 `Find()` 函数，可以判断动态森林上的两点是否连通．如果有 `Find(x)==Find(y)`，则说明 x, y 两点在一棵树上，相互连通．

例题 「SDOI2008」洞穴勘测[7]

一开始有 n 个独立的点，m 次操作．每次操作为以下之一：

1. `Connect u v`：在 u, v 两点之间连接一条边．
2. `Destroy u v`：删除在 u, v 两点之间的边，保证之前存在这样的一条边．
3. `Query u v`：询问 u, v 两点是否连通．

保证在任何时刻图的形态都是一个森林．

$n \leq 10^4, m \leq 2 \times 10^5$

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
constexpr int MAXN = 10010;

struct Splay {
  int ch[MAXN][2], fa[MAXN], tag[MAXN];

  void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = tag[x] = 0; }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) swap(ch[ch[x][0]][0], ch[ch[x][0]][1]), tag[ch[x][0]] ^= 1;
      if (ch[x][1]) swap(ch[ch[x][1]][0], ch[ch[x][1]][1]), tag[ch[x][1]] ^= 1;
      tag[x] = 0;
    }
  }

  void update(int x) {
    if (!isroot(x)) update(fa[x]);
    pushdown(x);
  }

  void rotate(int x) {
    int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
    fa[x] = z;
    if (!isroot(y)) ch[z][chy] = x;
    ch[y][chx] = ch[x][chx ^ 1];
    fa[ch[x][chx ^ 1]] = y;
    ch[x][chx ^ 1] = y;
    fa[y] = x;
  }

  void splay(int x) {
    update(x);
    for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
      if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
  }

  void access(int x) {
    for (int f = 0; x; f = x, x = fa[x]) splay(x), ch[x][1] = f;
  }

  void makeroot(int x) {
    access(x);
```

```
51      splay(x);
52      swap(ch[x][0], ch[x][1]);
53      tag[x] ^= 1;
54    }
55
56    int find(int x) {
57      access(x);
58      splay(x);
59      while (ch[x][0]) x = ch[x][0];
60      splay(x);
61      return x;
62    }
63  } st;
64
65  int n, q, x, y;
66  char op[MAXN];
67
68  int main() {
69    scanf("%d%d", &n, &q);
70    while (q--) {
71      scanf("%s%d%d", op, &x, &y);
72      if (op[0] == 'Q') {
73        if (st.find(x) == st.find(y))
74          printf("Yes\n");
75        else
76          printf("No\n");
77      }
78      if (op[0] == 'C')
79        if (st.find(x) != st.find(y)) st.makeroot(x), st.fa[x] = y;
80      if (op[0] == 'D') {
81        st.makeroot(x);
82        st.access(y);
83        st.splay(y);
84        if (st.ch[y][0] == x && !st.ch[x][1]) st.ch[y][0] = st.fa[x] = 0;
85      }
86    }
87    return 0;
88  }
```

维护边双连通分量

如果要求将边双连通分量缩成点，每次添加一条边，所连接的树上的两点如果相互连通，那么这条路径上的所有点都会被缩成一个点.

例题 「AHOI2005」航线规划[8]

给出 n 个点，初始时有 m 条无向边，q 次操作，每次操作为以下之一：

1. `0 u v`：删除 u, v 之间的连边，保证此时存在这样的一条边.
2. `1 u v`：查询此时 u, v 两点之间可能的所有路径必须经过的边的数量.

保证图在任意时刻都连通.

$1 < n < 3 \times 10^4, 1 < m < 10^5, 0 \leq q \leq 4 \times 10^4$

可以发现，u, v 两点之间的所有可能路径必须经过的边的数量为将所有边双连通分量缩成点之后 u 所在点和 v 所在点之间的路径上的结点数 -1.

由于题目中的删边操作不好进行，我们考虑离线逆向进行操作，改删边为加边.

加入一条边时，如果两点原来不连通，则在 LCT 上连接两点；否则提取出加这条边之前 LCT 上这两点之间的路径，遍历辅助树上的这个子树，相当于遍历了这条路径，将这些点合并，利用并查集维护合并的信息.

用合并后并查集的代表元素代替原来树上的路径. 注意之后的每次操作都要找到操作点在并查集上的代表元素进行操作.

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
using namespace std;
constexpr int MAXN = 200010;
int f[MAXN];

int findp(int x) { return f[x] ? f[x] = findp(f[x]) : x; }

void merge(int x, int y) {
  x = findp(x);
  y = findp(y);
  if (x != y) f[x] = y;
}

struct Splay {
  int ch[MAXN][2], fa[MAXN], tag[MAXN], siz[MAXN];

  void clear(int x) { ch[x][0] = ch[x][1] = fa[x] = tag[x] = siz[x] = 0; }

  int getch(int x) { return ch[findp(fa[x])][1] == x; }

  int isroot(int x) {
    return ch[findp(fa[x])][0] != x && ch[findp(fa[x])][1] != x;
  }

  void maintain(int x) {
    clear(0);
    if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]];
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) tag[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]]
[1]);
      if (ch[x][1]) tag[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]]
[1]);
      tag[x] = 0;
    }
```

```
}

void print(int x) {
  if (!x) return;
  pushdown(x);
  print(ch[x][0]);
  printf("%d ", x);
  print(ch[x][1]);
}

void update(int x) {
  if (!isroot(x)) update(findp(fa[x]));
  pushdown(x);
}

void rotate(int x) {
  x = findp(x);
  int y = findp(fa[x]), z = findp(fa[y]), chx = getch(x), chy = getch(y);
  fa[x] = z;
  if (!isroot(y)) ch[z][chy] = x;
  ch[y][chx] = ch[x][chx ^ 1];
  fa[ch[x][chx ^ 1]] = y;
  ch[x][chx ^ 1] = y;
  fa[y] = x;
  maintain(y);
  maintain(x);
  if (z) maintain(z);
}

void splay(int x) {
  x = findp(x);
  update(x);
  for (int f = findp(fa[x]); f = findp(fa[x]), !isroot(x); rotate(x))
    if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
}

void access(int x) {
  for (int f = 0; x; f = x, x = findp(fa[x]))
    splay(x), ch[x][1] = f, maintain(x);
}

void makeroot(int x) {
  x = findp(x);
  access(x);
  splay(x);
  tag[x] ^= 1;
  swap(ch[x][0], ch[x][1]);
}

int find(int x) {
  x = findp(x);
  access(x);
```

```
    splay(x);
    while (ch[x][0]) x = ch[x][0];
    splay(x);
    return x;
  }

  void dfs(int x) {
    pushdown(x);
    if (ch[x][0]) dfs(ch[x][0]), merge(ch[x][0], x);
    if (ch[x][1]) dfs(ch[x][1]), merge(ch[x][1], x);
  }
} st;

int n, m, q, x, y, cur, ans[MAXN];

struct oper {
  int op, a, b;
} s[MAXN];

map<pair<int, int>, int> mp;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) st.maintain(i);
  for (int i = 1; i <= m; i++)
    scanf("%d%d", &x, &y), mp[{x, y}] = mp[{y, x}] = 1;
  while (scanf("%d", &s[++q].op)) {
    if (s[q].op == -1) {
      q--;
      break;
    }
    scanf("%d%d", &s[q].a, &s[q].b);
    if (!s[q].op) mp[{s[q].a, s[q].b}] = mp[{s[q].b, s[q].a}] = 0;
  }
  reverse(s + 1, s + q + 1);
  for (map<pair<int, int>, int>::iterator it = mp.begin(); it != mp.end();
it++)
    if (it->second) {
      mp[{it->first.second, it->first.first}] = 0;
      x = findp(it->first.first);
      y = findp(it->first.second);
      if (st.find(x) != st.find(y))
        st.makeroot(x), st.fa[x] = y;
      else {
        if (x == y) continue;
        st.makeroot(x);
        st.access(y);
        st.splay(y);
        st.dfs(y);
        int t = findp(y);
        st.fa[t] = findp(st.fa[y]);
        st.ch[t][0] = st.ch[t][1] = 0;
```

```
142         st.maintain(t);
143       }
144     }
145   for (int i = 1; i <= q; i++) {
146     if (s[i].op == 0) {
147       x = findp(s[i].a);
148       y = findp(s[i].b);
149       st.makeroot(x);
150       st.access(y);
151       st.splay(y);
152       st.dfs(y);
153       int t = findp(y);
154       st.fa[t] = st.fa[y];
155       st.ch[t][0] = st.ch[t][1] = 0;
156       st.maintain(t);
157     }
158     if (s[i].op == 1) {
159       x = findp(s[i].a);
160       y = findp(s[i].b);
161       st.makeroot(x);
162       st.access(y);
163       st.splay(y);
164       ans[++cur] = st.siz[y] - 1;
165     }
166   }
167   for (int i = cur; i >= 1; i--) printf("%d\n", ans[i]);
168   return 0;
169 }
```

习题

- 洛谷 P3950 部落冲突[9]
- BZOJ 4998 星球联盟[10]
- BZOJ 2959 长跑[11]

维护边权

LCT 并不能直接处理边权，此时需要对每条边建立一个对应点，方便查询链上的边信息．利用这一技巧可以动态维护生成树．

例题 luogu P4234 最小差值生成树[12]

给定一个 n 个点，m 条边的带权无向图，求其边权最大值和边权最小值的差值最小的生成树，输出这个差值．

数据保证至少存在一棵生成树．

$1 \leq n \leq 5 \times 10^4, 1 \leq m \leq 2 \times 10^5, 1 \leq w_i \leq 10^4$

将边按照边权从小到大排序，枚举选择的最右边的一条边，要得到最优解，需要使边权最小边的边权最大．

每次按照顺序添加边，如果将要连接的这两个点已经连通，则删除这两点之间边权最小的一条边．如果整个图已经连通成了一棵树，则用当前边权减去最小边权更新答案．最小边权可用双指针法更新．

LCT 上没有固定的父子关系，所以不能将边权记录在点权中.

记录树链上的边的信息，可以使用 **拆边**. 对每条边建立一个对应的点，从这条边向其两个端点连接一条边，原先的连边与删边操作都变成两次操作.

参考代码

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <set>
using namespace std;
constexpr int MAXN = 5000010;

struct Splay {
  int ch[MAXN][2], fa[MAXN], tag[MAXN], val[MAXN], minn[MAXN];

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = tag[x] = val[x] = minn[x] = 0;
  }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }

  void maintain(int x) {
    if (!x) return;
    minn[x] = x;
    if (ch[x][0]) {
      if (val[minn[ch[x][0]]] < val[minn[x]]) minn[x] = minn[ch[x][0]];
    }
    if (ch[x][1]) {
      if (val[minn[ch[x][1]]] < val[minn[x]]) minn[x] = minn[ch[x][1]];
    }
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) tag[ch[x][0]] ^= 1, swap(ch[ch[x][0]][0], ch[ch[x][0]]
[1]);
      if (ch[x][1]) tag[ch[x][1]] ^= 1, swap(ch[ch[x][1]][0], ch[ch[x][1]]
[1]);
      tag[x] = 0;
    }
  }

  void update(int x) {
    if (!isroot(x)) update(fa[x]);
    pushdown(x);
  }

  void print(int x) {
    if (!x) return;
    pushdown(x);
```

```
46      print(ch[x][0]);
47      printf("%d ", x);
48      print(ch[x][1]);
49    }
50
51    void rotate(int x) {
52      int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
53      fa[x] = z;
54      if (!isroot(y)) ch[z][chy] = x;
55      ch[y][chx] = ch[x][chx ^ 1];
56      fa[ch[x][chx ^ 1]] = y;
57      ch[x][chx ^ 1] = y;
58      fa[y] = x;
59      maintain(y);
60      maintain(x);
61      if (z) maintain(z);
62    }
63
64    void splay(int x) {
65      update(x);
66      for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
67        if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
68    }
69
70    void access(int x) {
71      for (int f = 0; x; f = x, x = fa[x]) splay(x), ch[x][1] = f,
  maintain(x);
72    }
73
74    void makeroot(int x) {
75      access(x);
76      splay(x);
77      tag[x] ^= 1;
78      swap(ch[x][0], ch[x][1]);
79    }
80
81    int find(int x) {
82      access(x);
83      splay(x);
84      while (ch[x][0]) x = ch[x][0];
85      splay(x);
86      return x;
87    }
88
89    void link(int x, int y) {
90      makeroot(x);
91      fa[x] = y;
92    }
93
94    void cut(int x, int y) {
95      makeroot(x);
96      access(y);
```

```
    splay(y);
    ch[y][0] = fa[x] = 0;
    maintain(y);
  }
} st;

constexpr int inf = 2e9 + 1;
int n, m, ans, nww, x, y;

struct Edge {
  int u, v, w;

  bool operator<(Edge x) const { return w < x.w; };
} s[MAXN];

multiset<int> mp;

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) st.val[i] = inf, st.maintain(i);
  for (int i = 1; i <= m; i++) scanf("%d%d%d", &s[i].u, &s[i].v, &s[i].w);
  sort(s + 1, s + m + 1);
  for (int i = 1; i <= m; i++) st.val[n + i] = s[i].w, st.maintain(n + i);
  for (int i = 1; i <= m; i++) {
    x = s[i].u;
    y = s[i].v;
    if (x == y) continue;
    if (st.find(x) != st.find(y)) {
      nww++;
      st.link(x, n + i);
      st.link(n + i, y);
      mp.insert(s[i].w);
      if (nww == n - 1) ans = s[i].w - (*(mp.begin()++));
    } else {
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      int t = st.minn[y] - n;
      st.cut(s[t].u, t + n);
      st.cut(t + n, s[t].v);
      mp.erase(mp.find(s[t].w));
      st.link(x, n + i);
      st.link(n + i, y);
      mp.insert(s[i].w);
      if (nww == n - 1) ans = min(ans, s[i].w - (*(mp.begin()++)));
    }
  }
  printf("%d\n", ans);
  return 0;
}
```

习题

- 「WC2006」水管局长[13]
- 「BJWC2010」严格次小生成树[14]
- 「NOI2014」魔法森林[15]

维护子树信息

LCT 不擅长维护子树信息．统计一个结点所有虚子树的信息，就可以求得整棵树的信息．

> **例题 「BJOI2014」大融合**[16]
>
> 给定 n 个结点和 q 次操作，每个操作为如下形式：
> 1. `A x y` 在结点 x 和 y 之间连接一条边．
> 2. `Q x y` 给定一条已经存在的边 (x,y)，求有多少条简单路径，其中包含边 (x,y)．
>
> 保证在任意时刻，图的形态都是一棵森林．
>
> $1 \le n,q,x,y \le 10^5$

为询问 `Q` 考虑另一种表述，我们发现答案等于边 (x,y) 在 x 侧的结点数与 y 侧的结点数的乘积，即将边 (x,y) 断开后分别包含 x 和 y 的树的结点数．为了消除断边的影响，在询问后我们再次连接边 (x,y)．

题目中的操作既有连边，又有删边，还保证在任意时刻都是一棵森林，我们不由得想到用 LCT 来维护．但是这题中 LCT 维护的是子树的大小，不像我们印象中的维护一条链的信息，而 LCT 的构造 **认父不认子**，不方便我们直接进行子树的统计．怎么办呢?

方法是统计一个结点 x 所有虚儿子（即父亲为 x，但 x 在 Splay 中的左右儿子并不包含它）所代表的子树的贡献．

定义 $siz2[x]$ 为结点 x 的所有虚儿子代表的子树的结点数，$siz[x]$ 为 结点 x 子树中的结点数．

不同于以往我们维护 Splay 中子树结点个数的方法，我们在计算结点 x 子树中的结点数时，还要加上 $siz2[x]$，即

```
void maintain(int x) {
  clear(0);
  if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]] + siz2[x];
}
```

而且在我们 **改变 Splay 的形态**（即改变一个结点在 Splay 上的左右儿子指向时），需要及时修改 $siz2[x]$ 的值．

在 `Rotate(),Splay()` 操作中，我们都只是改变了 Splay 中结点的相对位置，没有改变任意一条边的虚实情况，所以不对 $siz2[x]$ 进行任何修改．

在 `access` 操作中，在每次 splay 完后，都会改变刚刚 splay 完的结点的右儿子，即该结点与其原右儿子的连边和该节点和新右儿子的连边的虚实情况发生了变化，我们需要加上新变成虚边所连的子树的贡献，减去刚刚变成实边所连的子树的贡献．代码如下：

```
void access(int x) {
  for (int f = 0; x; f = x, x = fa[x])
    splay(x), siz2[x] += siz[ch[x][1]] - siz[f], ch[x][1] = f, maintain(x);
}
```

在 `MakeRoot(),Find()` 操作中，我们都只是调用了之前的函数或者在 Splay 上条边，并不用做任何修改．

在连接两点时，我们修改了一个结点的父亲．我们需要在父亲结点的 $siz2$ 值中加上新子结点的子树大小贡献．

```
st.makeroot(x);
st.makeroot(y);
st.fa[x] = y;
st.siz2[y] += st.siz[x];
```

在断开一条边时，我们只是删除了 Splay 上的一条实边，`Maintain` 操作会维护这些信息，不需要做任何修改.

以上是代码修改的细节，最后总结一下 LCT 维护子树信息的要求与方法:

1. 维护的信息要有 **可减性**，如子树结点数，子树权值和，但不能直接维护子树最大最小值，因为在将一条虚边变成实边时要排除原先虚边的贡献.
2. 新建一个附加值存储虚子树的贡献，在统计时将其加入本结点答案，在改变边的虚实时及时维护.
3. 其余部分同普通 LCT，在统计子树信息时一定将其作为根节点.
4. 如果维护的信息没有可减性, 如维护区间最值, 可以对每个结点开一个平衡树维护结点的虚子树中的最值.

参考代码

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;
constexpr int MAXN = 100010;
using ll = long long;

struct Splay {
  int ch[MAXN][2], fa[MAXN], siz[MAXN], siz2[MAXN], tag[MAXN];

  void clear(int x) {
    ch[x][0] = ch[x][1] = fa[x] = siz[x] = siz2[x] = tag[x] = 0;
  }

  int getch(int x) { return ch[fa[x]][1] == x; }

  int isroot(int x) { return ch[fa[x]][0] != x && ch[fa[x]][1] != x; }

  void maintain(int x) {
    clear(0);
    if (x) siz[x] = siz[ch[x][0]] + 1 + siz[ch[x][1]] + siz2[x];
  }

  void pushdown(int x) {
    if (tag[x]) {
      if (ch[x][0]) swap(ch[ch[x][0]][0], ch[ch[x][0]][1]), tag[ch[x][0]]
^= 1;
      if (ch[x][1]) swap(ch[ch[x][1]][0], ch[ch[x][1]][1]), tag[ch[x][1]]
^= 1;
      tag[x] = 0;
    }
  }

  void update(int x) {
```

```
33      if (!isroot(x)) update(fa[x]);
34      pushdown(x);
35    }
36
37    void rotate(int x) {
38      int y = fa[x], z = fa[y], chx = getch(x), chy = getch(y);
39      fa[x] = z;
40      if (!isroot(y)) ch[z][chy] = x;
41      ch[y][chx] = ch[x][chx ^ 1];
42      fa[ch[x][chx ^ 1]] = y;
43      ch[x][chx ^ 1] = y;
44      fa[y] = x;
45      maintain(y);
46      maintain(x);
47      maintain(z);
48    }
49
50    void splay(int x) {
51      update(x);
52      for (int f = fa[x]; f = fa[x], !isroot(x); rotate(x))
53        if (!isroot(f)) rotate(getch(x) == getch(f) ? f : x);
54    }
55
56    void access(int x) {
57      for (int f = 0; x; f = x, x = fa[x])
58        splay(x), siz2[x] += siz[ch[x][1]] - siz[f], ch[x][1] = f,
  maintain(x);
59    }
60
61    void makeroot(int x) {
62      access(x);
63      splay(x);
64      swap(ch[x][0], ch[x][1]);
65      tag[x] ^= 1;
66    }
67
68    int find(int x) {
69      access(x);
70      splay(x);
71      while (ch[x][0]) x = ch[x][0];
72      splay(x);
73      return x;
74    }
75  } st;
76
77  int n, q, x, y;
78  char op;
79
80  int main() {
81    scanf("%d%d", &n, &q);
82    while (q--) {
83      scanf(" %c%d%d", &op, &x, &y);
```

```
    if (op == 'A') {
      st.makeroot(x);
      st.makeroot(y);
      st.fa[x] = y;
      st.siz2[y] += st.siz[x];
    }
    if (op == 'Q') {
      st.makeroot(x);
      st.access(y);
      st.splay(y);
      st.ch[y][0] = st.fa[x] = 0;
      st.maintain(x);
      st.makeroot(x);
      st.makeroot(y);
      printf("%lld\n", (ll)(st.siz[x] * st.siz[y]));
      st.makeroot(x);
      st.makeroot(y);
      st.fa[x] = y;
      st.siz2[y] += st.siz[x];
    }
  }
  return 0;
}
```

习题

- luogu P4299 首都[17]
- SPOJ QTREE5 - Query on a tree V[18]

外部参考资料

[1] 「BZOJ 3282」 Tree

[2] 「HNOI2010」弹飞绵羊

[3] 「国家集训队」 Tree II

[4] luogu P3690【模板】Link Cut Tree（动态树）

[5] 「SDOI2011」染色

[6] 「SHOI2014」三叉神经树

[7] 「SDOI2008」洞穴勘测

[8] 「AHOI2005」航线规划

[9] 洛谷 P3950 部落冲突

[10] BZOJ 4998 星球联盟

[11] BZOJ 2959 长跑

[12] luogu P4234 最小差值生成树

[13] 「WC2006」水管局长

[14]「BJWC2010」严格次小生成树

[15] 「NOI2014」魔法森林

[16] 「BJOI2014」大融合

[17] luogu P4299 首都

[18] SPOJ QTREE5 - Query on a tree V

10.20.2 全局平衡二叉树

引入

前置知识：树链剖分→11.6.6@p2658

由于树链剖分的时间复杂度为 $O(n\log^2 n)$，而我们熟知的 LCT 虽然时间复杂度为 $O(n\log n)$，但常数较大，可能比树链剖分还慢．那么有什么既是 $O(n\log n)$ 的，常数又相对较小的方法呢？这个时候全局平衡二叉树就出现了．

全局平衡二叉树实际上是一颗二叉树森林，其中的每颗二叉树维护一条重链．但是这个森林里的二叉树又互有联系，其中每个二叉树的根连向这个重链链头的父亲，就像 LCT 中一样．但全局平衡二叉树是静态树，区别于 LCT，建成后树的形态不变．

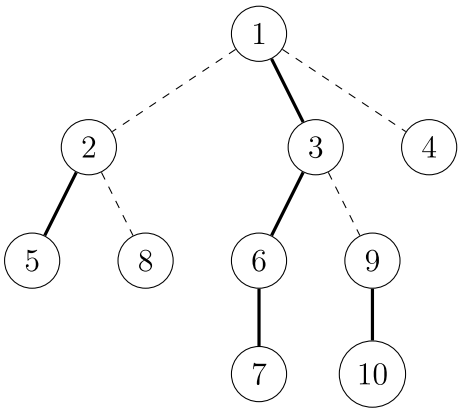
全局平衡二叉树是一种可以处理树上链修改/查询的数据结构，可以做到：

- $O(\log n)$ 一条链整体修改．
- $O(\log n)$ 一条链整体查询．
- $O(\log n)$ 求最近公共祖先，子树修改，子树查询等，这些复杂度和重链剖分是一样的．

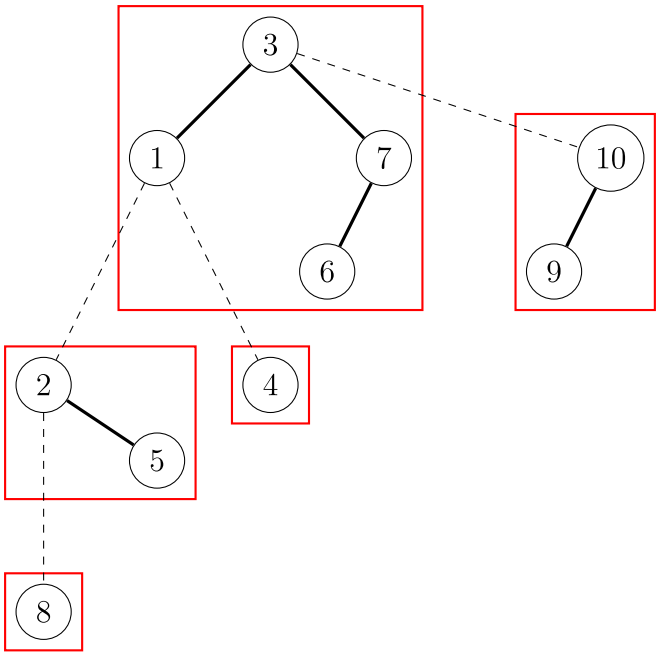
主要性质

1. 全局平衡二叉树由很多棵二叉树通过轻边连起来组成，每一棵二叉树维护了原树的一条重链，其中序遍历的顺序就是这条重链深度单调递增的顺序．每个节点都仅出现在一棵二叉树中．
2. 边分为重边和轻边，重边是包含在二叉树中的边，维护的时候就像正常维护二叉树一样，记录左右儿子和父节点．轻边从一颗二叉树的根节点指向它所对应的重链顶端节点的父节点．轻边维护的时候"认父不认子"，即只能从子节点访问到父节点，不能反过来．注意，全局平衡二叉树中的边和原树中的边没有对应关系．
3. 算上重边和轻边，全局平衡二叉树的高度是 $O(\log n)$ 级别的．这条是保证全局平衡二叉树时间复杂度的性质．

下面是一个全局平衡二叉树建树的例子．第一张图是原树，以节点 1 为根节点．实线是重边．



第二张图是建出来的全局平衡二叉树，其中虚线是轻边，实线是重边，每一棵二叉树用红圈表示．



建树

首先是像普通重链剖分一样，一次 DFS 求出每个节点的重儿子．然后从根开始，找到根节点所在的重链，对于这些点的轻儿子递归建树，并连上轻边．然后我们需要给重链上的点建一棵二叉树．我们先把重链上的点存到数组里，求出每个点轻儿子的子树大小之和加一（即该点本身所贡献的 size）．然后我们按照这个求出这条重链的加权中点，把它作为二叉树的根，两边递归建树，并连上重边．

代码如下：

实现

```
std::vector<int> G[N];
int n, fa[N], son[N], sz[N];

void dfsS(int u) {
  sz[u] = 1;
  for (int v : G[u]) {
    dfsS(v);
    sz[u] += sz[v];
    if (sz[v] > sz[son[u]]) son[u] = v;
  }
}

int b[N], bs[N], l[N], r[N], f[N], ss[N];

// 给b中[bl,br)内的点建二叉树，返回二叉树的根
int cbuild(int bl, int br) {
  int x = bl, y = br;
  while (y - x > 1) {
    int mid = (x + y) >> 1;
    if (2 * (bs[mid] - bs[bl]) <= bs[br] - bs[bl])
      x = mid;
    else
      y = mid;
```

```
  }
  // 二分求出按bs加权的中点
  y = b[x];
  ss[y] = br - bl;  // ss：二叉树中重子树的大小
  if (bl < x) {
    l[y] = cbuild(bl, x);
    f[l[y]] = y;
  }
  if (x + 1 < br) {
    r[y] = cbuild(x + 1, br);
    f[r[y]] = y;
  }
  return y;
}

int build(int x) {
  int y = x;
  do
    for (int v : G[y])
      if (v != son[y])
        f[build(v)] =
            y;  // 递归建树并连轻边，注意要从二叉树的根连边，不是从儿子连边
  while (y = son[y]);
  y = 0;
  do {
    b[y++] = x;                                  // 存放重链中的点
    bs[y] = bs[y - 1] + sz[x] - sz[son[x]];  // bs：轻儿子size和+1，求前缀和
  } while (x = son[x]);
  return cbuild(0, y);
}
```

由代码可以看出建树的时间复杂度是 $O(n \log n)$. 接下来我们可以证明树高是 $O(\log n)$ 的：考虑从任意一个点跳父节点到根. 跳轻边就相当于在原树中跳到另一条重链，由重链剖分的性质可得跳轻边最多 $O(\log n)$ 条；因为建二叉树的时候根节点找的是算轻儿子的加权中点，那么跳一次重边算上轻儿子的 size 至少翻倍，所以跳重边最多也是 $O(\log n)$ 条. 整体树高就是 $O(\log n)$ 的.

查询

以上就是关于全局平衡二叉树的部分. 剩下关于链修改和链查询的操作方法相对简单，只需要从要操作的点出发，一直跳跃到根节点. 要操作某个点所在的重链上比它深度小的所有点，本质上等同于在这条重链的二叉树中操作目标节点左侧的所有节点. 这些操作可以分解成一系列子树操作，与普通二叉树的维护方法类似，其中涉及到维护子树和以及打子树标记. 在这一过程中，使用的是标记永久化. 也可以用 pushdown 来打标记，用 pushup 维护子树和，不过这种方式可能相对复杂，因为通常情况下，处理二叉树是自上而下进行操作，但在这里，需要首先确定跳跃路径，然后再从上到下进行 pushdown，可能导致常数较大.

代码如下：

实现

```
// a：子树加标记
```

```
// s: 子树和（不算加标记的）
int a[N], s[N];

void add(int x) {
  bool t = true;
  int z = 0;
  while (x) {
    s[x] += z;
    if (t) {
      a[x]++;
      if (r[x]) a[r[x]]--;
      z += 1 + ss[l[x]];
      s[x] -= ss[r[x]];
    }
    t = (x != l[f[x]]);
    if (t && x != r[f[x]]) z = 0;  // 跳过轻边要清空
    x = f[x];
  }
}

int query(int x) {
  int ret = 0;
  bool t = true;
  int z = 0;
  while (x) {
    if (t) {
      ret += s[x] - s[r[x]];
      ret -= 1ll * ss[r[x]] * a[r[x]];
      z += 1 + ss[l[x]];
    }
    ret += 1ll * z * a[x];
    t = (x != l[f[x]]);
    if (t && x != r[f[x]]) z = 0;  // 跳过轻边要清空
    x = f[x];
  }
  return ret;
}
```

此外，对于子树操作，就是要考虑轻儿子的，需要再维护一个包括轻儿子的子树和、子树标记，可以去做 "P3384【模板】轻重链剖分[1]".

例题

P4751【模板】"动态 DP"& 动态树分治（加强版）[2]

```
#include <algorithm>
#include <cstdio>
#include <cstring>
constexpr int MAXN = 1000000;
constexpr int MAXM = 3000000;
constexpr int INF = 0x3FFFFFFF;
using namespace std;
```

```
8
9  struct edge {
10   int to;
11   edge *nxt;
12 } edges[MAXN * 2 + 5];
13
14 edge *ncnt = &edges[0], *Adj[MAXN + 5];
15 int n, m;
16
17 struct Matrix {
18   int M[2][2];
19
20   Matrix operator*(const Matrix &B) const {
21     static Matrix ret;
22     for (int i = 0; i < 2; i++)
23       for (int j = 0; j < 2; j++) {
24         ret.M[i][j] = -INF;
25         for (int k = 0; k < 2; k++)
26           ret.M[i][j] = max(ret.M[i][j], M[i][k] + B.M[k][j]);
27       }
28     return ret;
29   }
30 } matr1[MAXN + 5], matr2[MAXN + 5];  // 每个点维护两个矩阵
31
32 int root;
33 int w[MAXN + 5], dep[MAXN + 5], son[MAXN + 5], siz[MAXN + 5], lsiz[MAXN
   + 5];
34 int g[MAXN + 5][2], f[MAXN + 5][2], trfa[MAXN + 5], bstch[MAXN + 5][2];
35 int stk[MAXN + 5], tp;
36 bool vis[MAXN + 5];
37
38 void AddEdge(int u, int v) {
39   edge *p = ++ncnt;
40   p->to = v;
41   p->nxt = Adj[u];
42   Adj[u] = p;
43
44   edge *q = ++ncnt;
45   q->to = u;
46   q->nxt = Adj[v];
47   Adj[v] = q;
48 }
49
50 void DFS(int u, int fa) {
51   siz[u] = 1;
52   for (edge *p = Adj[u]; p != NULL; p = p->nxt) {
53     int v = p->to;
54     if (v == fa) continue;
55     dep[v] = dep[u] + 1;
56     DFS(v, u);
57     siz[u] += siz[v];
58     if (!son[u] || siz[son[u]] < siz[v]) son[u] = v;
```

```
  }
  lsiz[u] = siz[u] - siz[son[u]];  // 轻儿子的siz和+1
}

void DFS2(int u, int fa) {
  f[u][1] = w[u], f[u][0] = 0;
  g[u][1] = w[u], g[u][0] = 0;
  if (son[u]) {
    DFS2(son[u], u);
    f[u][0] += max(f[son[u]][0], f[son[u]][1]);
    f[u][1] += f[son[u]][0];
  }
  for (edge *p = Adj[u]; p != NULL; p = p->nxt) {
    int v = p->to;
    if (v == fa || v == son[u]) continue;
    DFS2(v, u);
    f[u][0] += max(f[v][0], f[v][1]);  // f[][]就是正常的DP数组
    f[u][1] += f[v][0];
    g[u][0] += max(f[v][0], f[v][1]);  // g[][]数组只统计了自己和轻儿子的信息
    g[u][1] += f[v][0];
  }
}

void PushUp(int u) {
  matr2[u] = matr1[u];  // matr1是单点加上轻儿子的信息，matr2是区间信息
  if (bstch[u][0]) matr2[u] = matr2[bstch[u][0]] * matr2[u];
  // 注意转移的方向，但是如果我们的矩乘定义不同，可能方向也会不同
  if (bstch[u][1]) matr2[u] = matr2[u] * matr2[bstch[u][1]];
}

int getmx2(int u) { return max(matr2[u].M[0][0], matr2[u].M[0][1]); }

int getmx1(int u) { return max(getmx2(u), matr2[u].M[1][0]); }

int SBuild(int l, int r) {
  if (l > r) return 0;
  int tot = 0;
  for (int i = l; i <= r; i++) tot += lsiz[stk[i]];
  for (int i = l, sumn = lsiz[stk[l]]; i <= r; i++, sumn += lsiz[stk[i]])
    if (sumn * 2 >= tot)  // 是重心了
    {
      int lch = SBuild(l, i - 1), rch = SBuild(i + 1, r);
      bstch[stk[i]][0] = lch;
      bstch[stk[i]][1] = rch;
      trfa[lch] = trfa[rch] = stk[i];
      PushUp(stk[i]);  // 将区间的信息统计上来
      return stk[i];
    }
  return 0;
}

int Build(int u) {
```

```
111   for (int pos = u; pos; pos = son[pos]) vis[pos] = true;
112   for (int pos = u; pos; pos = son[pos])
113     for (edge *p = Adj[pos]; p != NULL; p = p->nxt)
114       if (!vis[p->to])  // 是轻儿子
115       {
116         int v = p->to, ret = Build(v);
117         trfa[ret] = pos;  // 轻儿子的treefa[]接上来
118       }
119   tp = 0;
120   for (int pos = u; pos; pos = son[pos]) stk[++tp] = pos;  // 把重链取出来
121   int ret = SBuild(1, tp);  // 对重链进行单独的SBuild(我猜是Special Build?)
122   return ret;               // 返回当前重链的二叉树的根
123 }
124
125 void Modify(int u, int val) {
126   matr1[u].M[1][0] += val - w[u];
127   w[u] = val;
128   for (int pos = u; pos; pos = trfa[pos])
129     if (trfa[pos] && bstch[trfa[pos]][0] != pos && bstch[trfa[pos]][1] !=
    pos) {
130       matr1[trfa[pos]].M[0][0] -= getmx1(pos);
131       matr1[trfa[pos]].M[0][1] = matr1[trfa[pos]].M[0][0];
132       matr1[trfa[pos]].M[1][0] -= getmx2(pos);
133       PushUp(pos);
134       matr1[trfa[pos]].M[0][0] += getmx1(pos);
135       matr1[trfa[pos]].M[0][1] = matr1[trfa[pos]].M[0][0];
136       matr1[trfa[pos]].M[1][0] += getmx2(pos);
137     } else
138       PushUp(pos);
139 }
140
141 int read() {
142   int ret = 0, f = 1;
143   char c = 0;
144   while (c < '0' || c > '9') {
145     c = getchar();
146     if (c == '-') f = -f;
147   }
148   ret = 10 * ret + c - '0';
149   while (true) {
150     c = getchar();
151     if (c < '0' || c > '9') break;
152     ret = 10 * ret + c - '0';
153   }
154   return ret * f;
155 }
156
157 void print(int x) {
158   if (x == 0) return;
159   print(x / 10);
160   putchar(x % 10 + '0');
161 }
```

```
162
163 int main() {
164   scanf("%d %d", &n, &m);
165   for (int i = 1; i <= n; i++) w[i] = read();
166   int u, v;
167   for (int i = 1; i < n; i++) {
168     u = read(), v = read();
169     AddEdge(u, v);
170   }
171   DFS(1, -1);
172   // 求重儿子
173   DFS2(1, -1);
174   // 求初始的DP值，也可以在Build()里面求，但是这样写就和树剖的写法统一了
175   for (int i = 1; i <= n; i++) {
176     matr1[i].M[0][0] = matr1[i].M[0][1] = g[i][0];
177     matr1[i].M[1][0] = g[i][1], matr1[i].M[1][1] = -INF;  // 初始化矩阵
178   }
179   root = Build(1);  // root即为根节点所在重链的重心
180   int lastans = 0;
181   for (int i = 1; i <= m; i++) {
182     u = read(), v = read();
183     u ^= lastans;  // 强制在线
184     Modify(u, v);
185     lastans = getmx1(root);  // 直接取值
186     if (lastans == 0)
187       putchar('0');
188     else
189       print(lastans);
190     putchar('\n');
191   }
192   return 0;
193 }
```

参考

P4211 [LNOI2014] LCA | 全局平衡二叉树[3]

外部参考资料

[1] P3384【模板】轻重链剖分

[2] P4751【模板】"动态 DP"&动态树分治（加强版）

[3] P4211 [LNOI2014] LCA | 全局平衡二叉树

10.20.3 Euler Tour Tree

Authors: Backl1ght

Euler Tour Tree（欧拉游览树，欧拉回路树，后文简称 ETT）是一种可以解决 **动态树** 问题的数据结构. ETT 将动态树的操作转换成了其 DFS 序列上的区间操作，再用其他数据结构来维护序列的区间操作，从而维护动态树的操作. 例如，ETT 将动态树的加边操作转换成了多个序列拆分操作和序列合并操作，如果能维护序列拆分操作和序列合并操作，就能维护动态树的加边操作.

LCT 也是一种可以解决动态树问题的数据结构，相比 ETT 而言 LCT 会更加常见. LCT 其实更适用于维护树链的信息，而 ETT 更加适用于维护 **子树** 的信息. 例如，ETT 可以维护子树最小值而 LCT 不能.

ETT 可以使用任意数据结构维护，只需要该数据结构支持对应的序列区间操作，以及在复杂度上满足要求．一般情况下会使用例如 Splay，Treap 等平衡二叉搜索树来维护序列，而这些数据结构维护区间操作的复杂度均为 $O(\log n)$，由此也可以在 $O(\log n)$ 的时间内维护动态树的操作．如果使用多叉平衡搜索树例如 B 树来维护区间操作，也可以做到更优的复杂度．

其实 ETT 可以理解为一种思想，就是通过维护某种和原树一一对应的序列，从而达到维护原树的目的，本文介绍的只是这个思想的一些可行的实现和应用．

树的欧拉回路表示

如果把一条树边看成两条有向边的话，那么就可以把一棵树表示成一个有向图的欧拉回路，称为树的欧拉回路表示（Euler tour representation，ETR）．

后面要维护的序列其实是 ETR 的一个变种，把树中的点看成了自环也加到了 ETR 中，但是由于原始论文中作者没有给它起新的名字，就还是叫它 ETR 吧．

可以通过下述算法得到树 T 的 欧拉回路表示：

```
1 Input. A rooted tree T
2 Output. The dfs sequence of rooted tree T
3 ET(u)
4       visit vertex u
5       for all child v of u
6           visit directed edge u → v
7           ET(v)
8           visit directed edge v → u
```

树 T 的欧拉回路表示 $\mathrm{ETR}(T)$ 初始为空，DFS 的过程中每次访问一个节点或者一条有向边时就将其加到 $\mathrm{ETR}(T)$ 的尾部，如此便可得到 $\mathrm{ETR}(T)$．

若 T 中包含 n 个节点，则中包含 $2n-2$ 条有向边，而 DFS 的过程中，每个点和每条有向边都会被访问一次，所以 $\mathrm{ETR}(T)$ 的长度为 $3n-2$．

把点 u 看成是一个自环，这样 $\mathrm{ETR}(T)$ 就可以看成有向图中的一个欧拉回路．可以在欧拉回路的某处断开，将其看成是一些边的首尾相连组成的链；也可以把这样的链在断开处重新粘起来变回欧拉回路；还可以通过新增一些边把两个这样的链拼成一个新的欧拉回路．

后文中，如未说明默认维护的序列是树的欧拉回路表示．

ETT 的基本操作

以下 3 个操作算是 ETT 的基本操作，均可以转换成常数次序列的操作，所以这 3 个操作的复杂度和序列操作同阶．

这里给出的只是一种可行的实现，只要能用常数次序列操作把修改后对应的序列拼出来即可．

MakeRoot(u)

即换根操作．ETT 中的换根操作被转换成了 1 个序列拆分操作和 1 个序列合并操作，也可以理解成 1 个区间平移操作．

记包含点 u 的树为 T，当前其树根为 r，现在要将树根换成 u．树 T 对应的序列为 L，将 L 在 (u,u) 处拆分成序列 L^1 和 L^2，前者包含 L 中 (u,u) 之前的元素以及 (u,u)，后者包含剩余元素．则依次将 L^2 和 L^1 合并得到的序列，即为换根之后树对应的序列．

这里可以理解成对一个欧拉回路进行旋转操作，欧拉回路是一个环，旋转并不会改变欧拉回路的结构，也即不会改变树的结构，只是把点 u 旋转到了根的位置而已．

Insert(u, v)

即加边操作．ETT 中加边操作被转换成了 2 个序列拆分操作和 5 个序列合并操作．

记包含点 u 的树为 T_1，包含点 v 的树为 T_2，加边之后两颗树合并成了一颗树 T．树 T_1 对应的序列为 L_1，树 T_2 对应的序列为 L_2．

将 L_1 在 (u,u) 处拆分成序列 L_1^1 和 L_1^2，前者包含 L_1 中 (u,u) 之前的元素以及 (u,u)，后者包含剩余元素．类似地将 L_2 在 (v,v) 处拆分成序列 L_2^1 和 L_2^2．则依次将 $L_1^2, L_1^1, [(u,v)], L_2^2, L_2^1, [(v,u)]$ 合并即可得到树 T 对应的序列 L．

这里可以理解成两次换根操作，然后把两个欧拉回路在当前根的位置处断开，再用新加的两条有向边把两个欧拉回路拼成一个新的欧拉回路．

Delete(u, v)

即删边操作．ETT 中删边操作被转换成了 4 个序列拆分操作以及 1 个序列合并操作．

记包含边 (u,v) 和边 (v,u) 的树为 T，其对应序列为 L．删边之后 T 分成了两颗树．

将 L 拆分成 $L_1, [(u,v)], L_2, [(v,u)], L_3$，删边形成的两颗树对应的序列分别为 L_2 以及 L_1, L_3．注意，在序列 L 中 $[(u,v)]$ 有可能出现在 $[(v,u)]$ 的后面，此时可以先交换 u 和 v 的值然后再操作．

这里可以理解成把一个欧拉回路从两条有向边处断开形成两条链，然后两条链自己首尾相连形成两个新的欧拉回路．

实现

以下以非旋 Treap 为例介绍 ETT 的实现，需要读者事先了解使用非旋 Treap 维护区间操作的相关内容．

`Split` 和 `Merge` 都是非旋 Treap 的基本操作了，这里不再赘述．

SplitUp2(u)

假设 u 所在的序列为 L，将 L 在 u 处拆分成序列 L^1 和 L^2，前者包含 L 中 u 之前的元素以及 u，后者包含剩余元素．

如果 Treap 的每个节点额外维护自己的父亲的话，就可以实现 $O(\log n)$ 的时间内计算一个 Treap 节点对应的元素在序列中的位置，再根据位置去 `Split` 就可以实现上述功能．

也可以自底向上地拆分从而实现上述功能，这样做相比上述方法会更高效．具体就是，在从 u 对应的节点往根跳的过程中，根据二叉搜索树的性质就可以确定每个节点在 L 中位于 u 之前还是之后，根据这点就可以计算 u 在序列中的位置，也可以确定每个节点属于拆分后的哪一棵树．

```
/*
 * Bottom up split treap p into 2 treaps a and b.
 *   - a: a treap containing nodes with position less than or equal to p.
 *   - b: a treap containing nodes with postion greater than p.
 *
 * In the other word, split sequence containning p into two sequences, the first
 * one contains elements before p and element p, the second one contains
 * elements after p.
 */
static std::pair<Node*, Node*> SplitUp2(Node* p) {
  Node *a = nullptr, *b = nullptr;
  b = p->right_;
  if (b) b->parent_ = nullptr;
  p->right_ = nullptr;

  bool is_p_left_child_of_parent = false;
  bool is_from_left_child = false;
  while (p) {
    Node* parent = p->parent_;
```

```
20
21      if (parent) {
22        is_p_left_child_of_parent = (parent->left_ == p);
23        if (is_p_left_child_of_parent) {
24          parent->left_ = nullptr;
25        } else {
26          parent->right_ = nullptr;
27        }
28        p->parent_ = nullptr;
29      }
30
31      if (!is_from_left_child) {
32        a = Merge(p, a);
33      } else {
34        b = Merge(b, p);
35      }
36
37      is_from_left_child = is_p_left_child_of_parent;
38      p->Maintain();
39      p = parent;
40    }
41
42    return {a, b};
43  }
```

SplitUp3(u)

假设 u 所在的序列为 L，将 L 在 u 处拆分成序列 L^1,u 和 L^2，前者包含 L 中 u 之前的元素，后者包含剩余元素.

在 `SplitUp2` 的基础上稍作修改即可.

MakeRoot(u)

基于 `SplitUp2` 以及 `Merge` 易得.

```
1  void MakeRoot(int u) {
2    Node* vertex_u = vertices_[u];
3    auto [L1, L2] = Treap::SplitUp2(vertex_u);
4    Treap::Merge(L2, L1);
5  }
```

Insert(u, v)

基于 `SplitUp2` 以及 `Merge` 易得.

```
1  void Insert(int u, int v) {
2    Node* vertex_u = vertices_[u];
3    Node* vertex_v = vertices_[v];
4
5    Node* edge_uv = AllocateNode(u, v);
6    Node* edge_vu = AllocateNode(v, u);
7    tree_edges_[u][v] = edge_uv;
8    tree_edges_[v][u] = edge_vu;
9
10   auto [L11, L12] = Treap::SplitUp2(vertex_u);
11   auto [L21, L22] = Treap::SplitUp2(vertex_v);
```

```
12
13   Node* L = L12;
14   L = Treap::Merge(L, L11);
15   L = Treap::Merge(L, edge_uv);
16   L = Treap::Merge(L, L22);
17   L = Treap::Merge(L, L21);
18   L = Treap::Merge(L, edge_vu);
19 }
```

Delete(u, v)

基于 `SplitUp3` 以及 `Merge` 易得.

```
1  void Delete(int u, int v) {
2    Node* edge_uv = tree_edges_[u][v];
3    Node* edge_vu = tree_edges_[v][u];
4    tree_edges_[u].erase(v);
5    tree_edges_[v].erase(u);
6
7    int position_uv = Treap::GetPosition(edge_uv);
8    int position_vu = Treap::GetPosition(edge_vu);
9    if (position_uv > position_vu) {
10     std::swap(edge_uv, edge_vu);
11     std::swap(position_uv, position_vu);
12   }
13
14   auto [L1, uv, _] = Treap::SplitUp3(edge_uv);
15   auto [L2, vu, L3] = Treap::SplitUp3(edge_vu);
16   Treap::Merge(L1, L3);
17
18   FreeNode(edge_uv);
19   FreeNode(edge_vu);
20 }
```

维护连通性

点 u 和点 v 连通，当且仅当两个点属于同一棵树 T，即 (u,u) 和 (v,v) 属于 $\mathrm{ETR}(T)$，这可以根据点 u 和点 v 对应的 Treap 节点所在的 Treap 的根是否相同判断.

例题 P2147[SDOI2008] 洞穴勘测[1]

维护连通性的模板题.

参考代码

```
1  #include <cassert>
2  #include <cstdint>
3  #include <functional>
4  #include <iostream>
5  #include <map>
6  #include <random>
7  #include <sstream>
8
9  using i64 = int64_t;
10 using u64 = uint64_t;
```

```
11
12 void solve_case(int Case);
13
14 int main(int argc, char* argv[]) {
15   std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
16   int T = 1;
17   // std::cin >> T;
18   for (int t = 1; t <= T; ++t) {
19     solve_case(t);
20   }
21   return 0;
22 }
23
24 /**
25  * Dynamic Forest Maintained With Euler Tour Tree.
26  *
27  * As said in reference, link and cut operation of dynamic trees can be
28  * transformed into sequence split and sequence merge operation, which can
   be
29  * easily maintained using balanced search trees like Treap.
30  *
31  * @reference: Dynamic trees as search trees via euler tours, applied to the
32  * network simplex algorithm by Robert E. Tarjan.
33  * https://link.springer.com/article/10.1007/BF02614369
34  */
35 class DynamicForest {
36  private:
37   static std::mt19937 rng_;
38
39   struct Node {
40     int size_;
41     int priority_;
42
43     Node* left_;
44     Node* right_;
45     Node* parent_;
46
47     int from_;
48     int to_;
49
50     Node(int from, int to)
51         : size_(1),
52           priority_(rng_()),
53           left_(nullptr),
54           right_(nullptr),
55           parent_(nullptr),
56           from_(from),
57           to_(to) {}
58
59     void Maintain() {
60       size_ = 1;
61       if (left_) {
```

```
62          size_ += left_->size_;
63          left_->parent_ = this;
64        }
65        if (right_) {
66          size_ += right_->size_;
67          right_->parent_ = this;
68        }
69      }
70    };
71
72    static int GetSize(Node* p) { return p == nullptr ? 0 : p->size_; }
73
74    static Node* FindRoot(Node* p) {
75      if (!p) return nullptr;
76
77      while (p->parent_ != nullptr) p = p->parent_;
78      return p;
79    }
80
81    static std::string to_string(Node* p) {
82      std::stringstream ss;
83
84      ss << "Node [\n";
85
86      std::function<void(Node*)> dfs = [&](Node* p) {
87        if (!p) return;
88        dfs(p->left_);
89        ss << "(" << p->from_ << "," << p->to_ << "),";
90        dfs(p->right_);
91      };
92      dfs(p);
93
94      ss << "]\n\n";
95
96      return ss.str();
97    }
98
99    Node* AllocateNode(int u, int v) {
100     Node* p = new Node(u, v);
101     return p;
102   }
103
104   void FreeNode(Node*& p) {
105     if (p) {
106       delete p;
107       p = nullptr;
108     }
109   }
110
111   /*
112    * Dynamic Sequence Maintained using Treap.
113    */
```

```
114   class Treap {
115    public:
116     /**
117      * Merge two treap a and b into a single treap, with keys in a less than
118      * keys in b.
119      *
120      * In the other word, concating sequence a and sequence b.
121      */
122     static Node* Merge(Node* a, Node* b) {
123       if (a == nullptr) return b;
124       if (b == nullptr) return a;
125 
126       if (a->priority_ < b->priority_) {
127         a->right_ = Merge(a->right_, b);
128         a->Maintain();
129         return a;
130       } else {
131         b->left_ = Merge(a, b->left_);
132         b->Maintain();
133         return b;
134       }
135     }
136 
137     /**
138      * Get the number of nodes with keys less than or equal to the key of p.
139      *
140      * In the other word, the 1-based index of p inside the sequence
141      * containing p.
142      */
143     static int GetPosition(Node* p) {
144       assert(p != nullptr);
145 
146       int position = GetSize(p->left_) + 1;
147       while (p) {
148         if (p->parent_ && p == p->parent_->right_)
149           position += GetSize(p->parent_->left_) + 1;
150         p = p->parent_;
151       }
152       return position;
153     }
154 
155     /**
156      * Split sequence containning p into two sequences, the first one
contains
157      * the first k elements, the second one contains the remaining elements.
158      */
159     static std::pair<Node*, Node*> Split(Node* p, int k) {
160       if (!p) return {nullptr, nullptr};
161 
162       std::pair<Node*, Node*> result;
163 
164       if (GetSize(p->left_) < k) {
```

```
165         auto right_result = Split(p->right_, k - GetSize(p->left_) - 1);
166         p->right_ = right_result.first;
167 
168         result.first = p;
169         result.second = right_result.second;
170       } else {
171         auto left_result = Split(p->left_, k);
172         p->left_ = left_result.second;
173 
174         result.first = left_result.first;
175         result.second = p;
176       }
177 
178       p->Maintain();
179 
180       if (result.first) result.first->parent_ = nullptr;
181 
182       if (result.second) result.second->parent_ = nullptr;
183 
184       return result;
185     }
186 
187     /*
188      * Bottom up split treap p into 2 treaps a and b.
189      *   - a: a treap containing nodes with position less than or equal
    to p.
190      *   - b: a treap containing nodes with postion greater than p.
191      *
192      * In the other word, split sequence containning p into two sequences,
    the
193      * first one contains elements before p and element p, the second one
194      * contains elements after p.
195      */
196     static std::pair<Node*, Node*> SplitUp2(Node* p) {
197       assert(p != nullptr);
198 
199       Node *a = nullptr, *b = nullptr;
200       b = p->right_;
201       if (b) b->parent_ = nullptr;
202       p->right_ = nullptr;
203 
204       bool is_p_left_child_of_parent = false;
205       bool is_from_left_child = false;
206       while (p) {
207         Node* parent = p->parent_;
208 
209         if (parent) {
210           is_p_left_child_of_parent = (parent->left_ == p);
211           if (is_p_left_child_of_parent) {
212             parent->left_ = nullptr;
213           } else {
214             parent->right_ = nullptr;
```

```
215           }
216           p->parent_ = nullptr;
217         }
218
219         if (!is_from_left_child) {
220           a = Merge(p, a);
221         } else {
222           b = Merge(b, p);
223         }
224
225         is_from_left_child = is_p_left_child_of_parent;
226         p->Maintain();
227         p = parent;
228       }
229
230       return {a, b};
231     }
232
233     /*
234      * Bottom up split treap p into 3 treaps a, b and c.
235      *   - a: a treap containing nodes with key less than p.
236      *   - b: a treap containing nodes with key greater than p.
237      *   - c: a treap containing nodes with key equal p.
238      *
239      * In the other word, split sequence containning p into three sequences, the
240      * first one contains elements before p, the second one contains element p,
241      * the third one contains elements after p.
242      */
243     static std::tuple<Node*, Node*, Node*> SplitUp3(Node* p) {
244       assert(p != nullptr);
245
246       Node* a = p->left_;
247       if (a) a->parent_ = nullptr;
248       p->left_ = nullptr;
249
250       Node* b = p->right_;
251       if (b) b->parent_ = nullptr;
252       p->right_ = nullptr;
253
254       Node* c = p;
255
256       bool is_p_left_child_of_parent = false;
257       bool is_from_left_child = false;
258       Node* parent = p->parent_;
259       if (parent) {
260         is_p_left_child_of_parent = (parent->left_ == p);
261         if (is_p_left_child_of_parent) {
262           parent->left_ = nullptr;
263         } else {
264           parent->right_ = nullptr;
```

```
        }
        p->parent_ = nullptr;
      }
      is_from_left_child = is_p_left_child_of_parent;
      p->Maintain();
      p = parent;

      while (p) {
        Node* parent = p->parent_;

        if (parent) {
          is_p_left_child_of_parent = (parent->left_ == p);
          if (is_p_left_child_of_parent) {
            parent->left_ = nullptr;
          } else {
            parent->right_ = nullptr;
          }
          p->parent_ = nullptr;
        }

        if (!is_from_left_child) {
          a = Merge(p, a);
        } else {
          b = Merge(b, p);
        }

        is_from_left_child = is_p_left_child_of_parent;
        p->Maintain();
        p = parent;
      }

      return {a, c, b};
    }
  };

public:
  DynamicForest(int n) : n_(n), vertices_(n_), tree_edges_(n_) {
    assert(n_ > 0);

    for (int i = 0; i < n_; ++i) vertices_[i] = AllocateNode(i, i);
  }

  ~DynamicForest() {
    for (int i = 0; i < n_; ++i) {
      for (auto p : tree_edges_[i]) {
        FreeNode(p.second);
      }
    }
    for (int i = 0; i < n_; ++i) {
      FreeNode(vertices_[i]);
    }
  }
```

```
317
318   void Insert(int u, int v) {
319     assert(!tree_edges_[u].count(v));
320     assert(!tree_edges_[v].count(u));
321
322     Node* vertex_u = vertices_[u];
323     Node* vertex_v = vertices_[v];
324
325     Node* edge_uv = AllocateNode(u, v);
326     Node* edge_vu = AllocateNode(v, u);
327     tree_edges_[u][v] = edge_uv;
328     tree_edges_[v][u] = edge_vu;
329
330     int position_u = Treap::GetPosition(vertex_u);
331     int position_v = Treap::GetPosition(vertex_v);
332
333     auto p1 = Treap::SplitUp2(vertex_u);
334     auto p2 = Treap::SplitUp2(vertex_v);
335     auto L11 = p1.first, L12 = p1.second;
336     auto L21 = p2.first, L22 = p2.second;
337
338     assert(GetSize(L11) == position_u);
339     assert(GetSize(L21) == position_v);
340
341     Node* result = nullptr;
342     result = Treap::Merge(result, L12);
343     result = Treap::Merge(result, L11);
344     result = Treap::Merge(result, edge_uv);
345     result = Treap::Merge(result, L22);
346     result = Treap::Merge(result, L21);
347     result = Treap::Merge(result, edge_vu);
348   }
349
350   void Delete(int u, int v) {
351     assert(tree_edges_[u].count(v));
352     assert(tree_edges_[v].count(u));
353
354     Node* edge_uv = tree_edges_[u][v];
355     Node* edge_vu = tree_edges_[v][u];
356     tree_edges_[u].erase(v);
357     tree_edges_[v].erase(u);
358
359     int position_uv = Treap::GetPosition(edge_uv);
360     int position_vu = Treap::GetPosition(edge_vu);
361     if (position_uv > position_vu) {
362       std::swap(edge_uv, edge_vu);
363       std::swap(position_uv, position_vu);
364     }
365
366     auto p1 = Treap::SplitUp3(edge_uv);
367     auto L1 = std::get<0>(p1), uv = std::get<1>(p1);
368     assert(GetSize(L1) == position_uv - 1);
```

```
    assert(GetSize(uv) == 1);

    auto p2 = Treap::SplitUp3(edge_vu);
    auto L2 = std::get<0>(p2), vu = std::get<1>(p2), L3 = std::get<2>(p2);
    assert(GetSize(L2) == position_vu - position_uv - 1);
    assert(GetSize(vu) == 1);

    L1 = Treap::Merge(L1, L3);

    FreeNode(edge_uv);
    FreeNode(edge_vu);
  }

  bool IsConnected(int u, int v) {
    Node* vertex_u = vertices_[u];
    Node* vertex_v = vertices_[v];
    return FindRoot(vertex_u) == FindRoot(vertex_v);
  }

  std::string to_string() const {
    std::stringstream ss;

    ss << "DynamicForest [\n";

    std::function<void(Node*)> dfs = [&](Node* p) {
      if (!p) return;
      dfs(p->left_);
      ss << "(" << p->from_ << "," << p->to_ << "),";
      dfs(p->right_);
    };
    for (int i = 0; i < n_; ++i) {
      if (vertices_[i]->parent_ == nullptr) {
        ss << "  Component [";
        dfs(vertices_[i]);
        ss << "]\n";
      }
    }
    for (int i = 0; i < n_; ++i) {
      for (auto p : tree_edges_[i]) {
        if (p.second->parent_ == nullptr) {
          ss << "  Component [";
          dfs(p.second);
          ss << "]\n";
        }
      }
    }

    ss << "]\n\n";

    return ss.str();
  }

```

```
 private:
  int n_;
  std::vector<Node*> vertices_;
  std::vector<std::map<int, Node*>> tree_edges_;
};

std::mt19937 DynamicForest::rng_(std::random_device{}());

void solve_case(int Case) {
  int n, m;
  std::cin >> n >> m;

  DynamicForest t(n + 1);
  std::string op;
  int u, v;
  for (int i = 1; i <= m; ++i) {
    std::cin >> op;
    std::cin >> u >> v;
    if (op[0] == 'Q') {
      std::cout << (t.IsConnected(u, v) ? "Yes" : "No") << "\n";
    } else if (op[0] == 'C') {
      t.Insert(u, v);
    } else if (op[0] == 'D') {
      t.Delete(u, v);
    }
  }
}

```

维护子树信息

下面以子树节点数量为例进行说明.

对于 ETR(T) 中每一个元素，如果这个元素对应的是树中的点，则令其权值为 1；如果这个元素对应的是树中的边，则令其权值为 0. 现在树 T 的节点数量就可以看成 ETR(T) 中元素的权值和，只需要再维护序列权值和即可实现维护子树节点数量. 而序列权值和的维护是非旋 Treap 的经典操作了.

类似地，可以将子树最小值等操作转化成序列最小值等平衡树经典操作然后维护.

例题 LOJ #2230.「BJOI2014」大融合[2]

参考代码

```
#include <cassert>
#include <cstdint>
#include <functional>
#include <iostream>
#include <map>
#include <random>
#include <sstream>

using i64 = int64_t;
```

```cpp
using u64 = uint64_t;

void solve_case(int Case);

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int T = 1;
  // std::cin >> T;
  for (int t = 1; t <= T; ++t) {
    solve_case(t);
  }
  return 0;
}

/**
 * Dynamic Forest Maintained With Euler Tour Tree.
 *
 * As said in reference, link and cut operation of dynamic trees can be
 * transformed into sequence split and sequence merge operation, which can be
 * easily maintained using balanced search trees like Treap.
 *
 * @reference: Dynamic trees as search trees via euler tours, applied to the
 * network simplex algorithm by Robert E. Tarjan.
 * https://link.springer.com/article/10.1007/BF02614369
 */
class DynamicForest {
 private:
  static std::mt19937 rng_;

  struct Node {
    int size_;
    int priority_;

    Node* left_;
    Node* right_;
    Node* parent_;

    int from_;
    int to_;

    int num_vertex_;
    int num_edge_;

    Node(int from, int to)
        : size_(1),
          priority_(rng_()),
          left_(nullptr),
          right_(nullptr),
          parent_(nullptr),
          from_(from),
          to_(to),
```

```
61            num_vertex_(from_ == to_ ? 1 : 0),
62            num_edge_(from_ == to_ ? 0 : 1) {}
63
64      void Maintain() {
65        size_ = 1;
66        num_vertex_ = from_ == to_ ? 1 : 0;
67        num_edge_ = from_ == to_ ? 0 : 1;
68        if (left_) {
69          size_ += left_->size_;
70          left_->parent_ = this;
71
72          num_vertex_ += left_->num_vertex_;
73          num_edge_ += left_->num_edge_;
74        }
75        if (right_) {
76          size_ += right_->size_;
77          right_->parent_ = this;
78
79          num_vertex_ += right_->num_vertex_;
80          num_edge_ += right_->num_edge_;
81        }
82      }
83    };
84
85    static int GetSize(Node* p) { return p == nullptr ? 0 : p->size_; }
86
87    static Node* FindRoot(Node* p) {
88      if (!p) return nullptr;
89
90      while (p->parent_ != nullptr) p = p->parent_;
91      return p;
92    }
93
94    static std::string to_string(Node* p) {
95      std::stringstream ss;
96
97      ss << "Node [\n";
98
99      std::function<void(Node*)> dfs = [&](Node* p) {
100       if (!p) return;
101       dfs(p->left_);
102       ss << "(" << p->from_ << "," << p->to_ << "),";
103       dfs(p->right_);
104     };
105     dfs(p);
106
107     ss << "]\n\n";
108
109     return ss.str();
110   }
111
112   Node* AllocateNode(int u, int v) {
```

```
113     Node* p = new Node(u, v);
114     return p;
115   }
116
117   void FreeNode(Node*& p) {
118     if (p) {
119       delete p;
120       p = nullptr;
121     }
122   }
123
124   /*
125    * Dynamic Sequence Maintained using Treap.
126    */
127   class Treap {
128    public:
129     /**
130      * Merge two treap a and b into a single treap, with keys in a less than
131      * keys in b.
132      *
133      * In the other word, concating sequence a and sequence b.
134      */
135     static Node* Merge(Node* a, Node* b) {
136       if (a == nullptr) return b;
137       if (b == nullptr) return a;
138
139       if (a->priority_ < b->priority_) {
140         a->right_ = Merge(a->right_, b);
141         a->Maintain();
142         return a;
143       } else {
144         b->left_ = Merge(a, b->left_);
145         b->Maintain();
146         return b;
147       }
148     }
149
150     /**
151      * Get the number of nodes with keys less than or equal to the key of p.
152      *
153      * In the other word, the 1-based index of p inside the sequence
154      * containing p.
155      */
156     static int GetPosition(Node* p) {
157       assert(p != nullptr);
158
159       int position = GetSize(p->left_) + 1;
160       while (p) {
161         if (p->parent_ && p == p->parent_->right_)
162           position += GetSize(p->parent_->left_) + 1;
163         p = p->parent_;
164       }
```

```
165       return position;
166     }
167
168     /**
169      * Split sequence containning p into two sequences, the first one
    contains
170      * the first k elements, the second one contains the remaining elements.
171      */
172     static std::pair<Node*, Node*> Split(Node* p, int k) {
173       if (!p) return {nullptr, nullptr};
174
175       std::pair<Node*, Node*> result;
176
177       if (GetSize(p->left_) < k) {
178         auto right_result = Split(p->right_, k - GetSize(p->left_) - 1);
179         p->right_ = right_result.first;
180
181         result.first = p;
182         result.second = right_result.second;
183       } else {
184         auto left_result = Split(p->left_, k);
185         p->left_ = left_result.second;
186
187         result.first = left_result.first;
188         result.second = p;
189       }
190
191       p->Maintain();
192
193       if (result.first) result.first->parent_ = nullptr;
194
195       if (result.second) result.second->parent_ = nullptr;
196
197       return result;
198     }
199
200     /*
201      * Bottom up split treap p into 2 treaps a and b.
202      *   - a: a treap containing nodes with position less than or equal
    to p.
203      *   - b: a treap containing nodes with postion greater than p.
204      *
205      * In the other word, split sequence containning p into two sequences,
    the
206      * first one contains elements before p and element p, the second one
207      * contains elements after p.
208      */
209     static std::pair<Node*, Node*> SplitUp2(Node* p) {
210       assert(p != nullptr);
211
212       Node *a = nullptr, *b = nullptr;
213       b = p->right_;
```

```
214       if (b) b->parent_ = nullptr;
215       p->right_ = nullptr;
216
217       bool is_p_left_child_of_parent = false;
218       bool is_from_left_child = false;
219       while (p) {
220         Node* parent = p->parent_;
221
222         if (parent) {
223           is_p_left_child_of_parent = (parent->left_ == p);
224           if (is_p_left_child_of_parent) {
225             parent->left_ = nullptr;
226           } else {
227             parent->right_ = nullptr;
228           }
229           p->parent_ = nullptr;
230         }
231
232         if (!is_from_left_child) {
233           a = Merge(p, a);
234         } else {
235           b = Merge(b, p);
236         }
237
238         is_from_left_child = is_p_left_child_of_parent;
239         p->Maintain();
240         p = parent;
241       }
242
243       return {a, b};
244     }
245
246     /*
247      * Bottom up split treap p into 3 treaps a, b and c.
248      *   - a: a treap containing nodes with key less than p.
249      *   - b: a treap containing nodes with key greater than p.
250      *   - c: a treap containing nodes with key equal p.
251      *
252      * In the other word, split sequence containning p into three sequences,
    the
253      * first one contains elements before p, the second one contains element
    p,
254      * the third one contains elements after p.
255      */
256     static std::tuple<Node*, Node*, Node*> SplitUp3(Node* p) {
257       assert(p != nullptr);
258
259       Node* a = p->left_;
260       if (a) a->parent_ = nullptr;
261       p->left_ = nullptr;
262
263       Node* b = p->right_;
```

```
264       if (b) b->parent_ = nullptr;
265       p->right_ = nullptr;
266
267       Node* c = p;
268
269       bool is_p_left_child_of_parent = false;
270       bool is_from_left_child = false;
271       Node* parent = p->parent_;
272       if (parent) {
273         is_p_left_child_of_parent = (parent->left_ == p);
274         if (is_p_left_child_of_parent) {
275           parent->left_ = nullptr;
276         } else {
277           parent->right_ = nullptr;
278         }
279         p->parent_ = nullptr;
280       }
281       is_from_left_child = is_p_left_child_of_parent;
282       p->Maintain();
283       p = parent;
284
285       while (p) {
286         Node* parent = p->parent_;
287
288         if (parent) {
289           is_p_left_child_of_parent = (parent->left_ == p);
290           if (is_p_left_child_of_parent) {
291             parent->left_ = nullptr;
292           } else {
293             parent->right_ = nullptr;
294           }
295           p->parent_ = nullptr;
296         }
297
298         if (!is_from_left_child) {
299           a = Merge(p, a);
300         } else {
301           b = Merge(b, p);
302         }
303
304         is_from_left_child = is_p_left_child_of_parent;
305         p->Maintain();
306         p = parent;
307       }
308
309       return {a, c, b};
310     }
311   };
312
313 public:
314   DynamicForest(int n) : n_(n), vertices_(n_), tree_edges_(n_) {
315     assert(n_ > 0);
```

```
316
317     for (int i = 0; i < n_; ++i) vertices_[i] = AllocateNode(i, i);
318   }
319
320   ~DynamicForest() {
321     for (int i = 0; i < n_; ++i) {
322       for (auto p : tree_edges_[i]) {
323         FreeNode(p.second);
324       }
325     }
326     for (int i = 0; i < n_; ++i) {
327       FreeNode(vertices_[i]);
328     }
329   }
330
331   void MakeRoot(int u) {
332     Node* vertex_u = vertices_[u];
333
334     int position_u = Treap::GetPosition(vertex_u);
335
336     auto p1 = Treap::SplitUp2(vertex_u);
337     auto L1 = p1.first, L2 = p1.second;
338     assert(GetSize(L1) == position_u);
339
340     Treap::Merge(L2, L1);
341   }
342
343   void Insert(int u, int v) {
344     assert(!tree_edges_[u].count(v));
345     assert(!tree_edges_[v].count(u));
346
347     Node* vertex_u = vertices_[u];
348     Node* vertex_v = vertices_[v];
349
350     Node* edge_uv = AllocateNode(u, v);
351     Node* edge_vu = AllocateNode(v, u);
352     tree_edges_[u][v] = edge_uv;
353     tree_edges_[v][u] = edge_vu;
354
355     int position_u = Treap::GetPosition(vertex_u);
356     int position_v = Treap::GetPosition(vertex_v);
357
358     auto p1 = Treap::SplitUp2(vertex_u);
359     auto p2 = Treap::SplitUp2(vertex_v);
360     auto L11 = p1.first, L12 = p1.second;
361     auto L21 = p2.first, L22 = p2.second;
362
363     assert(GetSize(L11) == position_u);
364     assert(GetSize(L21) == position_v);
365
366     Node* result = nullptr;
367     result = Treap::Merge(result, L12);
```

```
368     result = Treap::Merge(result, L11);
369     result = Treap::Merge(result, edge_uv);
370     result = Treap::Merge(result, L22);
371     result = Treap::Merge(result, L21);
372     result = Treap::Merge(result, edge_vu);
373   }
374
375   void Delete(int u, int v) {
376     assert(tree_edges_[u].count(v));
377     assert(tree_edges_[v].count(u));
378
379     Node* edge_uv = tree_edges_[u][v];
380     Node* edge_vu = tree_edges_[v][u];
381     tree_edges_[u].erase(v);
382     tree_edges_[v].erase(u);
383
384     int position_uv = Treap::GetPosition(edge_uv);
385     int position_vu = Treap::GetPosition(edge_vu);
386     if (position_uv > position_vu) {
387       std::swap(edge_uv, edge_vu);
388       std::swap(position_uv, position_vu);
389     }
390
391     auto p1 = Treap::SplitUp3(edge_uv);
392     auto L1 = std::get<0>(p1), uv = std::get<1>(p1);
393     assert(GetSize(L1) == position_uv - 1);
394     assert(GetSize(uv) == 1);
395
396     auto p2 = Treap::SplitUp3(edge_vu);
397     auto L2 = std::get<0>(p2), vu = std::get<1>(p2), L3 = std::get<2>(p2);
398     assert(GetSize(L2) == position_vu - position_uv - 1);
399     assert(GetSize(vu) == 1);
400
401     L1 = Treap::Merge(L1, L3);
402
403     FreeNode(edge_uv);
404     FreeNode(edge_vu);
405   }
406
407   bool IsConnected(int u, int v) {
408     Node* vertex_u = vertices_[u];
409     Node* vertex_v = vertices_[v];
410     return FindRoot(vertex_u) == FindRoot(vertex_v);
411   }
412
413   int GetComponentSize(int u) {
414     Node* vertex_u = vertices_[u];
415     Node* root_of_vertex_u = FindRoot(vertex_u);
416     return GetSize(root_of_vertex_u);
417   }
418
419   int GetComponentNumberOfVertex(int u) {
```

```
    Node* vertex_u = vertices_[u];
    Node* root_of_vertex_u = FindRoot(vertex_u);
    return root_of_vertex_u ? root_of_vertex_u->num_vertex_ : 0;
  }

  std::string to_string() const {
    std::stringstream ss;

    ss << "DynamicForest [\n";

    std::function<void(Node*)> dfs = [&](Node* p) {
      if (!p) return;
      dfs(p->left_);
      ss << "(" << p->from_ << "," << p->to_ << "),";
      dfs(p->right_);
    };
    for (int i = 0; i < n_; ++i) {
      if (vertices_[i]->parent_ == nullptr) {
        ss << "  Component [";
        dfs(vertices_[i]);
        ss << "]\n";
      }
    }
    for (int i = 0; i < n_; ++i) {
      for (auto p : tree_edges_[i]) {
        if (p.second->parent_ == nullptr) {
          ss << "  Component [";
          dfs(p.second);
          ss << "]\n";
        }
      }
    }

    ss << "]\n\n";

    return ss.str();
  }

 private:
  int n_;
  std::vector<Node*> vertices_;
  std::vector<std::map<int, Node*>> tree_edges_;
};

std::mt19937 DynamicForest::rng_(std::random_device{}());

void solve_case(int Case) {
  int n, q;
  std::cin >> n >> q;

  DynamicForest t(n + 1);
  std::string op;
```

```
  int u, v;
  for (int i = 1; i <= q; ++i) {
    std::cin >> op >> u >> v;
    if (op[0] == 'A') {
      t.Insert(u, v);
    } else if (op[0] == 'Q') {
      t.Delete(u, v);
      int ans = i64(1) * t.GetComponentNumberOfVertex(u) *
                t.GetComponentNumberOfVertex(v);
      t.Insert(u, v);
      std::cout << ans << "\n";
    }
  }
}

```

维护树链信息

可以使用一个比较常见的技巧就是借助括号序的性质将树链信息转化成区间信息，然后就可以借助数据结构维护序列从而维护树链信息了．但是这个技巧要求维护的信息满足 **可减性**.

前面介绍的动态树操作对应的序列操作可能会把括号序中的右括号移动到左括号前，所以维护树链点权和之类的信息时还需要额外注意，操作时不能改变对应左右括号的先后顺序，而这可能需要重新思考动态树操作对应的序列操作，甚至重新思考维护什么 DFS 序.

此外，ETT 很难维护树链修改.

例题 「星际探索」[3]

这题的动态树操作只有换父亲，可以看成删边再加边，但是这样可能会改变对应括号的先后顺序.

可以把点权转成边权，维护树的括号序，换父亲操作转化成把整个子树对应的括号序列平移至父亲左括号后面.

参考代码

```
/*
虽然上文提到过块状链表实现 ETT
在某些情况下可能较简单，但对于此题块状链表复杂度有可能无法通过而且实现较繁琐，所以这份代码采用
FHQ Treap 实现。
*/
#include <iostream>
#include <vector>
constexpr int N = 1000000;
using namespace std;
/*FHQ TREAP*/
long long rt, tot, f[N], rnd[N], ls[N], rs[N], siz[N], tag[N], val[N], sum[N],
    pd[N], pds[N];

void pushup(long long x) {
  siz[x] = siz[ls[x]] + siz[rs[x]] + 1;
```

```
  sum[x] = sum[ls[x]] + sum[rs[x]] + val[x];
  pds[x] = pds[ls[x]] + pds[rs[x]] + pd[x];
}

void link(long long x, long long c, long long y) {
  if (c)
    rs[x] = y;
  else
    ls[x] = y;
  if (y) f[y] = x;
  pushup(x);
}

long long newNode(long long x, long long y) {
  siz[++tot] = 1;
  val[tot] = sum[tot] = x;
  pd[tot] = pds[tot] = y;
  rnd[tot] = rand();
  return tot;
}

void setTag(long long x, long long v) {
  tag[x] += v;
  sum[x] += v * pds[x];
  val[x] += v * pd[x];
}

void pushdown(long long x) {
  if (ls[x]) setTag(ls[x], tag[x]);
  if (rs[x]) setTag(rs[x], tag[x]);
  tag[x] = 0;
}

void split(long long now, long long k, long long &x, long long &y) {
  f[now] = 0;
  if (!now) {
    x = y = 0;
    return;
  }
  pushdown(now);
  if (siz[ls[now]] + 1 <= k) {
    x = now;
    split(rs[now], k - siz[ls[now]] - 1, rs[x], y);
    link(x, 1, rs[x]);
  } else {
    y = now;
    split(ls[now], k, x, ls[y]);
    link(y, 0, ls[y]);
  }
}

long long merge(long long x, long long y) {
```

```
  if (!x || !y) return x | y;
  if (rnd[x] < rnd[y]) {
    pushdown(x);
    link(x, 1, merge(rs[x], y));
    return x;
  } else {
    pushdown(y);
    link(y, 0, merge(x, ls[y]));
    return y;
  }
}

long long rnk(long long x) {
  long long c = 1, ans = 0;
  while (x) {
    if (c) ans += siz[ls[x]] + 1;
    c = (rs[f[x]] == x);
    x = f[x];
  }
  return ans;
}

/*ETT*/
long long s[N], e[N];

void add(long long x, long long v) {
  long long a, b, c;
  split(rt, rnk(s[x]) - 1, a, b);
  split(b, rnk(e[x]) - rnk(s[x]) + 1, b,
        c);  // 这里 b 是我们要进行操作的子树的括号序列。
  setTag(b, v);
  rt = merge(merge(a, b), c);
}

long long query(long long x) {
  long long a, b;
  split(rt, rnk(s[x]), a, b);
  long long ans = sum[a];
  rt = merge(a, b);
  return ans;
}

void changeFa(long long x, long long y) {
  long long a, b, c, d;
  split(rt, rnk(s[x]) - 1, a, b);
  split(b, rnk(e[x]) - rnk(s[x]) + 1, b, c);
  a = merge(
      a,
      c);  // 因为我们确定不了要设置为父亲的节点在括号序列中的哪边，所以先把两边合并。
  split(a, rnk(s[y]), a, d);
  rt = merge(merge(a, b), d);  // 把要进行操作的子树放在父亲括号序列的最前面。
}
```

```
120
121 /*main function*/
122 long long n, m, w[N];
123 vector<long long> v[N];
124
125 void dfs(long long x) {
126   rt = merge(rt, s[x] = newNode(w[x], 1));
127   for (auto to : v[x]) dfs(to);
128   rt = merge(rt, e[x] = newNode(-w[x], -1));
129 }
130
131 signed main() {
132   cin >> n;
133   for (long long i = 2; i <= n; i++) {
134     long long f;
135     cin >> f;
136     v[f].push_back(i);
137   }
138   for (long long i = 1; i <= n; i++) cin >> w[i];
139   dfs(1);
140   cin >> m;
141   for (long long i = 1; i <= m; i++) {
142     char c;
143     cin >> c;
144     if (c == 'Q') {
145       long long d;
146       cin >> d;
147       cout << query(d) << endl;
148     } else if (c == 'C') {
149       long long x, y;
150       cin >> x >> y;
151       changeFa(x, y);
152     } else {
153       long long p, q;
154       cin >> p >> q;
155       add(p, q);
156     }
157   }
158   return 0;
159 }
```

参考资料

- Dynamic trees as search trees via euler tours, applied to the network simplex algorithm - Robert E. Tarjan
- Randomized fully dynamic graph algorithms with polylogarithmic time per operation - Henzinger et al.

外部参考资料

[1] P2147[SDOI2008] 洞穴勘测  [2] LOJ #2230.「BJOI2014」大融合  [3] 「星际探索」 

10.20.4 Top Tree

Authors: F7487

Self-Adjusting Top Tree

简介

Self-Adjusting Top Tree，是 2005 年 Tarjan 和 Werneck 在他们的论文 Self-Adjusting Top Trees 中提出的一种基于 Top Tree 理论的维护完全动态森林的数据结构，简称为 SATT.

Self-Adjusting Top Tree 可以实现森林中任一棵树的链修改/查询、子树修改/查询以及非局部搜索等操作.

Splay Tree 是 SATT 的基础，但是 SATT 用的 Splay Tree 和普通的 Splay 在细节处不太一样（进行了一些扩展）.

问题引入

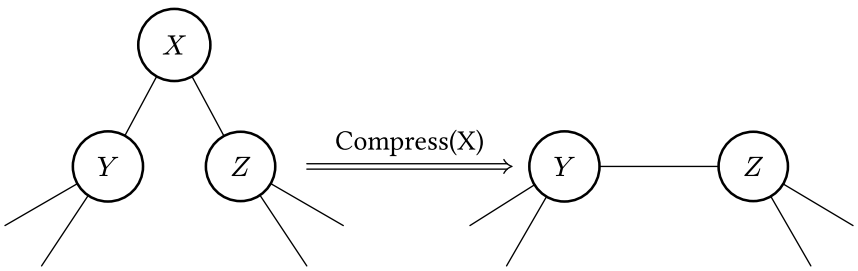
维护一个森林，支持如下操作：

- 删除，添加一条边，保证操作前后仍是一个森林.
- 修改某棵树上某条简单路径的权值.
- 修改以某个点为根的子树权值.
- 查询某棵树上的某条简单路径权值和.
- 查询以某个点为根的子树权值和.

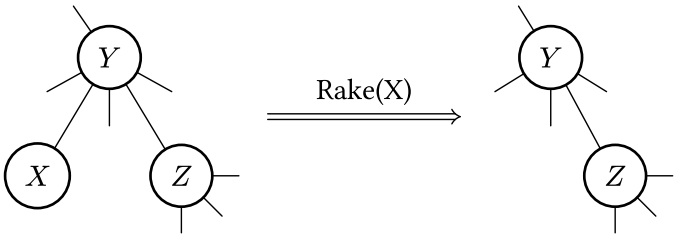
树收缩

对于任意一棵树，我们都可以运用 **树收缩** 理论来将它收缩为一条边.

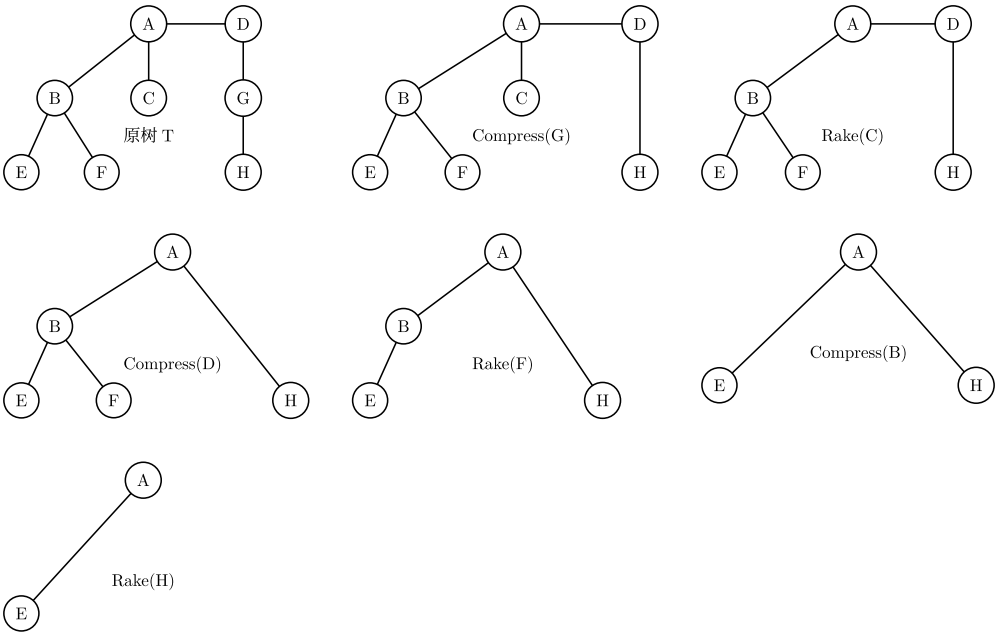
具体地，树收缩有两个基本操作：**Compress** 和 **Rake**，Compress 操作指定一个度数为 2 的点 x，与点 x 相邻的那两个点记为 y、z，我们连一条新边 yz；将点 x、边 xz、边 xy 的信息放到 yz 中储存，并删去它们. 如图所示.



Rake 操作指定一个度为 1 的点 x，而且与点 x 相邻的点 y 的度数需大于 1，设点 y 的另一个邻点为 z，我们将点 x、边 xy 的信息放入边 yz 中储存，并删去它们. 如图所示.



不难证明，任何一棵树都可以只用 Compress 操作和 Rake 操作来将它收缩为一条边，如图所示.

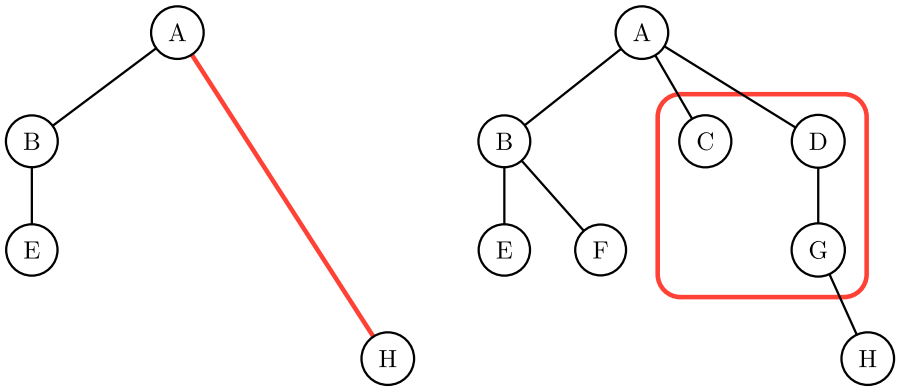


簇

为了表达方便，我们记在进行任何操作之前的原树为 T. 在对 T 进行某些树收缩操作（可以不做任何操作）之后的树记为 T_x.

我们研究某个 T_x 中某一条边所包含的信息情况.

这条边除了带有它本身的信息（当然，如果这条边在 T 中不存在，这条边就没有本身的信息）之外，还可能包含其它通过 Compress/Rake 操作合并到它上面的点、边的信息．我们不妨先从下图中的树收缩过程中选取一条边，看看它所包含的信息在 T 中代表哪些点、边．



如图，选取的边和对应的图已用红线圈出．

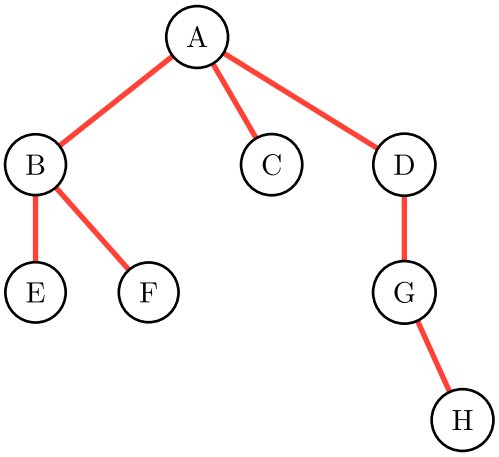
可以看出，这条边所包含的信息在 T 中代表的点、边是连通的．我们可以推及，对于任一 T_x 中的任一条边储存的信息在 T 中总体现为一个连通子图．我们将这样的连通子图称为 **簇（Cluster）**．

然而，簇是 **不完整的子图**，它包含的某些边的端点不被簇它自己包含．于是我们将这些端点称作簇的 **端点（Endpoint）**，将它包含的那些连通子图的点称作 **内点（Internal Node）**，连通子图的边称作 **内边（Internal Edge）**．

对于任意一个簇，都有以下性质：

1. 簇只存储和维护内点和内边的信息．
2. 簇有两个端点．这两个端点即为 T_x 中代表那个簇的边相连的那两个点．两个端点之间的路径我们称之为 **簇路径（Cluster Path）**；记一个簇的两个端点分别为 x、y，我们下面用 $C(x,y)$ 来表示这个簇．
3. 内点仅与端点或内点相连．

特别地，对于 T 中的每条边，都各自独立为一个簇（仅包含边自己的信息），这种簇我们称之为 **基簇（Base Cluster）**．对于由 T 收缩到只有一条边的最终的 T_x，那条边代表的簇包含除了两个端点之外的整棵 T 的信息，这个簇我们称之为 **根簇（Root Cluster）**．

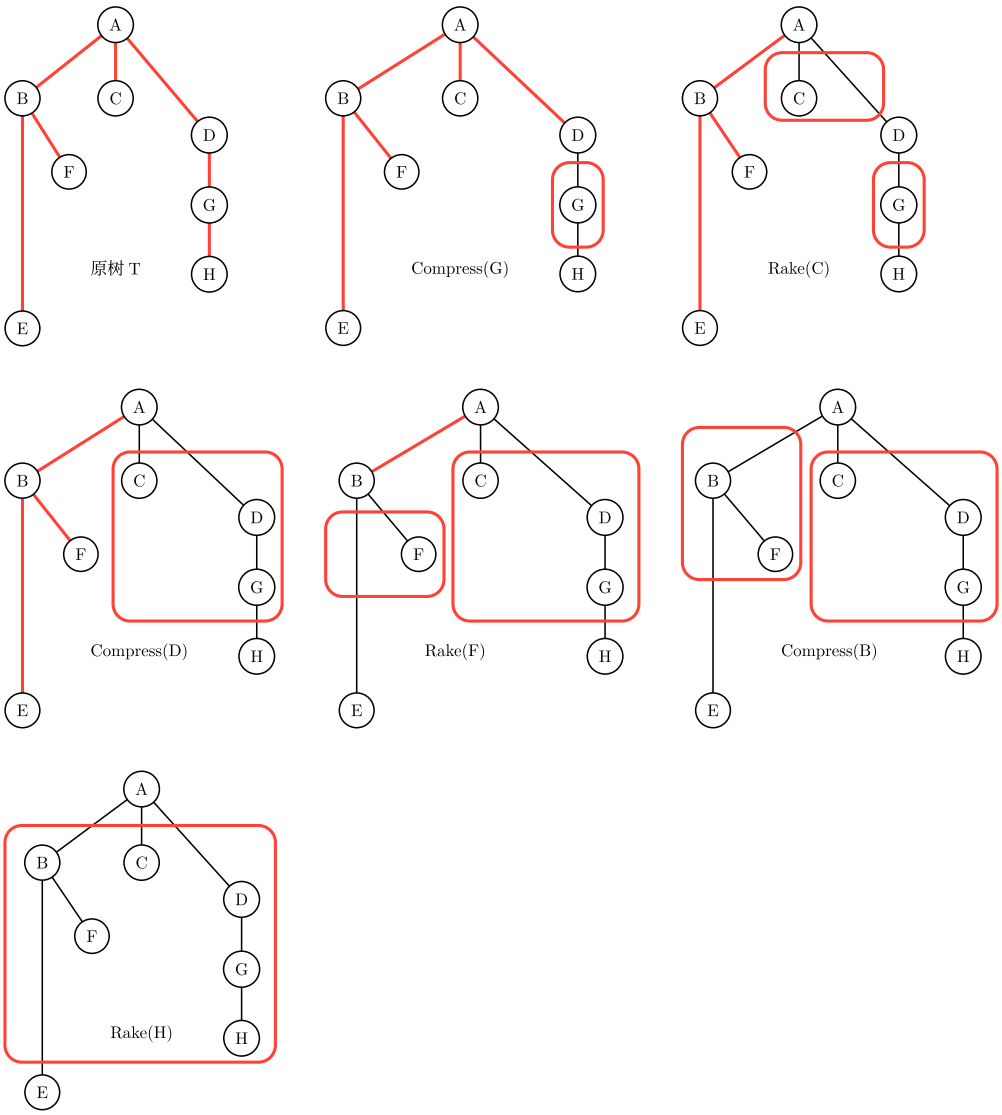


原树 T

如图，上文提到的基簇已用红线标出.

从簇的视角来看 Compress/Rake 操作，我们发现这两个操作会将两个簇「合二为一」，剩下一个新簇，所以树收缩的过程也是所有的基簇合并为一个簇的过程.

所以我们也可以得到下图，是对一系列树收缩操作的另一表示.

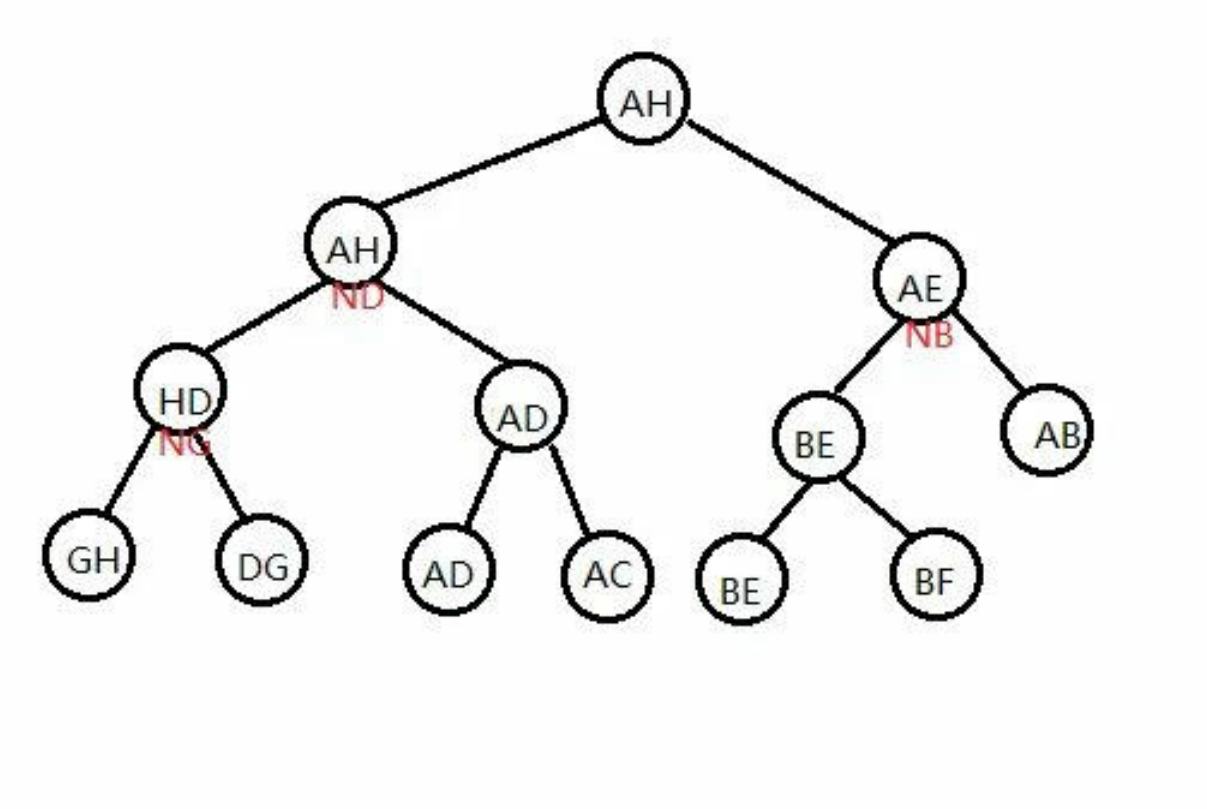


Top Tree

我们现在想表示某一棵树进行树收缩的全过程.

我们可以用上文的两种方法来表示这一过程，但这样十分麻烦，如果树收缩进行了 n 步，我们就要用 n 棵树来表示整个树收缩.

考虑一个对某棵树进行某一树收缩的更简便表示，我们引入 **Top Tree**.



如图，是以上文的收缩方法和原树为基础的一棵 Top Tree.

Top Tree 有以下性质；

1. 一棵 Top Tree 对应一棵原树和一种对其进行树收缩的方法，Top Tree 的每个节点都表示在某个 T_x 中的某一条边，也就是树收缩过程中形成的某一个簇．图中的形如 N_x 的点表示 `compress(x)` 这一操作形成的簇.
2. Top Tree 中的一个节点有两个儿子（都分别代表一个簇），这个节点代表的簇是这两个簇通过 Compress 或 Rake 操作合并得到的新簇.
3. Top Tree 的叶子节点是基簇，其根节点是根簇．因此我们按一棵 Top Tree 的拓扑序分层，它的每一层就代表了一棵 T_x.

用三度化 Self-Adjusting Top Tree 实现信息维护

原理

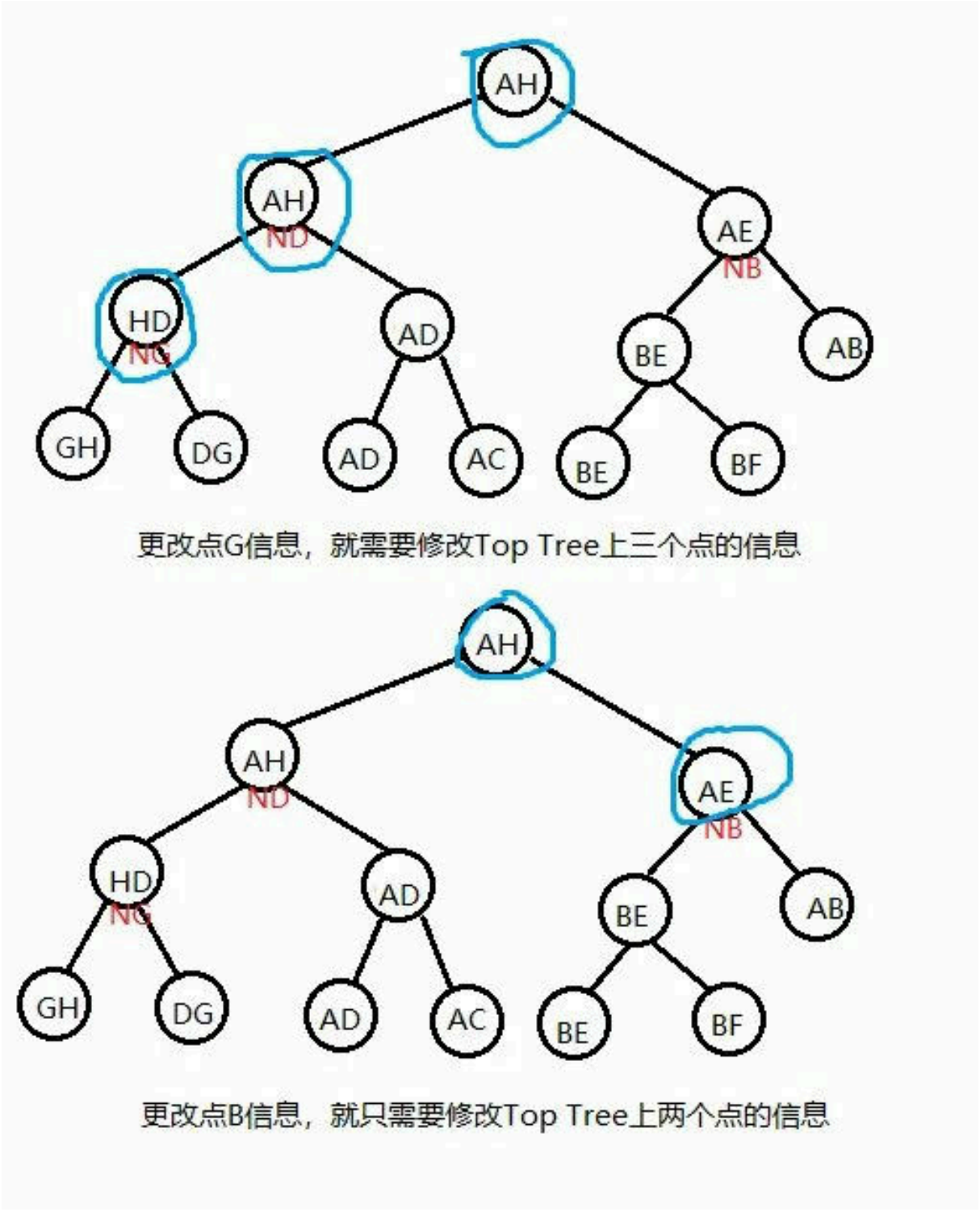
Top Tree 对树收缩过程的极大简化，使我们看到通过维护树收缩过程来维护树上信息的可能性，SATT 即是通过这一原理来维护树上信息的.

注意到树收缩的过程也是树上信息不断加入的过程，我们执行一次 `compress(x)`，x 点的信息从此刻起就开始在某个簇中出现，影响着我们的统计结果.

假如我们现在用 Top Tree 来维护某棵树 T，树上的每个点，边都有权值，我们要维护的是 T 的权值和.

现在我们在维护时要对 T 中某个点 x 的权值进行修改，很明显，我们就需要更改 Top Tree 中所有簇信息包含 x 的节点信息，这样做单次时间复杂度会是 $O(n)$ 级别的.

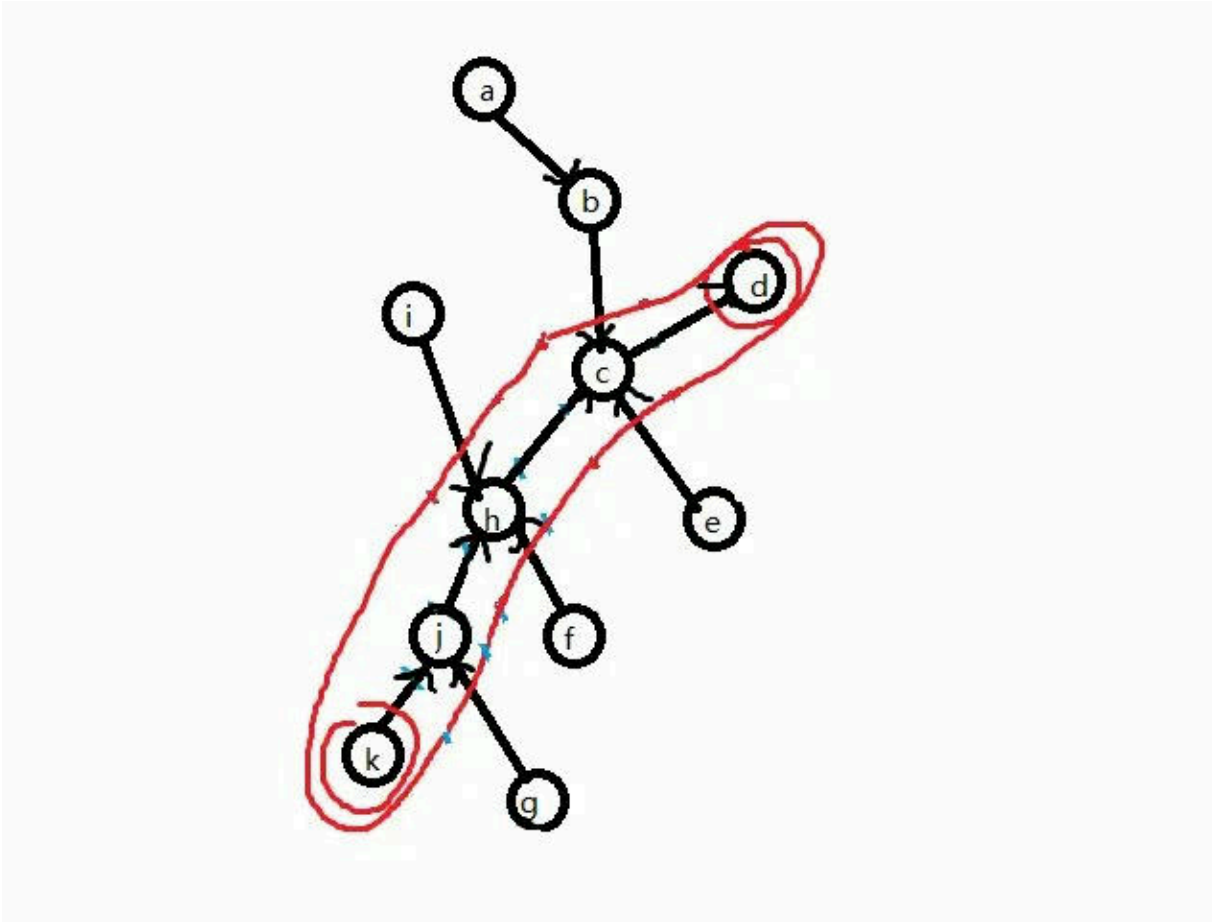
然而，如果我们选的点它在 Top Tree 中簇信息包含 x 的节点个数很少，也就是说使它的信息尽可能晚地加入簇中，我们单次操作的时间复杂度就会有一个很大的提升．如图.



SATT 就是通过修改 **某个点/某条路径** 在树收缩过程中信息被加入簇中的先后顺序（以降低其在被修改时的单次时间复杂度）来维护树上信息的.

实际结构

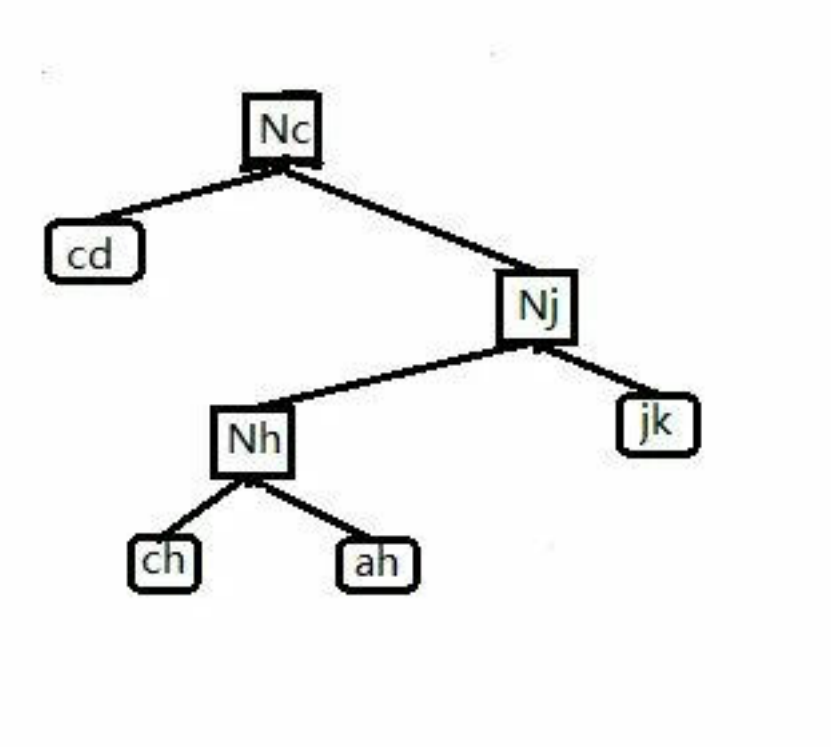
我们先将一棵原树 T 分层定根，然后我们考虑对某种树收缩顺序的 Top Tree 的根簇，它有两个端点，我们令这其中一个端点就是原树的根，另一个端点任选.



如图，给根簇选出一组端点，这里标注簇时将端点也圈进去了.

由树收缩的基本操作可知，簇路径上的点、边 (j, h, c, jh, hc) 的信息最后是通过 Compress 操作才加入 $C(k, g)$ 的，而的非簇路径点 $(a, b, i, f, g, e, ig, \cdots)$ 是通过 Rake 操作才加入 $C(k, g)$ 的.

我们将簇路径单独拿出来，这是一条形态特殊（为链）的树，我们为这棵树建出一棵 top tree（其代表的树收缩顺序任意）.

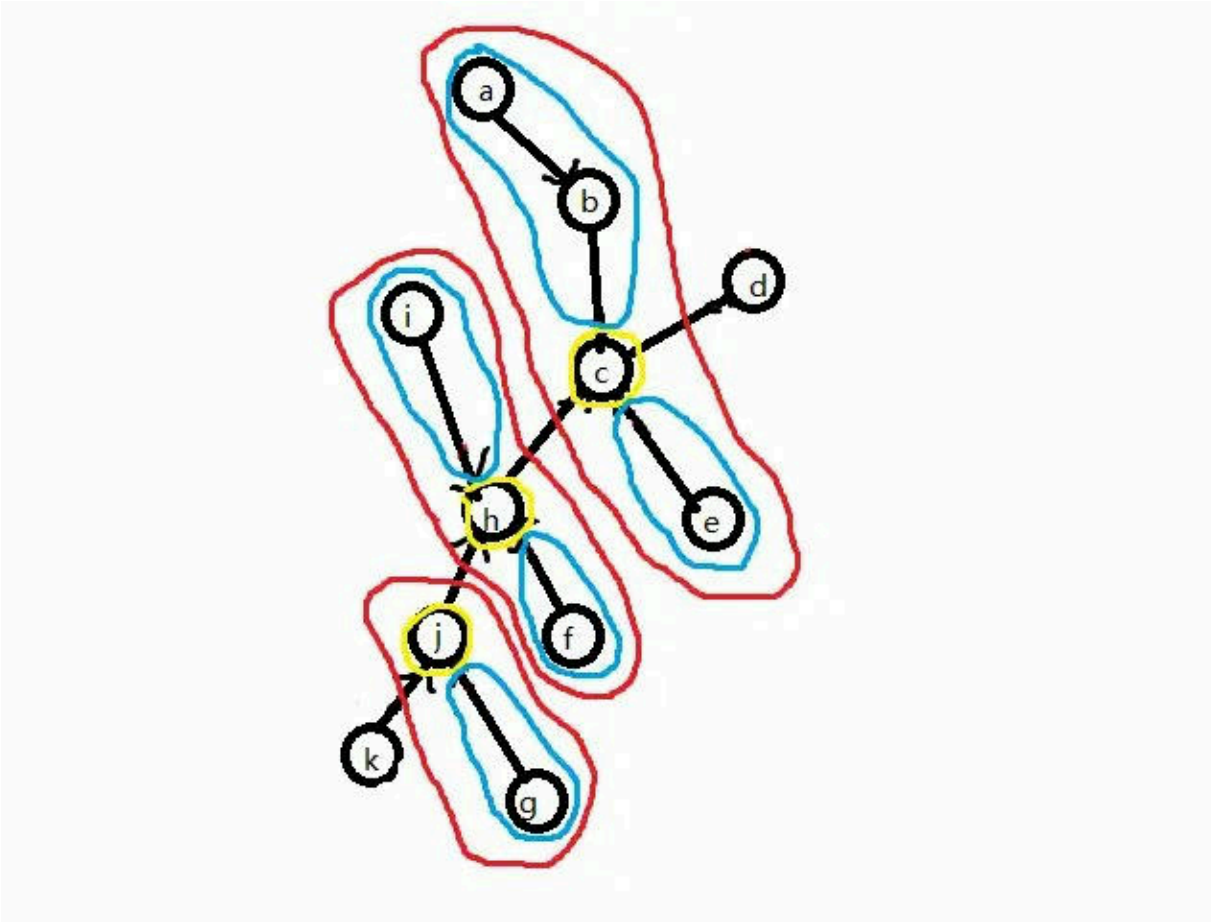


我们将这一结构称之为 **Compress Tree**，因为在这棵 Top Tree 中任一个点的两个儿子之间是通过 Compress 操作来合并成它们的父亲.

Compress Tree 里的节点称为 **Compress Node**. 只考虑当前这条簇路径，一个非叶子的 Compress Node 就代表一次 compress 过程，表示将左儿子和右儿子信息合并起来，再将这个 `compress(x)` 本身存储的点 x 信息加入. 这棵 Compress Tree 就维护了 $C(k,g)$ 簇路径的信息.

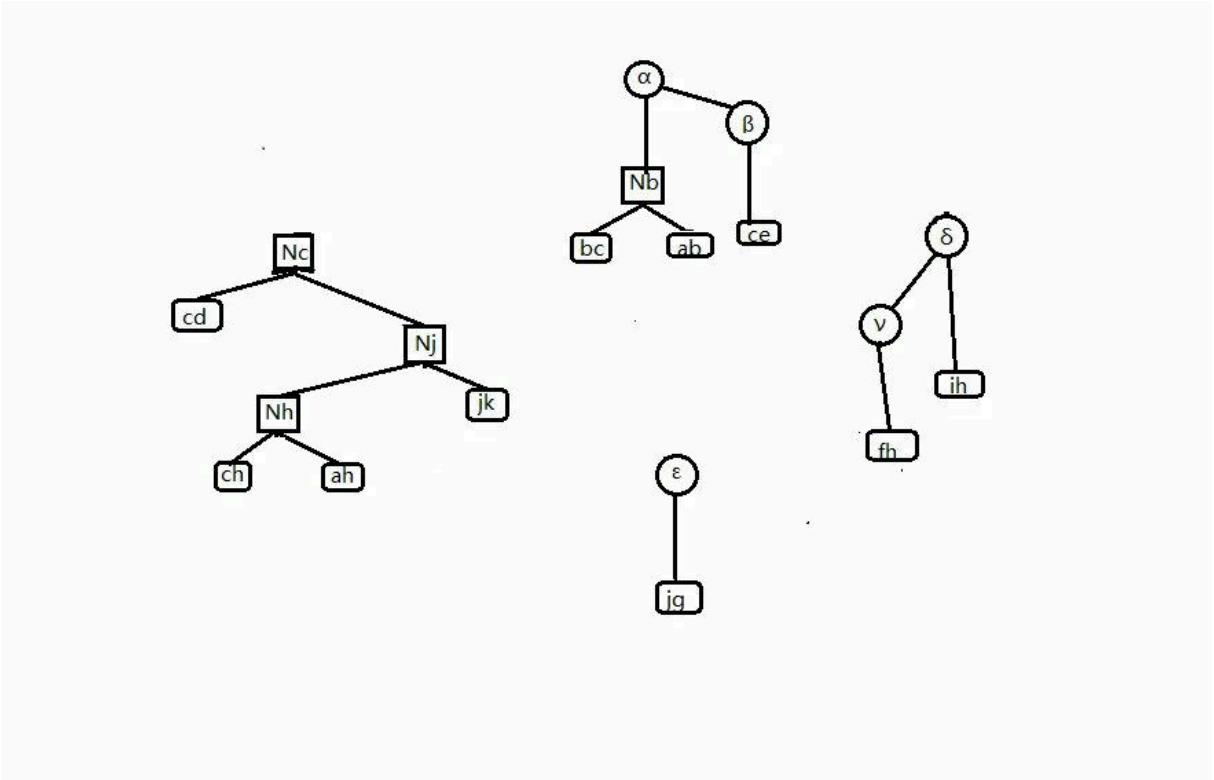
另外，在 Compress Tree 中，我们实际上还对使用的 Top Tree 做了一些限制. 注意到 Compress Tree 维护的是一个 T 中点的深度两两不同的链，我们规定在 Compress Tree 中基簇的中序遍历顺序与对应的 T 中边的深度是一致的，且中序遍历越小深度越浅. 同样，对于每个点 x 对应的 `compress(x)` 的关系也是如此.

现在来维护那些非簇路径的信息，我们假设这些非簇路径上的点、边已经形成了一个个极大簇，而这些极大簇是由这些用蓝线圈出的更小簇之间互相 Rake 形成的，对由一些更小簇合并形成一个极大簇的过程，我们用一个三叉树来表示，类似地，我们称这一结构为 **Rake Tree**，对应地 Rake Tree 里的点就是 **Rake Node**. 每个 Rake Node 都代表一个簇，是由其左儿子和右儿子 Rake 到其中儿子代表的更小簇上形成的. 具体可见下图，可知 Rake Tree 中的每个点都代表了 T 中具有相同端点的更小簇.



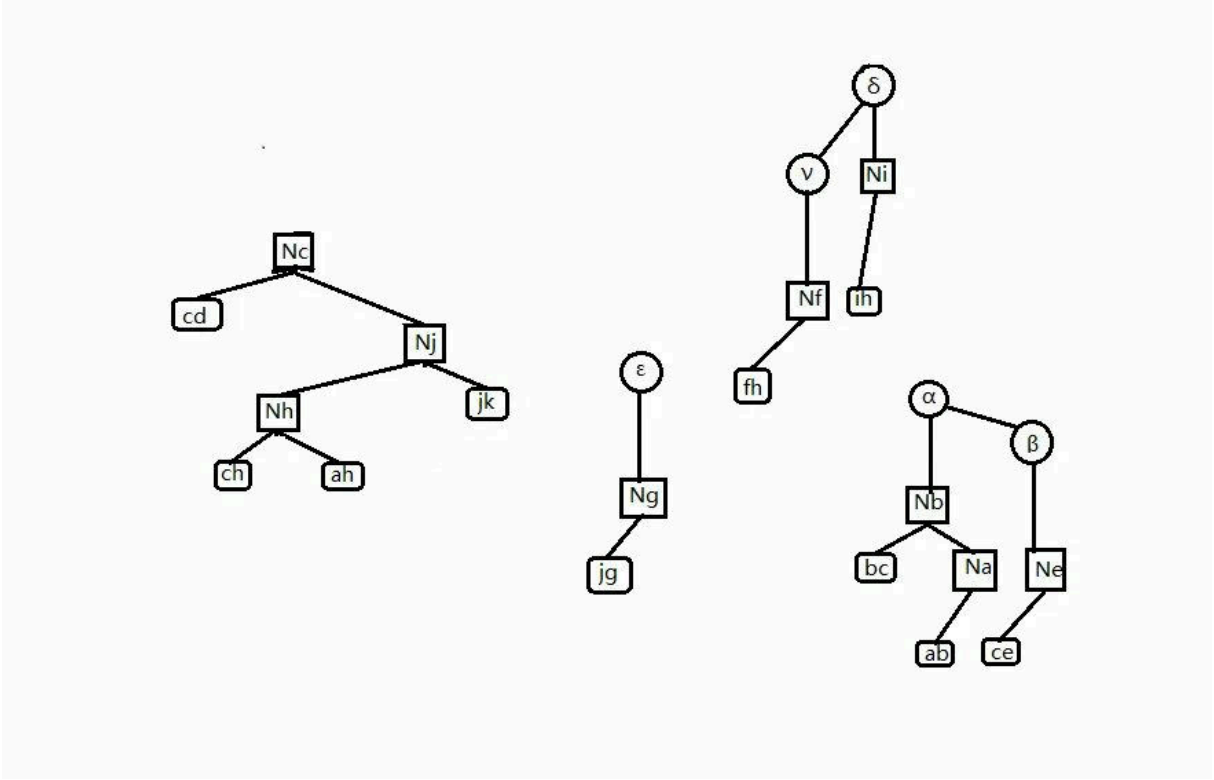
如图，蓝线圈出的是一个个极大簇，黄线圈出的是一个个更小簇.

对于那些更小簇，我们对它们进行相同处理，给它们选择簇路径、建出 Compress Tree、……如此递归下去，就建出了许多表示树收缩过程的 Compress Tree， Rake Tree.

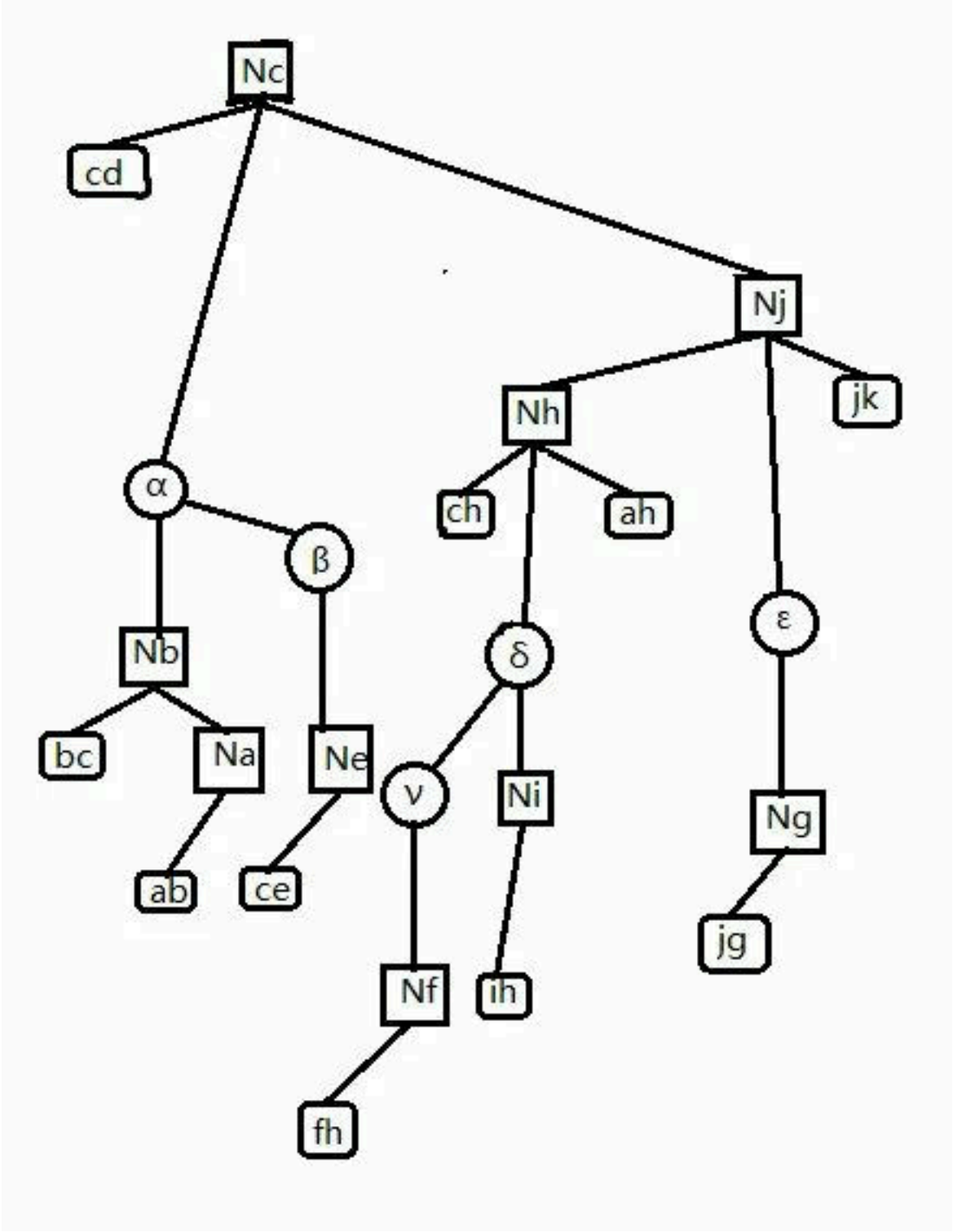


上图为原树的 Rake-Compress Tree（因为每个 Rake Node 都连着一棵 Compress Tree，所以表现为一棵 Rake Tree 连着许多 Compress Tree 的形态）和代表根簇路径的 Compress Tree.

考虑将这些树以某种方式拼接在一起，使它们形成一个有序的整体．记一个 Rake Tree 代表的最小簇的集合的公共端点是点 x．我们给这些 Rake Node 的中儿子（一个 Compress Tree 集合）都加入非 x 的另一端点，但仍保持其中序遍历和 Top Tree 的基本性质，如图.



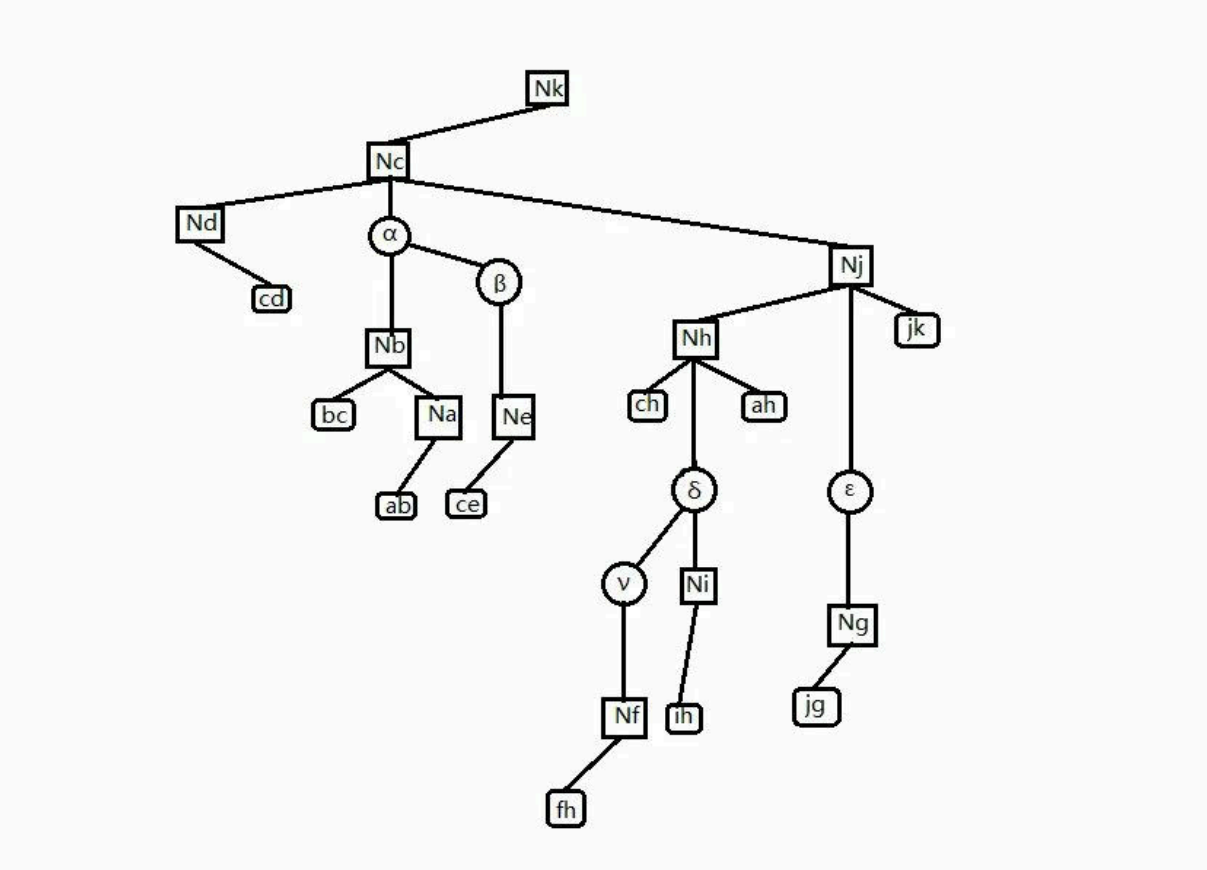
这一步相当于是让 Rake 操作加入某个 T 中点的操作直接发生在 Compress Tree 中，这不仅使我们能正确维护 Rake Node 的信息（只需将三个儿子信息合并即可），还使我们 Compress Tree 的结构更完整．下一步，我们将 Compress Tree 改为三叉树，若某个 Rake Tree 的公共端点是点 x，我们就将 Rake Tree 挂在 `compress(x)` 的中儿子处，如图.



此时经过三叉化的 `compress(x)` 点，它的意义就变成先将其中儿子 Rake 到簇路径上，再统计左右儿子和点 x 的信息.

最后，我们再处理一下根簇路径的那棵 Compress Tree：与其它所有 Compress Tree 一致地，按中序遍历加入它的两个端点，使得它的根储存整棵 T 的信息.

于是我们就实现了用三度化 Self-Adjusting Top Tree 实现一棵树的信息维护.



总结一下，SATT 有以下性质：

1. SATT 由 Compress Tree 和 Rake Tree 组成，Compress Tree 是一棵特殊的 Top Tree；Rake Tree 是一个三叉树，它们都对应一棵树进行树收缩的过程.
2. Compress Tree 里的点最多有三个儿子. Compress Tree 可以做类似于 Splay 树的旋转操作（只需保证其中序遍历不变即可，旋转一个点时保持其中儿子不动）.
3. Rake Tree 里的点一定有一个中儿子. Rake Tree 可以做类似于 Splay 树的旋转操作（只需保证其中序遍历不变即可，旋转一个点时保持其中儿子不动）.
4. SATT 的拓扑序反映了原树 T 的树收缩顺序.

我们在上文中提到的「修改某个点/某条路径在树收缩过程中信息被加入簇中的先后顺序」SATT 是否能实现呢，答案是肯定的.

在 SATT 中，有一个 `access(x)` 的操作，它的作用是使某点 x 成为根簇的非根端点，同时在 SATT 中使 `compress(x)` 成为 SATT 的根.

我们可以通过 `access(x)` 操作以均摊 $O(\log n)$ 的复杂度使 SATT 中代表 `compress(x)` 的点旋到整棵 SATT 的树根，根据 SATT 的第四个性质，我们改变了 `compress(x)` 的操作顺序，使得它最晚执行，x 点的信息也就被最晚加入；这样当我们要修改 x 点的信息时，就只需要更新 `compress(x)`.

代码实现

Push 类函数

首先考虑上传信息，即 `Pushup(x)` 函数．在考虑对 SATT 的某个节点维护信息时，首先分这个点在 Compress Tree 还是在 Rake Tree 进行讨论，原因可见上文，不再赘述，下面以维护某个点的子树大小为例

```
// ls(x) x的左儿子
// rs(x) x的右儿子
// ms(x) x的中儿子
// type==0 是 Compress Node
// type==1 是 Rake Node
void pushup(int x, int type) {
  if (type == 0)
    size[x] = size[rs(x)] + size[ms(x)] + 1;
  else
    size[x] = size[rs(x)] + size[ms(x)] + size[ls(x)];
  return;
}
```

查询点 x 的子树大小，就将其 Access 到 SATT 根，答案是其中儿子的 size $+1$；因为根据上文，在 Access 之后，其中儿子才是它的真子树．

然后考虑下传信息，即 `Pushdown(x)` 函数．我们如果要对原树中的某个子树做整体修改，一个很自然的想法是：将这个节点直接 Access 到 SATT 根节点，给它的中儿子打上一个标记即可．同理，查询子树就直接 Access 后查询中儿子．

我们如果要对原树中的某条路径做整体修改，我们就 expose 路径的两个端点，其中 `expose(x, y)` 是指使点 x 成为 T 的根节点，使点 y 成为根簇的另一个端点．对应在 SATT 上，此时根簇的 Compress Tree 就是 x 到 y 的路径．于是直接给根簇的 Compress Tree 打上一个标记即可．同理查询链 expose 后查询根节点即可．

于是我们就知道问题引入的问题怎么做了．

```
void pushdown(int x, int type) {
  if (type == 0) {
    // 处理链
    chain[ls(x)] += chain[x] chain[rs(x)] += chain[x];
    val[ls(x)] += chain[x];
    val[rs(x)] += chain[x];
    // 处理子树
    subtree[ls(x)] += subtree[x];
    subtree[rs(x)] += subtree[x];
    subtree[ms(x)] += subtree[x];
    val[ls(x)] += subtree[x];
    val[rs(x)] += subtree[x];
    val[ms(x)] += subtree[x];
    subtree[x] = 0;
  } else {
    subtree[ls(x)] += subtree[x];
    subtree[rs(x)] += subtree[x];
    subtree[ms(x)] += subtree[x];
    val[ls(x)] += subtree[x];
    val[rs(x)] += subtree[x];
    val[ms(x)] += subtree[x];
    subtree[x] = 0;
  }
```

```
  return;
}

// 下传标记
void pushall(int x, int type) {
  if (!isroot(x)) pushall(father[x], type);
  pushdown(x, type);
  return;
}
```

Splay 类函数

我们知道 SATT 中的 Rake Tree 和 Compress Tree 都是可以旋转的，也就是说它们可以用 Splay 来维护. 因此我们可以写出以下代码：

```
// 是一个节点的中儿子或无父亲
// ls 一个SATT节点的左儿子
// rs 一个SATT节点的右儿子
// ms 一个SATT节点的中儿子
// type==1 在 Rake Tree中
// type==0 在 Compress Tree中
bool isroot(int x) { return rs(father[x]) != x && ls(father[x]) != x; }

bool direction(int x) { return rs(father[x]) == x; }

void rotate(int x, int type) {
  int y = father[x], z = father[y], d = direction(x), w = son[x][d ^ 1];
  if (z) son[z][ms(z) == y ? 2 : direction(y)] = x;
  son[x][d ^ 1] = y;
  son[y][d] = w;
  if (w) father[w] = y;
  father[y] = x;
  father[x] = z;
  pushup(y, type);
  pushup(x, type);
  return;
}

void splay(int x, int type, int goal = 0) {
  pushall(x, ty);  // 下传标记
  for (int y; y = father[x], (!isroot(x)) && y != goal; rotate(x, ty)) {
    if (father[y] != goal && (!isroot(y))) {
      rotate(direction(x) ^ diretion(y) ? x : y, type);
    }
  }
  return;
}
```

值得注意的是，函数 `direction` 和 `isroot` 与普通 Splay 的不同. 因为无论这个点怎么转，这个点的中儿子是不会变的.

Access 类函数

`access(x)` 的意义是：将点 x 旋转到整个 SATT 的根处，使点 x 成为根簇的两个端点之一（另一端点即为 T 的根节点），同时不能改变原树的结构和原树的根.

为了实现 `access(x)`，我们先将其旋转到其所在 Compress Tree 的树根，再把点 x 的右儿子去掉，使点 x 成为其所在 Compress Tree 对应簇的端点.

```
if (rs(x)) {
  int y = new_node();
  setfather(ms(x), y, 0);
  setfather(rs(x), y, 2);
  rs(x) = 0;
  setfather(y, x, 2);
  pushup(y, 1);
  pushup(x, 0);
}
```

如果这时点 x 已经到了根部，则退出；若没有，则执行以下步骤，以让它跨过它上面的 Rake Tree：

1. 将其父亲节点（一定是一个 Rake Node），splay 到其 Rake Tree 的树根；
2. 将 x 的爷节点（一定是一个 Compress Node）splay 到其 Compress Tree 根部.
3. 若 x 的爷节点有一个右儿子，则将点 x 和爷节点的右儿子互换，更新信息，然后退出.
4. 若爷节点没有右儿子，则先让点 x 成为爷节点的右儿子，此时点 x 原来的父节点没有中儿子，根据上文 Rake Node 的性质，它不能存在. 于是调用 `Delete` 函数，将其删除，然后退出.

1，2 两个步骤合称为 **Local Splay**. 3，4 两个步骤合称为 **Splice**. 但我们方便起见，将它们都写在 `Splice(x)` 函数里.

上文提到的 `Delete(x)` 函数是这样的：

1. 检视将要删除的点 x 有没有左儿子，若有，则将左儿子的子树后继续旋转到点 x 下方（成为新的左儿子），然后将右儿子（若有）变成左儿子的右儿子，此时点 x 的左儿子就代替了点 x. 这相当于 Splay 的合并操作.
2. 若没有左儿子，则直接让其右儿子代替点 x.

不难发现，`Splice(x)` 改变了原树的一些簇的端点选取. 一次 splice 完了之后，我们将点 x 的父亲节点当作新的点 x，进行下一次 splice.

最终我们会发现，我们最开始要操作的点 x 一定在根簇的 Compress Tree 最右端. 我们只需最后做一次 **Global Splay**，将其旋至 SATT 根部即可.

```
// ls 一个SATT节点的左儿子
// rs 一个SATT节点的右儿子
// ms 一个SATT节点的中儿子
// son[x][0] ls
// son[x][1] rs
// son[x][2] ms
// type==1 在 Rake Tree中
// type==0 在 Compress Tree中
int new_node() {
  if (top) {
    top--;
    return Stack[top + 1];
  }
  return ++tot;
}

void setfather(int x, int fa, int type) {
  if (x) father[x] = fa;
  son[fa][type] = x;
}
```

```
21
22 void Delete(int x) {
23   setfather(ms(x), father[x], 1);
24   if (ls(x)) {
25     int p = ls(x);
26     pushdown(p, 1);
27     while (rs(p)) p = rs(p), pushdown(p, 1);
28     splay(p, 1, x);
29     setfather(rs(x), p, 1);
30     setfather(p, father[x], 2);
31     pushup(p, 1);
32     pushup(father[x], 0);
33   } else
34     setfather(rs(x), father[x], 2);
35   Clear(x);
36 }
37
38 void splice(int x) {
39   // local splay
40   splay(x, 1);
41   int y = father[x];
42   splay(y, 0);
43   pushdown(x, 1);
44   // splice
45   if (rs(y)) {
46     swap(father[ms(x)], father[rs(y)]);
47     swap(ms(x), rs(y));
48   } else
49     Delete(x);
50   pushup(x, 1);
51   pushup(y, 0);
52 }
53
54 void access(int x) {
55   splay(x, 0);
56   if (rs(x)) {
57     int y = new_node();
58     setfather(ms(x), y, 0);
59     setfather(rs(x), y, 2);
60     rs(x) = 0;
61     setfather(y, x, 2);
62     pushup(y, 1);
63     pushup(x, 0);
64   }
65   while (father[x]) {
66     splice(father[x]);
67     x = father[x];
68     pushup(x, 0);
69   }
70   splay(x, 0)  // global splay
71 }
```

若要让一个点成为原树的根，那么我们就将点 x Access 到 SATT 的根节点，可知此时点 x 已经是最终状态的簇一个端点．由 Compress Tree 的中序遍历性质可知，将点 x 所在的 Compress Tree 左右颠倒（所有

点的左右儿子互换），就使点 x 成为原树的根．在具体实现中，我们通过给点 x 打上一个翻转标记，之后下传来进行这一过程．

```
void makeroot(int x) {
  access(x);
  push_rev(x);
}
```

于是 `expose(x, y)` 就呼之欲出：

```
void expose(int x, int y) {
  makeroot(x);
  access(y);
}
```

Link & Cut

现在我们要将原树中两个不连通的点之间连一条边，我们先让其中的一个点 x 成为原树的根，再将另一个点 y 旋转到根处，可知此时应该使点 y 成为点 x 的右儿子．然后在点 y 的右儿子上挂上这一条边（在只需维护点的 SATT 中，这一步可省）．

```
void Link(int x, int y, int z) {
  // z代表连接 x, y的边
  access(x);
  makeroot(y);
  setfather(y, x, 1);
  setfather(z, y, 0);
  pushup(x, 0);
  pushup(y, 0);
}
```

`Cut` 跟 `Link` 原理差不多

```
void cut(int x, int y) {
  expose(x, y);
  clear(rs(x));  // 删掉 xy 这一基簇
  father[x] = ls(y) = rs(x);
  pushup(y, 0);
}
```

完整代码

Luogu P3690【模板】动态树[1]

```
#include <iostream>
#define ls(x) T[x][0]
#define rs(x) T[x][1]
#define ms(x) T[x][2]
using namespace std;

constexpr int MAXN = 300005;

int T[MAXN][3], s[MAXN][2], tot, v[MAXN], r[MAXN], top, st[MAXN], f[MAXN];

int nnd() {
```

```
  if (top) {
    top--;
    return st[top + 1];
  }
  return ++tot;
}

bool isr(int x) { return rs(f[x]) != x && ls(f[x]) != x; }

bool dir(int x) { return rs(f[x]) == x; }

void psu(int x, int ty) {
  if (ty) {
    s[x][1] = s[ls(x)][1] ^ s[rs(x)][1] ^ s[ms(x)][1];
    return;
  }
  s[x][0] = s[ls(x)][0] ^ v[x] ^ s[rs(x)][0];
  s[x][1] = s[ls(x)][1] ^ s[ms(x)][1] ^ s[rs(x)][1] ^ v[x];
}

void psr(int x) {
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
}

void psd(int x, int ty) {
  if (ty) return;
  if (r[x]) {
    psr(ls(x));
    psr(rs(x));
    r[x] = 0;
  }
}

void upd(int x, int ty) {
  if (!isr(x)) upd(f[x], ty);
  psd(x, ty);
}

void stf(int x, int fa, int ty) {
  if (x) f[x] = fa;
  T[fa][ty] = x;
}

void rtt(int x, int ty) {
  int y = f[x], z = f[y], d = dir(x), w = T[x][d ^ 1];
  if (z) T[z][ms(z) == y ? 2 : dir(y)] = x;
  T[x][d ^ 1] = y;
  T[y][d] = w;
  if (w) f[w] = y;
  f[y] = x;
```

```
  f[x] = z;
  psu(y, ty);
  psu(x, ty);
}

void spy(int x, int ty, int gl = 0) {
  upd(x, ty);
  for (int y; y = f[x], (!isr(x)) && y != gl; rtt(x, ty)) {
    if (f[y] != gl && (!isr(y))) rtt(dir(x) ^ dir(y) ? x : y, ty);
  }
}

void cle(int x) {
  ls(x) = ms(x) = rs(x) = s[x][0] = s[x][1] = r[x] = v[x] = 0;
  st[++top] = x;
}

void del(int x) {
  stf(ms(x), f[x], 1);
  if (ls(x)) {
    int p = ls(x);
    psd(p, 1);
    while (rs(p)) p = rs(p), psd(p, 1);
    spy(p, 1, x);
    stf(rs(x), p, 1);
    stf(p, f[x], 2);
    psu(p, 1);
    psu(f[x], 0);
  } else
    stf(rs(x), f[x], 2);
  cle(x);
}

void spl(int x) {
  spy(x, 1);
  int y = f[x];
  spy(y, 0);
  psd(x, 1);
  if (rs(y)) {
    swap(f[ms(x)], f[rs(y)]);
    swap(ms(x), rs(y));
    psu(x, 1);
  } else
    del(x);
  psu(rs(y), 0);
  psu(y, 0);
}

void acs(int x) {
  spy(x, 0);
  int ys = x;
  if (rs(x)) {
```

```
116     int y = nnd();
117     stf(ms(x), y, 0);
118     stf(rs(x), y, 2);
119     rs(x) = 0;
120     stf(y, x, 2);
121     psu(y, 1);
122     psu(x, 0);
123   }
124   while (f[x]) {
125     spl(f[x]);
126     x = f[x];
127   }
128   spy(ys, 0);
129 }
130
131 int fdr(int x) {
132   acs(x);
133   psd(x, 0);
134   while (ls(x)) x = ls(x), psd(x, 0);
135   spy(x, 0);
136   return x;
137 }
138
139 void mkr(int x) {
140   acs(x);
141   psr(x);
142 }
143
144 void epo(int x, int y) {
145   mkr(x);
146   acs(y);
147 }
148
149 void lnk(int x, int y) {
150   if (fdr(x) == fdr(y)) return;
151   acs(x);
152   mkr(y);
153   stf(y, x, 1);
154   psu(x, 0);
155   psu(y, 0);
156 }
157
158 void cu(int x, int y) {
159   epo(x, y);
160   if (ls(y) != x || rs(x)) return;
161   f[x] = ls(y) = 0;
162   psu(y, 0);
163 }
164
165 int main() {
166   cin.tie(nullptr)->sync_with_stdio(false);
167   int n, m;
```

```
  cin >> n >> m;
  tot = n;
  for (int i = 1; i <= n; i++) {
    cin >> v[i];
    psu(i, 0);
  }
  for (int i = 1; i <= m; i++) {
    int op, U, V;
    cin >> op >> U >> V;
    if (op == 0) {
      epo(U, V);
      cout << s[V][0] << '\n';
    }
    if (op == 1) lnk(U, V);
    if (op == 2) cu(U, V);
    if (op == 3) {
      acs(U);
      v[U] = V;
      psu(U, 0);
    }
  }
  return 0;
}

```

SATT 的时间复杂度证明

设在一棵 SATT（点数为 n）中，其当前状态 x 的势能函数为

$$\varphi(x) = \sum_{i=1}^{n} r(i)$$

其中 $r(i) = \lceil \log_2 \operatorname{siz}(i) \rceil$. $\operatorname{siz}(i)$ 为以 i 为根的子树大小.

则 SATT 的 splay 的均摊复杂度显然仍是 $3n \log n + 1$，即使 SATT 是一个三叉树.

因此对于 SATT，我们只要证得 Access 函数复杂度正确，就能证得 SATT 的时间复杂度.

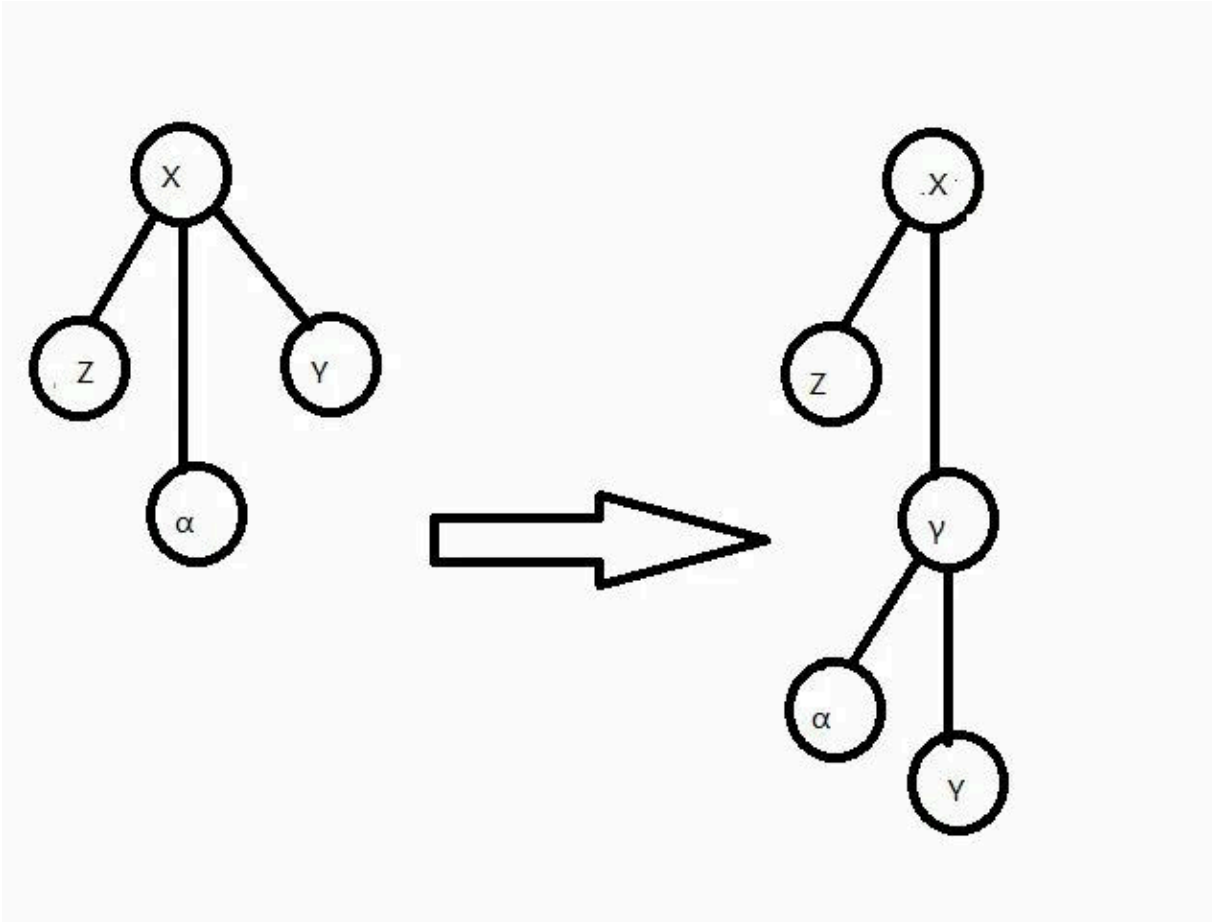
我们逐步分析 Accese 的均摊复杂度.

我们先要将点 x 旋至其所在 Compress Tree 的根，则这一步的均摊复杂度

$$a \le 3 \log n + 1$$

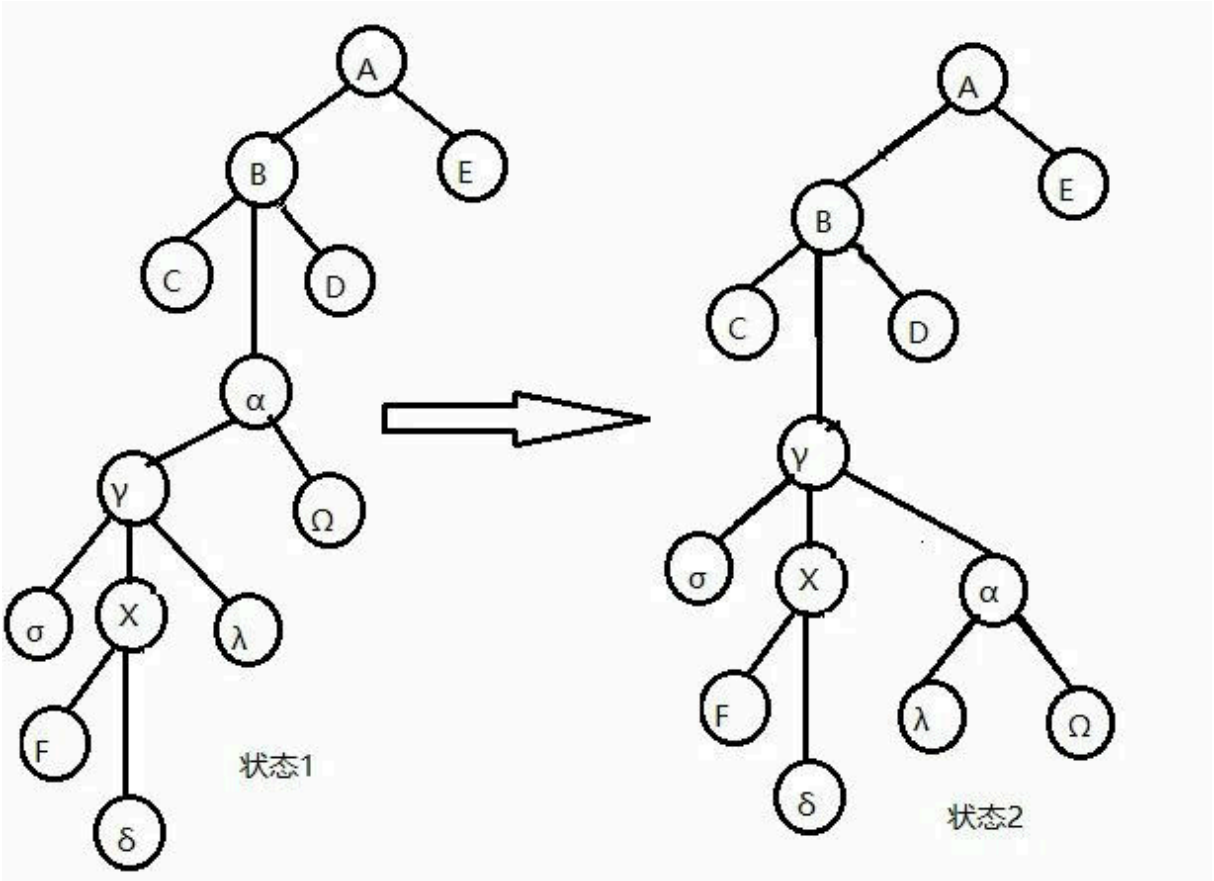
接着我们要使点 x 无右儿子，则这一步的均摊复杂度

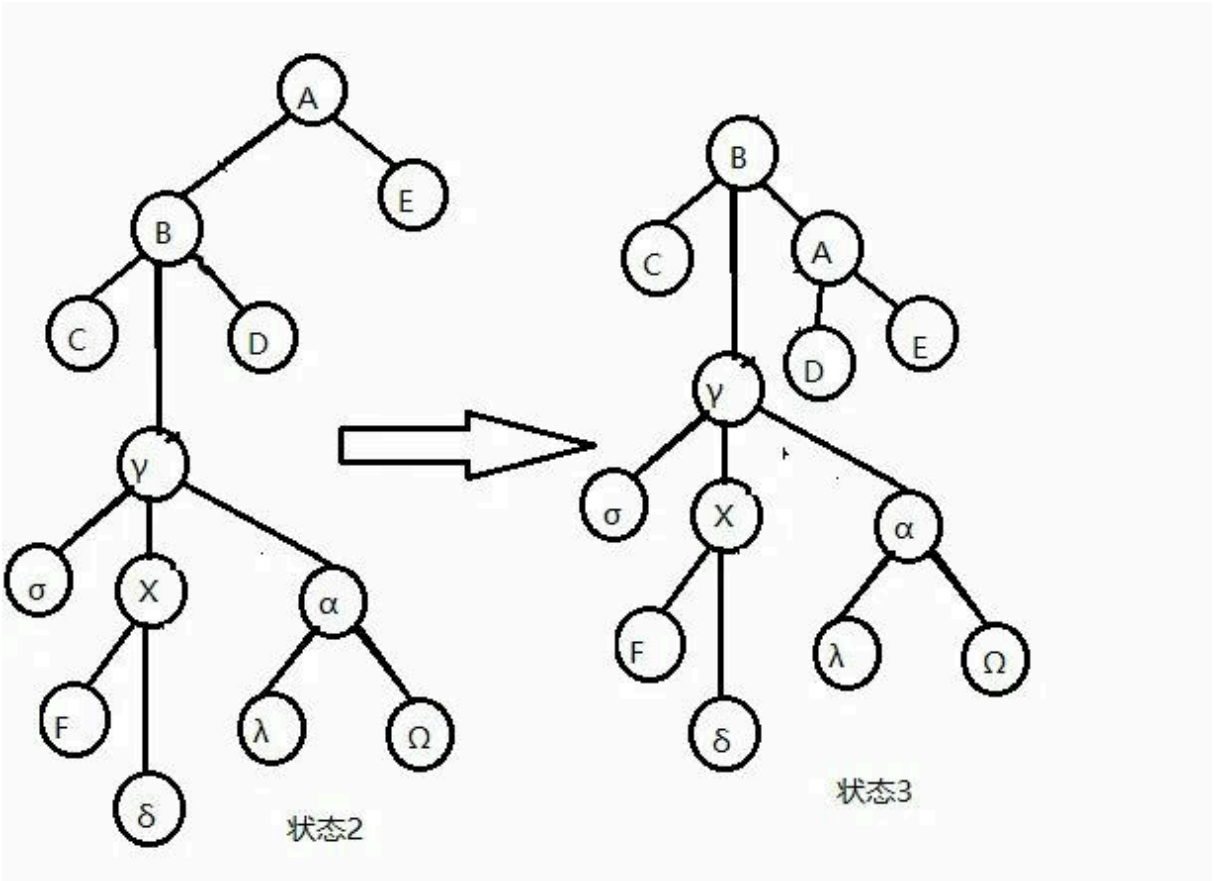
$$a = 1 + r'(\gamma) - 0 \le \log n + 1$$

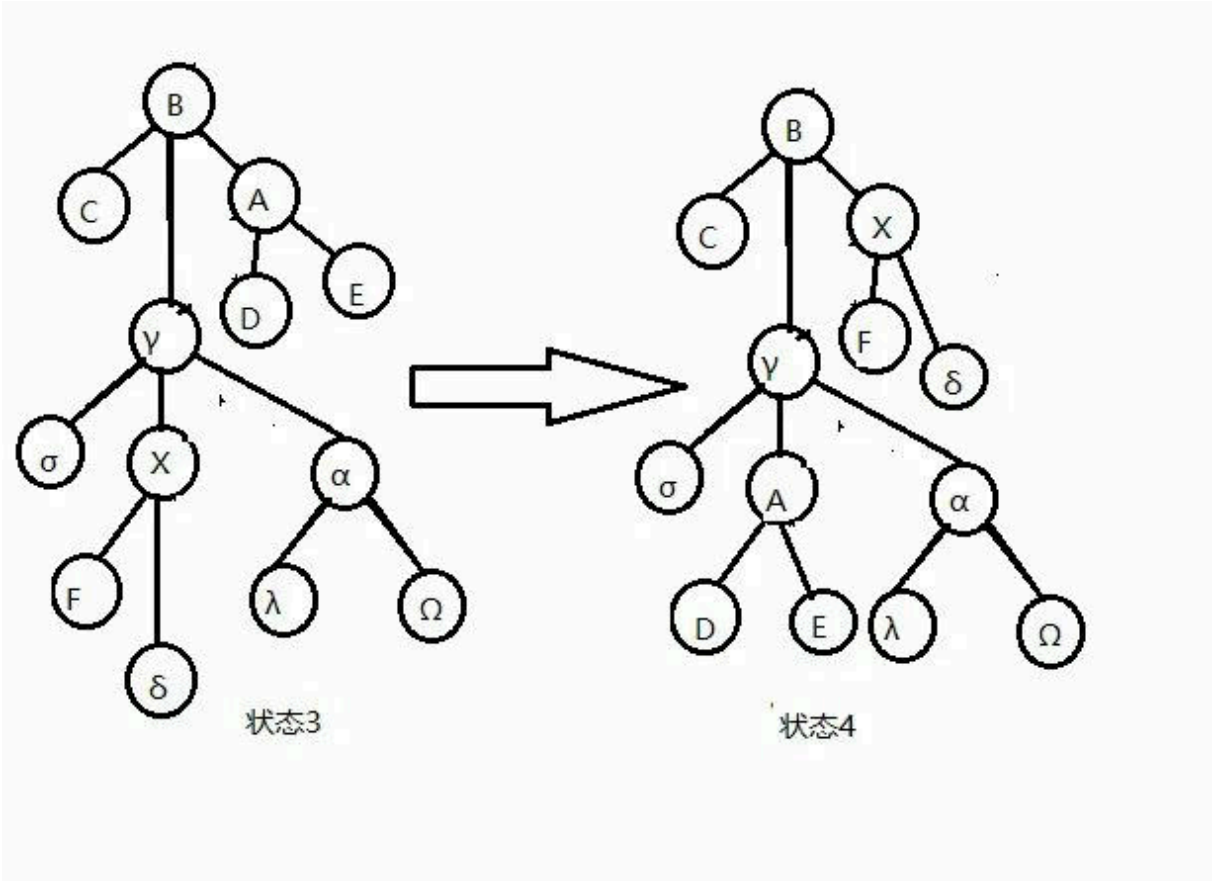


如图，为去掉点 x 的右儿子过程.

然后是 Local Splay，Splice 交替进行的过程，经过若干次 Splice，点 x 被旋至 SATT 的根．我们对其中一组 Local Splay，Splice 进行分析：


A
B
E
C
D
α
γ
Ω
σ
X
λ
F
δ
状态1
A
E
B
C
D
γ
σ
X
α
F
λ
Ω
δ
状态2


A
E
B
C
D
γ
σ
X
α
F
λ
Ω
δ
状态2
B
C
A
D
E
γ
σ
X
α
F
λ
Ω
δ
状态3



如图，体现了对点 x 做一次 Splice 的过程，不包括最后左旋点 x 的部分.

为表达方便，设 $r_x(i)$ 为点 i 在状态 x 时的 r 值.

由图，易知由状态 1 到状态 2 的操作（将点 x 的父亲旋至其 Rake Tree 的根部的 Local Splay 操作）的均摊复杂度

$$a \le 3(r_2(\gamma) - r_1(\gamma)) + 1$$

由图，易知由状态 2 到状态 3 的操作（将点 x 的爷节点旋至其 Compress Tree 的根部的 Local Splay 操作）的均摊复杂度

$$a \le 3(r_3(B) - r_2(B)) + 1$$

重点分析由状态 3 到状态 4 的操作（Splice）

$$a = r_4(\gamma) - r_3(\gamma) + 1$$

不难发现 $r_4(\gamma) \le r_3(B)$

故这一次操作的均摊复杂度为

$$\begin{aligned} a &\le r_3(B) - r_3(\gamma) + 1 \\ &\le 3(r_3(B) - r_3(\gamma)) + 1 \end{aligned}$$

综合上述过程，一次 Splice 的复杂度为

$$a \le 3r_3(B) + 3r_3(B) + 3r_2(\gamma) - 3r_3(\gamma) - 3r_2(B) - 3r_1(\gamma) + 3$$

记下一次 Splice 的点 X（即状态 4 中的点 B）的 r 值为 $r'(X)$，并注意到 $r_3(\gamma), r_1(\gamma) \ge r_1(X)$，$r_3(B), r_2(\gamma) \le r'(X)$ 且 $r_3(B) = r_2(B)$，所以

$$a \le 9(r'(X) - r(X)) + 3$$

除了上面这个复杂度以外，在 Splice 中可能还会有因 `delete(x)` 产生的额外均摊复杂度，记这一部分为 $a' \le 3\log n + 1$.

先不管 a' 部分，每次 Splice 的 $r'(X)$ 等于下一次的 $r(X)$，且第一次 Splice 的 $r(X)$ 等于我们一开始旋转点 x 到其 Compress Tree 树根时的 $r(X)$，则对于不计 `delete(x)` 的一次 `access(x)` 复杂度，我们有：

$$a \le 9(r'(x) - r(x)) + 3k + 1$$

其中 k 为 Splice 次数.

看样子 a 会带一个 $3k+1$ 导致均摊复杂度无法分析，但我们有办法来对付它，注意到 zig-zig/zig-zag 的旋转可以这么均摊

$$\begin{aligned} a &\le 3(r'(X) - r(X)) + q \\ &\le 3(q-1)(r'(X) - r(X)) \end{aligned}$$

如果我们能找到足够多的 zig-zig，zig-zag 操作，我们就可以将这 $3k+1$ 平摊到这些操作上去，从而消掉这个 $3k+1$.

我们发现 Globel Splay 里面就有这么多的 zig-zig，zag-zig 来给我们使用，因为 Globel Splay 里面点的个数一定大于 k，而从点 x 到 Globel Splay 根部路径的点数一定不少于 k，也就是说一次 `access(x)` 中一定会至少有 $\frac{k}{2}$ 个 zig-zag 操作，算上 Globel Splay 的均摊复杂度 $a \le 3\log n + 1$，一次 `access(x)` 不记 `delete(x)` 的均摊复杂度为

$$\begin{aligned} &a \le 9(r'(X) - r(X)) + 3k + 1 + 18(r''(X) - r'(X)) - S + 1 + 3\log n + 1, S \ge 3k \\ &a \le 18(r''(X) - r(X)) + 2 + 3\log n + 1 \\ &a \le 21(r''(X) - r(X)) + 3 \end{aligned}$$

现在算上 a'，列出进行 m 次 `access(x)` 操作的总式子.

$$\sum_{i=1}^{m} a_{(i)'} + \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} c_i + \varphi(x_n) - \varphi(x_0)$$

我们要求的是实际复杂度

$$\begin{aligned} \sum_{i=1}^{m} c_i &= \sum_{i=1}^{m} a_i + \sum_{i=1}^{m} a_{(i)'} - \varphi(x_n) + \varphi(x_0) \\ &\le \sum_{i=1}^{m} a_{(i)'} + 21m\log n + n\log n + 3m \end{aligned}$$

注意到 `delete(x)` 操作的本质是删掉一个 Rake Node，但我们在 m 次操作中最多只会添加 m 个 Rake Node，由 Rake Node 的定义，我们初始时最多有 n 个 Rake Node，也就是说我们总共只会做 $m+n$ 次 `delete(x)` 操作，由 $a' \le 3\log n + 1$ 可知

$$\sum_{i=1}^{m} c_i \le 3(m+n)\log n + 21m\log n + n\log n + 4m + n$$

所以我们就证明了 Access 的复杂度，而其他函数要么基于 Access 要么单次时间复杂度为常数，所以我们就证明了 SATT 的复杂度.

顺便一提，如果像 LCT 一样省略 Global Splay 的过程，改为在每次 Splice 时直接将要 Access 的点旋转一下，这样做时间复杂度也是对的（实测省略 Global Splay 的版本要快很多，能与 LCT 在 Luogu P3690 跑得不分上下）.

例题

例题 1

CEOI 2019 Dynamic Diameter[2]

给定一棵 n 个节点的树，每条边有边权，有 q 次更新，每次修改一条边的边权，并询问树的直径. 强制在线.

维护动态直径，建出 SATT 后，我们只需要在 `Pushup(x)` 里面维护每个点的答案，最后查询根节点的答案（即整棵树的直径）就可以了.

```
void pushup(int x, int op) {
  if (op == 0) {
    // 是 Compress Node
    len[x] = len[ls(x)] + len[rs(x)];
    diam[x] = maxs[ls(x)][1] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], max(maxs[ls(x)][1], maxs[rs(x)][0]) + maxs[ms(x)][0]);
    diam[x] = max(diam[x], max(max(diam[ls(x)], diam[rs(x)]), diam[ms(x)]));
    maxs[x][0] =
        max(maxs[ls(x)][0], len[ls(x)] + max(maxs[ms(x)][0], maxs[rs(x)][0]));
    maxs[x][1] =
        max(maxs[rs(x)][1], len[rs(x)] + max(maxs[ms(x)][0], maxs[ls(x)][1]));
  } else {
    // 是 Rake Node
    diam[x] = maxs[ls(x)][0] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], maxs[ms(x)][0] + max(maxs[ls(x)][0], maxs[rs(x)][0]));
    diam[x] = max(max(diam[x], diam[ms(x)]), max(diam[ls(x)], diam[rs(x)]));
    maxs[x][0] = max(maxs[ms(x)][0], max(maxs[ls(x)][0], maxs[rs(x)][0]));
  }
  return;
}
```

其中 $diam$ 是当前点的答案（这个点代表的簇的直径）. len 表示当前 Compress Node 所在簇路径的长度，$maxs_{0/1}$ 表示 Compress Node 到簇内点和端点的不选簇路径儿子/不选父亲的最大距离（如果是 Rake Node 则只存储选取当前簇的上端点到簇内点和端点的最大距离 $maxs_0$）. 每次查询 SATT 根节点的 diam 即可，正确性显然.

注意对 `Pushrev(x)` 做一些改动.

```
void pushrev(int x) {
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
  swap(maxs[x][0], maxs[x][1]);
}
```

例题 2

> **「CSP-S 2019」树的重心**[3]
>
> 给定一棵树，求出单独删去树的每条边后，分裂出的两个子树的重心编号和之和.

假如我们能动态 $O(\log n)$ 维护树的重心，我们就做出这个题了.

SATT 支持动态 $O(\log n)$ 维护树的重心，做到这需要 **非局部搜索（Non-local Search）**.

对于一种树上的性质，如果一个点/一条边在整棵树中有这种性质，且在所有包含它的子树中都包含此种性质，我们就称这个性质是 **局部的（Local）**，否则称它是 **非局部的（Non-local）**. 局部信息一般可以通过 `pushup(x)` 来维护

例如，权值最小值是局部的，因为一个点/一条边如果在整棵树中权值最小，那么在所有包含它的子树中它也是权值最小的，而权值第二小显然就是非局部的.

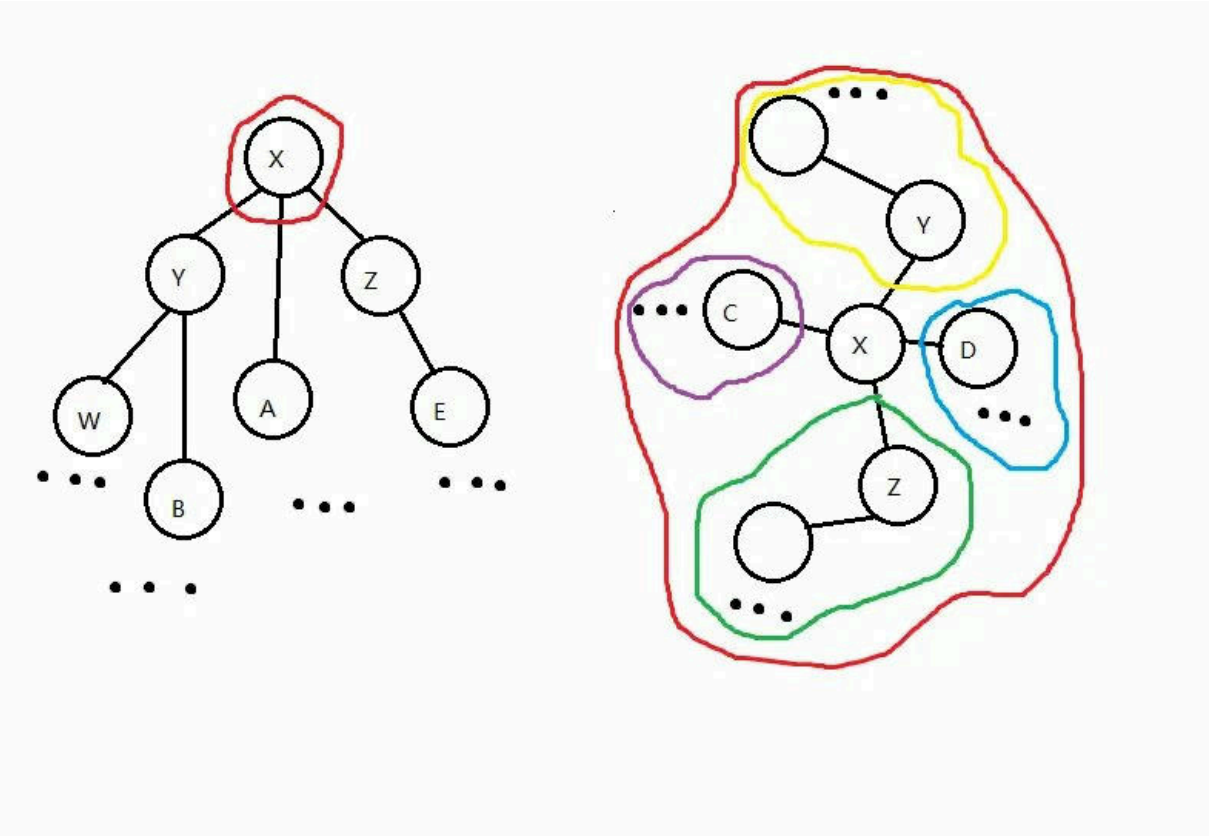
我们上文维护的 $diam$ 也是局部信息.

回到正题，重心显然是一个非局部信息，无法通过简单的 `pushup(x)` 来维护．我们考虑在 SATT 上搜索：

我们的搜索从 SATT 的根节点，即根簇开始．注意到重心有很好的性质：假如有一条边的一侧点的个数大于等于另一侧点的个数，那么边的这一侧一定至少有一个重心（重心可能有两个）.

记 sum 表示某一个簇的点个数，$maxs$ 为一棵 Rake Tree 的所有 Rake Node 中儿子的 sum 最大值.

```
void pushup(int x, int op) {
  if (op == 0) {
    // 是 Compress Node
    sum[x] = sum[ls(x)] + sum[rs(x)] + sum[ms(x)] + 1;
  } else {
    // 是 Rake Node
    maxs[x] = max(maxs[ls(x)], max(maxs[rs(x)], sum[ms(x)]));
    sum[x] = sum[ls(x)] + sum[rs(x)] + sum[ms(x)];
  }
}
```



如图，为在进行 Non-local Search 时的 SATT 和对应的原树 T.

我们做如下比较：

1. 比较簇 $compress(Y)$ 的 sum 值与簇 $compress(Z)$、簇 A 和点 X 的并（我们暂称为簇 α）的 sum 值．若 $compress(Y)$ 的 sum 值大于等于后者，说明至少有一个重心在 $compress(Y)$ 的子树中，我们递归到 $compress(Y)$ 搜索．（如果此处取等，点 X 也是一个重心，需要记录）
2. 比较簇 $compress(Z)$ 的 sum 值与簇 $compress(Y)$、簇 A 和点 X 的并（我们暂称为簇 β）的 sum 值．若 $compress(Z)$ 的 sum 值大于等于后者，说明至少有一个重心在 $compress(Z)$ 的子树中，我们递归到 $compress(Z)$ 搜索．（如果此处取等，点 X 也是一个重心，需要记录）

3. 比较点 x 中儿子 Rake tree 之中 sum 最大的更小簇的 sum 值与簇 $compress(Y)$、簇 A、点 X 及其它更小簇的并（我们暂称为簇 Y）的 sum 值，若那个更小簇的 sum 值大于等于后者，说明至少有一个重心在那个更小簇的子树中，我们递归到它搜索．如果此处取等，点 X 也是一个重心，需要记录．
4. 若以上比较都不递归，则点 X 一定是一个重心，记录并退出．

第一步的搜索显然正确，之后应该怎么搜呢?

假如我们递归到 Y，则现在 Y 储存信息的并不完整，因为 $compress(Y)$ 里面只存储了它自己这个簇的信息，而我们要求的是整棵树的重心．解决方法是，将之前簇的信息记录下来，在点 Y 上比较计算时将上一个簇的信息与点 Y 自己的信息合并处理．具体实现如下：

```
void non_local_search(int x, int lv, int rv, int op) {
  // lv 和 rv 都是搜索的上一个簇的信息
  if (!x) return;
  psd(x, 0);
  if (op == 0) {
    if (maxs[ms(x)] >=
        sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + lv + 1 + rv) {
      if (maxs[ms(x)] ==
          sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + lv + 1 + rv) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(
          ms(x),
          sum[ms(x)] - maxs[ms(x)] + sum[rs(x)] + sum[ls(x)] + 1 + lv + rv, 0,
          1);
      return;
    }
    if (ss[rs(x)] + rv >= ss[ms(x)] + ss[ls(x)] + lv + 1) {
      if (ss[rs(x)] + rv == ss[ms(x)] + ss[ls(x)] + lv + 1) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(rs(x), sum[ms(x)] + 1 + sum[ls(x)] + lv, rv, 0);
      return;
    }
    if (sum[ls(x)] + lv >= sum[ms(x)] + sum[rs(x)] + 1 + rv) {
      if (sum[ls(x)] + lv == sum[ms(x)] + sum[rs(x)] + 1 + rv) {
        if (ans1)
          ans2 = x;
        else
          ans1 = x;
      }
      non_local_search(ls(x), lv, rv + sum[ms(x)] + 1 + sum[rs(x)], 0);
      return;
    }
  } else {
    if (maxs[ls(x)] == maxs[x]) {
      non_local_search(ls(x), lv, rv, 1);
```

```
44        return;
45      }
46      if (maxs[rs(x)] == maxs[x]) {
47        non_local_search(rs(x), lv, rv, 1);
48        return;
49      }
50      non_local_search(ms(x), lv, rv, 0);
51      return;
52    }
53    if (ans1)
54      ans2 = x;
55    else
56      ans1 = x;
57  }
```

示例代码

```
1  #include <algorithm>
2  #include <iostream>
3  #define ls(x) T[x][0]
4  #define rs(x) T[x][1]
5  #define ms(x) T[x][2]
6  using namespace std;
7
8  constexpr int MAXN = 600005;
9
10 int T[MAXN][3], tot, r[MAXN], top, st[MAXN], f[MAXN], maxs[MAXN], ss[MAXN];
11
12 int nnd() {
13   if (top) {
14     top--;
15     return st[top + 1];
16   }
17   return ++tot;
18 }
19
20 bool isr(int x) { return rs(f[x]) != x && ls(f[x]) != x; }
21
22 bool dir(int x) { return rs(f[x]) == x; }
23
24 void psr(int x) {
25   if (!x) return;
26   r[x] ^= 1;
27   swap(ls(x), rs(x));
28 }
29
30 void psd(int x, int ty) {
31   if (ty) return;
32   if (r[x]) {
33     psr(ls(x));
34     psr(rs(x));
35     r[x] = 0;
36   }
```

```
}

void psu(int x, int op) {
  psd(x, op); /*不知道哪没 psd*/
  if (op == 0) {
    ss[x] = ss[ls(x)] + ss[rs(x)] + ss[ms(x)] + 1;
  } else {
    maxs[x] = max(maxs[ls(x)], max(maxs[rs(x)], ss[ms(x)]));
    ss[x] = ss[ls(x)] + ss[rs(x)] + ss[ms(x)];
  }
}

void upd(int x, int ty) {
  if (!isr(x)) upd(f[x], ty);
  psd(x, ty);
}

void stf(int x, int fa, int ty) {
  if (x) f[x] = fa;
  T[fa][ty] = x;
}

void rtt(int x, int ty) {
  int y = f[x], z = f[y], d = dir(x), w = T[x][d ^ 1];
  if (z) T[z][ms(z) == y ? 2 : dir(y)] = x;
  T[x][d ^ 1] = y;
  T[y][d] = w;
  if (w) f[w] = y;
  f[y] = x;
  f[x] = z;
  psu(y, ty);
  psu(x, ty);
}

void spy(int x, int ty, int gl = 0) {
  upd(x, ty);
  for (int y; y = f[x], (!isr(x)) && y != gl; rtt(x, ty)) {
    if (f[y] != gl && (!isr(y))) rtt(dir(x) ^ dir(y) ? x : y, ty);
  }
}

void cle(int x) {
  ls(x) = ms(x) = rs(x) = ss[x] = r[x] = maxs[x] = f[x] = 0;
  st[++top] = x;
}

void del(int x) {
  stf(ms(x), f[x], 1);
  if (ls(x)) {
    int p = ls(x);
    psd(p, 1);
    while (rs(p)) p = rs(p), psd(p, 1);
```

```
    spy(p, 1, x);
    stf(rs(x), p, 1);
    stf(p, f[x], 2);
    psu(p, 1);
    psu(f[x], 0);
  } else
    stf(rs(x), f[x], 2);
  cle(x);
}

void spl(int x) {
  spy(x, 1);
  int y = f[x];
  spy(y, 0);
  psd(x, 1);
  if (rs(y)) {
    swap(f[ms(x)], f[rs(y)]);
    swap(ms(x), rs(y));
  } else
    del(x);
  psu(x, 1);
  psu(y, 0);
  rtt(rs(y), 0);
}

void acs(int x) {
  spy(x, 0);
  if (rs(x)) {
    int y = nnd();
    stf(ms(x), y, 0);
    stf(rs(x), y, 2);
    rs(x) = 0;
    stf(y, x, 2);
    psu(y, 1);
    psu(x, 0);
  }
  while (f[x]) spl(f[x]);
}

void mkr(int x) {
  acs(x);
  psr(x);
}

void epo(int x, int y) {
  mkr(x);
  acs(y);
}

void lnk(int x, int y) {
  acs(x);
  mkr(y);
```

```
141   stf(y, x, 1);
142   psu(x, 0);
143 }
144
145 void cu(int x, int y) {
146   epo(x, y);
147   f[x] = ls(y) = 0;
148   psu(y, 0);
149 }
150
151 int ans1, ans2;
152
153 void non_local_search(int x, int lv, int rv, int op) {
154   if (!x) return;
155   psd(x, 0);
156   if (op == 0) {
157     if (maxs[ms(x)] >=
158         ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + lv + 1 + rv) {
159       if (maxs[ms(x)] ==
160           ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + lv + 1 + rv) {
161         if (ans1)
162           ans2 = x;
163         else
164           ans1 = x;
165       }
166       non_local_search(
167           ms(x), ss[ms(x)] - maxs[ms(x)] + ss[rs(x)] + ss[ls(x)] + 1 + lv
    + rv,
168           0, 1);
169       return;
170     }
171     if (ss[rs(x)] + rv >= ss[ms(x)] + ss[ls(x)] + lv + 1) {
172       if (ss[rs(x)] + rv == ss[ms(x)] + ss[ls(x)] + lv + 1) {
173         if (ans1)
174           ans2 = x;
175         else
176           ans1 = x;
177       }
178       non_local_search(rs(x), ss[ms(x)] + 1 + ss[ls(x)] + lv, rv, 0);
179       return;
180     }
181     if (ss[ls(x)] + lv >= ss[ms(x)] + ss[rs(x)] + 1 + rv) {
182       if (ss[ls(x)] + lv == ss[ms(x)] + ss[rs(x)] + 1 + rv) {
183         if (ans1)
184           ans2 = x;
185         else
186           ans1 = x;
187       }
188       non_local_search(ls(x), lv, rv + ss[ms(x)] + 1 + ss[rs(x)], 0);
189       return;
190     }
191   } else {
```

```
    if (maxs[ls(x)] == maxs[x]) {
      non_local_search(ls(x), lv, rv, 1);
      return;
    }
    if (maxs[rs(x)] == maxs[x]) {
      non_local_search(rs(x), lv, rv, 1);
      return;
    }
    non_local_search(ms(x), lv, rv, 0);
    return;
  }
  if (ans1)
    ans2 = x;
  else
    ans1 = x;
  return;
}

int qu[MAXN], qv[MAXN];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int TT;
  cin >> TT;
  while (TT--) {
    int n;
    cin >> n;
    tot = n;
    long long ANS = 0;
    for (int i = 1; i <= n; i++) ss[i] = 1;
    for (int i = 1; i < n; i++) {
      cin >> qu[i] >> qv[i];
      lnk(qu[i], qv[i]);
    }
    for (int i = 1; i < n; i++) {
      cu(qu[i], qv[i]);
      ans1 = 0;
      ans2 = 0;
      non_local_search(qu[i], 0, 0, 0);
      ANS += ans1 + ans2;
      if (ans1) acs(ans1);
      if (ans2) acs(ans2);
      ans1 = 0;
      ans2 = 0;
      non_local_search(qv[i], 0, 0, 0);
      ANS += ans1 + ans2;
      if (ans1) acs(ans1);
      if (ans2) acs(ans2);
      lnk(qu[i], qv[i]);
    }
    cout << ANS << '\n';
    for (int i = 1; i <= tot; i++)
```

```
244         T[i][0] = T[i][1] = T[i][2] = ss[i] = r[i] = maxs[i] = f[i] = 0;
245       tot = top = 0;
246     }
247     return 0;
248 }
249
```

Reference

1. Robert E. Tarjan and Renato F. Werneck. 2005. Self-adjusting top trees. In Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms (SODA '05). Society for Industrial and Applied Mathematics, USA, 813–822. DOI 10.5555/1070432.1070547
2. negiizhao 的博客[4]

外部参考资料

[1] Luogu P3690【模板】动态树 [2] CEOI 2019 Dynamic Diameter [3] 「CSP-S 2019」树的重心

[4] negiizhao 的博客

10.21 析合树

关于段的问题

我们由一个小清新的问题引入：

> 对于一个 $1-n$ 的排列，我们称一个值域连续的区间为段．问一个排列的段的个数．比如，$\{5,3,4,1,2\}$ 的段有：$[1,1],[2,2],[3,3],[4,4],[5,5],[2,3],[4,5],[1,3],[2,5],[1,5]$.

看到这个东西，感觉要维护区间的值域集合，复杂度好像挺不友好的．线段树可以查询某个区间是否为段，但不太能统计段的个数．

这里我们引入这个神奇的数据结构——析合树！

连续段

在介绍析合树之前，我们先做一些前提条件的限定．鉴于 LCA 的课件中给出的定义不易理解，为方便读者理解，这里给出一些不太严谨（但更容易理解）的定义．

排列与连续段

排列：定义一个 n 阶排列 P 是一个大小为 n 的序列，使得 P_i 取遍 $1,2,\cdots,n$．说得形式化一点，n 阶排列 P 是一个有序集合满足：

1. $|P|=n$.
2. $\forall i, P_i\in[1,n]$.
3. $\nexists i,j\in[1,n], P_i=P_j$.

连续段：对于排列 P，定义连续段 $(P,[l,r])$ 表示一个区间 $[l,r]$，要求 $P_{l\sim r}$ 值域是连续的．说得更形式化一点，对于排列 P，连续段表示一个区间 $[l,r]$ 满足：

$$(\nexists\, x,z\in[l,r], y\notin[l,r], P_x<P_y<P_z)$$

特别地，当 $l>r$ 时，我们认为这是一个空的连续段，记作 $(P,\emptyset)$.

我们称排列 P 的所有连续段的集合为 I_P，并且我们认为 $(P,\emptyset)\in I_P$.

连续段的运算

连续段是依赖区间和值域定义的，于是我们可以定义连续段的交并差的运算.

定义 $A=(P,[a,b]),B=(P,[x,y])$，且 $A,B\in I_P$. 于是连续段的关系和运算可以表示为:

1. $A\subseteq B\iff x\le a\wedge b\le y$.
2. $A=B\iff a=x\wedge b=y$.
3. $A\cap B=(P,[\max(a,x),\min(b,y)])$.
4. $A\cup B=(P,[\min(a,x),\max(b,y)])$.
5. $A\setminus B=(P,\{i\mid i\in[a,b]\wedge i\notin[x,y]\})$.

其实这些运算就是普通的集合交并差放在区间上而已.

连续段的性质

连续段的一些显而易见的性质. 我们定义 $A,B\in I_P,A\cap B\neq\emptyset,A\notin B,B\notin A$，那么有 $A\cup B,A\cap B,A\setminus B,B\setminus A\in I_P$.

证明? 证明的本质就是集合的交并差的运算.

析合树

好的，现在讲到重点了. 你可能已经猜到了，析合树正是由连续段组成的一棵树. 但是要知道一个排列可能有多达 $O(n^2)$ 个连续段，因此我们就要抽出其中更基本的连续段组成析合树.

本原段

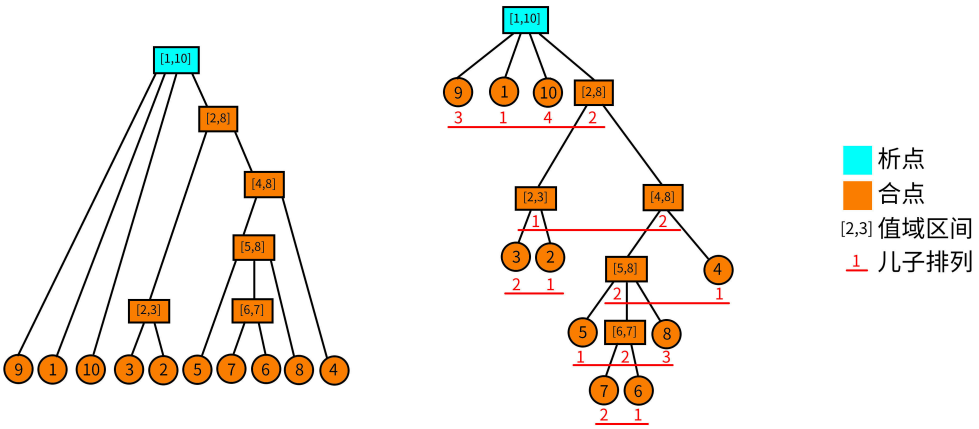
其实这个定义全称叫作 **本原连续段**. 但笔者认为本原段更为简洁.

对于排列 P，我们认为一个本原段 M 表示在集合 I_P 中，不存在与之相交且不包含的连续段. 形式化地定义，我们认为 $X\in I_P$ 且满足 $\forall A\in I_P,X\cap A=(P,\emptyset)\vee X\subseteq A\vee A\subseteq X$.

所有本原段的集合为 M_P. 显而易见，$(P,\emptyset)\in M_P$.

显然，本原段之间只有相离或者包含关系. 并且你发现 **一个连续段可以由几个互不相交的本原段构成**. 最大的本原段就是整个排列本身，它包含了其他所有本原段，因此我们认为本原段可以构成一个树形结构，我们称这个结构为 **析合树**. 更严格地说，排列 P 的析合树由排列 P 的 **所有本原段** 组成.

前面干讲这么多的定义，不来点图怎么行. 考虑排列 $P=\{9,1,10,3,2,5,7,6,8,4\}$. 它的本原段构成的析合树如下:



在图中我们没有标明本原段. 而图中 **每个结点都代表一个本原段**. 我们只标明了每个本原段的值域. 举个例子, 结点 $[5,8]$ 代表的本原段就是 $(P,[6,9])=\{5,7,6,8\}$. 于是这里就有一个问题: **什么是析点合点?**

析点与合点

这里我们直接给出定义，稍候再来讨论它的正确性.

1. **值域区间**：对于一个结点 u，用 $[u_l,u_r]$ 表示该结点的值域区间.
2. **儿子序列**：对于析合树上的一个结点 u，假设它的儿子结点是一个 **有序** 序列，该序列是以值域区间为元素的（单个的数 x 可以理解为 $[x,x]$ 的区间）. 我们把这个序列称为儿子序列. 记作 S_u.
3. **儿子排列**：对于一个儿子序列 S_u，把它的元素离散化成正整数后形成的排列称为儿子排列. 举个例子，对于结点 $[5,8]$，它的儿子序列为 $\{[5,5],[6,7],[8,8]\}$，那么把区间排序标个号，则它的儿子排列就为 $\{1,2,3\}$；类似的，结点 $[4,8]$ 的儿子排列为 $\{2,1\}$. 结点 u 的儿子排列记为 P_u.
4. **合点**：我们认为，儿子排列为顺序或者逆序的点为合点. 形式化地说，满足 $P_u=\{1,2,\cdots,|\ S_u|\}$ 或者 $P_u=\{|\ S_u|,|\ S_u-1\ |,\cdots,1\}$ 的点称为合点. **叶子结点没有儿子排列，我们也认为它是合点.**
5. **析点**：不是合点的就是析点.

从图中可以看到，只有 $[1,10]$ 不是合点. 因为 $[1,10]$ 的儿子排列是 $\{3,1,4,2\}$.

析点与合点的性质

析点与合点的命名来源于他们的性质. 首先我们有一个非常显然的性质：对于析合树中任何的结点 u，其儿子序列区间的并集就是结点 u 的值域区间. 即 $\bigcup_{i=1}^{|\ S_u|}S_u[i]=[u_l,u_r]$.

对于一个合点 u：其儿子序列的任意 **子区间** 都构成一个 **连续段**. 形式化地说，$\forall S_u[l\sim r]$，有 $\bigcup_{i=l}^{r}S_u[i]\in I_P$.

对于一个析点 u：其儿子序列的任意 **长度大于 1（这里的长度是指儿子序列中的元素数，不是下标区间的长度）** 的子区间都 **不** 构成一个 **连续段**. 形式化地说，$\forall S_u[l\sim r],l<r$，有 $\bigcup_{i=l}^{r}S_u[i]\notin I_P$.

合点的性质不难证明. 因为合点的儿子排列要么是顺序，要么是倒序，而值域区间也是首位相接，因此只要是连续的一段子序列（区间）都是一个连续段.

对于析点的性质可能很多读者就不太能理解了：为什么 **任意** 长度大于 1 的子区间都不构成连续段?

使用反证法. 假设对于一个点 u，它的儿子序列中有一个 **最长的** 区间 $S_u[l\sim r]$ 构成了连续段. 那么这个 $A=\bigcup_{i=l}^{r}S_u[i]\in I_P$，也就意味着 A 是一个本原段！（因为 A 是儿子序列中最长的，因此找不到一个与它相交又不包含的连续段）于是你就没有使用所有的本原段构成这个析合树. 矛盾.

析合树的构造

对于具体构造析合树，LCA 提供了一种线性构造算法[1]，下面给出一种比较好懂的 $O(n\log n)$ 算法.

增量法

我们考虑增量法. 用一个栈维护前 $i-1$ 个元素构成的析合森林. 在这里需要 **着重强调**，析合森林的意思是，在任何时候，栈中结点要么是析点要么是合点. 现在考虑当前结点 P_i.

1. 我们先判断它能否成为栈顶结点的儿子，如果能就变成栈顶的儿子，然后把栈顶取出，作为当前结点. 重复上述过程直到栈空或者不能成为栈顶结点的儿子.
2. 如果不能成为栈顶的儿子，就看能不能把栈顶的若干个连续的结点都合并成一个结点（判断能否合并的方法在后面），把合并后的点，作为当前结点.
3. 重复上述过程直到不能进行为止. 然后结束此次增量，直接把当前结点压栈.

接下来我们仔细解释一下.

具体的策略

我们认为，如果当前点能够成为栈顶结点的儿子，那么栈顶结点是一个合点. 如果是析点，那么你合并后这个析点就存在一个子连续段，不满足析点的性质. 因此一定是合点.

[1]刘承奥. 简单的连续段数据结构. WC2019 营员交流.

如果无法成为栈顶结点的儿子，那么我们就看栈顶连续的若干个点能否与当前点一起合并．设 l 为当前点所在区间的左端点．我们计算 L_i 表示右端点下标为 i 的连续段中，左端点 $< l$ 的最大值．当前结点为 P_i，栈顶结点记为 t．

1. 如果 L_i 不存在，那么显然当前结点无法合并；
2. 如果 $t_l = L_i$，那么这就是两个结点合并，合并后就是一个 **合点**；
3. 否则在栈中一定存在一个点 t' 的左端点 $t'_l = L_i$，那么一定可以从当前结点合并到 t' 形成一个 **析点**；

判断能否合并

最后，我们考虑如何处理 L_i．事实上，一个连续段 $(P, [l, r])$ 等价于区间极差与区间长度 -1 相等．即

$$\max_{l\le i\le r} P_i - \min_{l\le i\le r} P_i = r - l$$

而且由于 P 是一个排列，因此对于任意的区间 $[l, r]$ 都有

$$\max_{l\le i\le r} P_i - \min_{l\le i\le r} P_i \ge r - l$$

于是我们就维护 $\max_{l\le i\le r} P_i - \min_{l\le i\le r} P_i - (r - l)$，那么要找到一个连续段相当于查询一个最小值！

有了上述思路，不难想到这样的算法．对于增量过程中的当前的 i，我们维护一个数组 Q 表示区间 $[j, i]$ 的极差减长度．即

$$Q_j = \max_{j\le k\le i} P_k - \min_{j\le k\le i} P_k - (i - j),\ 0 < j < i$$

现在我们想知道在 $1 \sim i - 1$ 中是否存在一个最小的 j 使得 $Q_j = 0$．这等价于求 $Q_{1\sim i-1}$ 的最小值．求得最小的 j 就是 L_i．如果没有，那么 $L_i = i$．

但是当第 i 次增量结束时，我们需要快速把 Q 数组更新到 i+1 的情况．原本的区间从 $[j, i]$ 变成 $[j, i + 1]$，如果 $P_{i+1} > \max$ 或者 $P_{i+1} < \min$ 都会造成 Q_j 发生变化．如何变化？如果 $P_{i+1} > \max$，相当于我们把 Q_j 先减掉 $\max$ 再加上 P_{i+1} 就完成了 Q_j 的更新；$P_{i+1} < \min$ 同理，相当于 $Q_j = Q_j + \min - P_{i+1}$．

那么如果对于一个区间 $[x, y]$，满足 $P_{x\sim i}, P_{x+1\sim i}, P_{x+2\sim i}, \cdots, P_{y\sim i}$ 的区间 $\max$ 都相同呢？你已经发现了，那么相当于我们在做一个区间加的操作；同理，当 $P_{x\sim i}, P_{x+1\sim i}, \cdots, P_{y\sim i}$ 的区间 $\min$ 都想同时也是一个区间加的操作．同时，$\max$ 和 $\min$ 的更新是相互独立的，因此可以各自更新．

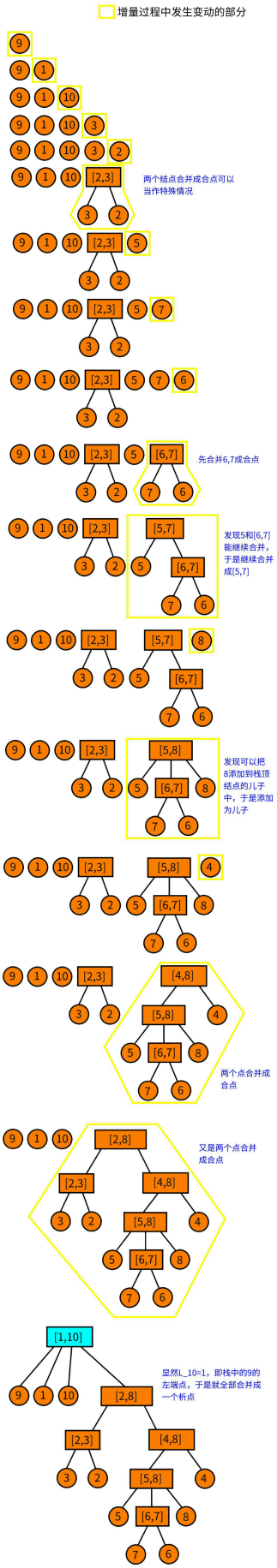
因此我们对 Q 的维护可以这样描述：

1. 找到最大的 j 使得 $P_j > P_{i+1}$，那么显然，$P_{j+1\sim i}$ 这一段数全部小于 P_{i+1}，于是就需要更新 $Q_{j+1\sim i}$ 的最大值．由于 $P_i, \max(P_i, P_{i-1}), \max(P_i, P_{i-1}, P_{i-2}), \cdots, \max(P_i, P_{i-1}, \cdots, P_{j+1})$ 是（非严格）单调递增的，因此可以每一段相同的 $\max$ 做相同的更新，即区间加操作．
2. 更新 $\min$ 同理．
3. 把每一个 Q_j 都减 1．因为区间长度加 1．
4. 查询 L_i：即查询 Q 的最小值的所在的 **下标**．

没错，我们可以使用线段树维护 Q！现在还有一个问题：怎么找到相同的一段使得他们的 $\max$ / $\min$ 都相同？使用单调栈维护！维护两个单调栈分别表示 $\max$ / $\min$．那么显然，栈中以相邻两个元素为端点的区间的 $\max$ / $\min$ 是相同的，于是在维护单调栈的时候顺便更新线段树即可．

具体的维护方法见代码．

讲这么多干巴巴的想必小伙伴也听得云里雾里的，那么我们就先上图吧．长图警告！


增量过程中发生变动的部分
两个结点合并成合点可以
当作特殊情况
先合并6,7成合点
发现5和[6,7]
能继续合并，
于是继续合并
成[5,7]
发现可以把
8添加到栈顶
结点的儿子
中，于是添加
为儿子
两个点合并成
合点
又是两个点合并
成合点
显然L_10=1，即栈中的9的
左端点，于是就全部合并成
一个析点

实现

最后放一个实现的代码供参考. 代码转自 大米饼的博客[1], 添加了一些注释.

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
constexpr int N = 200010;

int n, m, a[N], st1[N], st2[N], tp1, tp2, rt;
int L[N], R[N], M[N], id[N], cnt, typ[N], bin[20], st[N], tp;

// 本篇代码原题应为 CERC2017 Intrinsic Interval
// a 数组即为原题中对应的排列
// st1 和 st2 分别两个单调栈，tp1、tp2 为对应的栈顶，rt 为析合树的根
// L、R 数组表示该析合树节点的左右端点，M 数组的作用在析合树构造时有提到
// id 存储的是排列中某一位置对应的节点编号，typ 用于标记析点还是合点
// st 为存储析合树节点编号的栈，tp为其栈顶
struct RMQ {  // 预处理 RMQ（Max & Min）
  int lg[N], mn[N][17], mx[N][17];

  void chkmn(int& x, int y) {
    if (x > y) x = y;
  }

  void chkmx(int& x, int y) {
    if (x < y) x = y;
  }

  void build() {
    for (int i = bin[0] = 1; i < 20; ++i) bin[i] = bin[i - 1] << 1;
    for (int i = 2; i <= n; ++i) lg[i] = lg[i >> 1] + 1;
    for (int i = 1; i <= n; ++i) mn[i][0] = mx[i][0] = a[i];
    for (int i = 1; i < 17; ++i)
      for (int j = 1; j + bin[i] - 1 <= n; ++j)
        mn[j][i] = min(mn[j][i - 1], mn[j + bin[i - 1]][i - 1]),
        mx[j][i] = max(mx[j][i - 1], mx[j + bin[i - 1]][i - 1]);
  }

  int ask_mn(int l, int r) {
    int t = lg[r - l + 1];
    return min(mn[l][t], mn[r - bin[t] + 1][t]);
  }

  int ask_mx(int l, int r) {
    int t = lg[r - l + 1];
    return max(mx[l][t], mx[r - bin[t] + 1][t]);
  }
} D;

// 维护 L_i

struct SEG {  // 线段树
#define ls (k << 1)
#define rs (k << 1 | 1)
  int mn[N << 1], ly[N << 1];  // 区间加；区间最小值
```

```
53
54    void pushup(int k) { mn[k] = min(mn[ls], mn[rs]); }
55
56    void mfy(int k, int v) { mn[k] += v, ly[k] += v; }
57
58    void pushdown(int k) {
59      if (ly[k]) mfy(ls, ly[k]), mfy(rs, ly[k]), ly[k] = 0;
60    }
61
62    void update(int k, int l, int r, int x, int y, int v) {
63      if (l == x && r == y) {
64        mfy(k, v);
65        return;
66      }
67      pushdown(k);
68      int mid = (l + r) >> 1;
69      if (y <= mid)
70        update(ls, l, mid, x, y, v);
71      else if (x > mid)
72        update(rs, mid + 1, r, x, y, v);
73      else
74        update(ls, l, mid, x, mid, v), update(rs, mid + 1, r, mid + 1, y, v);
75      pushup(k);
76    }
77
78    int query(int k, int l, int r) {  // 询问 0 的位置
79      if (l == r) return l;
80      pushdown(k);
81      int mid = (l + r) >> 1;
82      if (!mn[ls])
83        return query(ls, l, mid);
84      else
85        return query(rs, mid + 1, r);
86      // 如果不存在 0 的位置就会自动返回当前你查询的位置
87    }
88  } T;
89
90  int o = 1, hd[N], dep[N], fa[N][18];
91
92  struct Edge {
93    int v, nt;
94  } E[N << 1];
95
96  void add(int u, int v) {  // 树结构加边
97    E[o] = Edge{v, hd[u]};
98    hd[u] = o++;
99  }
100
101 void dfs(int u) {
102   for (int i = 1; bin[i] <= dep[u]; ++i) fa[u][i] = fa[fa[u][i - 1]][i - 1];
103   for (int i = hd[u]; i; i = E[i].nt) {
104     int v = E[i].v;
105     dep[v] = dep[u] + 1;
106     fa[v][0] = u;
```

```
107     dfs(v);
108   }
109 }
110
111 int go(int u, int d) {
112   for (int i = 0; i < 18 && d; ++i)
113     if (bin[i] & d) d ^= bin[i], u = fa[u][i];
114   return u;
115 }
116
117 int lca(int u, int v) {
118   if (dep[u] < dep[v]) swap(u, v);
119   u = go(u, dep[u] - dep[v]);
120   if (u == v) return u;
121   for (int i = 17; ~i; --i)
122     if (fa[u][i] != fa[v][i]) u = fa[u][i], v = fa[v][i];
123   return fa[u][0];
124 }
125
126 // 判断当前区间是否为连续段
127 bool judge(int l, int r) { return D.ask_mx(l, r) - D.ask_mn(l, r) == r - l; }
128
129 // 建树
130 void build() {
131   for (int i = 1; i <= n; ++i) {
132     // 单调栈
133     // 在区间 [st1[tp1-1]+1,st1[tp1]] 的最小值就是 a[st1[tp1]]
134     // 现在把它出栈，意味着要把多减掉的 Min 加回来.
135     // 线段树的叶结点位置 j 维护的是从 j 到当前的 i 的
136     // Max{j,i}-Min{j,i}-(i-j)
137     // 区间加只是一个 Tag.
138     // 维护单调栈的目的是辅助线段树从 i-1 更新到 i.
139     // 更新到 i 后，只需要查询全局最小值即可知道是否有解
140
141     while (tp1 && a[i] <= a[st1[tp1]])  // 单调递增的栈，维护 Min
142       T.update(1, 1, n, st1[tp1 - 1] + 1, st1[tp1], a[st1[tp1]]), tp1--;
143     while (tp2 && a[i] >= a[st2[tp2]])
144       T.update(1, 1, n, st2[tp2 - 1] + 1, st2[tp2], -a[st2[tp2]]), tp2--;
145
146     T.update(1, 1, n, st1[tp1] + 1, i, -a[i]);
147     st1[++tp1] = i;
148     T.update(1, 1, n, st2[tp2] + 1, i, a[i]);
149     st2[++tp2] = i;
150
151     id[i] = ++cnt;
152     L[cnt] = R[cnt] = i;  // 这里的 L,R 是指节点所对应区间的左右端点
153     int le = T.query(1, 1, n), now = cnt;
154     while (tp && L[st[tp]] >= le) {
155       if (typ[st[tp]] && judge(M[st[tp]], i)) {
156         // 判断是否能成为儿子，如果能就做
157         R[st[tp]] = i, M[st[tp]] = L[now], add(st[tp], now), now = st[tp--];
158       } else if (judge(L[st[tp]], i)) {
159         typ[++cnt] = 1;  // 合点一定是被这样建出来的
160         L[cnt] = L[st[tp]], R[cnt] = i, M[cnt] = L[now];
```

```
        // 这里M数组是记录节点最右面的儿子的左端点，用于上方能否成为儿子的判断
        add(cnt, st[tp--]), add(cnt, now);
        now = cnt;
      } else {
        add(++cnt, now);  // 新建一个结点，把 now 添加为儿子
        // 如果从当前结点开始不能构成连续段，就合并.
        // 直到找到一个结点能构成连续段. 而且我们一定能找到这样
        // 一个结点.
        do add(cnt, st[tp--]);
        while (tp && !judge(L[st[tp]], i));
        L[cnt] = L[st[tp]], R[cnt] = i, add(cnt, st[tp--]);
        now = cnt;
      }
    }
    st[++tp] = now;  // 增量结束，把当前点压栈

    T.update(1, 1, n, 1, i, -1);  // 因为区间右端点向后移动一格，因此整体 -1
  }

  rt = st[1];  // 栈中最后剩下的点是根结点
}

// 分 lca 为析或和，这里把叶子看成析的
void query(int l, int r) {
  int x = id[l], y = id[r];
  int z = lca(x, y);
  if (typ[z] & 1)
    l = L[go(x, dep[x] - dep[z] - 1)], r = R[go(y, dep[y] - dep[z] - 1)];
  // 合点这里特判的原因是因为这个合点不一定是最小的包含l，r的连续段.
  // 因为合点所代表的区间的子区间也都是连续段，而我们只需要其中的一段就够了.
  else
    l = L[z], r = R[z];
  printf("%d %d\n", l, r);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
  D.build();
  build();
  dfs(rt);
  scanf("%d", &m);
  for (int i = 1; i <= m; ++i) {
    int x, y;
    scanf("%d%d", &x, &y);
    query(x, y);
  }
  return 0;
}

// 20190612
// 析合树
```

参考文献与链接

大米饼的博客 -【学习笔记】析合树[2]

[1] 大米饼的博客

[2] 大米饼的博客 -【学习笔记】析合树

10.22 PQ 树

Authors: isdanni,xyf007

PQ 树是一种基于树的数据结构，代表一组元素上的一系列排列，由 Kellogg S. Booth 和 George S. Lueker 于 1976 年发现命名，用来解决以下问题

> 给出 m 个集合 S_i，你要找到一个 $1 \sim n$ 的排列，使得每个集合内的元素都相邻.

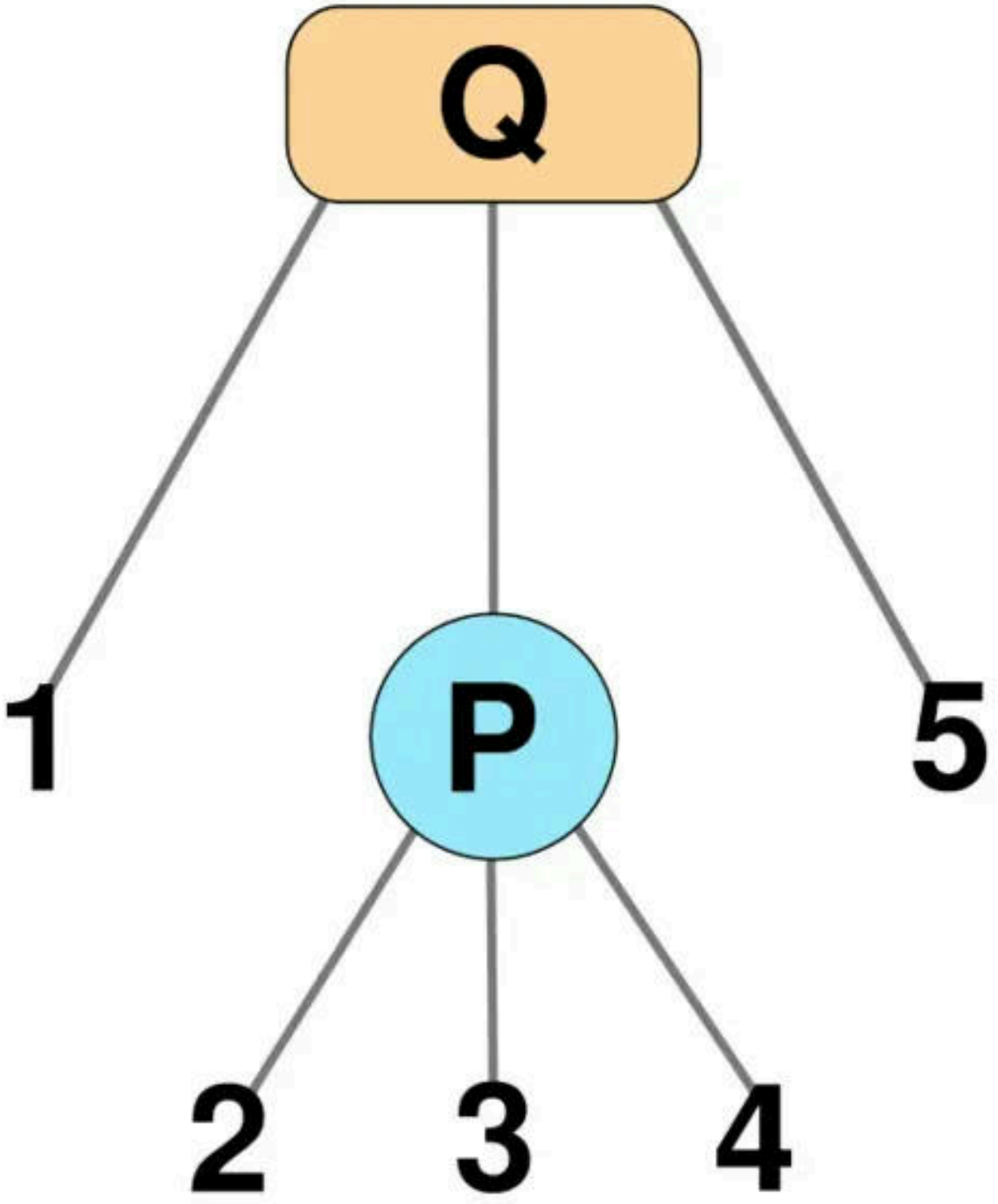
PQ 树可以在 $O(n+\sum|S_i|)$ 时间内构建．本文中介绍的构建方法时间复杂度为 $O(nm)$.

定义

PQ 树有三种结点：**叶子结点**、**P 结点** 和 **Q 结点**．其中叶子结点代表排列中的一个元素，P 结点表示它的子结点可以任意排列，Q 结点表示它的儿子顺序可以反转．所有非叶子结点都是 P 结点或 Q 结点中的一种．P 结点至少有 2 个儿子，Q 结点至少有 3 个儿子.

由于结点的定义，PQ 树本身代表了 **所有的** 合法方案，其先序遍历就是其中之一.

下图是一棵 PQ 树.



其先序遍历 1,2,3,4,5 代表了一个合法方案．如果 P 结点的儿子重排列为 4,2,3，我们得到了另一个合法方案 1,4,2,3,5．保持 P 结点儿子顺序不变，Q 结点的儿子顺序反转，得到了另一个合法方案 5,3,2,4,1.

构建

PQ 树使用儿子 - 兄弟表示法．

我们增量构建一棵 PQ 树．

首先建立一棵树，其根为 P，总共 n 个儿子，分别是 $1, 2, ..., n$，代表没有任何限制时的 PQ 树．随着限制的不断加入，我们不断修改这棵树．

当加入一个新的限制集合 S 时，我们把所有属于这个集合的叶子结点标记为 **黑色**，不在这个集合内的叶子结点标记为 **白色**. 对于所有非叶子结点，如果其所有儿子均为黑色，将其也标记为黑色；如果其所有儿子均为白色，将其也标记为白色；否则将其标记为 **灰色**. 在下面的图中，黑色结点、白色结点、灰色结点分别用黑色、灰色、一半黑一半灰来表示.

我们要求 PQ 树中的结点按照颜色排序.

自底向上法

包含所有黑色结点的最小子树被称为 **相关子树**, 相关子树的根（不一定是整棵树的根）被称为 **相关根**.

添加一个限制的过程被称为 reduction. 一次 reduction 分为两个阶段：冒泡阶段和减少阶段.

冒泡阶段

冒泡阶段只处理相关子树. 我们将相关子树中的所有结点标记为黑色或灰色, 并为每个结点计算其拥有的相关子结点数量. 为了高效地完成这个过程，我们从叶子往根处理相关子树. 这需要记录每个点的父亲结点，但在减少阶段一个点的父亲结点经常要被修改. 为了在线性时间内构造，只有 P 结点的儿子和 Q **结点的最后一个儿子** 始终记录正确的父亲结点. 对于 Q 结点的其他儿子，在冒泡阶段用最后一个儿子的父亲更新他们的父亲.

当遇到一个中间的结点时，我们看一下它的兄弟是否已经有合法的父亲结点. 如果没有，将其标记为 **阻塞** 的. 如果后面它的兄弟有了合法的父亲，那么修改这个结点的父亲并且取消标记. 如果在冒泡阶段结束时，仍然有一段连续的阻塞结点（如下面的情况 Q3），一个没有父结点的「伪结点」成为该块的父结点，并在减少阶段时被去除.

减少阶段

减少阶段用一个队列来处理结点. 首先将所有限制内的叶子结点加入队列. 每次取出队首的结点 u 并处理. 如果 u 的父亲也是相关子树内的结点，那么将 fa_u 入队.

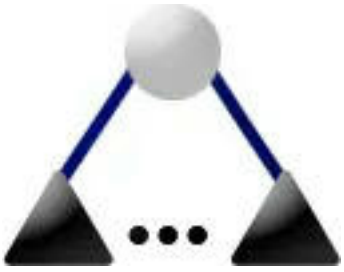
对于每一个结点 u，我们分情况讨论. 如果不属于其中任何一种情况，则无解.

叶子结点

将 u 标记为黑色.

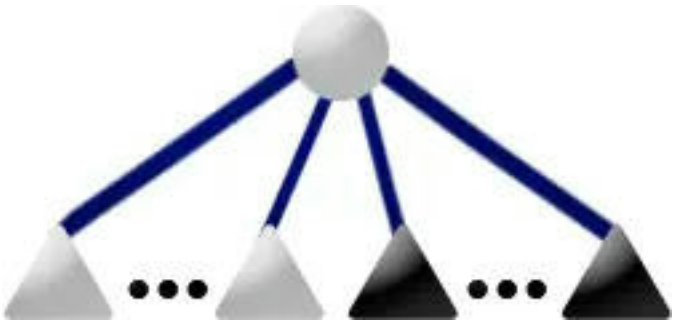
P 结点

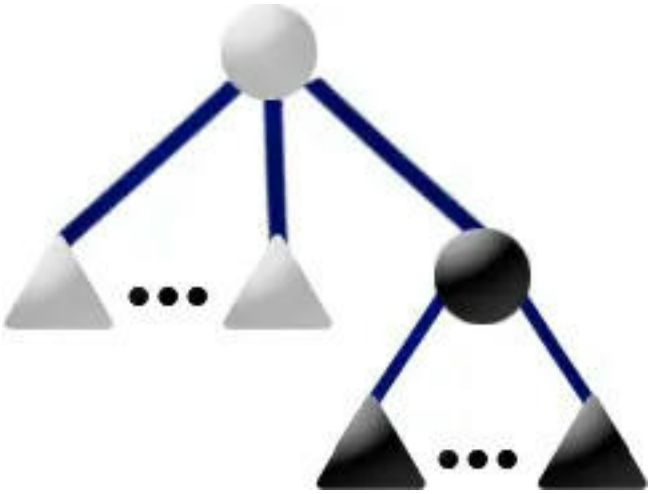
如果所有儿子均为黑色，将 u 标记为黑色.





如果 u 有黑色儿子和白色儿子，且 u 是相关根，那么新建一个 P 结点 v 成为它所有黑色儿子的根.

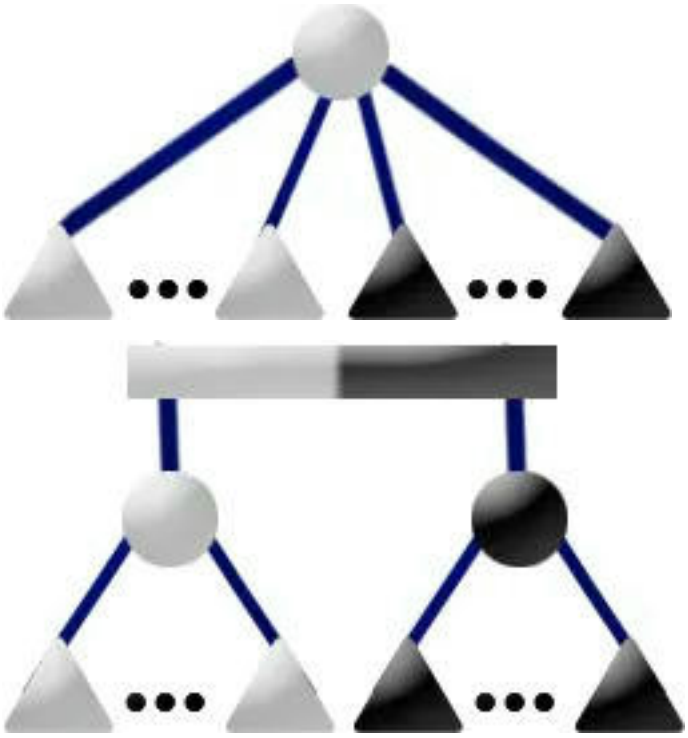




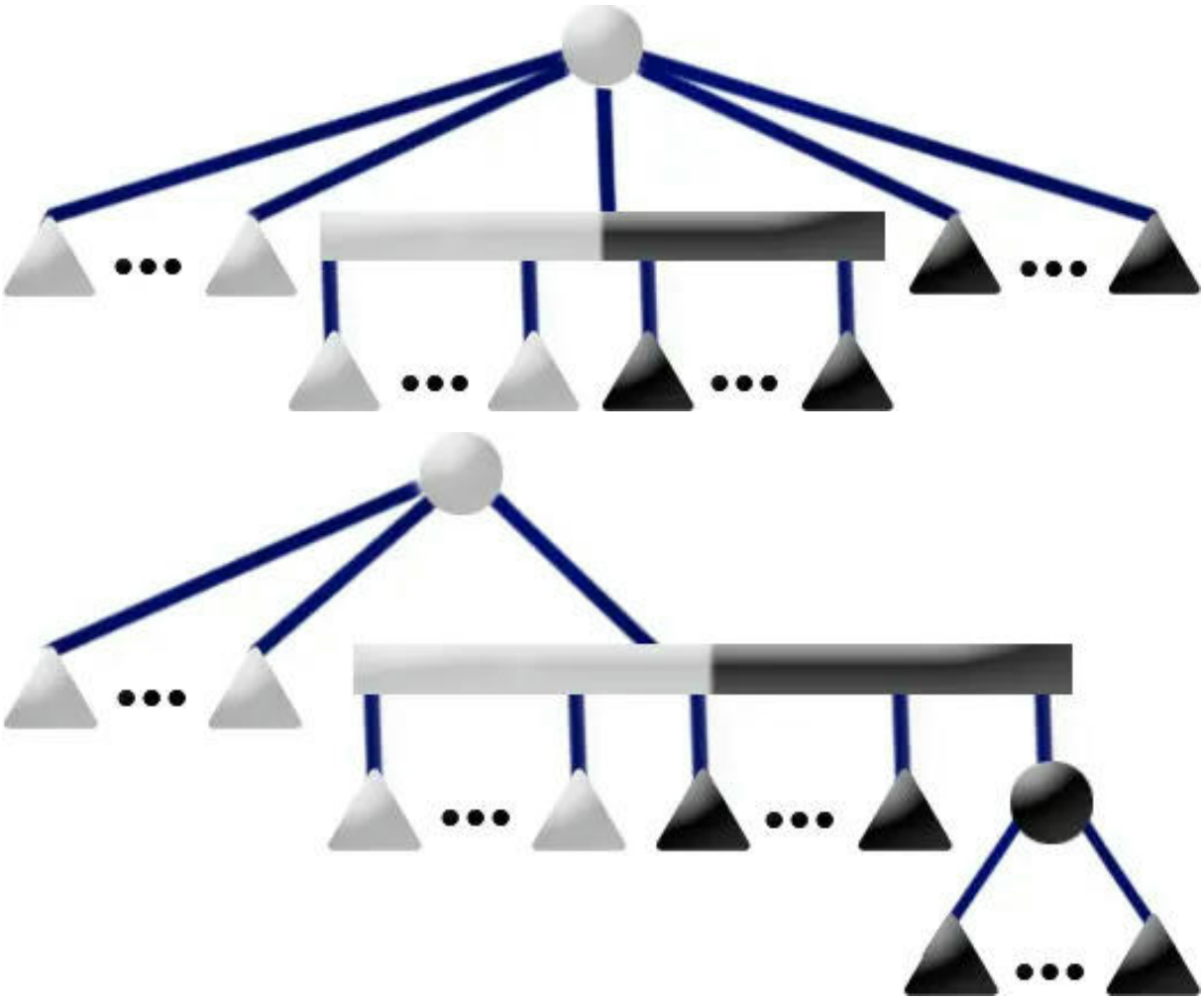
如果 u 有黑色儿子和白色儿子，且 u 不是相关根，那么做以下操作：

- 新建一个 P 结点 f 成为所有黑色儿子的根.
- 新建一个 P 结点 e 成为所有白色儿子的根.
- 如果 e（和/或 f）只有一个儿子，那么不要新建结点，而是将 e（和/或 f）直接赋值成那个儿子.
- 将 u 改成 Q 结点并把其儿子设为 e 和 f，将其标记为灰色.

注意到根据之前的定义，Q 结点至少有 3 个儿子，因此这里的 u 被视为一个「伪结点」，并且将在后面被继续处理.

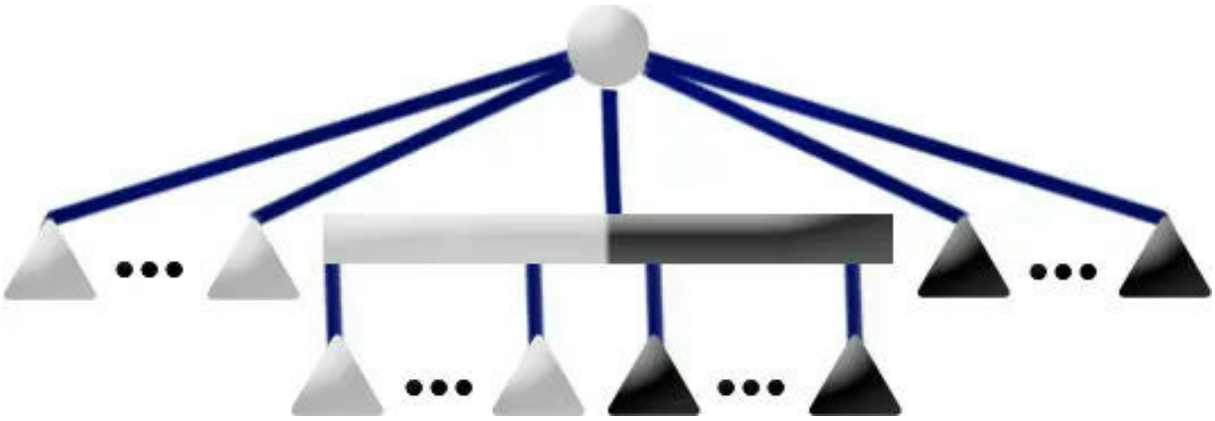


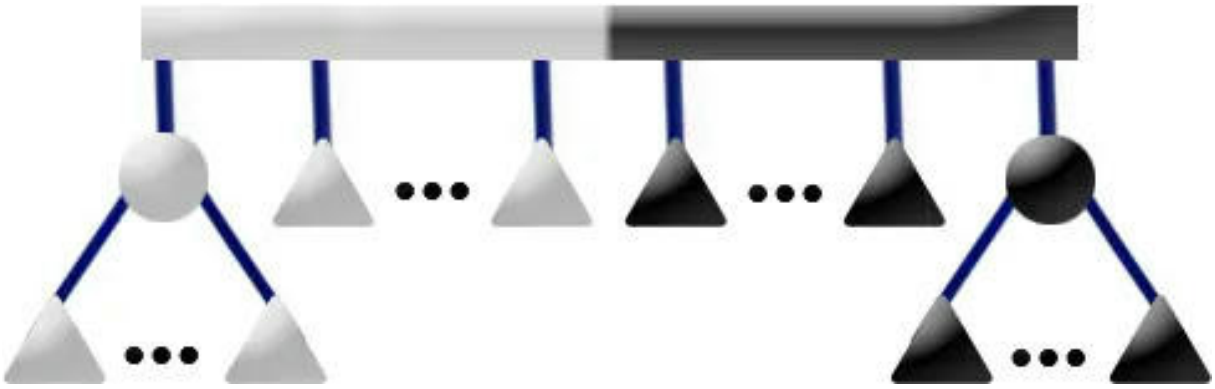
如果 u 有一个灰色儿子 p，且 u 是相关根，那么新建一个 P 结点 v 作为其所有黑色儿子的根，将 v 的兄弟设为 p 最后一个一个黑色儿子，然后把 v 设为 p 的最后一个儿子.



如果 u 有一个灰色儿子 p，且 u 不是相关根，那么进行以下操作：

- 新建一个 P 结点 f 成为所有黑色儿子的根.
- 新建一个 P 结点 e 成为所有白色儿子的根.
- 如果 e（和/或 f）只有一个儿子，那么不要新建结点，而是将 e（和/或 f）直接赋值成那个儿子.
- 将 e 的兄弟设为 p 最后一个白色儿子，然后把 e 设为 p 的最后一个儿子.
- 将 f 的兄弟设为 p 最后一个黑色儿子，然后把 f 设为 p 的最后一个儿子.

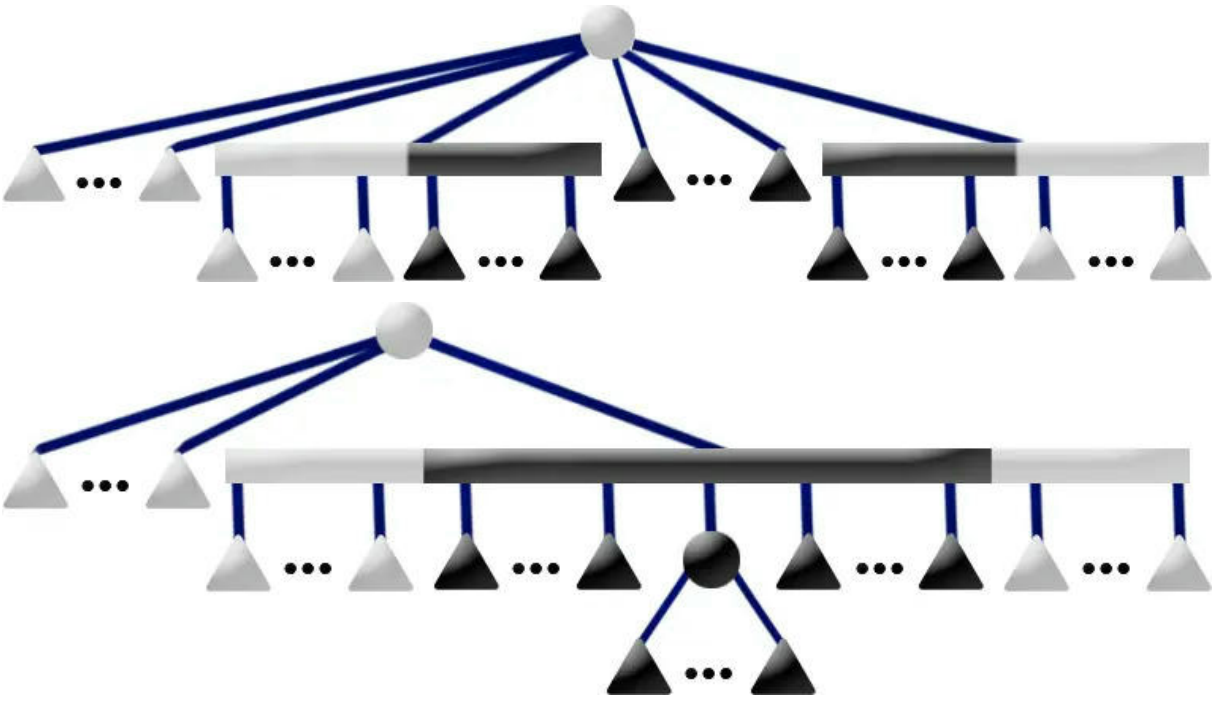




如果 u 恰有两个灰色儿子 p_1, p_2，那么进行以下操作：

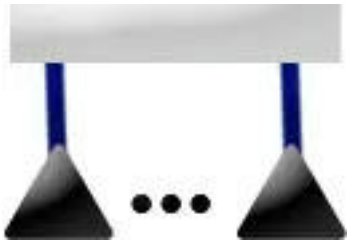
- 新建一个 P 结点 f 成为所有黑色儿子的根.
- 如果 f 只有一个儿子，那么不要新建结点，而是将 f 直接赋值成那个儿子.
- 把 p_1 的最后一个黑色儿子的兄弟设为 f.
- 把 f 的兄弟设为 p_2 的最后一个黑色儿子.
- 把 p_2 的最后一个儿子设为 p_2 的最后一个白色儿子.

可以发现这样 p_2 就被合并进了 p_1.



Q 结点

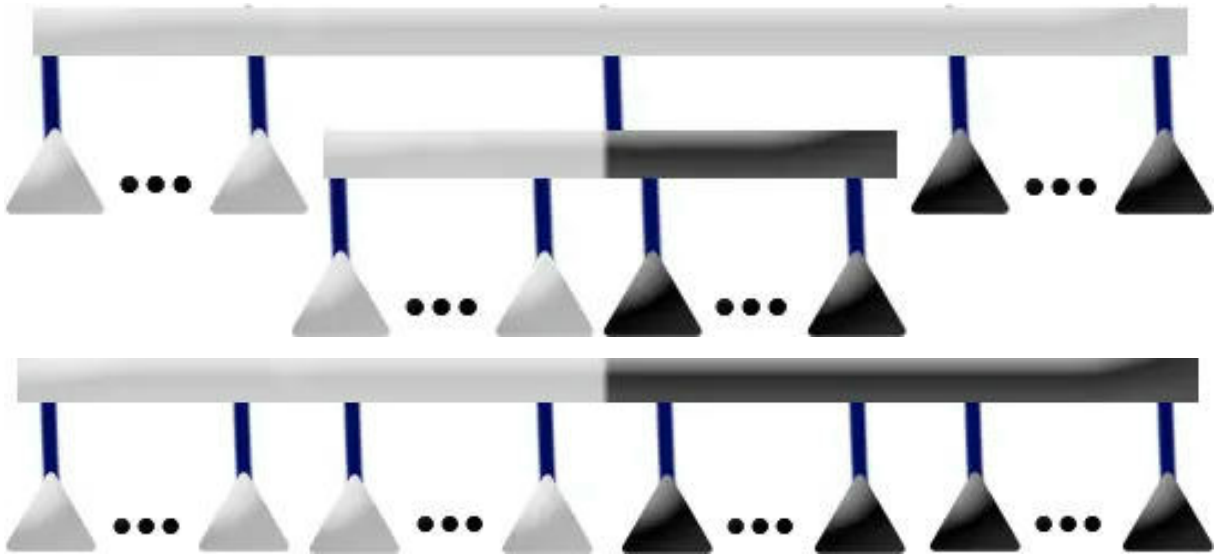
如果 u 只有黑色儿子，那么将 u 标记成黑色.（下面的图的形状错了.）



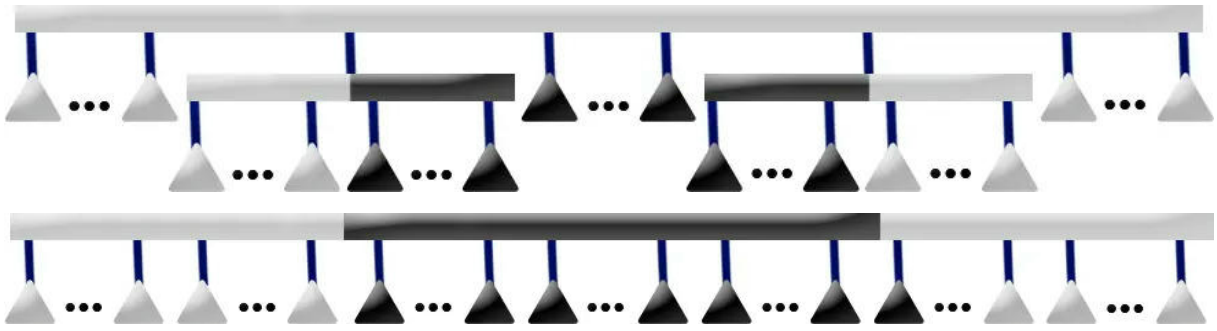


如果 u 有一个灰色儿子 p，且所有标记相同的儿子均连续出现，那么进行如下操作：

- 设 p_f 为 p 最后一个黑色儿子，p_e 为 p 最后一个白色儿子，f 为 p 的黑色兄弟，e 为 p 的白色兄弟.
- 将 f 的兄弟设为 p_f，e 的兄弟设为 p_e.
- 如果 p 没有一个白色兄弟或黑色兄弟，将 u 的最后一个儿子设成 p 的最后一个儿子.
- 删除 p.



如果 u 恰有两个灰色儿子 p_1, p_2，且所有标记相同的儿子均连续出现，那么对 p_1, p_2 都进行上一种操作即可.



该构建方法是原论文中的，但是实现较为不便.

自顶向下法

目前 OI 中的实现大多采用该方法. 其实方法类似，下面出现的情况基本都能在上面找到.

注意到根据之前的染色过程，所有黑色和白色的点都已经满足条件，因此我们 **只需要处理灰色结点**.

P 结点

- 如果 u 有多于两个灰色儿子，无解.
- 如果 u 只有一个灰色儿子，且没有黑色儿子，递归处理灰色儿子.

- 否则先清空 u 的儿子，然后加入所有的白色儿子．新建一个 Q 结点 q_1 并成为 u 的儿子．在 q_1 中加入所有的灰色儿子．新建一个 P 结点 p 作为所有黑色儿子的根，将 p 插入 q_1 的中间．（对应了自底向上法 P 结点的所有情况．）

注意到我们会要求两个灰色节点白色全在左侧，黑色全在右侧（或相反），因此我们需要实现一个分裂函数 `split`，可以把这个子树的点分裂成黑白部分，并同时保留分裂成的子树的节点的 **所有可能**．

Q 结点

- 找到最左边和最右边的非白色节点位置 l, r．如果 $[l+1, r-1]$ 内有非黑色节点，无解．
- 如果没有黑色节点，只有一个灰色节点，递归处理这个灰色节点，否则只需要将 l 和 r 位置的节点分裂．

分裂函数

令要分裂的点为 u，我们想把 u 分裂成左边全是白色，右边全是黑色的森林．如果 u 不是灰色结点则直接返回子树．只考虑灰色结点的情况． 如果 u 是 P 类结点：

- 如果 u 有至少两个灰色儿子，则无解．
- 否则左边是所有白色儿子，中间递归处理灰色儿子，右边是所有黑色儿子．注意到要保留所有的可能，因此要新建两个 P 结点分别作为白色儿子和黑色儿子的根．（对应自底向上法的 P4 情况．）
- 删除 u．

如果 u 是 Q 类结点：

- 如果正序和反序都不满足白 - 灰 - 黑，则无解．
- 如果有至少两个灰色儿子，也无解．
- 否则递归分裂灰色儿子即可．
- 删除 u．

最后把所有多余的结点（只有一个儿子的结点）删除．

代码实现

```cpp
class PQTree {
 public:
  PQTree() {}

  void Init(int n) {
    n_ = n, rt_ = tot_ = n + 1;
    for (int i = 1; i <= n; i++) g_[rt_].emplace_back(i);
  }

  void Insert(const std::string &s) {
    s_ = s;
    Dfs0(rt_);
    Work(rt_);
    while (g_[rt_].size() == 1) rt_ = g_[rt_][0];
    Remove(rt_);
  }

  std::vector<int> ans() {
    DfsAns(rt_);
    return ans_;
  }

  ~PQTree() {}

```

```
25  private:
26   int n_, rt_, tot_, pool_[100001], top_, typ_[100001] /* 0-P 1-Q */,
27       col_[100001] /* 0-black 1-white 2-grey */;
28   std::vector<int> g_[100001], ans_;
29   std::string s_;
30
31   void Fail() {
32     std::cout << "NO\n";
33     std::exit(0);
34   }
35
36   int NewNode(int ty) {
37     int x = top_ ? pool_[top_--] : ++tot_;
38     typ_[x] = ty;
39     return x;
40   }
41
42   void Delete(int u) { g_[u].clear(), pool_[++top_] = u; }
43
44   void Dfs0(int u) {  // get color of each node
45     if (u >= 1 && u <= n_) {
46       col_[u] = s_[u] == '1';
47       return;
48     }
49     bool c0 = false, c1 = false;
50     for (auto &&v : g_[u]) {
51       Dfs0(v);
52       if (col_[v]) c1 = true;
53       if (col_[v] != 1) c0 = true;
54     }
55     if (c0 && !c1)
56       col_[u] = 0;
57     else if (!c0 && c1)
58       col_[u] = 1;
59     else
60       col_[u] = 2;
61   }
62
63   bool Check(const std::vector<int> &v) {
64     int p2 = -1;
65     for (int i = 0; i < static_cast<int>(v.size()); i++)
66       if (col_[v[i]] == 2) {
67         if (p2 != -1) return false;
68         p2 = i;
69       }
70     if (p2 == -1)
71       for (int i = 0; i < static_cast<int>(v.size()); i++)
72         if (col_[v[i]]) {
73           p2 = i;
74           break;
75         }
76     for (int i = 0; i < p2; i++)
77       if (col_[v[i]]) return false;
78     for (int i = p2 + 1; i < static_cast<int>(v.size()); i++)
```

```
79        if (col_[v[i]] != 1) return false;
80      return true;
81    }
82
83    std::vector<int> Split(int u) {
84      if (col_[u] != 2) return {u};
85      std::vector<int> ng;
86      if (typ_[u]) {  // Q
87        if (!Check(g_[u])) {
88          std::reverse(g_[u].begin(), g_[u].end());
89          if (!Check(g_[u])) Fail();
90        }
91        for (auto &&v : g_[u])
92          if (col_[v] != 2) {
93            ng.emplace_back(v);
94          } else {
95            auto s = Split(v);
96            ng.insert(ng.end(), s.begin(), s.end());
97          }
98      } else {  // P
99        std::vector<int> son[3];
100       for (auto &&x : g_[u]) son[col_[x]].emplace_back(x);
101       if (son[2].size() > 1) Fail();
102       if (!son[0].empty()) {
103         int n0 = NewNode(0);
104         g_[n0] = son[0];
105         ng.emplace_back(n0);
106       }
107       if (!son[2].empty()) {
108         auto s = Split(son[2][0]);
109         ng.insert(ng.end(), s.begin(), s.end());
110       }
111       if (!son[1].empty()) {
112         int n1 = NewNode(0);
113         g_[n1] = son[1];
114         ng.emplace_back(n1);
115       }
116     }
117     Delete(u);
118     return ng;
119   }
120
121   void Work(int u) {
122     if (col_[u] != 2) return;
123     if (typ_[u]) {  // Q
124       int l = 1e9, r = -1e9;
125       for (int i = 0; i < static_cast<int>(g_[u].size()); i++)
126         if (col_[g_[u][i]]) checkmin(l, i), checkmax(r, i);
127       for (int i = l + 1; i < r; i++)
128         if (col_[g_[u][i]] != 1) Fail();
129       if (l == r && col_[g_[u][l]] == 2) {
130         Work(g_[u][l]);
131         return;
132       }
```

```
      std::vector<int> ng;
      for (int i = 0; i < l; i++) ng.emplace_back(g_[u][i]);
      auto s = Split(g_[u][l]);
      ng.insert(ng.end(), s.begin(), s.end());
      for (int i = l + 1; i < r; i++) ng.emplace_back(g_[u][i]);
      if (l != r) {
        s = Split(g_[u][r]);
        std::reverse(s.begin(), s.end());
        ng.insert(ng.end(), s.begin(), s.end());
      }
      for (int i = r + 1; i < static_cast<int>(g_[u].size()); i++)
        ng.emplace_back(g_[u][i]);
      g_[u] = ng;
    } else {  // P
      std::vector<int> son[3];
      for (auto &&x : g_[u]) son[col_[x]].emplace_back(x);
      if (son[1].empty() && son[2].size() == 1) {
        Work(son[2][0]);
        return;
      }
      g_[u].clear();
      if (son[2].size() > 2) Fail();
      g_[u] = son[0];
      int n1 = NewNode(1);
      g_[u].emplace_back(n1);
      if (son[2].size() >= 1) {
        auto s = Split(son[2][0]);
        g_[n1].insert(g_[n1].end(), s.begin(), s.end());
      }
      if (son[1].size()) {
        int n2 = NewNode(0);
        g_[n1].emplace_back(n2);
        g_[n2] = son[1];
      }
      if (son[2].size() >= 2) {
        auto s = Split(son[2][1]);
        std::reverse(s.begin(), s.end());
        g_[n1].insert(g_[n1].end(), s.begin(), s.end());
      }
    }
  }

  void Remove(int u) {  // remove the nodes with only one child
    for (auto &&v : g_[u]) {
      int tv = v;
      while (g_[tv].size() == 1) {
        int t = tv;
        tv = g_[tv][0];
        Delete(t);
      }
      v = tv, Remove(v);
    }
  }

```

```
187   void DfsAns(int u) {
188     if (u >= 1 && u <= n_) {
189       ans_.emplace_back(u);
190       return;
191     }
192     for (auto &&v : g_[u]) DfsAns(v);
193   }
194 } T;
```

习题

- CF243E Matrix[1]
- CF1552I Organizing a Music Festival[2]

参考资料

- Booth, Kellogg S. & Lueker, George S. (1976)."Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms"[3].*Journal of Computer and System Sciences*[4].13(3): 335–379.doi[5]:10.1016/S0022-0000(76)80045-1[6].
- PQ Tree Algorithm and Consecutive Ones Problem[7]
- CF243E Matrix PQTree - RainAir's Blog[8]

外部参考资料

[1] CF243E Matrix

[2] CF1552I Organizing a Music Festival

[3] "Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms"

[4] Journal of Computer and System Sciences

[5] doi

[6] 10.1016/S0022-0000(76)80045-1

[7] PQ Tree Algorithm and Consecutive Ones Problem

[8] CF243E Matrix PQTree - RainAir's Blog

10.23 手指树

Authors: isdanni

> ⚠ **注意**
>
> 此章是选读内容，在阅读前请确定你对函数式编程（Functional Programming）有一定了解.

简介

手指树（Finger Tree）是一种 **纯函数式** 数据结构，由 Ralf Hinze 和 Ross Paterson 提出.

为什么需要手指树

在函数式编程中, 列表是十分常见的数据类型. 对于基于序列的操作, 包括在两端添加和删除元素（双端队列操作），在任意节点插入、连接、删除，查找某个满足要求的元素，将序列拆分为子序列，几乎所有的函数型语言都支持. 但是对于高效的更多操作，这些语言很难做到. 即使有相对应的实现，通常也都非常复杂，实际很难使用.

而指状树提供了一种纯函数式的序列数据结构，它可以在均摊常量时间（amortized constant time）内完成访问，添加到序列的前端和末尾等操作，以及在对数时间（logarithmic time）内完成串联和随机访问.

除了良好的渐近运行时边界外，手指树还非常灵活：当与元素上的幺半群标记（monoidal tag[1]）结合时，指状树可用于实现高效的随机访问序列、有序序列、间隔树和优先级队列.

基本结构

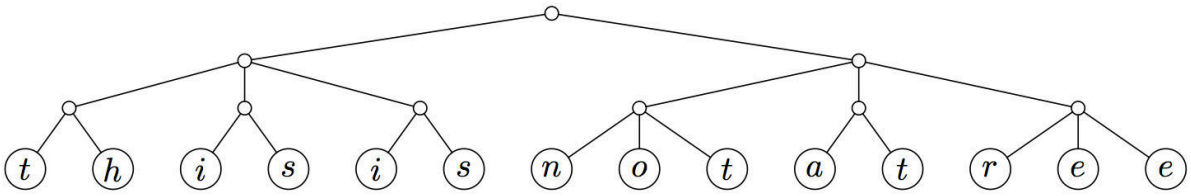
手指树在树的「手指」（叶子）的地方存储数据，访问时间为分摊常量. 手指是一个可以访问部分数据结构的点. 在命令式语言（imperative language）中，这被称做指针. 在手指树中，「手指」是指向序列末端或叶节点的结构. 手指树还在每个内部节点中存储对其后代应用一些关联操作的结果. 存储在内部节点中的数据可用于提供除树类数据结构之外的功能.

1. 手指树的深度由下到上计算.
2. 手指树的第一级，即树的叶节点，仅包含值，深度为 0. 第二级为深度 1. 第三级为深度 2，依此类推.
3. 离根越近，节点指向的原始树（在它是手指树之前的树）的子树越深. 这样，沿着树向下工作就是从叶子到树的根，这与典型的树数据结构相反. 为了获得这种的结构，我们必须确保原始树具有统一的深度. 在声明节点对象时，必须通过子节点的类型进行参数化. 深度为 1 及以上的脊椎上的节点指向树，通过这种参数化，它们可以由嵌套节点表示.

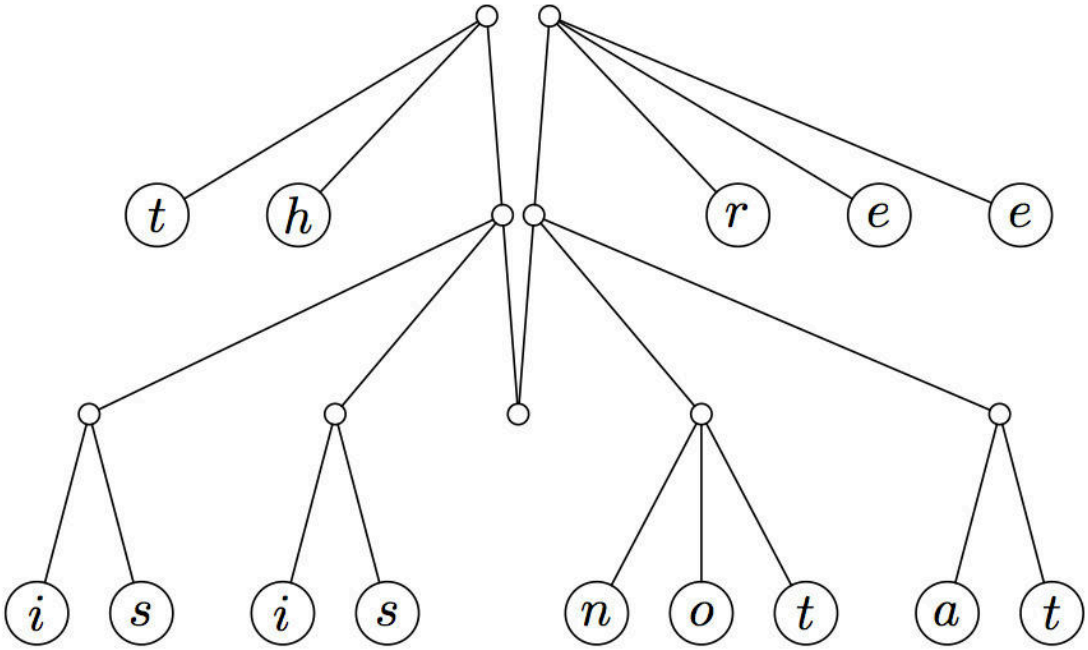
将一棵树变成手指树

> **注释**
>
> **2-3 树** 是一种树状数据结构，其中每个带有子节点（内部节点）的节点具有两个子节点（2 节点）和一个数据元素或三个子节点（3 节点）和两个数据元素. 2-3 树是 3 阶 B 树. 树外部的节点（叶节点）没有子节点和一两个数据元素.

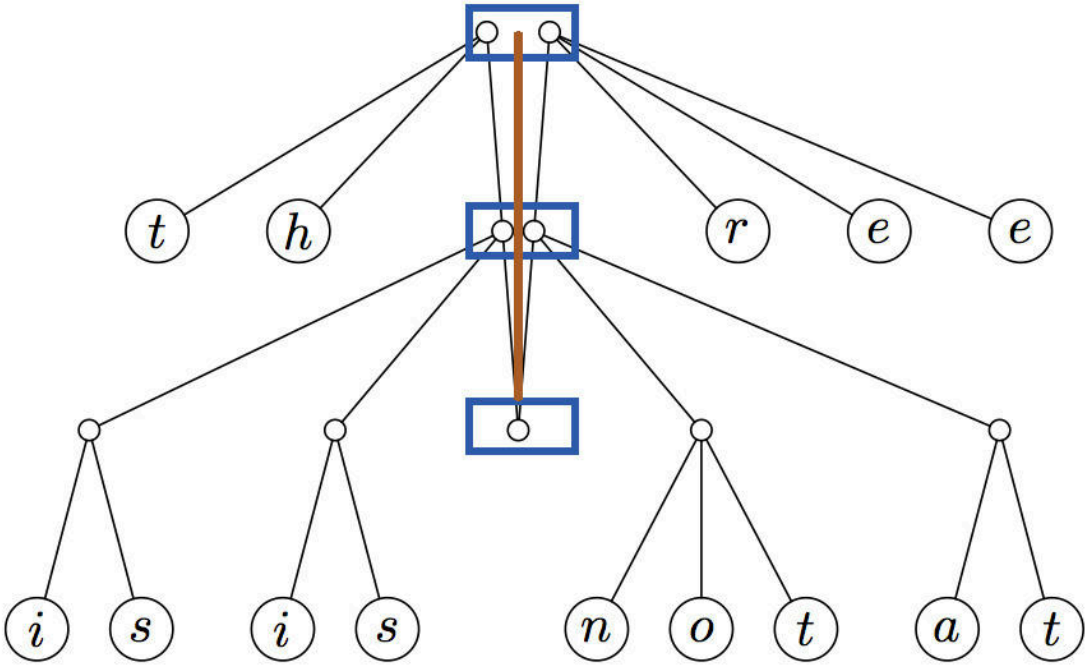
我们将从平衡 2-3 树开始这个过程. 为了使手指树正常工作，所有的叶节点需要是水平的. 如下图所示（图片取自手指树论文）：



手指是「一种结构，可以有效地访问靠近特定位置的树的节点.」要制作手指树，我们需要将手指放在树的左右两端，取树的最左边和最右边的内部节点并将它们拉起来，使树的其余部分悬在它们之间，这为我们提供了对序列末尾的均摊常量访问时间.



这种新的数据结构被称为手指树．手指树由沿其树脊（棕色线）分布的几层（下方蓝色框）组成：




```haskell
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)

data Digit a = One a | Two a a | Three a a a | Four a a a a
data Node a = Node2 a a | Node3 a a a
```

示例中的数字是带有字母的节点．每个列表由树脊上每个节点的前缀或后缀划分．在转换后的 2-3 树中，顶层的数字列表似乎可以有两个或三个长度，而较低级别的长度只有一或两个．为了使手指树的某些

应用程序能够如此高效地运行，手指树允许在每个级别上有 1 到 4 个子树．手指树的数字可以转换成一个列表，如：

```
type Digit a = One a | Two a a | Three a a a | Four a a a a
```

顶层具有类型 a 的元素，下一层具有类型节点 a 的元素，因为树脊和叶子之间的节点，这通常意味着树的第 n 层具有元素类型为 $Node^n\ a$，或 2-3 个深度为 n 的树．这意味着 n 个元素的序列由深度为 `θ(log n)` 的树表示．距离最近端 d 的元素存储在树中 `θ(log d)` 深度处．

双向队列操作

指状树也可以制作高效的双向队列．无论结构是否持久，所有操作都需要 `θ(1)` 时间．它可以被看作是的隐式双端队列的扩展[1]：

1. 用 2-3 个节点替换对提供了足够的灵活性来支持有效的串联．(为了保持恒定时间的双端队列操作，必须将 Digit 扩展为四．)
2. 用幺半群（monoid）注释内部节点允许有效的分裂．

```
data ImplicitDeque a = Empty
                     | Single a
                     | Deep (Digit a) (ImplicitDeque (a, a)) (Digit a)

data Digit a = One a | Two a a | Three a a a
```

时间复杂度

手指树提供了对树的「手指」（叶子）的分摊常量时间访问，这是存储数据的地方，以及在较小部分的大小中连接和拆分对数时间．它还在每个内部节点中存储对其后代应用一些关联操作的结果．存储在内部节点中的「摘要」数据可用于提供除树之外的数据结构的功能．

操作	手指树	注释 2-3 树 (annotated 2-3 tree)	列表（list）	向量（vector）
`cons,snoc`	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(n)$
`viewl,viewr`	$O(1)$	$O(\log n)$	$O(1)/O(n)$	$O(1)$
`measure/length`	$O(1)$	$O(1)$	$O(n)$	$O(1)$
`append`	$O(\log \min(l1, l2))$	$O(\log n)$	$O(n)$	$O(m+n)$
`split`	$O(\log \min(n, l-n))$	$O(\log n)$	$O(n)$	$O(1)$
`replicate`	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$
`fromList,toList,reverse`	$O(l)/O(l)/O(l)$	$O(l)$	$O(1)/O(1)/O(n)$	$O(n)$
`index`	$O(\log \min(n, l-n))$	$O(\log n)$	$O(n)$	$O(1)$

应用

指状树可用于建造其他树．例如，优先级队列可以通过树中子节点的最小优先级标记内部节点来实现，或者索引列表/数组可以通过节点的子节点中叶子的计数来标记节点来实现．其他应用包括随机访问序列（如下所述）、有序序列和区间树．

手指树可以提供平均 $O(1)$ 的推、反转、弹出，$O(\log n)$ 追加和拆分；并且可以适应索引或排序序列．和所有函数式数据结构一样，它本质上是持久的；也就是说，始终保留旧版本的树．

[1] Purely Functional Data Structures[2], Chris Okasaki (1999)

对于代码实现，Haskell 核心库中的有限序列 `Seq` 的实现使用了 2-3 手指树（Data.Sequence[3]），OCaml 中 `BatFingerTree` 模块的 实现[4] 也使用了通用手指树数据结构．手指树可以使用或不使用惰性求值来实现，但惰性允许更简单的实现．

参考资料与拓展阅读

1. Ralf Hinze and Ross Paterson, "Finger trees: a simple general-purpose data structure[5]", Journal of Functional Programming 16:2 (2006) pp 197-217.
2. Finger Tree - Wikipedia[6]

[1] monoidal tag

[2] Purely Functional Data Structures

[3] Data.Sequence

[4] 实现

[5] Finger trees: a simple general-purpose data structure

[6] Finger Tree - Wikipedia

10.24 霍夫曼树

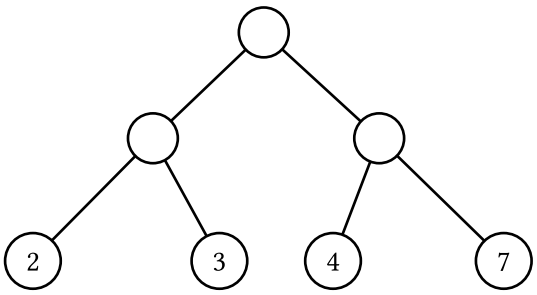
Authors: Alex-McAvoy, lingkerio, LvCGame

树的带权路径长度

设二叉树具有 n 个带权叶结点，从根结点到各叶结点的路径长度与相应叶节点权值的乘积之和称为**树的带权路径长度（Weighted Path Length of Tree，WPL）**．

设 w_i 为二叉树第 i 个叶结点的权值，l_i 为从根结点到第 i 个叶结点的路径长度，则 WPL 计算公式如下：

$$WPL=\sum_{i=1}^{n}w_il_i$$



如上图所示，其 WPL 计算过程与结果如下：

$$WPL=2*2+3*2+4*2+7*2=4+6+8+14=32$$

结构

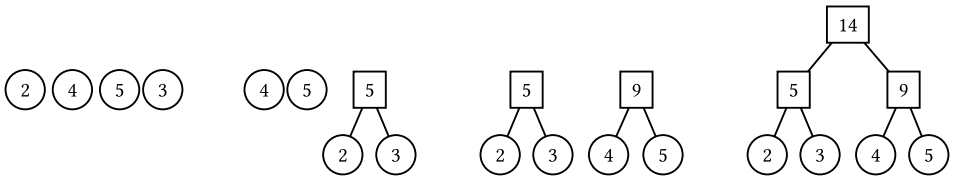
对于给定一组具有确定权值的叶结点，可以构造出不同的二叉树，其中，**WPL 最小的二叉树** 称为 **霍夫曼树（Huffman Tree）**.

对于霍夫曼树来说，其叶结点权值越小，离根越远，叶结点权值越大，离根越近，此外其仅有叶结点的度为 0，其他结点度均为 2.

霍夫曼算法

霍夫曼算法用于构造一棵霍夫曼树，算法步骤如下：

1. **初始化**：由给定的 n 个权值构造 n 棵只有一个根节点的二叉树，得到一个二叉树集合 F.
2. **选取与合并**：从二叉树集合 F 中选取根节点权值 **最小的两棵** 二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根节点的权值为其左、右子树根结点的权值和.
3. **删除与加入**：从 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中.
4. 重复 2、3 步，当集合中只剩下一棵二叉树时，这棵二叉树就是霍夫曼树.



正确性证明

引理

最优前缀编码树（Huffman 树）中的权值最小的两个叶结点总是最深的叶结点，并且将这两个结点调整为兄弟结点至少不会破坏编码树的最优性.

证明

我们采用反证法来证明该命题. 假设在一棵最优前缀编码树中，存在两个权值最小的叶结点，它们不是最深的叶结点. 设这两个结点为 a 和 b，且它们的深度小于某个最深的叶结点. 对于这个最深的叶结点 c，我们可以将 a 和 c 交换位置，或将 b 和 c 交换位置. 由于 Huffman 算法保证树的每一层按权值最小的叶结点合并，因此在交换后，树的带权路径长度（WPL）将减少. 由此矛盾可得出假设不成立，因此权值最小的两个叶结点必须是最深的叶结点.

接下来，假设这两个权值最小的叶结点分别为 a 和 b，它们的深度相同. 如果在一棵最优前缀编码树中这两个结点不是兄弟结点，假设存在其他结点 c 和 d 与 a 和 b 分别是兄弟结点（假设 a 和 c 是兄弟结点，b 和 d 是兄弟结点）. 我们可以将 a 和 b 合并为一个子树.

- 如果 a 和 b 合并后的权值之和小于 c 或 d 的权值，那么我们可以将合并后的子树与权值较大的结点（如 c 或 d）合并，形成新的子树，WPL 会减少.

- 如果 a 和 b 的权值之和不小于 c 和 d 的权值，我们可以直接将 a 和 b 调整为兄弟结点，c 和 d 作为另一个兄弟结点，WPL 不会增加.

因此，经过这样的调整，最优性不会被破坏，得证.

定理

Huffman 算法得到的前缀编码树是最优前缀编码树.

证明

我们使用数学归纳法来证明该定理.

- **基本情况**: 当字母数 $n = 2$ 时，显然，直接将两个字母合并成一棵树即为最优编码树.
- **归纳假设**: 假设对于字母数 $n = k$（$k \geq 2$）时，Huffman 算法能够得到最优前缀编码树.
- **归纳步骤**: 对于字母数 $n = k + 1$，我们从 $k + 1$ 个字母中选出两个权值最小的字母，将它们合并为一棵子树，子树的根作为虚拟字母（虚拟结点）. 根据引理可知，这一操作不会破坏前缀编码树的最优性. 此时，虚拟字母与剩下的 k 个字母一同构成 $k + 1$ 个字母，根据归纳假设，当字母数为 k 时，Huffman 算法能够得到最优前缀编码树.

因此，通过数学归纳法，Huffman 算法对于任意字母数 n 都能够得到最优前缀编码树，得证.

霍夫曼编码

在进行程序设计时，通常给每一个字符标记一个单独的代码来表示一组字符，即 **编码**.

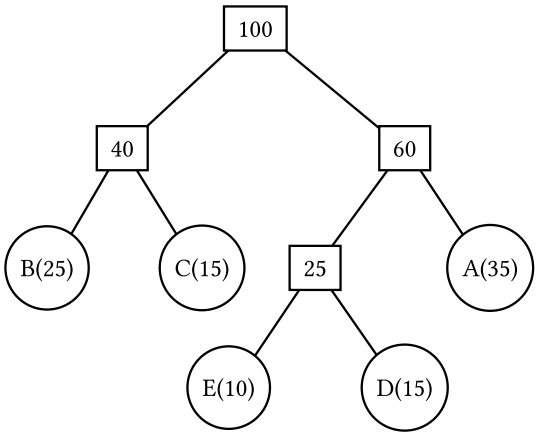
在进行二进制编码时，假设所有的代码都等长，那么表示 n 个不同的字符需要 $\lceil \log_2 n \rceil$ 位，称为 **等长编码**.

如果每个字符的 **使用频率相等**, 那么等长编码无疑是空间效率最高的编码方法, 而如果字符出现的频率不同, 则可以让频率高的字符采用尽可能短的编码, 频率低的字符采用尽可能长的编码, 来构造出一种 **不等长编码**，从而获得更好的空间效率.

在设计不等长编码时, 要考虑解码的唯一性, 如果一组编码中任一编码都不是其他任何一个编码的前缀，那么称这组编码为 **前缀编码**，其保证了编码被解码时的唯一性.

霍夫曼树可用于构造 **最短的前缀编码**，即 **霍夫曼编码（Huffman Code）**，其构造步骤如下:

1. 设需要编码的字符集为：$d_1, d_2, ..., d_n$，他们在字符串中出现的频率为：$w_1, w_2, ..., w_n$.
2. 以 $d_1, d_2, ..., d_n$ 作为叶结点，$w_1, w_2, ..., w_n$ 作为叶结点的权值，构造一棵霍夫曼树.
3. 规定霍夫曼编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶结点所经过的路径组成的 0、1 序列即为该叶结点对应字符的编码.



字符	频率	编码
A	35	11
B	25	00
C	15	01
D	15	101
E	10	100

示例代码

霍夫曼树的构建

```
struct HNode {
  int weight;
  HNode *lchild, *rchild;
};

using Htree = HNode *;

Htree createHuffmanTree(int arr[], int n) {
  Htree forest[N];
  Htree root = NULL;
  for (int i = 0; i < n; i++) {  // 将所有点存入森林
    Htree temp;
    temp = (Htree)malloc(sizeof(HNode));
    temp->weight = arr[i];
    temp->lchild = temp->rchild = NULL;
    forest[i] = temp;
  }

  for (int i = 1; i < n; i++) {  // n-1 次循环建霍夫曼树
    int minn = -1, minnSub;  // minn 为最小值树根下标，minnsub 为次小值树根下标
    for (int j = 0; j < n; j++) {
      if (forest[j] != NULL && minn == -1) {
        minn = j;
        continue;
      }
      if (forest[j] != NULL) {
```

```
27          minnSub = j;
28          break;
29        }
30      }
31
32      for (int j = minnSub; j < n; j++) {  // 根据 minn 与 minnSub 赋值
33        if (forest[j] != NULL) {
34          if (forest[j]->weight < forest[minn]->weight) {
35            minnSub = minn;
36            minn = j;
37          } else if (forest[j]->weight < forest[minnSub]->weight) {
38            minnSub = j;
39          }
40        }
41      }
42
43      // 建新树
44      root = (Htree)malloc(sizeof(HNode));
45      root->weight = forest[minn]->weight + forest[minnSub]->weight;
46      root->lchild = forest[minn];
47      root->rchild = forest[minnSub];
48
49      forest[minn] = root;      // 指向新树的指针赋给 minn 位置
50      forest[minnSub] = NULL;   // minnSub 位置为空
51    }
52    return root;
53  }
```

计算构成霍夫曼树的 WPL

```
1  struct HNode {
2    int weight;
3    HNode *lchild, *rchild;
4  };
5
6  using Htree = HNode *;
7
8  int getWPL(Htree root, int len) {  // 递归实现，对于已经建好的霍夫曼树，求 WPL
9    if (root == NULL)
10     return 0;
11   else {
12     if (root->lchild == NULL && root->rchild == NULL)  // 叶节点
13       return root->weight * len;
14     else {
15       int left = getWPL(root->lchild, len + 1);
16       int right = getWPL(root->rchild, len + 1);
17       return left + right;
18     }
19   }
20 }
```

对于未建好的霍夫曼树，直接求其 WPL

```cpp
int getWPL(int arr[], int n) {  // 对于未建好的霍夫曼树，直接求其 WPL
  priority_queue<int, vector<int>, greater<int>> huffman;  // 小根堆
  for (int i = 0; i < n; i++) huffman.push(arr[i]);

  int res = 0;
  for (int i = 0; i < n - 1; i++) {
    int x = huffman.top();
    huffman.pop();
    int y = huffman.top();
    huffman.pop();
    int temp = x + y;
    res += temp;
    huffman.push(temp);
  }
  return res;
}
```

对于给定序列，计算霍夫曼编码

```cpp
struct HNode {
  int weight;
  HNode *lchild, *rchild;
};

using Htree = HNode *;

void huffmanCoding(Htree root, int len, int arr[]) {  // 计算霍夫曼编码
  if (root != NULL) {
    if (root->lchild == NULL && root->rchild == NULL) {
      printf("结点为 %d 的字符的编码为: ", root->weight);
      for (int i = 0; i < len; i++) printf("%d", arr[i]);
      printf("\n");
    } else {
      arr[len] = 0;
      huffmanCoding(root->lchild, len + 1, arr);
      arr[len] = 1;
      huffmanCoding(root->rchild, len + 1, arr);
    }
  }
}
```

第十一章

图论

11.1 图论部分简介

图论 (Graph theory) 是数学的一个分支，图是图论的主要研究对象．**图** (Graph) 是由若干给定的顶点及连接两顶点的边所构成的图形，这种图形通常用来描述某些事物之间的某种特定关系．顶点用于代表事物，连接两顶点的边则用于表示两个事物间具有这种关系．

参考资料

图论 - 维基百科，自由的百科全书[1]

外部参考资料

[1] 图论 - 维基百科，自由的百科全书

11.2 图论相关概念

本页面概述了图论中的一些概念，这些概念并不全是在 OI 中常见的，对于 OIer 来说，只需掌握本页面中的基础部分即可，如果在学习中碰到了不懂的概念，可以再来查阅．

> **Warning**
>
> 图论相关定义在不同教材中往往会有所不同，遇到的时候需根据上下文加以判断．

图

图 (graph) 是一个二元组 $G=(V(G),E(G))$．其中 $V(G)$ 是非空集，称为 **点集** (vertex set)，对于 V 中的每个元素，我们称其为 **顶点** (vertex) 或 **节点** (node)，简称 **点**；$E(G)$ 为 $V(G)$ 各结点之间边的集合，称为 **边集** (edge set)．

常用 $G=(V,E)$ 表示图．

当 V,E 都是有限集合时，称 G 为 **有限图**．

当 V 或 E 是无限集合时，称 G 为 **无限图**．

图有多种，包括 **无向图** (undirected graph)，**有向图** (directed graph)，**混合图** (mixed graph) 等．

若 G 为无向图，则 E 中的每个元素为一个无序二元组 (u,v)，称作 **无向边** (undirected edge)，简称 **边** (edge)，其中 $u,v\in V$．设 $e=(u,v)$，则 u 和 v 称为 e 的 **端点** (endpoint)．

若 G 为有向图，则 E 中的每一个元素为一个有序二元组 (u,v)，有时也写作 $u\to v$，称作 **有向边** (directed edge) 或 **弧** (arc)，在不引起混淆的情况下也可以称作 **边** (edge)．设 $e=u\to v$，则此时 u 称为 e 的 **起点** (tail)，v 称为 e 的 **终点** (head)，起点和终点也称为 e 的 **端点** (endpoint)．并称 u 是 v 的直接前驱，v 是 u 的直接后继．

> **为什么起点是 tail，终点是 head?**
>
> 边通常用箭头表示，而箭头是从「尾」指向「头」的．

若 G 为混合图，则 E 中既有 **有向边**，又有 **无向边**．

若 G 的每条边 $e_k=(u_k,v_k)$ 都被赋予一个数作为该边的 **权**，则称 G 为 **赋权图**．如果这些权都是正实数，就称 G 为 **正权图**．

图 G 的点数 $|V(G)|$ 也被称作图 G 的 **阶** (order)．

形象地说，图是由若干点以及连接点与点的边构成的．

相邻

在无向图 $G=(V,E)$ 中，若点 v 是边 e 的一个端点，则称 v 和 e 是 **关联的 (incident)** 或 **相邻的 (adjacent)**. 对于两顶点 u 和 v，若存在边 (u,v)，则称 u 和 v 是 **相邻的 (adjacent)**.

一个顶点 $v\in V$ 的 **邻域 (neighborhood)** 是所有与之相邻的顶点所构成的集合，记作 $N(v)$.

一个点集 S 的邻域是所有与 S 中至少一个点相邻的点所构成的集合，记作 $N(S)$，即：

$$N(S)=\bigcup_{v\in S}N(v)$$

简单图

自环 (loop)：对 E 中的边 $e=(u,v)$，若 $u=v$，则 e 被称作一个自环.

重边 (multiple edge)：若 E 中存在两个完全相同的元素（边）e_1,e_2，则它们被称作（一组）重边.

简单图 (simple graph)：若一个图中没有自环和重边，它被称为简单图. 具有至少两个顶点的简单无向图中一定存在度相同的结点.（鸽巢原理→9.13.2@p1618）

如果一张图中有自环或重边，则称它为 **多重图 (multigraph)**.

> ⚠ **Warning**
>
> 在无向图中 (u,v) 和 (v,u) 算一组重边，而在有向图中，$u\to v$ 和 $v\to u$ 不为重边.

> ⚠ **Warning**
>
> 在题目中，如果没有特殊说明，是可以存在自环和重边的，在做题时需特殊考虑.

度数

与一个顶点 v 关联的边的条数称作该顶点的 **度 (degree)**，记作 $d(v)$. 特别地，对于边 (v,v)，则每条这样的边要对 $d(v)$ 产生 2 的贡献.

对于无向简单图，有 $d(v)=|N(v)|$.

握手定理（又称图论基本定理）：对于任何无向图 $G=(V,E)$，有 $\sum_{v\in V}d(v)=2|E|$.

推论：在任意图中，度数为奇数的点必然有偶数个.

若 $d(v)=0$，则称 v 为 **孤立点 (isolated vertex)**.

若 $d(v)=1$，则称 v 为 **叶节点 (leaf vertex)/悬挂点 (pendant vertex)**.

若 $2\mid d(v)$，则称 v 为 **偶点 (even vertex)**.

若 $2\nmid d(v)$，则称 v 为 **奇点 (odd vertex)**. 图中奇点的个数是偶数.

若 $d(v)=|V|-1$，则称 v 为 **支配点 (universal vertex)**.

对一张图，所有节点的度数的最小值称为 G 的 **最小度 (minimum degree)**，记作 $\delta(G)$；最大值称为 **最大度 (maximum degree)**，记作 $\Delta(G)$. 即：$\delta(G)=\min_{v\in G}d(v)$，$\Delta(G)=\max_{v\in G}d(v)$.

在有向图 $G=(V,E)$ 中，以一个顶点 v 为起点的边的条数称为该顶点的 **出度 (out-degree)**，记作 $d^+(v)$. 以一个顶点 v 为终点的边的条数称为该节点的 **入度 (in-degree)**，记作 $d^-(v)$. 显然 $d^+(v)+d^-(v)=d(v)$.

对于任何有向图 $G=(V,E)$，有：

$$\sum_{v\in V}d^+(v)=\sum_{v\in V}d^-(v)=|E|$$

若对一张无向图 $G=(V,E)$，每个顶点的度数都是一个固定的常数 k，则称 G 为 k- **正则图** (k-regular graph).

如果给定一个序列 a，可以找到一个图 G，以其为度数列，则称 a 是 **可图化** 的.

如果给定一个序列 a，可以找到一个简单图 G，以其为度数列，则称 a 是 **可简单图化** 的.

路径

途径 (walk)：途径是连接一连串顶点的边的序列，可以为有限或无限长度．形式化地说，一条有限途径 w 是一个边的序列 $e_1,e_2,...,e_k$，使得存在一个顶点序列 $v_0,v_1,...,v_k$ 满足 $e_i=(v_{i-1},v_i)$，其中 $i\in[1,k]$．这样的途径可以简写为 $v_0\to v_1\to v_2\to\cdots\to v_k$．通常来说，边的数量 k 被称作这条途径的 **长度**（如果边是带权的，长度通常指途径上的边权之和，题目中也可能另有定义）.

迹 (trail)：对于一条途径 w，若 $e_1,e_2,...,e_k$ 两两互不相同，则称 w 是一条迹.

路径 (path)（又称 **简单路径** (simple path)）：对于一条迹 w，若其连接的点的序列中点两两不同，则称 w 是一条路径.

回路 (circuit)：对于一条迹 w，若 $v_0=v_k$，则称 w 是一条回路.

环/圈 (cycle)（又称 **简单回路/简单环** (simple circuit)）：对于一条回路 w，若 $v_0=v_k$ 是点序列中唯一重复出现的点对，则称 w 是一个环.

> ⚠ **Warning**
>
> 关于路径的定义在不同地方可能有所不同，如，「路径」可能指本文中的「途径」，「环」可能指本文中的「回路」．如果在题目中看到类似的词汇，且没有「简单路径」/「非简单路径」（即本文中的「途径」）等特殊说明，最好询问一下具体指什么.

子图

对一张图 $G=(V,E)$，若存在另一张图 $H=(V',E')$ 满足 $V'\subseteq V$ 且 $E'\subseteq E$，则称 H 是 G 的 **子图** (subgraph)，记作 $H\subseteq G$.

若对 $H\subseteq G$，满足 $\forall u,v\in V'$，只要 $(u,v)\in E$，均有 $(u,v)\in E'$，则称 H 是 G 的 **导出子图/诱导子图** (induced subgraph).

容易发现，一个图的导出子图仅由子图的点集决定，因此点集为 $V'(V'\subseteq V)$ 的导出子图称为 V' 导出的子图，记作 $G[V']$.

若 $H\subseteq G$ 满足 $V'=V$，则称 H 为 G 的 **生成子图/支撑子图** (spanning subgraph).

显然，G 是自身的子图，支撑子图，导出子图；无边图→11@p2597 是 G 的支撑子图．原图 G 和无边图都是 G 的平凡子图.

如果一张无向图 G 的某个生成子图 F 为 k- 正则图，则称 F 为 G 的一个 k- **因子** (k-factor).

如果有向图 $G=(V,E)$ 的导出子图 $H=G[V^*]$ 满足 $\forall v\in V^*,(v,u)\in E$，有 $u\in V^*$，则称 H 为 G 的一个 **闭合子图** (closed subgraph).

连通

无向图

对于一张无向图 $G=(V,E)$，对于 $u,v\in V$，若存在一条途径使得 $v_0=u,v_k=v$，则称 u 和 v 是 **连通的** (connected)．由定义，任意一个顶点和自身连通，任意一条边的两个端点连通.

若无向图 $G=(V,E)$，满足其中任意两个顶点均连通，则称 G 是 **连通图** (connected graph)，G 的这一性质称作 **连通性** (connectivity).

若 H 是 G 的一个连通子图，且不存在 F 满足 $H \subsetneq F \subseteq G$ 且 F 为连通图，则 H 是 G 的一个 **连通块/连通分量** (connected component)（极大连通子图）.

有向图

对于一张有向图 $G = (V, E)$，对于 $u, v \in V$，若存在一条途径使得 $v_0 = u, v_k = v$，则称 u **可达** v. 由定义，任意一个顶点可达自身，任意一条边的起点可达终点.（无向图中的连通也可以视作双向可达.）

若一张有向图的节点两两互相可达，则称这张图是 **强连通的** (strongly connected).

若一张有向图的边替换为无向边后可以得到一张连通图，则称原来这张有向图是 **弱连通的** (weakly connected).

与连通分量类似，也有 **弱连通分量** (weakly connected component)（极大弱连通子图）和 **强连通分量** (strongly connected component)（极大强连通子图）.

相关算法请参见 强连通分量→11.13.1@p2806.

割

相关算法请参见 割点和桥→11.13.3@p2823 以及 双连通分量→11.13.2@p2813.

在本部分中，有向图的「连通」一般指「强连通」.

对于连通图 $G = (V, E)$，若 $V' \subseteq V$ 且 $G[V \setminus V']$（即从 G 中删去 V' 中的点）不是连通图，则 V' 是图 G 的一个 **点割集** (vertex cut/separating set). 大小为一的点割集又被称作 **割点** (cut vertex).

对于连通图 $G = (V, E)$ 和整数 k，若 $|V| \geq k + 1$ 且 G 不存在大小为 $k - 1$ 的点割集，则称图 G 是 k-**点连通的** (k-vertex-connected)，而使得上式成立的最大的 k 被称作图 G 的 **点连通度** (vertex connectivity)，记作 $\kappa(G)$.（对于非完全图，点连通度即为最小点割集的大小，而完全图 K_n 的点连通度为 $n - 1$.）

对于图 $G = (V, E)$ 以及 $u, v \in V$ 满足 $u \neq v$，u 和 v 不相邻，u 可达 v，若 $V' \subseteq V$，$u, v \notin V'$，且在 $G[V \setminus V']$ 中 u 和 v 不连通，则 V' 被称作 u 到 v 的点割集. u 到 v 的最小点割集的大小被称作 u 到 v 的 **局部点连通度** (local connectivity)，记作 $\kappa(u, v)$.

还可以在边上作类似的定义：

对于连通图 $G = (V, E)$，若 $E' \subseteq E$ 且 $G' = (V, E \setminus E')$（即从 G 中删去 E' 中的边）不是连通图，则 E' 是图 G 的一个 **边割集** (edge cut). 大小为一的边割集又被称作 **桥** (bridge).

对于连通图 $G = (V, E)$ 和整数 k，若 G 不存在大小为 $k - 1$ 的边割集，则称图 G 是 k- **边连通的** (k-edge-connected)，而使得上式成立的最大的 k 被称作图 G 的 **边连通度** (edge connectivity)，记作 $\lambda(G)$.（对于任何图，边连通度即为最小边割集的大小.）

对于图 $G = (V, E)$ 以及 $u, v \in V$ 满足 $u \neq v$，u 可达 v，若 $E' \subseteq E$，且在 $G' = (V, E \setminus E')$ 中 u 和 v 不连通，则 E' 被称作 u 到 v 的边割集. u 到 v 的最小边割集的大小被称作 u 到 v 的 **局部边连通度** (local edge-connectivity)，记作 $\lambda(u, v)$.

点双连通 (biconnected) 几乎与 2- 点连通完全一致，除了一条边连接两个点构成的图，它是点双连通的，但不是 2- 点连通的. 换句话说，没有割点的连通图是点双连通的.

边双连通 (2-edge-connected) 与 2- 边双连通完全一致. 换句话说，没有桥的连通图是边双连通的.

与连通分量类似，也有 **点双连通分量** (biconnected component)（极大点双连通子图）和 **边双连通分量** (2-edge-connected component)（极大边双连通子图）.

Whitney 定理：对任意的图 G，有 $\kappa(G) \leq \lambda(G) \leq \delta(G)$.（不等式中的三项分别为点连通度、边连通度、最小度.）

稀疏图/稠密图

若一张图的边数远小于其点数的平方，那么它是一张 **稀疏图 (sparse graph)**.

若一张图的边数接近其点数的平方，那么它是一张 **稠密图 (dense graph)**.

这两个概念并没有严格的定义，一般用于讨论 时间复杂度→5.2.1@p544 为 $O(|V|^2)$ 的算法与 $O(|E|)$ 的算法的效率差异（在稠密图上这两种算法效率相当，而在稀疏图上 $O(|E|)$ 的算法效率明显更高）.

补图

对于无向简单图 $G=(V,E)$，它的 **补图 (complement graph)** 指的是这样的一张图：记作 $\bar{G}$，满足 $V(\bar{G})=V(G)$，且对任意节点对 (u,v)，$(u,v)\in E(\bar{G})$ 当且仅当 $(u,v)\notin E(G)$.

反图

对于有向图 $G=(V,E)$，它的 **反图 (transpose graph)** 指的是点集不变，每条边反向得到的图，即：若 G 的反图为 $G'=(V,E')$，则 $E'=\{(v,u)|\ (u,v)\in E\}$.

特殊的图

若无向简单图 G 满足任意不同两点间均有边，则称 G 为 **完全图 (complete graph)**，n 阶完全图记作 K_n. 若有向图 G 满足任意不同两点间都有两条方向不同的边，则称 G 为 **有向完全图 (complete digraph)**.

边集为空的图称为 **无边图 (edgeless graph)**、**空图 (empty graph)** 或 **零图 (null graph)**，n 阶无边图记作 $\overline{K_n}$ 或 N_n. N_n 与 K_n 互为补图.

> ⚠ **Warning**
>
> **零图 (null graph)** 也可指 **零阶图 (order-zero graph)** K_0，即点集与边集均为空的图.

若有向简单图 G 满足任意不同两点间都有恰好一条边（单向），则称 G 为 **竞赛图 (tournament graph)**.

若无向简单图 $G=(V,E)$ 的所有边恰好构成一个圈，则称 G 为 **环图/圈图 (cycle graph)**，$n(n\geq 3)$ 阶圈图记作 C_n. 易知，一张图为圈图的充分必要条件是，它是 2- 正则连通图.

若无向简单图 $G=(V,E)$ 满足，存在一个点 v 为支配点，其余点之间没有边相连，则称 G 为 **星图/菊花图 (star graph)**，$n+1(n\geq 1)$ 阶星图记作 S_n.

若无向简单图 $G=(V,E)$ 满足，存在一个点 v 为支配点，其它点之间构成一个圈，则称 G 为 **轮图 (wheel graph)**，$n+1(n\geq 3)$ 阶轮图记作 W_n.

若无向简单图 $G=(V,E)$ 的所有边恰好构成一条简单路径，则称 G 为 **链 (chain/path graph)**，n 阶的链记作 P_n. 易知，一条链由一个圈图删去一条边而得.

如果一张无向连通图不含环，则称它是一棵 **树 (tree)**. 相关内容详见 树基础→11.6.1@p2621.

如果一张无向连通图包含恰好一个环，则称它是一棵 **基环树 (pseudotree)**.

如果一张有向弱连通图每个点的入度都为 1，则称它是一棵 **基环外向树**.

如果一张有向弱连通图每个点的出度都为 1，则称它是一棵 **基环内向树**.

多棵树可以组成一个 **森林 (forest)**，多棵基环树可以组成 **基环森林 (pseudoforest)**，多棵基环外向树可以组成 **基环外向树森林**，多棵基环内向树可以组成 **基环内向森林 (functional graph)**.

如果一张无向连通图的每条边最多在一个环内，则称它是一棵 **仙人掌 (cactus)**. 多棵仙人掌可以组成 **沙漠**.

如果一张图的点集可以被分为两部分，每一部分的内部都没有连边，那么这张图是一张 **二分图 (bipartite graph)**. 如果二分图中任何两个不在同一部分的点之间都有连边，那么这张图是一张 **完全二分图**

(complete bipartite graph/biclique)，一张两部分分别有 n 个点和 m 个点的完全二分图记作 $K_{n,m}$．相关内容详见 二分图→11.19@p2877.

如果一张图可以画在一个平面上，且没有两条边在非端点处相交，那么这张图是一张 **平面图 (planar graph)**．对于简单连通平面图 $G=(V,E)$ 且 $V\geq 3$，$|E|\leq 3|V|-6$.

Kuratowski 定理：一张图是平面图当且仅当其不存在与 K_5 或 $K_{3,3}$ **同胚 (homeomorphism)** 的子图．其中图 G 和图 G' 同胚指的是两图均可通过在边上新增若干度为 2 的顶点[1]的方式变为相同的图.

同构

两个图 G 和 H，如果存在一个双射 $f:V(G)\to V(H)$，且满足 $(u,v)\in E(G)$，当且仅当 $(f(u),f(v))\in E(H)$，则我们称 f 为 G 到 H 的一个 **同构 (isomorphism)**，且图 G 与图 H 是 **同构的 (isomorphic)**，记作 $G\cong H$.

从定义可知，若 $G\cong H$，必须满足：

- $|V(G)|=|V(H)|,|E(G)|=|E(H)|$
- G 和 H 结点度的非增序列相同
- G 和 H 存在同构的导出子图

无向简单图的二元运算

对于无向简单图，我们可以定义如下二元运算：

交 (intersection)：图 $G=(V_1,E_1),H=(V_2,E_2)$ 的交定义成图 $G\cap H=(V_1\cap V_2,E_1\cap E_2)$.

容易证明两个无向简单图的交还是无向简单图.

并 (union)：图 $G=(V_1,E_1),H=(V_2,E_2)$ 的并定义成图 $G\cup H=(V_1\cup V_2,E_1\cup E_2)$.

和 (sum)/直和 (direct sum)：对于 $G=(V_1,E_1),H=(V_2,E_2)$，任意构造 $H'\cong H$ 使得 $V(H')\cap V_1=\emptyset$（H' 可以等于 H）．此时与 $G\cup H'$ 同构的任何图称为 G 和 H 的和/直和/不交并，记作 $G+H$ 或 $G\oplus H$.

若 G 与 H 的点集本身不相交，则 $G\cup H=G+H$.

比如，森林可以定义成若干棵树的和.

> **并与和的区别**
>
> 可以理解为，「并」会让两张图中「名字相同」的点、边合并，而「和」则不会.

特殊的点集/边集

支配集

对于无向图 $G=(V,E)$，若 $V'\subseteq V$ 且 $\forall v\in(V\setminus V')$ 存在边 $(u,v)\in E$ 满足 $u\in V'$，则 V' 是图 G 的一个 **支配集 (dominating set)**.

无向图 G 最小的支配集的大小记作 $\gamma(G)$．求一张图的最小支配集是 NP 困难→13.9@p3306 的.

对于有向图 $G=(V,E)$，若 $V'\subseteq V$ 且 $\forall v\in(V\setminus V')$ 存在边 $(u,v)\in E$ 满足 $u\in V'$，则 V' 是图 G 的一个 **出 - 支配集 (out-dominating set)**．类似地，可以定义有向图的 **入 - 支配集 (in-dominating set)**.

有向图 G 最小的出 - 支配集大小记作 $\gamma^+(G)$，最小的入 - 支配集大小记作 $\gamma^-(G)$.

边支配集

对于图 $G=(V,E)$，若 $E'\subseteq E$ 且 $\forall e\in(E\setminus E')$ 存在 E' 中的边与其有公共点，则称 E' 是图 G 的一个 **边支配集 (edge dominating set)**.

[1]该操作又叫细分（subdivision）.

求一张图的最小边支配集是 NP 困难→13.9@p3306 的.

独立集

对于图 $G=(V,E)$，若 $V'\subseteq V$ 且 V' 中任意两点都不相邻，则 V' 是图 G 的一个 **独立集** (independent set).

图 G 最大的独立集的大小记作 $\alpha(G)$. 求一张图的最大独立集是 NP 困难→13.9@p3306 的.

匹配

对于图 $G=(V,E)$，若 $E'\subseteq E$ 且 E' 中任意两条不同的边都没有公共的端点，且 E' 中任意一条边都不是自环，则 E' 是图 G 的一个 **匹配** (matching)，也可以叫作 **边独立集** (independent edge set). 如果一个点是匹配中某条边的一个端点，则称这个点是 **被匹配的** (matched)/**饱和的** (saturated)，否则称这个点是 **不被匹配的** (unmatched).

边数最多的匹配被称作一张图的 **最大匹配** (maximum-cardinality matching). 图 G 的最大匹配的大小记作 $\nu(G)$.

如果边带权，那么权重之和最大的匹配被称作一张图的 **最大权匹配** (maximum-weight matching).

如果一个匹配在加入任何一条边后都不再是一个匹配，那么这个匹配是一个 **极大匹配** (maximal matching). 最大的极大匹配就是最大匹配，任何最大匹配都是极大匹配. 极大匹配一定是边支配集，但边支配集不一定是匹配. 最小极大匹配和最小边支配集大小相等，但最小边支配集不一定是匹配. 求最小极大匹配是 NP 困难的.

如果在一个匹配中所有点都是被匹配的，那么这个匹配是一个 **完美匹配** (perfect matching). 如果在一个匹配中只有一个点不被匹配，那么这个匹配是一个 **准完美匹配** (near-perfect matching).

求一张普通图或二分图的匹配或完美匹配个数都是 #P 完全→13.9@p3306 的.

对于一个匹配 M，若一条路径以非匹配点为起点，每相邻两条边的其中一条在匹配中而另一条不在匹配中，则这条路径被称作一条 **交替路径** (alternating path)；一条在非匹配点终止的交替路径，被称作一条 **增广路径** (augmenting path).

托特定理：n 阶无向图 G 有完美匹配当且仅当对于任意的 $V'\subset V(G)$，$p_{\text{odd}}(G-V')\leq |V'|$，其中 p_{odd} 表示奇数阶连通分支数.

托特定理（推论）：任何无桥 3 - 正则图都有完美匹配.

点覆盖

对于图 $G=(V,E)$，若 $V'\subseteq V$ 且 $\forall e\in E$ 满足 e 的至少一个端点在 V' 中，则称 V' 是图 G 的一个 **点覆盖** (vertex cover).

点覆盖集必为支配集，但极小点覆盖集不一定是极小支配集.

一个点集是点覆盖的充要条件是其补集是独立集，因此最小点覆盖的补集是最大独立集. 求一张图的最小点覆盖是 NP 困难→13.9@p3306 的.

一张图的任何一个匹配的大小都不超过其任何一个点覆盖的大小. 完全二分图 $K_{n,m}$ 的最大匹配和最小点覆盖大小都为 $\min(n,m)$.

边覆盖

对于图 $G=(V,E)$，若 $E'\subseteq E$ 且 $\forall v\in V$ 满足 v 与 E' 中的至少一条边相邻，则称 E' 是图 G 的一个 **边覆盖** (edge cover).

最小边覆盖的大小记作 $\rho(G)$，可以由最大匹配贪心扩展求得：对于所有非匹配点，将其一条邻边加入最大匹配中，即得到了一个最小边覆盖.

最大匹配也可以由最小边覆盖求得：对于最小边覆盖中每对有公共点的边删去其中一条.

一张图的最小边覆盖的大小加上最大匹配的大小等于图的点数，即 $\rho(G)+\nu(G)=|V(G)|$.

一张图的最大匹配的大小不超过最小边覆盖的大小，即 $\nu(G)\leq\rho(G)$. 特别地，完美匹配一定是一个最小边覆盖，这也是上式取到等号的唯一情况.

一张图的任何一个独立集的大小都不超过其任何一个边覆盖的大小. 完全二分图 $K_{n,m}$ 的最大独立集和最小边覆盖大小都为 $\max(n,m)$.

团

对于图 $G=(V,E)$，若 $V'\subseteq V$ 且 V' 中任意两个不同的顶点都相邻，则 V' 是图 G 的一个 **团 (clique)**. 团的导出子图是完全图.

如果一个团在加入任何一个顶点后都不再是一个团，则这个团是一个 **极大团 (maximal clique)**.

一张图的最大团的大小记作 $\omega(G)$，最大团的大小等于其补图最大独立集的大小，即 $\omega(G)=\alpha(\bar{G})$. 求一张图的最大团是 NP 困难→13.9@p3306 的.

参考资料

OI 中转站 - 图论概念梳理[1]

Wikipedia[2]（以及相关概念的对应词条）

离散数学（修订版），田文成 周禄新 编著，天津文学出版社，P184-187

戴一奇，胡冠章，陈卫. 图论与代数结构 [M]. 北京：清华大学出版社，1995.

[1] OI 中转站 - 图论概念梳理 [2] Wikipedia

11.3 图的存储

Authors: Ir1d, sshwy, Xeonacid, partychicken, Anguei, HeRaNO

在 OI 中，想要对图进行操作，就需要先学习图的存储方式.

约定

本文默认读者已阅读并了解了 图论相关概念→11.2@p2598 中的基础内容，如果在阅读中遇到困难，也可以在 图论相关概念→11.2@p2598 中进行查阅.

在本文中，用 n 代指图的点数，用 m 代指图的边数，用 $d^+(u)$ 代指点 u 的出度，即以 u 为出发点的边数.

直接存边

方法

使用一个数组来存边，数组中的每个元素都包含一条边的起点与终点（带边权的图还包含边权）.（或者使用多个数组分别存起点，终点和边权.）

参考代码

- C++

```cpp
#include <iostream>
#include <vector>
```

```cpp

using namespace std;

struct Edge {
  int u, v;
};

int n, m;
vector<Edge> e;
vector<bool> vis;

bool find_edge(int u, int v) {
  for (int i = 1; i <= m; ++i) {
    if (e[i].u == u && e[i].v == v) {
      return true;
    }
  }
  return false;
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = 1; i <= m; ++i) {
    if (e[i].u == u) {
      dfs(e[i].v);
    }
  }
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  e.resize(m + 1);

  for (int i = 1; i <= m; ++i) cin >> e[i].u >> e[i].v;

  return 0;
}
```

- Python

```python
class Edge:
    def __init__(self, u=0, v=0):
        self.u = u
        self.v = v


n, m = map(int, input().split())

e = [Edge() for _ in range(m)]
vis = [False] * n
```

```python

for i in range(m):
    e[i].u, e[i].v = map(int, input().split())


def find_edge(u, v):
    for i in range(m):
        if e[i].u == u and e[i].v == v:
            return True
    return False


def dfs(u):
    if vis[u]:
        return
    vis[u] = True
    for i in range(m):
        if e[i].u == u:
            dfs(e[i].v)
```

复杂度

查询是否存在某条边：$O(m)$.

遍历一个点的所有出边：$O(m)$.

遍历整张图：$O(nm)$.

空间复杂度：$O(m)$.

应用

由于直接存边的遍历效率低下，一般不用于遍历图.

在 Kruskal 算法→11.10.1@p2764 中，由于需要将边按边权排序，需要直接存边.

在有的题目中，需要多次建图（如建一遍原图，建一遍反图），此时既可以使用多个其它数据结构来同时存储多张图，也可以将边直接存下来，需要重新建图时利用直接存下的边来建图.

邻接矩阵

方法

使用一个二维数组 `adj` 来存边，其中 `adj[u][v]` 为 1 表示存在 u 到 v 的边，为 0 表示不存在. 如果是带边权的图，可以在 `adj[u][v]` 中存储 u 到 v 的边的边权.

参考代码

- C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
```

```cpp
vector<bool> vis;
vector<vector<bool>> adj;

bool find_edge(int u, int v) { return adj[u][v]; }

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int v = 1; v <= n; ++v) {
    if (adj[u][v]) {
      dfs(v);
    }
  }
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1);
  adj.resize(n + 1, vector<bool>(n + 1));

  for (int i = 1; i <= m; ++i) {
    int u, v;
    cin >> u >> v;
    adj[u][v] = true;
  }

  return 0;
}
```

- Python

```python
vis = [False] * (n + 1)
adj = [[False] * (n + 1) for _ in range(n + 1)]

for i in range(1, m + 1):
    u, v = map(lambda x: int(x), input().split())
    adj[u][v] = True


def find_edge(u, v):
    return adj[u][v]


def dfs(u):
    if vis[u]:
        return
    vis[u] = True
    for v in range(1, n + 1):
        if adj[u][v]:
            dfs(v)
```

复杂度

查询是否存在某条边：$O(1)$.

遍历一个点的所有出边：$O(n)$.

遍历整张图：$O(n^2)$.

空间复杂度：$O(n^2)$.

应用

邻接矩阵只适用于没有重边（或重边可以忽略）的情况.

其最显著的优点是可以 $O(1)$ 查询一条边是否存在.

由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图上使用邻接矩阵.

邻接表

方法

使用一个支持动态增加元素的数据结构构成的数组，如 `vector<int> adj[n + 1]` 来存边，其中 `adj[u]` 存储的是点 u 的所有出边的相关信息（终点、边权等）.

参考代码

- C++

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<bool> vis;
vector<vector<int>> adj;

bool find_edge(int u, int v) {
  for (int i = 0; i < adj[u].size(); ++i) {
    if (adj[u][i] == v) {
      return true;
    }
  }
  return false;
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = 0; i < adj[u].size(); ++i) dfs(adj[u][i]);
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1);
  adj.resize(n + 1);
```

```
30
31   for (int i = 1; i <= m; ++i) {
32     int u, v;
33     cin >> u >> v;
34     adj[u].push_back(v);
35   }
36
37   return 0;
38 }
```

- Python

```
1  vis = [False] * (n + 1)
2  adj = [[] for _ in range(n + 1)]
3
4  for i in range(1, m + 1):
5      u, v = map(lambda x: int(x), input().split())
6      adj[u].append(v)
7
8
9  def find_edge(u, v):
10     for i in range(0, len(adj[u])):
11         if adj[u][i] == v:
12             return True
13     return False
14
15
16 def dfs(u):
17     if vis[u]:
18         return
19     vis[u] = True
20     for i in range(0, len(adj[u])):
21         dfs(adj[u][i])
```

复杂度

查询是否存在 u 到 v 的边：$O(d^+(u))$（如果事先进行了排序就可以使用 二分查找→5.9@p629 做到 $O(\log(d^+(u)))$）.

遍历点 u 的所有出边：$O(d^+(u))$.

遍历整张图：$O(n+m)$.

空间复杂度：$O(m)$.

应用

存各种图都很适合，除非有特殊需求（如需要快速查询一条边是否存在，且点数较少，可以使用邻接矩阵）.

尤其适用于需要对一个点的所有出边进行排序的场合.

链式前向星

方法

本质上是用链表实现的邻接表，核心代码如下：

- C++

```cpp
// head[u] 和 cnt 的初始值都为 -1
void add(int u, int v) {
  nxt[++cnt] = head[u];  // 当前边的后继
  head[u] = cnt;         // 起点 u 的第一条边
  to[cnt] = v;           // 当前边的终点
}

// 遍历 u 的出边
for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
  int v = to[i];
}
```

- Python

```python
# head[u] 和 cnt 的初始值都为 -1
def add(u, v):
    cnt = cnt + 1
    nex[cnt] = head[u]  # 当前边的后继
    head[u] = cnt  # 起点 u 的第一条边
    to[cnt] = v  # 当前边的终点


# 遍历 u 的出边
i = head[u]
while ~i:  # ~i 表示 i != -1
    v = to[i]
    i = nxt[i]
```

参考代码

```cpp
#include <iostream>
#include <vector>

using namespace std;

int n, m;
vector<bool> vis;
vector<int> head, nxt, to;

void add(int u, int v) {
  nxt.push_back(head[u]);
  head[u] = to.size();
  to.push_back(v);
}

bool find_edge(int u, int v) {
  for (int i = head[u]; ~i; i = nxt[i]) {  // ~i 表示 i != -1
    if (to[i] == v) {
      return true;
    }
  }
  return false;
```

```
}

void dfs(int u) {
  if (vis[u]) return;
  vis[u] = true;
  for (int i = head[u]; ~i; i = nxt[i]) dfs(to[i]);
}

int main() {
  cin >> n >> m;

  vis.resize(n + 1, false);
  head.resize(n + 1, -1);

  for (int i = 1; i <= m; ++i) {
    int u, v;
    cin >> u >> v;
    add(u, v);
  }

  return 0;
}
```

复杂度

查询是否存在 u 到 v 的边：$O(d^+(u))$.

遍历点 u 的所有出边：$O(d^+(u))$.

遍历整张图：$O(n+m)$.

空间复杂度：$O(m)$.

应用

存各种图都很适合，但不能快速查询一条边是否存在，也不能方便地对一个点的出边进行排序.

优点是边是带编号的，有时会非常有用，而且如果 `cnt` 的初始值为奇数，存双向边时 `i ^ 1` 即是 `i` 的反边（常用于 网络流→11.23.1@p2906）.

11.4 DFS（图论）

> **Authors:** Ir1d, greyqz, yjl9903, partychicken, ChungZH, qq1010903229, Marcythm, Acfboy, shenshuaijie, Craneplayz

引入

DFS 全称是 Depth First Search[1]，中文名是深度优先搜索，是一种用于遍历或搜索树或图的算法. 所谓深度优先，就是说每次都尝试向更深的节点走.

该算法讲解时常常与 BFS 并列，但两者除了都能遍历图的连通块以外，用途完全不同，很少有能混用两种算法的情况.

DFS 常常用来指代用递归函数实现的搜索，但实际上两者并不一样. 有关该类搜索思想请参阅 DFS（搜索）→6.2@p654.

过程

DFS 最显著的特征在于其 **递归调用自身**. 同时与 BFS 类似，DFS 会对其访问过的点打上访问标记，在遍历图时跳过已打过标记的点，以确保 **每个点仅访问一次**. 符合以上两条规则的函数，便是广义上的 DFS.

具体地说，DFS 大致结构如下：

```
DFS(v) // v 可以是图中的一个顶点，也可以是抽象的概念，如 dp 状态等.
  在 v 上打访问标记
  for u in v 的相邻节点
    if u 没有打过访问标记 then
      DFS(u)
    end
  end
end
```

以上代码只包含了 DFS 必需的主要结构. 实际的 DFS 会在以上代码基础上加入一些代码，利用 DFS 性质进行其他操作.

性质

该算法通常的时间复杂度为 $O(n+m)$，空间复杂度为 $O(n)$，其中 n 表示点数，m 表示边数. 注意空间复杂度包含了栈空间，栈空间的空间复杂度是 $O(n)$ 的. 在平均 $O(1)$ 遍历一条边的条件下才能达到此时间复杂度，例如用前向星或邻接表存储图；如果用邻接矩阵则不一定能达到此复杂度.

> 备注：目前大部分算法竞赛（包括 NOIP、大部分省选以及 CCF 举办的各项赛事）都支持 **无限栈空间**，即：栈空间不单独限制，但总内存空间仍然受题面限制. 但大部分操作系统会对栈空间做额外的限制，因此在本地调试时需要一些方式来取消栈空间限制.
>
> - 在 Windows 上，通常的方法是在 **编译选项** 中加入 `-Wl,--stack=1000000000`，表示将栈空间限制设置为 1000000000 字节.
> - 在 Linux 上，通常的方法是在运行程序前 **在终端内** 执行 `ulimit -s unlimited`，表示栈空间无限. 每个终端只需执行一次，对之后每次程序运行都有效.

实现

栈实现

DFS 可以使用 栈（Stack）→10.2@p1978 为遍历中节点的暂存容器来实现；这与用 队列（Queue）→10.3@p1980 实现的 BFS 形成高度对应.

- C++

```cpp
vector<vector<int>> adj;  // 邻接表
vector<bool> vis;         // 记录节点是否已经遍历

void dfs(int s) {
  stack<int> st;
  st.push(s);
  vis[s] = true;

  while (!st.empty()) {
    int u = st.top();
    st.pop();

    for (int v : adj[u]) {
      if (!vis[v]) {
```

```
        vis[v] = true;  // 确保栈里没有重复元素
        st.push(v);
      }
    }
  }
}
```

- Python

```
# adj : List[List[int]] 邻接表
# vis : List[bool] 记录节点是否已经遍历


def dfs(s: int) -> None:
    stack = [s]  # 用列表来模拟栈，把起点加入栈中
    vis[s] = True  # 起点被遍历

    while stack:  # 当栈非空时继续执行
        u = (
            stack.pop()
        )  # 拿取并丢弃掉最后一个元素（栈顶的元素），可以理解为走到u这个元素

        for v in adj[u]:  # 对于与u相邻的每个元素v
            if not vis[v]:  # 如果v在此前没有走过
                vis[v] = True  # 确保栈里没有重复元素
                stack.append(v)  # 把v加入栈中
```

递归实现

函数在递归调用时的求值如同对栈的添加和删除元素的顺序，故函数调用所占据的虚拟地址被称为函数调用栈（Call Stack），DFS 可用递归的方式实现.

以 邻接表（Adjacency List）→11.3@p2605 作为图的存储方式：

- C++

```
vector<vector<int>> adj;  // 邻接表
vector<bool> vis;         // 记录节点是否已经遍历

void dfs(const int u) {
  vis[u] = true;
  for (int v : adj[u])
    if (!vis[v]) dfs(v)
}
```

- Python

```
# adj : List[List[int]] 邻接表
# vis : List[bool] 记录节点是否已经遍历


def dfs(u: int) -> None:
    vis[u] = True
    for v in adj[u]:
        if not vis[v]:
            dfs(v)
```

以 链式前向星→11.3@p2605 为例：

- C++

```cpp
void dfs(int u) {
  vis[u] = 1;
  for (int i = head[u]; i; i = e[i].x) {
    if (!vis[e[i].t]) {
      dfs(v);
    }
  }
}
```

- Java

```java
public void dfs(int u) {
    vis[u] = true;
    for (int i = head[u]; i != 0; i = e[i].x) {
        if (!vis[e[i].t]) {
            dfs(v);
        }
    }
}
```

- Python

```python
def dfs(u):
    vis[u] = True
    i = head[u]
    while i:
        if vis[e[i].t] == False:
            dfs(v)
        i = e[i].x
```

DFS 序列

DFS 序列是指 DFS 调用过程中访问的节点编号的序列.

我们发现，每个子树都对应 DFS 序列中的连续一段（一段区间）.

括号序列

DFS 进入某个节点的时候记录一个左括号 `(`，退出某个节点的时候记录一个右括号 `)`.

每个节点会出现两次. 相邻两个节点的深度相差 1.

一般图上 DFS

对于非连通图，只能访问到起点所在的连通分量.

对于连通图，DFS 序列通常不唯一.

注：树的 DFS 序列也是不唯一的.

在 DFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，称为 DFS 树. DFS 树是原图的一个生成树.

DFS 树→11.13.1@p2806 有很多性质，比如可以用来求 强连通分量→11.13.1@p2806.

外部参考资料

[1] Depth First Search 

11.5 BFS（图论）

Authors: Ir1d, greyqz, yjl9903, Anguei, Marcythm, ChungZH, Xeonacid, ylxmf2005

BFS 全称是 Breadth First Search[1]，中文名是宽度优先搜索，也叫广度优先搜索.

是图上最基础、最重要的搜索算法之一.

所谓宽度优先. 就是每次都尝试访问同一层的节点. 如果同一层都访问完了，再访问下一层.

这样做的结果是，BFS 算法找到的路径是从起点开始的 **最短** 合法路径. 换言之，这条路径所包含的边数最小.

在 BFS 结束时，每个节点都是通过从起点到该点的最短路径访问的.

算法过程可以看做是图上火苗传播的过程：最开始只有起点着火了，在每一时刻，有火的节点都向它相邻的所有节点传播火苗.

实现

下文中 C++ 与 Python 的代码实现是基于链式前向星的存图方式，其实现可参考 图的存储→11.3@p2605 页面.

- 伪代码

```
bfs(s) {
  q = new queue()
  q.push(s), visited[s] = true
  while (!q.empty()) {
    u = q.pop()
    for each edge(u, v) {
      if (!visited[v]) {
        q.push(v)
        visited[v] = true
      }
    }
  }
}
```

- C++

```
void bfs(int u) {
  while (!Q.empty()) Q.pop();
  Q.push(u);
  vis[u] = 1;
  d[u] = 0;
  p[u] = -1;
  while (!Q.empty()) {
    u = Q.front();
    Q.pop();
    for (int i = head[u]; i; i = e[i].nxt) {
      if (!vis[e[i].to]) {
        Q.push(e[i].to);
        vis[e[i].to] = 1;
        d[e[i].to] = d[u] + 1;
        p[e[i].to] = u;
      }
    }
  }
```

```
}

void restore(int x) {
  vector<int> res;
  for (int v = x; v != -1; v = p[v]) {
    res.push_back(v);
  }
  std::reverse(res.begin(), res.end());
  for (int i = 0; i < res.size(); ++i) printf("%d", res[i]);
  puts("");
}
```

- Python

```
from queue import Queue


def bfs(u):
    Q = Queue()
    Q.put(u)
    vis[u] = True
    d[u] = 0
    p[u] = -1
    while Q.qsize() != 0:
        u = Q.get()
        i = head[u]
        while i:
            if vis[e[i].to] == False:
                Q.put(e[i].to)
                vis[e[i].to] = True
                d[e[i].to] = d[u] + 1
                p[e[i].to] = u
            i = e[i].nxt


def restore(x):
    res = []
    v = x
    while v != -1:
        res.append(v)
        v = p[v]
    res.reverse()
    for i in range(0, len(res)):
        print(res[i])
```

具体来说，我们用一个队列 Q 来记录要处理的节点，然后开一个布尔数组 `vis[]` 来标记是否已经访问过某个节点.

开始的时候，我们将所有节点的 `vis` 值设为 0，表示没有访问过；然后把起点 s 放入队列 Q 中并将 `vis[s]` 设为 1.

之后，我们每次从队列 Q 中取出队首的节点 u，然后把与 u 相邻的所有节点 v 标记为已访问过并放入队列 Q.

循环直至当队列 Q 为空，表示 BFS 结束.

在 BFS 的过程中，也可以记录一些额外的信息．例如上述代码中，d 数组用于记录起点到某个节点的最短距离（要经过的最少边数），p 数组记录是从哪个节点走到当前节点的．

有了 d 数组，可以方便地得到起点到一个节点的距离．

有了 p 数组，可以方便地还原出起点到一个点的最短路径．上述代码中的 `restore` 函数使用该数组依次输出从起点到节点 x 的最短路径所经过的节点．

时间复杂度 $O(n+m)$

空间复杂度 $O(n)$（`vis` 数组和队列）

open-closed 表

在实现 BFS 的时候，本质上我们把未被访问过的节点放在一个称为 open 的容器中，而把已经访问过了的节点放在一个称为 closed 容器中．

在树/图上 BFS

BFS 序列

类似 DFS 序列，BFS 序列是指在 BFS 过程中访问的节点编号的序列．

一般图上 BFS

如果原图不连通，只能访问到从起点出发能够到达的点．

BFS 序列通常也不唯一．

类似的我们也可以定义 BFS 树：在 BFS 过程中，通过记录每个节点从哪个点访问而来，可以建立一个树结构，即为 BFS 树．

应用

- 在一个无权图上求从起点到其他所有点的最短路径．
- 在 $O(n+m)$ 时间内求出所有连通块．（我们只需要从每个没有被访问过的节点开始做 BFS，显然每次 BFS 会走完一个连通块）
- 如果把一个游戏的动作看做是状态图上的一条边（一个转移），那么 BFS 可以用来找到在游戏中从一个状态到达另一个状态所需要的最小步骤．
- 在一个有向无权图中找最小环．（从每个点开始 BFS，在我们即将抵达一个之前访问过的点开始的时候，就知道遇到了一个环．图的最小环是每次 BFS 得到的最小环的平均值．）
- 找到一定在 (a,b) 最短路上的边．（分别从 a 和 b 进行 BFS，得到两个 d 数组．之后对每一条边 (u,v)，如果 $d_a[u]+1+d_b[v]=d_a[b]$，则说明该边在最短路上）
- 找到一定在 (a,b) 最短路上的点．（分别从 a 和 b 进行 BFS，得到两个 d 数组．之后对每一个点 v，如果 $d_a[v]+d_b[v]=d_a[b]$，则说明该点在某条最短路上）
- 找到一条长度为偶数的最短路．（我们需要一个构造一个新图，把每个点拆成两个新点，原图的边 (u,v) 变成 $((u,0),(v,1))$ 和 $((u,1),(v,0))$．对新图做 BFS，$(s,0)$ 和 $(t,0)$ 之间的最短路即为所求）
- 在一个边权为 0/1 的图上求最短路，见下方双端队列 BFS．

双端队列 BFS

如果你不了解双端队列 `deque` 的话，请参阅 deque 相关章节→4.3.2.3@p373．

双端队列 BFS 又称 0-1 BFS．

适用范围

边权值为可能有，也可能没有（由于 BFS 适用于权值为 1 的图，所以一般权值是 0 或 1），或者能够转化为这种边权值的最短路问题.

例如在走迷宫问题中，你可以花 1 个金币走 5 步，也可以不花金币走 1 步，这就可以用 0-1 BFS 解决.

实现

一般情况下，我们把没有权值的边扩展到的点放到队首，有权值的边扩展到的点放到队尾. 这样即可保证像普通 BFS 一样整个队列队首到队尾权值单调不下降.

下面是伪代码：

```
while (队列不为空) {
  int u = 队首;
  弹出队首;
  for (枚举 u 的邻居) {
    更新数据
    if (...)
      添加到队首;
    else
      添加到队尾;
  }
}
```

例题

Codeforces 173B[2]

一个 $n \times m$ 的图，现在有一束激光从左上角往右边射出，每遇到 '#'，你可以选择光线往四个方向射出，或者什么都不做，问最少需要多少个 '#' 往四个方向射出才能使光线在第 n 行往右边射出.

此题目正解不是 0-1 BFS，但是适用 0-1 BFS，减小思维强度，赛时许多大佬都是这么做的.

做法很简单，一个方向射出不需要花费（0），而往四个方向射出需要花费（1），然后直接来就可以了.

代码

```cpp
#include <deque>
#include <iostream>
using namespace std;

constexpr int INF = 1 << 29;
int n, m;
char grid[1001][1001];
int dist[1001][1001][4];
int fx[] = {1, -1, 0, 0};
int fy[] = {0, 0, 1, -1};
deque<int> q;  // 双端队列

void add_front(int x, int y, int dir, int d) {  // 向前方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_front(dir);
    q.push_front(y);
    q.push_front(x);
  }
```

```
}

void add_back(int x, int y, int dir, int d) {  // 向后方加
  if (d < dist[x][y][dir]) {
    dist[x][y][dir] = d;
    q.push_back(x);
    q.push_back(y);
    q.push_back(dir);
  }
}

int main() {
  cin >> n >> m;
  for (int i = 0; i < n; i++) cin >> grid[i];

  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      for (int k = 0; k < 4; k++) dist[i][j][k] = INF;

  add_front(n - 1, m - 1, 3, 0);

  while (!q.empty()) {  // 具体搜索的过程，可以参考上面写的题解
    int x = q[0], y = q[1], dir = q[2];
    q.pop_front();
    q.pop_front();
    q.pop_front();
    int d = dist[x][y][dir];
    int nx = x + fx[dir], ny = y + fy[dir];
    if (nx >= 0 && nx < n && ny >= 0 && ny < m)
      add_front(nx, ny, dir, d);  // 判断条件
    if (grid[x][y] == '#')
      for (int i = 0; i < 4; i++)
        if (i != dir) add_back(x, y, i, d + 1);
  }
  if (dist[0][0][3] == INF)
    cout << -1 << endl;
  else
    cout << dist[0][0][3] << endl;
  return 0;
}
```

优先队列 BFS

优先队列，相当于一个二叉堆，STL 中提供了 std::priority_queue→4.3.2.6@p384，可以方便我们使用优先队列.

在基于优先队列的 BFS 中，我们每次从队首取出代价最小的结点进行进一步搜索. 容易证明这个贪心思想是正确的，因为从这个结点开始扩展的搜索，一定不会更新原来那些代价更高的结点. 换句话说，其余那些代价更高的结点，我们不回去考虑更新它.

当然，每个结点可能会被入队多次，只是每次入队的代价不同. 当该结点第一次从优先队列中取出，以后便无需再在该结点进行搜索，直接忽略即可. 所以，优先队列的 BFS 当中，每个结点只会被处理一次.

相对于普通队列的 BFS，时间复杂度多了一个 $\log n$，毕竟要维护这个优先队列嘛. 不过普通 BFS 有可能每个结点入队、出队多次，时间复杂度会达到 $O(n^2)$，不是 $O(n)$. 所以优先队列 BFS 通常还是快的.

诶？这怎么听起来这么像堆优化的 Dijkstra→11.9.1@p2735 算法呢？事实上，堆优化 Dijkstra 就是优先队列 BFS.

习题

- 「NOIP2017」奶酪[3]

双端队列 BFS：

- CF1063B. Labyrinth[4]
- CF173B. Chamber of Secrets[5]
- 「BalticOI 2011 Day1」打开灯泡 Switch the Lamp On[6]

参考

https://cp-algorithms.com/graph/breadth-first-search.html[7]

外部参考资料

[1] Breadth First Search

[2] Codeforces 173B

[3] 「NOIP2017」奶酪

[4] CF1063B. Labyrinth

[5] CF173B. Chamber of Secrets

[6] 「BalticOI 2011 Day1」打开灯泡 Switch the Lamp On

[7] https://cp-algorithms.com/graph/breadth-first-search.html

11.6 树上问题

11.6.1 树基础

引入

图论中的树和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑. 这种数据结构看起来像是一个倒挂的树，因此得名.

定义

一个没有固定根结点的树称为 **无根树**（unrooted tree）. 无根树有几种等价的形式化定义：

- 有 n 个结点，$n-1$ 条边的连通无向图
- 无向无环的连通图
- 任意两个结点之间有且仅有一条简单路径的无向图
- 任何边均为桥的连通图
- 没有圈，且在任意不同两点间添加一条边之后所得图含唯一的一个圈的图

在无根树的基础上，指定一个结点称为 **根**，则形成一棵 **有根树**（rooted tree）. 有根树在很多时候仍以无向图表示，只是规定了结点之间的上下级关系，详见下文.

有关树的定义

适用于无根树和有根树

- **森林（forest）**：每个连通分量（连通块）都是树的图. 按照定义，一棵树也是森林.
- **生成树（spanning tree）**：一个连通无向图的生成子图，同时要求是树. 也即在图的边集中选择 $n-1$ 条，将所有顶点连通.
- **无根树的叶结点（leaf node）**：度数不超过 1 的结点.

为什么不是度数恰为 1？

考虑 $n = 1$.

- **有根树的叶结点**（leaf node）：没有子结点的结点.

只适用于有根树

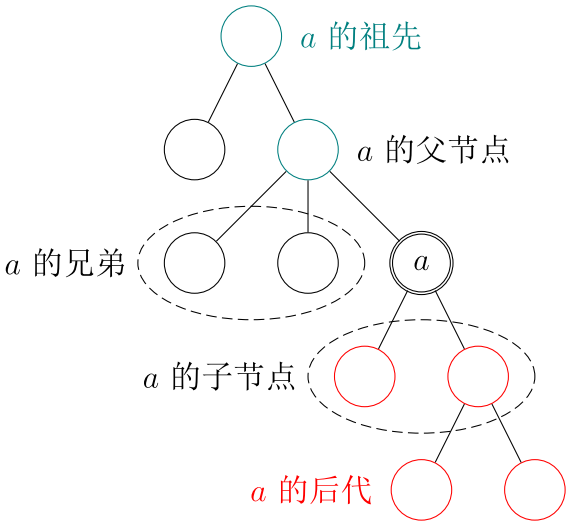
- **父亲**（parent node）：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点.

  根结点没有父结点.
- **祖先**（ancestor）：一个结点到根结点的路径上，除了它本身外的结点.

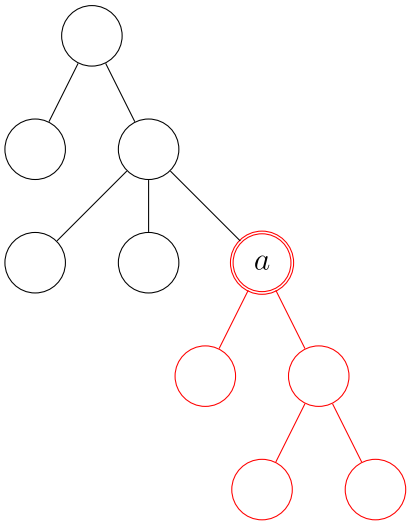
  根结点的祖先集合为空.
- **子结点**（child node）：如果 u 是 v 的父亲，那么 v 是 u 的子结点.

  子结点的顺序一般不加以区分，二叉树是一个例外.
- **结点的深度**（depth）：到根结点的路径上的边数.
- **树的高度**（height）：所有结点的深度的最大值.
- **兄弟**（sibling）：同一个父亲的多个子结点互为兄弟.
- **后代**（descendant）：子结点和子结点的后代.

  或者理解成：如果 u 是 v 的祖先，那么 v 是 u 的后代.



- **子树**（subtree）：删掉与父亲相连的边后，该结点所在的子图.

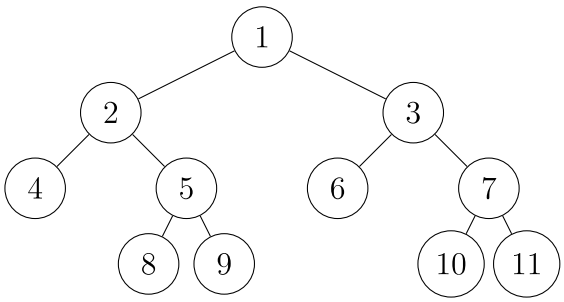


a 的子树

特殊的树

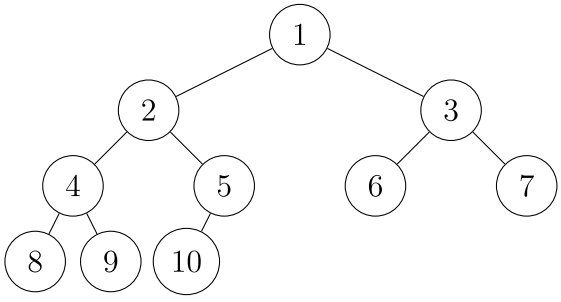
- **链（chain/path graph）**：满足与任一结点相连的边不超过 2 条的树称为链.
- **菊花/星星（star）**：满足存在 u 使得所有除 u 以外结点均与 u 相连的树称为菊花.
- **有根二叉树（rooted binary tree）**：每个结点最多只有两个儿子（子结点）的有根树称为二叉树. 常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点.

  大多数情况下，**二叉树** 一词均指有根二叉树.
- **完整二叉树（full/proper binary tree）**：每个结点的子结点数量均为 0 或者 2 的二叉树. 换言之，每个结点或者是树叶，或者左右子树均非空.



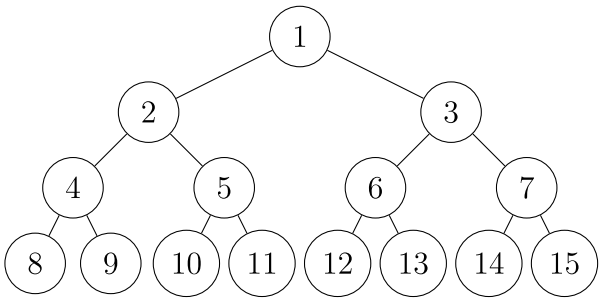
完整二叉树（proper binary tree）

- **完全二叉树（complete binary tree）**：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上.



完全二叉树（complete binary tree）

- **完美二叉树（perfect binary tree）**：所有叶结点的深度均相同，且所有非叶节点的子节点数量均为 2 的二叉树称为完美二叉树.



完美二叉树（perfect binary tree）

> **Warning**
>
> Proper binary tree 的汉译名称不固定，且完全二叉树和满二叉树的定义在不同教材中定义不同，遇到的时候需根据上下文加以判断.

OIers 所说的「满二叉树」多指完美二叉树.

存储

只记录父结点

用一个数组 `parent[N]` 记录每个结点的父亲结点.

这种方式可以获得的信息较少，不便于进行自顶向下的遍历. 常用于自底向上的递推问题中.

邻接表

- 对于无根树：为每个结点开辟一个线性列表，记录所有与之相连的结点.

```cpp
std::vector<int> adj[N];
```

- 对于有根树：
  - 方法一：若给定的是无向图，则仍可以上述形式存储. 下文将介绍如何区分结点的上下关系.
  - 方法二：若输入数据能够确保结点的上下关系，则可以利用这个信息. 为每个结点开辟一个线性列表，记录其所有子结点；若有需要，还可在另一个数组中记录其父结点.

    ```cpp
    std::vector<int> children[N];
    int parent[N];
    ```

    当然也可以用其他方式（如链表）替代 `std::vector`.

左孩子右兄弟表示法

过程

对于有根树，存在一种简单的表示方法.

首先，给每个结点的所有子结点任意确定一个顺序.

此后为每个结点记录两个值：其 **第一个子结点** `child[u]` 和其 **下一个兄弟结点** `sib[u]`. 若没有子结点，则 `child[u]` 为空；若该结点是其父结点的最后一个子结点，则 `sib[u]` 为空.

实现

遍历一个结点的所有子结点可由如下方式实现.

```cpp
int v = child[u];  // 从第一个子结点开始
```

```
while (v != EMPTY_NODE) {
  // ...
  // 处理子结点 v
  // ...
  v = sib[v];  // 转至下一个子结点，即 v 的一个兄弟
}
```

也可简写为以下形式.

```
for (int v = child[u]; v != EMPTY_NODE; v = sib[v]) {
  // ...
  // 处理子结点 v
  // ...
}
```

二叉树

需要记录每个结点的左右子结点.

实现

```
int parent[N];
int lch[N], rch[N];
// -- or --
int child[N][2];
```

树的遍历

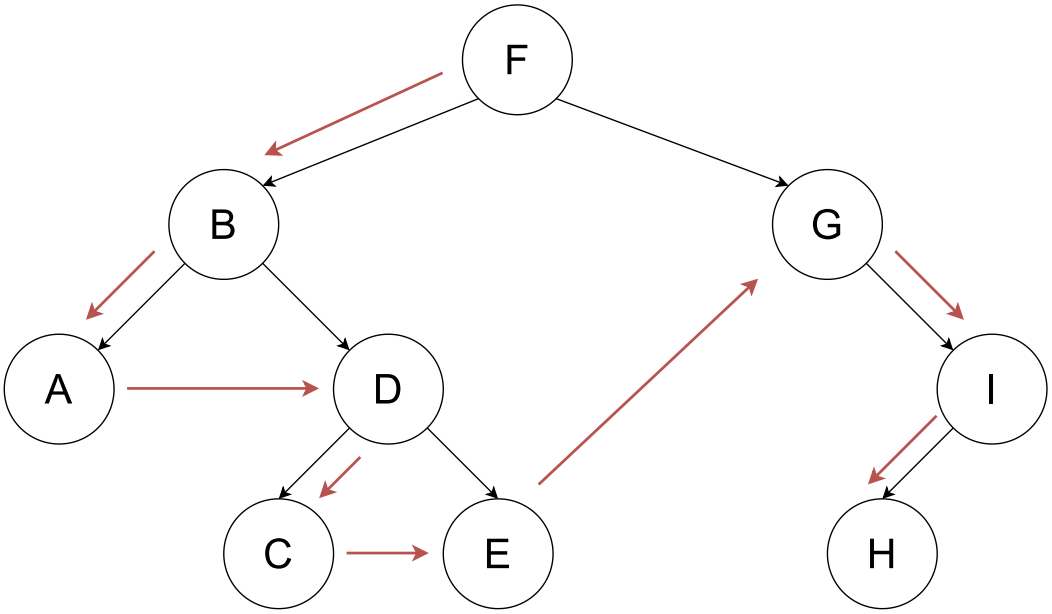
树上 DFS

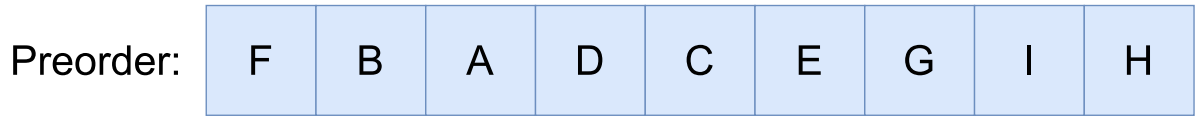
在树上 DFS 是这样的一个过程：先访问根节点，然后分别访问根节点每个儿子的子树.

可以用来求出每个节点的深度、父亲等信息.

二叉树 DFS 遍历

先序遍历



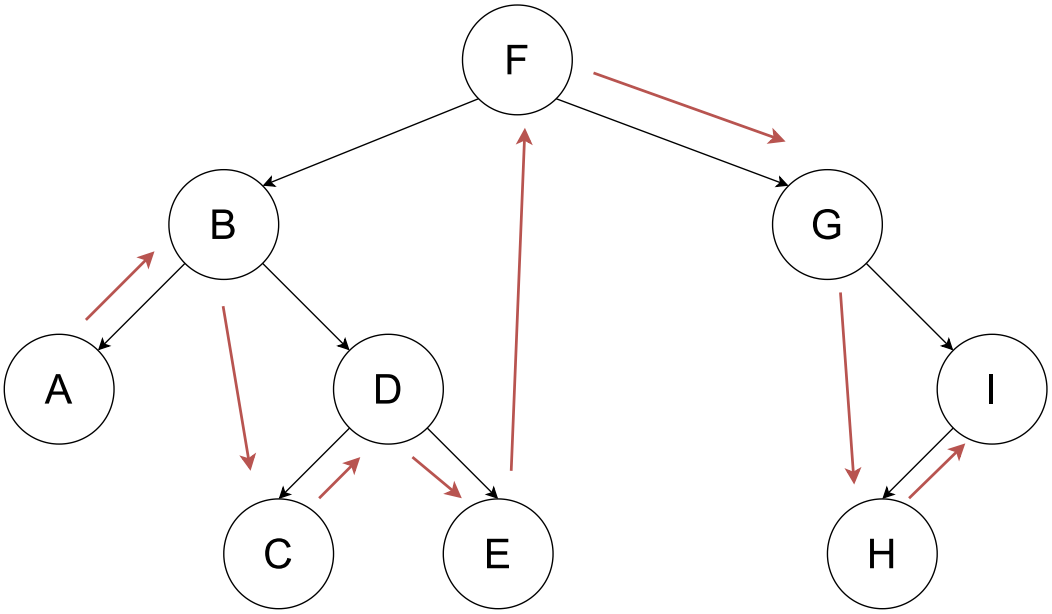


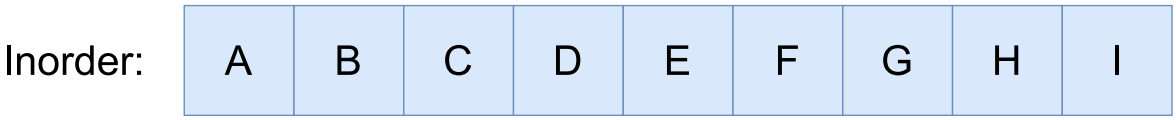
按照 **根，左，右** 的顺序遍历二叉树.

实现

```
void preorder(BiTree* root) {
  if (root) {
    cout << root->key << " ";
    preorder(root->left);
    preorder(root->right);
  }
}
```

中序遍历



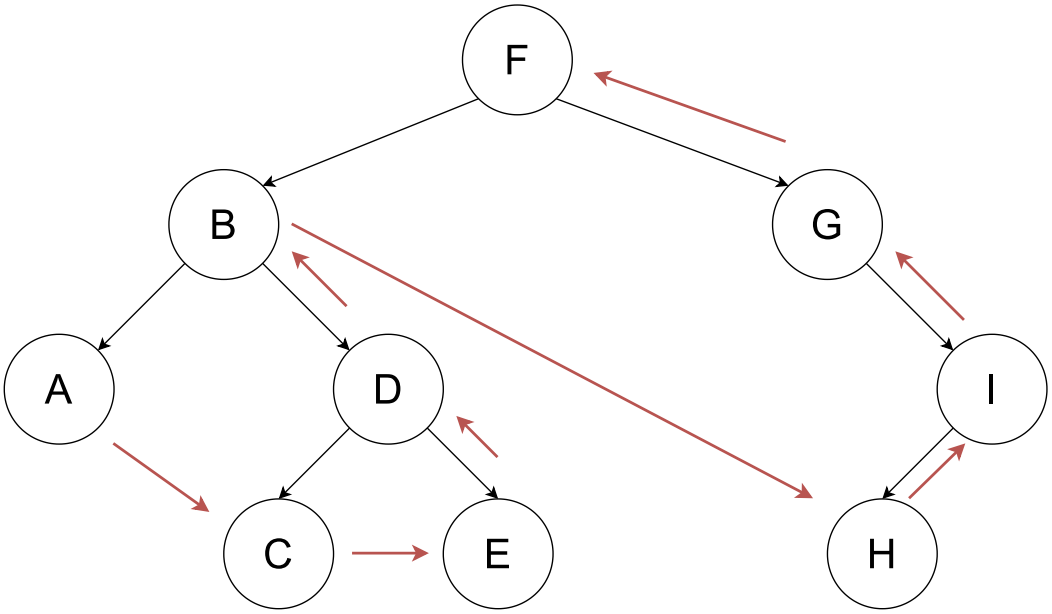


按照 **左，根，右** 的顺序遍历二叉树.

实现

```
void inorder(BiTree* root) {
  if (root) {
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
  }
}
```

后序遍历



Postorder:	A	C	E	D	B	H	I	G	F

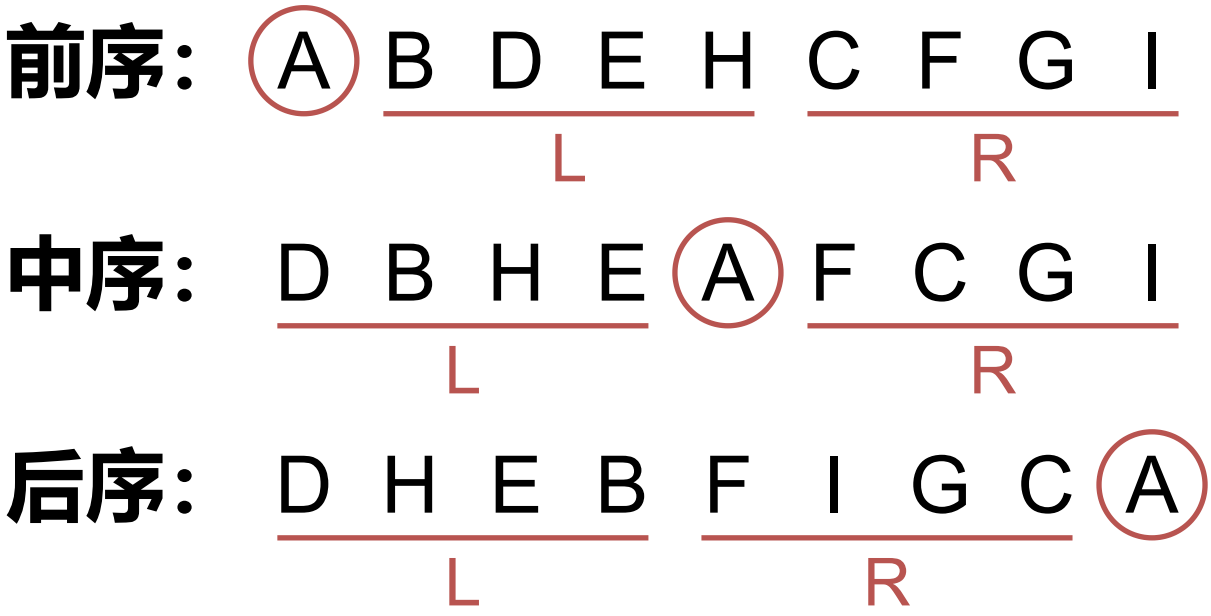
按照 **左，右，根** 的顺序遍历二叉树.

实现

```
void postorder(BiTree* root) {
  if (root) {
    postorder(root->left);
    postorder(root->right);
    cout << root->key << " ";
  }
}
```

反推

已知中序遍历序列和另外一个序列可以求第三个序列.



1. 前序的第一个是 `root`，后序的最后一个是 `root`.
2. 先确定根节点，然后根据中序遍历，在根左边的为左子树，根右边的为右子树.
3. 对于每一个子树可以看成一个全新的树，仍然遵循上面的规律.

树上 BFS

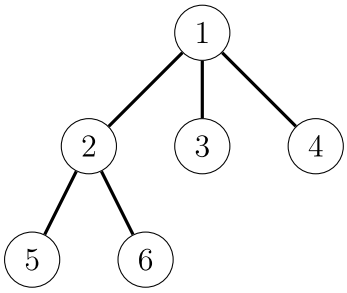
从树根开始，严格按照层次来访问节点.

BFS 过程中也可以顺便求出各个节点的深度和父亲节点.

树的层序遍历

树层序遍历是指按照从根节点到叶子节点的层次关系，一层一层的横向遍历各个节点. 根据 BFS 的定义可以知道，BFS 所得到的遍历顺序就是一种层序遍历. 但层序遍历要求将不同的层次区分开来，所以其结果通常以二维数组的形式表示.

例如，下图的树的层序遍历的结果是 `[[1], [2, 3, 4], [5, 6]]`（每一层从左向右）.



实现

```
vector<vector<int>> levelOrder(Node* root) {
  if (!root) {
    return {};
  }
  vector<vector<int>> res;
  queue<Node*> q;
  q.push(root);
```

```
8   while (!q.empty()) {
9     int currentLevelSize = q.size();  // 当前层的节点个数
10    res.push_back(vector<int>());
11    for (int i = 0; i < currentLevelSize; ++i) {
12      Node* cur = q.front();
13      q.pop();
14      res.back().push_back(cur->val);
15      for (Node* child : cur->children) {  // 把子节点都加入
16        q.push(child);
17      }
18    }
19  }
20  return res;
21 }
```

二叉树 Morris 遍历

二叉树遍历的核心问题是，当遍历当前节点的子节点后，如何返回当前节点并继续遍历. 遍历二叉树的递归方法和非递归方法都使用了栈结构，记录返回路径，来实现从下层到上层的移动. 其空间复杂度最好时为 $O(\log n)$，最坏时为 $O(n)$（二叉树呈线性）.

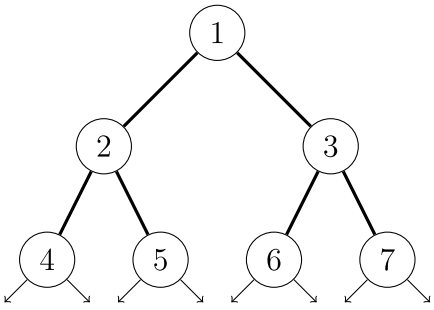
Morris 遍历的实质是避免使用栈，利用底层节点空闲的 `right` 指针指回上层的某个节点，从而完成下层到上层的移动.

Morris 遍历的过程

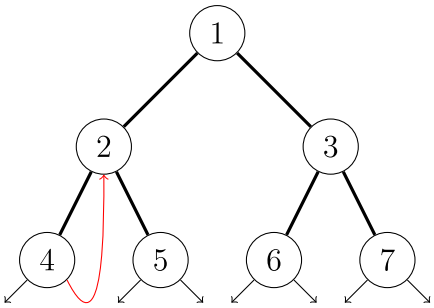
假设来到当前节点 `cur`，开始时来到根节点位置.

1. 如果 `cur` 为空时遍历停止，否则进行以下过程.
2. 如果 `cur` 没有左子树，`cur` 向右移动（`cur = cur->right`）.
3. 如果 `cur` 有左子树，找到左子树上最右的节点，记为 `mostRight`.
    - 如果 `mostRight` 的 `right` 指针指向空，让其指向 `cur`，然后 `cur` 向左移动（`cur = cur->left`）.
    - 如果 `mostRight` 的 `right` 指针指向 `cur`，将其修改为 `null`，然后 `cur` 向右移动（`cur = cur->right`）.

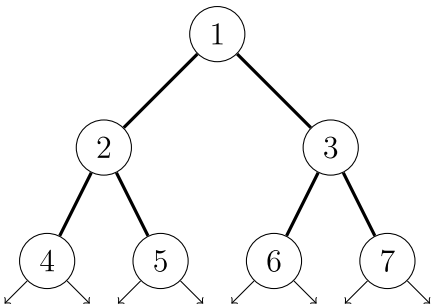
例如，`cur` 从节点 1 开始访问.



`cur` 第一次访问节点 2 时，找到左子树上最右的节点 4，将 4 的 `right` 指针指向 `cur`（节点 2）.



cur 通过 4 的 right 指针返回上层，第二次访问节点 2 时，找到左子树上最右节点 4，将 4 的 right 指针修改为 null，然后继续访问右子树．之后的过程省略．



整棵树的访问顺序是 1242513637．可以发现有左子树的节点访问两次，没有左子树的节点只访问一次．

实现

```cpp
void morris(TreeNode* root) {
  TreeNode* cur = root;
  while (cur) {
    if (!cur->left) {
      // 如果当前节点没有左子节点，则输出当前节点的值并进入右子树
      std::cout << cur->val << " ";
      cur = cur->right;
      continue;
    }
    // 找到当前节点的左子树的最右节点
    TreeNode* mostRight = cur->left;
    while (mostRight->right && mostRight->right != cur) {
      mostRight = mostRight->right;
    }
    if (!mostRight->right) {
      // 如果最右节点的right指针为空，将其指向当前节点，并进入左子树
      mostRight->right = cur;
      cur = cur->left;
    } else {
      // 如果最右节点的right指针指向当前节点，说明左子树已经遍历完毕，输出当前节点的值并进入右子树
      mostRight->right = nullptr;
      std::cout << cur->val << " ";
      cur = cur->right;
    }
```

```
  }
}
```

无根树

过程

树的遍历一般为深度优先遍历，这个过程中最需要注意的是避免重复访问结点.

由于树是无环图, 因此只需记录当前结点是由哪个结点访问而来, 此后进入除该结点外的所有相邻结点，即可避免重复访问.

实现

```
void dfs(int u, int from) {
  // 递归进入除了 from 之外的所有子结点
  // 对于出发结点，from 为空，故会访问所有相邻结点，这与期望一致
  for (int v : adj[u])
    if (v != from) {
      dfs(v, u);
    }
}

// 开始遍历时
int EMPTY_NODE = -1;  // 一个不存在的编号
int root = 0;         // 任取一个结点作为出发点
dfs(root, EMPTY_NODE);
```

有根树

对于有根树，需要区分结点的上下关系.

考察上面的遍历过程，若从根开始遍历，则访问到一个结点时 `from` 的值，就是其父结点的编号.

通过这个方式，可以对于无向的输入求出所有结点的父结点，以及子结点列表.

本页面部分内容引用自博文 二叉树：前序遍历、中序遍历、后续遍历[1]，遵循 CC 4.0 BY-SA 版权协议.

外部参考资料

[1] 二叉树: 前序遍历、中序遍历、后续遍历

11.6.2 树的直径

树上任意两节点之间最长的简单路径即为树的「直径」.

前置知识：树基础→11.6.1@p2621.

引入

显然，一棵树可以有多条直径，他们的长度相等.

可以用两次 DFS 或者树形 DP 的方法在 $O(n)$ 时间求出树的直径.

两次 DFS

首先从任意节点 y 开始进行第一次 DFS，到达距离其最远的节点，记为 z，然后再从 z 开始做第二次 DFS，到达距离 z 最远的节点，记为 z'，则 $\delta(z,z')$ 即为树的直径.

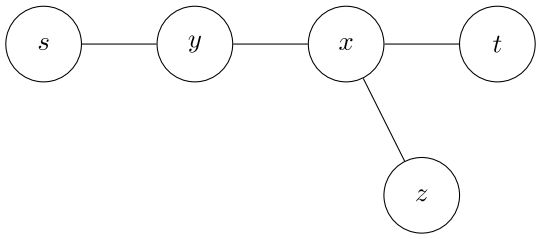
显然，如果第一次 DFS 到达的节点 z 是直径的一端，那么第二次 DFS 到达的节点 z' 一定是直径的一端. 我们只需证明在任意情况下，z 必为直径的一端.

定理：在一棵树上，从任意节点 y 开始进行一次 DFS，到达的距离其最远的节点 z 必为直径的一端.

证明

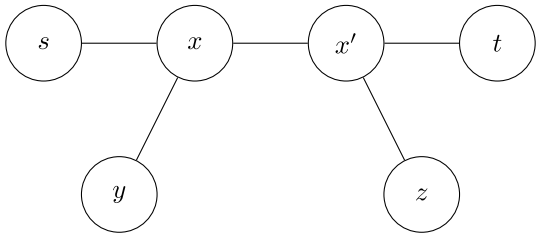
使用反证法. 记出发节点为 y. 设真实的直径是 $\delta(s,t)$，而从 y 进行的第一次 DFS 到达的距离其最远的节点 z 不为 t 或 s. 共分三种情况：

- 若 y 在 $\delta(s,t)$ 上：



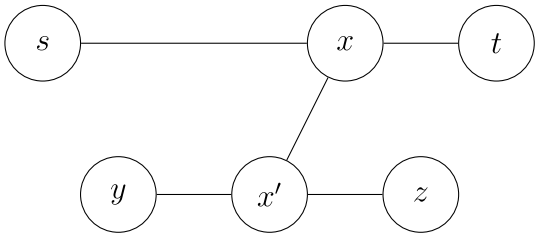
有 $\delta(y,z)>\delta(y,t)\Longrightarrow\delta(x,z)>\delta(x,t)\Longrightarrow\delta(s,z)>\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾.

- 若 y 不在 $\delta(s,t)$ 上，且 $\delta(y,z)$ 与 $\delta(s,t)$ 存在重合路径：



有 $\delta(y,z)>\delta(y,t)\Longrightarrow\delta(x,z)>\delta(x,t)\Longrightarrow\delta(s,z)>\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾.

- 若 y 不在 $\delta(s,t)$ 上，且 $\delta(y,z)$ 与 $\delta(s,t)$ 不存在重合路径：



有 $\delta(y,z)>\delta(y,t)\Longrightarrow\delta(x',z)>\delta(x',t)\Longrightarrow\delta(x,z)>\delta(x,t)\Longrightarrow\delta(s,z)>\delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾.

综上，三种情况下假设均会产生矛盾，故原定理得证.

负权边

上述证明过程建立在所有路径均不为负的前提下. 如果树上存在负权边，则上述证明不成立. 故若存在负权边，则无法使用两次 DFS 的方式求解直径.

如果需要求出一条直径上所有的节点，则可以在第二次 DFS 的过程中，记录每个点的前序节点，即可从直径的一端一路向前，遍历直径上所有的节点.

树形 DP

方法 1

我们记录当 1 为树的根时，每个节点作为子树的根向下，所能延伸的最长路径长度 d_1 与次长路径（与最长路径无公共边）长度 d_2，那么直径就是对于每一个点，该点 $d_1 + d_2$ 能取到的值中的最大值.

树形 DP 可以在存在负权边的情况下求解出树的直径.

如果需要求出一条直径上所有的节点，则可以在 DP 的过程中，记录下每个节点能向下延伸的最长路径与次长路径（定义同上）所对应的子节点，在求 d 的同时记下对应的节点 u，使得 $d = d_1[u] + d_2[u]$，即可分别沿着从 u 开始的最长路径的次长路径对应的子节点一路向某个方向（对于无根树，虽然这里指定了 1 为树的根，但仍需记录每点跳转的方向；对于有根树，一路向上跳即可），遍历直径上所有的节点.

方法 2

这里提供一种只使用一个数组进行的树形 DP 方法.

我们定义 $dp[u]$ 为以 u 为根的子树中，从 u 出发的最长路径. 那么容易得出转移方程：$dp[u] = \max(dp[u], dp[v] + w(u, v))$，其中的 v 为 u 的子节点，$w(u, v)$ 表示所经过边的权重.

对于树的直径，实际上是可以通过枚举从某个节点出发不同的两条路径相加的最大值求出. 因此，在 DP 求解的过程中，我们只需要在更新 $dp[u]$ 之前，计算 $d = \max(d, dp[u] + dp[v] + w(u, v))$ 即可算出直径 d.

例题

Luogu B4016 树的直径[1]

给定一棵 n 个节点的树，求其直径的长度. $1 \le n \le 10^5$.

两次 DFS 的参考实现

```cpp
#include <cstdio>
#include <vector>
using namespace std;

constexpr int N = 100000 + 10;

int n, c, d[N];
vector<int> E[N];

void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    d[v] = d[u] + 1;
    if (d[v] > d[c]) c = v;
    dfs(v, u);
  }
}

int main() {
```

```
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  d[c] = 0, dfs(c, 0);
  printf("%d\n", d[c]);
  return 0;
}

```

使用两个数组的树形 DP 参考实现

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

constexpr int N = 100000 + 10;

int n, d;
int d1[N], d2[N];
vector<int> E[N];

void dfs(int u, int fa) {
  d1[u] = d2[u] = 0;
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    int t = d1[v] + 1;
    if (t > d1[u])
      d2[u] = d1[u], d1[u] = t;
    else if (t > d2[u])
      d2[u] = t;
  }
  d = max(d, d1[u] + d2[u]);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  printf("%d\n", d);
  return 0;
}
```

```
37
```

使用一个数组的树形 DP 参考实现

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

constexpr int N = 100000 + 10;

int n, d;
int dp[N];
vector<int> E[N];

void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    d = max(d, dp[u] + dp[v] + 1);
    dp[u] = max(dp[u], dp[v] + 1);
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  printf("%d\n", d);
  return 0;
}

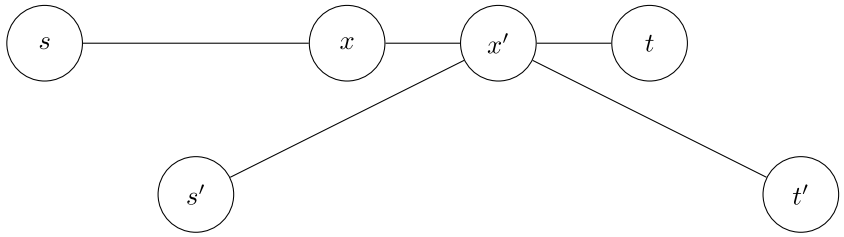
```

性质

树的直径具有如下性质：若树上所有边边权均为正，则树的所有直径中点重合.

证明

证明：使用反证法. 设两条中点不重合的直径分别为 $\delta(s,t)$ 与 $\delta(s',t')$，中点分别为 x 与 x'. 显然，$\delta(s,x)=\delta(x,t)=\delta(s',x')=\delta(x',t')$.



有 $\delta(s,t') = \delta(s,x) + \delta(x,x') + \delta(x',t') > \delta(s,x) + \delta(x,t) = \delta(s,t)$，与 $\delta(s,t)$ 为树上任意两节点之间最长的简单路径矛盾，故性质得证.

习题

- CodeChef, Diameter of Tree[2]
- Educational Codeforces Round 35, Problem F, Tree Destruction[3]
- ZOJ 3820 Building Fire Stations[4]
- CEOI2019/CodeForces 1192B. Dynamic Diameter[5]
- ICPC 2019 上海赛区网络赛 Lightning Routing I[6]
- NOIP2007 提高组 树网的核[7]
- SDOI2011 消防[8]
- APIO2010 巡逻[9]

外部参考资料

[1] Luogu B4016 树的直径

[2] CodeChef, Diameter of Tree

[3] Educational Codeforces Round 35, Problem F, Tree Destruction

[4] ZOJ 3820 Building Fire Stations

[5] CEOI2019/CodeForces 1192B. Dynamic Diameter

[6] ICPC 2019 上海赛区网络赛 Lightning Routing I

[7] NOIP2007 提高组 树网的核

[8] SDOI2011 消防

[9] APIO2010 巡逻

11.6.3 树的中心

Authors: littleparrot12345

定义

在树中，如果节点 x 作为根节点时，从 x 出发的最长链最短，那么称 x 为这棵树的中心.

性质

- 树的中心不一定唯一，但最多有 2 个，且这两个中心是相邻的.
- 树的中心一定位于树的直径上.
- 树上所有点到其最远点的路径一定交会于树的中心.
- 当树的中心为根节点时，其到达直径端点的两条链分别为最长链和次长链.
- 当通过在两棵树间连一条边以合并为一棵树时，连接两棵树的中心可以使新树的直径最小.
- 树的中心到其他任意节点的距离不超过树直径的一半.

求法

寻找一个点 x，使其作为根节点时，最长链的长度最短.

具体步骤

1. 维护 $len1_x$，表示节点 x 子树内的最长链.
2. 维护 $len2_x$，表示不与 $len1_x$ 重叠的最长链.
3. 维护 up_x，表示节点 x 子树外的最长链，该链必定经过 x 的父节点.
4. 找到点 x 使得 $\max(len1_x, up_x)$ 最小，那么 x 即为树的中心.

参考代码

```
// 这份代码默认节点编号从 1 开始，即 i ∈ [1,n]，使用vector存图
int d1[N], d2[N], up[N], x, y, mini = 1e9;  // d1,d2对应上文中的len1,len2

struct node {
  int to, val;  // to为边指向的节点，val为边权
};

vector<node> nbr[N];

void dfsd(int cur, int fa) {  // 求取len1和len2
  for (node nxtn : nbr[cur]) {
    int nxt = nxtn.to, w = nxtn.val;  // nxt为这条边通向的节点，val为边权
    if (nxt == fa) {
      continue;
    }
    dfsd(nxt, cur);
    if (d1[nxt] + w > d1[cur]) {  // 可以更新最长链
      d2[cur] = d1[cur];
      d1[cur] = d1[nxt] + w;
    } else if (d1[nxt] + w > d2[cur]) {  // 不能更新最长链，但可更新次长链
      d2[cur] = d1[nxt] + w;
    }
  }
}

void dfsu(int cur, int fa) {
  for (node nxtn : nbr[cur]) {
    int nxt = nxtn.to, w = nxtn.val;
    if (nxt == fa) {
      continue;
    }
    up[nxt] = up[cur] + w;
    if (d1[nxt] + w != d1[cur]) {  // 如果自己子树里的最长链不在nxt子树里
      up[nxt] = max(up[nxt], d1[cur] + w);
    } else {  // 自己子树里的最长链在nxt子树里，只能使用次长链
      up[nxt] = max(up[nxt], d2[cur] + w);
    }
    dfsu(nxt, cur);
  }
}

void GetTreeCenter() {  // 统计树的中心，记为x和y（若存在）
  dfsd(1, 0);
  dfsu(1, 0);
  for (int i = 1; i <= n; i++) {
```

```
    if (max(d1[i], up[i]) < mini) {  // 找到了当前max(len1[x],up[x])最小点
      mini = max(d1[i], up[i]);
      x = i;
      y = 0;
    } else if (max(d1[i], up[i]) == mini) {  // 另一个中心
      y = i;
    }
  }
}
```

示例

假设我们有一棵树，如下所示：

```
          A
         / \
        B   C
       / \   \
      D   E   F
```

- 树的直径为 $D \to B \to A \to C \to F$. 直径长度为 4.
- 树的中心为节点 A，因为从 A 出发的最长链（到 D 或 F）均为 2.
- 如果将 B 或 C 作为树的根，则从这些节点出发的最长链将增加，因此它们不是树的中心.

时间复杂度

上述算法的时间复杂度为 $O(n)$，其中 n 是树中节点的数量.

参考

- TutorialsPoint: Centers of a Tree[1]
- ProofWiki: Definition of Center of Tree[2]
- Wikipedia: Tree (graph theory)[3]

外部参考资料

[1] TutorialsPoint: Centers of a Tree 

[2] ProofWiki: Definition of Center of Tree 

[3] Wikipedia: Tree (graph theory) 

11.6.4 树的重心

Authors: Ir1d, Marcythm, LucienShui, Anguei, H-J-Granger, CornWorld, ttzc

本文介绍树的重心的概念和基本性质.

定义

如果在树 T 中删去某个结点 v 后，得到的图 $T \setminus \{v\}$ 中每个连通分量的大小均不超过原树结点数的一半，就称这个结点 v 为整棵树的 **重心**（centroid）. 删去某一结点后得到的最大连通分量的大小也称为该结点的 **重量**（weight）. 利用这一概念，重心的定义可以叙述为重量不超过树结点数的一半的结点.

「子树」

本文可能同时涉及无根树、有根树，以及将有根树换根到非根结点得到的树. 为了避免混淆，本文将用 T 表示无根树，用 $T^{(v)}$ 表示以结点 v 为根的有根树. 本文中提到的「子树」均是指 **有根树** 中，

一个结点及其所有子孙结点构成的树．有根树 $T^{(v)}$ 中，结点 u 对应子树记为 $T_u^{(v)}$．这样定义的子树当然包括整棵树本身．如果要明确不包括整棵树本身，将称呼它为「真子树」．

无根树中的「子树」通常指它的一个连通子图．在讨论重心时，部分作者会用「子树」一词特指不包含某结点的极大连通子图，或者特指将某条边删去后得到的两个连通分量之一．容易验证，这两种方式定义得到的「子树」集合是一致的，且不包括整棵树自身．由于它和有根树的子树集合并不一致，本文将避免对无根树使用「子树」概念．

在实际求解重心或处理某些问题时，通常存在一个默认的树根．此时，将非根结点 v 删掉得到的连通分量中，除了该结点的子结点对应子树外，还有一棵「向上」的子树．设结点 v 的父结点为 u，这棵「向上」的子树就是 $T_u^{(v)}$．本文在提及这类子图时，会显式地称呼它为「向上」的子树．若无特殊说明，本文提及的子树均不包括这类「向上」的子树．

注意到，得到的这些连通分量同样是无根树．通过删去树的重心，一棵树将变为若干棵至多原树一半大小的树．重心的这一特性使得在树上应用分治思想解决问题成为可能．这就是 点分治→11.6.9@p2692，也称为树的重心分解．

性质

本节讨论重心的性质．首先，树的重心有如下等价定义：

等价定义

树 T 中的结点 v 是它的重心，当且仅当以下任意一条成立：

- 无根树版本
  1. 在树中删去结点 v 后，得到的图 $T \setminus \{v\}$ 中每个连通分量的大小均不超过原树结点数的一半．
  2. 在所有删去某个结点后得到的最大连通分量大小中，删去结点 v 时所得到的值最小．
  3. 树中所有结点到某个结点的距离和中，到结点 v 的距离和最小．
- 有根树版本
  1. 当树以结点 v 为根时，任一真子树的大小均不超过原树结点数的一半．
  2. 在所有以某个结点为根时的最大真子树大小中，以结点 v 为根时所得到的值最小．
  3. 在所有以某个结点为根时所有结点的深度和中，以结点 v 为根时深度和最小．

证明

首先，引入一些记号．有根树和无根树两个版本的表述显然是等价的．定义 $W(x) = \max_{u\sim x} |\, T_u^{(x)} |$，其中，$u \sim x$ 表示 u 和 x 相邻．定义 $S(x) = \sum_{u\in T} d(u,x)$，其中，$d(u,x)$ 表示结点 u 和 x 之间的距离．那么，定义 1 相当于要求 $W(v) \le |\, T \,| /2$，定义 2 相当于要求 $v \in \arg\min_{x\in T} W(x)$，定义 3 相当于要求 $v \in \arg\min_{x\in T} S(x)$．需要证明的是，这三个条件是等价的．

将 $S(x)$ 理解为以 x 为根时的结点深度和，考虑它在树根从结点 v 换为相邻结点 u 时发生的变化．注意到，在树中删去边 (v,u) 后，得到的两个连通分量分别是子树 $T_v^{(u)}$ 和 $T_u^{(v)}$．在换根前后，子树 $T_v^{(u)}$ 中每个结点深度都增加 1，子树 $T_u^{(v)}$ 中每个结点深度都减少 1，所以，深度和的变化为

$$\Delta S_{v\to u} = S(u) - S(v) = |\, T_v^{(u)} | - |\, T_u^{(v)} | = |\, T \,| - 2 \,|\, T_u^{(v)} | \,.$$

所以，定义 1 的条件相当于要求 $\Delta S_{v\to u} \ge 0$ 对 v 的所有相邻结点 u 都成立，也就是说，v 是 $S(\cdot)$ 的极小值点．

转而设 v 是 $S(\cdot)$ 的（一个）最小值点（即定义 3），它必然存在，且一定是极小值点．考虑以 v 为根的有根树 $T^{(v)}$．设 $u \neq v$ 是一个非根结点，且从 v 到 u 的有向路径上，结点 v 的后一个结点为 y（可能就是 u），结点 u 的前一个结点为 x（可能就是 v）．那么，因为 $T_u^{(x)} \subseteq T_y^{(v)}$，所以有

$$2 \mid T_x^{(u)} \mid = 2 \mid T \mid - 2 \mid T_u^{(x)} \mid \geq 2 \mid T \mid - 2 \mid T_y^{(v)} \mid \geq \mid T \mid .$$

其中，最后一步用到 v 是 $S(\cdot)$ 的极小值点这一事实．此时，有两种情形：

- 存在结点 u 使得 $2 \mid T_x^{(u)} \mid = \mid T \mid$ 成立．此时，根据上述不等式，必然有 $(x, u) = (v, y)$，且 $\mid T_v^{(u)} \mid = \mid T_u^{(v)} \mid = \mid T \mid /2$．也就是说，使得等式成立的结点 u 必然与 v 相邻；又因为 v 的所有相邻结点对应子树的大小之和为 $\mid T \mid - 1 < \mid T \mid$，这样的结点 u 只能有一个．此时，对于所有其他结点 $u' \neq u, v$，必然存在 $x' \sim u'$ 使得 $\mid T_{x'}^{(u')} \mid > \mid T \mid /2$ 成立．满足条件 1 的结点集合为 $\{v, u\}$．

  注意到，删去任何结点时，得到的连通分量大小之和总是 $\mid T \mid - 1$，所以，只要有一个连通分量大小不小于 $\mid T \mid /2$，它就必然是最大连通分量．因此，对于这种情形，$W(v) = W(u) = \mid T \mid /2$，且对于所有 $u' \neq u, v$ 都有 $W(u') > \mid T \mid /2$．因此，满足条件 2 的结点集合为 $\arg\min W(\cdot) = \{v, u\}$．

  又因为 $\Delta S_{v \to u} = 0$，所以 $S(v) = S(u)$．因为 v 是最小值点，u 也一定是最小值点．而对于 $u' \neq u, v$ 都存在 $x' \sim u'$ 使得 $\mid T_{x'}^{(u')} \mid > \mid T \mid /2$，这违反了极小值点需要满足的条件，所以，$u'$ 一定也不是最小值点．因此，满足条件 3 的结点集合为 $\arg\min S(\cdot) = \{v, u\}$．
- 不存在结点 u 使得 $2 \mid T_x^{(u)} \mid = \mid T \mid$ 成立．此时，对于所有结点 $u \neq v$，都存在结点 $x \sim u$ 使得 $\mid T_x^{(u)} \mid > \mid T \mid /2$．重复前文分析可知，对于所有结点 $u \neq v$，都有 $W(u) > \mid T \mid /2$，且 u 不是 $S(\cdot)$ 的极小值点．因此，满足条件 1 的结点只有 v，且 $\arg\min W(\cdot) = \arg\min S(\cdot) = \{v\}$．

无论在哪一情形中，满足三个条件的集合都是相同的．这就证明了三个定义是等价的．

除了这些等价定义外，树的重心还有如下常见性质：

性质

1. 树的重心如果不唯一，则恰有两个．这两个重心相邻．而且，删去它们的连边后，树将变为两个大小相同的连通分量．
2. 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离．
3. 把两棵树通过一条边相连得到一棵新的树，那么新树的重心在连接原来两棵树的重心的路径上．
4. 一棵有根树的重心一定在根结点所在的重链上．一棵树的重心一定是该树根结点重子结点对应子树的重心的祖先．

证明

性质 1 可以从对重心等价定义的证明中得到．

性质 2 只需要考虑添加一个叶子结点的情形．这进一步分为两种情形：

- 树 T 只有一个重心 v．设 x 为新添加的叶子结点，且在新树中删去结点 v 得到的图 $T \cup \{x\} \setminus \{v\}$ 中，x 所在连通分量为 $B \cup \{x\}$．因为 v 是树 T 的唯一重心，所以 $2 \mid B \mid < \mid T \mid$，也就是 $2 \mid B \mid + 1 \leq \mid T \mid$．进而，有

  $2 \mid B \cup \{x\} \mid = 2(\mid B \mid + 1) \leq \mid T \mid + 1 = \mid T \cup \{x\} \mid .$

  因此，v 仍然是新树 $T \cup \{x\}$ 的重心．即使新树的重心不唯一，它也必然是 v 的相邻结点．因此，重心至多移动一条边．

- 树 T 有两个重心 u, v．此时，删去 (u, v) 得到的两个连通分量 $T_u^{(v)}$ 和 $T_v^{(u)}$ 大小相等，均为 $|T|/2$．不妨设新添加的叶子结点 x 连接在 v 所在的连通分量 $T_v^{(u)}$ 上．那么，因为

  $$|T_v^{(u)} \cup \{x\}| = |T|/2 + 1 > (|T| + 1)/2 = |T \cup \{x\}|/2,$$

  所以 u 不再是新树的重心．反过来，因为删去 v 后，仍然有大小为 $|T|/2$ 的连通分量 $T_u^{(v)}$，而其他连通分量的大小之和为

  $$|T \cup \{x\}| - 1 - |T_u^{(v)}| = |T|/2 \le |T_u^{(v)}|,$$

  所以，v 仍然是新树的重心．因为新树的结点数是奇数，重心必然唯一．所以，重心也至多移动一条边．

总结两种情形的分析可以发现，新树的重心一定在旧树的重心与新添加的叶子结点的路径上．

性质 3 可以通过归纳法说明．设 T 和 T' 连接时，新添加的边为 (x, y)，且 $x \in T, y \in T'$．不妨设新树的（一个）重心在 T 中．考虑从树 T 开始，将树 T' 中结点逐一添加为其叶子结点的过程．取定树 T 的一个重心 c．可以归纳地证明，树的（一个）重心始终在连接 c 与结点 x 的路径上．归纳起点显然．假设直到某一时刻为止，命题仍然成立．设此时重心为 v．由性质 2 的分析可知，新树的重心必然在连接新添加结点和当前重心 v 的路径上，又因为它不会移动到树 T 外，所以，只需要考虑路径与树 T 的公共部分，即连接当前重心 v 和结点 x 的路径上．根据归纳假设，v 就在连接 c 和结点 x 的路径上，所以，新树重心也必然在连接 c 和结点 x 的路径上．由归纳法可知，命题成立．

性质 4 只需要结合 重链剖分性质→11.6.6@p2658 即可说明．设树 T 的（一个）重心为 v．结点 v 是根结点时，命题显然成立．下面设 v 是非根结点，u 是它的父结点．因为 $|T_u^{(v)}| \le |T|/2$，所以，v 所在子树大小至少是 $|T|/2$．但是，只要它到根结点的路径上经过一次轻边，它所在子树大小将严格小于 $|T|/2$，矛盾．所以，它必然在根结点所在重链上．进而，根据重链的定义，根结点重子结点对应子树中根结点所在重链就是原树中根结点所在重链的一部分；而且，根据性质 3，在重子结点对应子树上添加根结点及其所有轻子结点对应子树后，重心位置将沿着当前重心和根结点路径移动，所以新重心必然是旧重心的祖先（包括其自身）．

求法

根据重心的等价定义，有两种方法可以在 $O(n)$ 时间内求出树的所有重心，其中，n 为树的大小．

DFS 统计子树大小

通过 DFS 计算每个子树的大小．对每个结点，记录它的所有子结点对应子树的大小，并利用总结点数减去当前子树大小得到「向上」的子树的大小，然后就可以依据定义找到重心了．

参考实现

```cpp
const int MAXN = 50005;

int n;
// 这份代码默认节点编号从 1 开始，即 i ∈ [1,n]
int siz[MAXN],  // 这个节点的「大小」(所有子树上节点数 + 该节点)
    weight[MAXN];  // 这个节点的「重量」，即所有子树「大小」的最大值
vector<int> centroids;  // 用于记录树的重心（存的是节点编号）
vector<int> g[MAXN];

void dfs(int cur, int fa) {  // cur 表示当前节点 (current)
  siz[cur] = 1;
  weight[cur] = 0;
  for (int v : g[cur]) {
```

```
    if (v != fa) {  // v 表示这条有向边所通向的节点
      dfs(v, cur);
      siz[cur] += siz[v];
      weight[cur] = max(weight[cur], siz[v]);
    }
  }
  weight[cur] = max(weight[cur], n - siz[cur]);
  if (weight[cur] <= n / 2) {  // 依照树的重心的定义统计
    centroids.push_back(cur);
  }
}

void get_centroids() { dfs(1, 0); }

```

换根 DP 统计深度和

还可以通过换根 DP 计算出以不同结点为根时，所有结点的深度和（即到当前根结点的距离和）. 根据定义，只需要找到使得这一深度和最小的结点即可.

参考实现

```
const int N = 50005;

int n, siz[N];
long long dp[N], ans[N];
vector<int> g[N], centroids;

// 求 1 号节点到所有其他节点的距离和
void dfs1(int u, int fa) {
  siz[u] = 1;
  dp[u] = 0;
  for (int v : g[u]) {
    if (v == fa) continue;
    dfs1(v, u);
    siz[u] += siz[v];
    dp[u] += dp[v] + siz[v];  // 子树节点到 u 的距离和
  }
}

// 通过换根 DP 求所有节点为树根时对应的距离和
void dfs2(int u, int fa) {
  for (int v : g[u]) {
    if (v == fa) continue;
    ans[v] = ans[u] - siz[v] + (n - siz[v]);
    dfs2(v, u);
  }
}

// 求树的重心
void get_centroids() {
  dfs1(1, 0);
```

```cpp
  ans[1] = dp[1];
  dfs2(1, 0);

  long long mini = std::numeric_limits<long long>::max();
  for (int i = 1; i <= n; i++) {
    if (ans[i] < mini) {
      mini = ans[i];
      centroids = {i};
    } else if (ans[i] == mini)
      centroids.push_back(i);
  }
}

```

例题

Codeforces Round 359 (Div. 1) B. Kay and Snowflake[1]

给定一棵有根树，求出每一棵子树的重心.

解题思路

根据性质 4，对于一棵以点 u 为根的子树，其重心一定在以 u 的重子结点为根的子树的重心到点 u 的路径上.

类似于上文提到的 DFS 求重心方法，对于每棵以结点 u 为根的子树，先求出其重子结点对应子树的重心（叶子结点的重心是其本身），然后从该重心出发向上判断路径上的结点是不是重心即可；若路径上的结点都不是重心，则 u 本身就是重心.

由于同一条重链上各子树的重心只会沿重链向上移动，每条重链上向上判断的总次数与重链长度同阶，所以在 $O(n)$ 时间内可以求出所有子树的重心.

参考代码

```cpp
#include <iostream>
#include <vector>
using namespace std;

constexpr int N = 3e5 + 5;

int n, q;  // 点数，询问数
int fa[N];
vector<int> son[N];
int siz[N],     // 子树大小
    ans[N],     // 以结点 u 为根的子树重心是 ans[u]
    weight[N];  // 结点重量（不包括向上的子树）

void dfs(int u) {
  siz[u] = 1, ans[u] = u;
  int hson = 0;  // 重子结点
```

```cpp
  for (int v : son[u]) {
    dfs(v);
    siz[u] += siz[v];
    if (siz[v] > weight[u]) weight[u] = siz[v], hson = v;
  }
  if (hson) {
    int p = ans[hson];
    while (p != u) {
      if (max(weight[p], siz[u] - siz[p]) <= siz[u] / 2) {
        ans[u] = p;
        break;
      } else
        p = fa[p];
    }
  }
}

int main() {
  ios::sync_with_stdio(false);
  cin >> n >> q;
  for (int v = 2; v <= n; v++) cin >> fa[v], son[fa[v]].push_back(v);
  dfs(1);
  while (q--) {
    int u;
    cin >> u;
    cout << ans[u] << '\n';
  }
  return 0;
}

```

习题

- Gym 101649G Godfather[2]
- POJ 1655 Balancing Act[3]
- 洛谷 P1364 医院设置[4]
- Codeforces 1406C Link Cut Centroids[5]
- Codeforces 708C Centroids[6]

参考资料

- 树的 "重心" 的一些性质及动态维护 - fanhq666[7]（博客园转载[8]）
- 树的直径、树的重心与树的点分治 - cyendra[9]
- 树的重心的性质及其证明 - suxxsfe[10]
- 《信息学奥林匹克辞典》2.4.7.11 章 1. 树的重心

外部参考资料

[1] Codeforces Round 359 (Div. 1) B. Kay and Snowflake 

[2] Gym 101649G Godfather 

[3] POJ 1655 Balancing Act 

[4] 洛谷 P1364 医院设置 

[5] Codeforces 1406C Link Cut Centroids 

[6] Codeforces 708C Centroids 

[7] 树的 "重心" 的一些性质及动态维护 - fanhq666

[8] 博客园转载

[9] 树的直径、树的重心与树的点分治 - cyendra

[10] 树的重心的性质及其证明 - suxxsfe

11.6.5 最近公共祖先

> **Authors:** ouuan, Backl1ght, billchenchina, CCXXXI, ChickenHu, ChungZH, cjsoft, countercurrent-time, diauweb, Early0v0, Enter-tainer, EtaoinWu, H-J-Granger, H-Shen, Henry-ZHR, HeRaNO, hsfzLZH1, huaruoji, iamtwz, imp2002, Ir1d, kenlig, Konano, Lyccrius, Marcythm, Menci, NachtgeistW, PeterlitsZo, psz2007, shuzhouliu, SkqLiao, sshwy, SukkaW, therehello, TrisolarisHD, ttzztztz, vincent-163, WAAutoMaton, Hunter19019

定义

最近公共祖先简称 LCA（Lowest Common Ancestor）．两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个． 为了方便，我们记某点集 $S=\{v_1,v_2,...,v_n\}$ 的最近公共祖先为 $\text{LCA}(v_1,v_2,...,v_n)$ 或 $\text{LCA}(S)$.

性质

> 本节 **性质** 部分内容翻译自 wcipeg[1]，并做过修改.

1. $\text{LCA}(\{u\})=u$;
2. u 是 v 的祖先，当且仅当 $\text{LCA}(u,v)=u$;
3. 如果 u 不为 v 的祖先并且 v 不为 u 的祖先，那么 u,v 分别处于 $\text{LCA}(u,v)$ 的两棵不同子树中;
4. 前序遍历中, $\text{LCA}(S)$ 出现在所有 S 中元素之前, 后序遍历中 $\text{LCA}(S)$ 则出现在所有 S 中元素之后;
5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\text{LCA}(A\cup B)=\text{LCA}(\text{LCA}(A),\text{LCA}(B))$;
6. 两点的最近公共祖先必定处在树上两点间的最短路上;
7. $d(u,v)=h(u)+h(v)-2h(\text{LCA}(u,v))$，其中 d 是树上两点间的距离，h 代表某点到树根的距离.

求法

朴素算法

过程

可以每次找深度比较大的那个点，让它向上跳．显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA． 或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇.

性质

朴素算法预处理时需要 dfs 整棵树，时间复杂度为 $O(n)$，单次查询时间复杂度为 $\Theta(n)$．如果树满足随机性质，则时间复杂度与这种随机树的期望高度有关.

倍增算法

过程

倍增算法是最经典的 LCA 求法，他是朴素算法的改进算法．通过预处理 $\text{fa}_{x,i}$ 数组，游标可以快速移动，大幅减少了游标跳转次数．$\text{fa}_{x,i}$ 表示点 x 的第 2^i 个祖先．$\text{fa}_{x,i}$ 数组可以通过 dfs 预处理出来.

现在我们看看如何优化这些跳转：在调整游标的第一阶段中，我们要将 u,v 两点跳转到同一深度．我们可以计算出 u,v 两点的深度之差，设其为 y．通过将 y 进行二进制拆分，我们将 y 次游标跳转优化为「y

的二进制表示所含 1 的个数」次游标跳转．在第二阶段中，我们从最大的 i 开始循环尝试，一直尝试到 0（包括 0），如果 $\mathrm{fa}_{u,i} \neq \mathrm{fa}_{v,i}$，则 $u \leftarrow \mathrm{fa}_{u,i}, v \leftarrow \mathrm{fa}_{v,i}$，那么最后的 LCA 为 $\mathrm{fa}_{u,0}$．

性质

倍增算法的预处理时间复杂度为 $O(n \log n)$，单次查询时间复杂度为 $O(\log n)$．另外倍增算法可以通过交换 fa 数组的两维使较小维放在前面．这样可以减少 cache miss 次数，提高程序效率．

例题

HDU 2586 How far away?[2] 树上最短路查询．

可先求出 LCA，再结合性质 7 进行解答．也可以直接在求 LCA 时求出结果．

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <vector>

constexpr int MXN = 40005;
using namespace std;
vector<int> v[MXN];
vector<int> w[MXN];

int fa[MXN][31], cost[MXN][31], dep[MXN];
int n, m;
int a, b, c;

// dfs，用来为 lca 算法做准备。接受两个参数：dfs 起始节点和它的父亲节点。
void dfs(int root, int fno) {
  // 初始化：第 2^0 = 1 个祖先就是它的父亲节点，dep 也比父亲节点多 1。
  fa[root][0] = fno;
  dep[root] = dep[fa[root][0]] + 1;
  // 初始化：其他的祖先节点：第 2^i 的祖先节点是第 2^(i-1) 的祖先节点的第
  // 2^(i-1) 的祖先节点。
  for (int i = 1; i < 31; ++i) {
    fa[root][i] = fa[fa[root][i - 1]][i - 1];
    cost[root][i] = cost[fa[root][i - 1]][i - 1] + cost[root][i - 1];
  }
  // 遍历子节点来进行 dfs。
  int sz = v[root].size();
  for (int i = 0; i < sz; ++i) {
    if (v[root][i] == fno) continue;
    cost[v[root][i]][0] = w[root][i];
    dfs(v[root][i], root);
  }
}

// lca。用倍增算法算取 x 和 y 的 lca 节点。
int lca(int x, int y) {
  // 令 y 比 x 深。
  if (dep[x] > dep[y]) swap(x, y);
  // 令 y 和 x 在一个深度。
  int tmp = dep[y] - dep[x], ans = 0;
```

```cpp
  for (int j = 0; tmp; ++j, tmp >>= 1)
    if (tmp & 1) ans += cost[y][j], y = fa[y][j];
  // 如果这个时候 y = x，那么 x，y 就都是它们自己的祖先。
  if (y == x) return ans;
  // 不然的话，找到第一个不是它们祖先的两个点。
  for (int j = 30; j >= 0 && y != x; --j) {
    if (fa[x][j] != fa[y][j]) {
      ans += cost[x][j] + cost[y][j];
      x = fa[x][j];
      y = fa[y][j];
    }
  }
  // 返回结果。
  ans += cost[x][0] + cost[y][0];
  return ans;
}

void Solve() {
  cin.tie(nullptr)->sync_with_stdio(false);
  // 初始化表示祖先的数组 fa，代价 cost 和深度 dep。
  memset(fa, 0, sizeof(fa));
  memset(cost, 0, sizeof(cost));
  memset(dep, 0, sizeof(dep));
  // 读入树：节点数一共有 n 个，查询 m 次，每一次查找两个节点的 lca 点。
  cin >> n >> m;
  // 初始化树边和边权
  for (int i = 1; i <= n; ++i) {
    v[i].clear();
    w[i].clear();
  }
  for (int i = 1; i < n; ++i) {
    cin >> a >> b >> c;
    v[a].push_back(b);
    v[b].push_back(a);
    w[a].push_back(c);
    w[b].push_back(c);
  }
  // 为了计算 lca 而使用 dfs。
  dfs(1, 0);
  for (int i = 0; i < m; ++i) {
    cin >> a >> b;
    cout << lca(a, b) << '\n';
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) Solve();
  return 0;
}
```

```
92
```

Tarjan 算法

过程

Tarjan 算法是一种 **离线算法**，需要使用 并查集→10.6.1@p1999 记录某个结点的祖先结点．做法如下：

1. 首先接受输入边（邻接链表）、查询边（存储在另一个邻接链表内）．查询边其实是虚拟加上去的边，为了方便，每次输入查询边的时候，将这个边及其反向边都加入到 `queryEdge` 数组里．
2. 然后对其进行一次 DFS 遍历，同时使用 `visited` 数组进行记录某个结点是否被访问过、`parent` 记录当前结点的父亲结点．
3. 其中涉及到了 **回溯思想**，我们每次遍历到某个结点的时候，认为这个结点的根结点就是它本身．让以这个结点为根节点的 DFS 全部遍历完毕了以后，再将这个结点的根节点设置为这个结点的父一级结点．
4. 回溯的时候，如果以该节点为起点，`queryEdge` 查询边的另一个结点也恰好访问过了，则直接更新查询边的 LCA 结果．
5. 最后输出结果．

性质

Tarjan 算法需要初始化并查集，所以预处理的时间复杂度为 $O(n)$．

朴素的 Tarjan 算法处理所有 m 次询问的时间复杂度为 $O(m\alpha(m+n,n)+n)$，但是 Tarjan 算法的常数比倍增算法大．存在 $O(m+n)$ 的实现．

⚠ 注意

并不存在「朴素 Tarjan LCA 算法中使用的并查集性质比较特殊，单次调用 `find()` 函数的时间复杂度为均摊 $O(1)$」这种说法．

以下的朴素 Tarjan 实现复杂度为 $O(m\alpha(m+n,n)+n)$．如果需要追求严格线性，可以参考 Gabow 和 Tarjan 于 1983 年的论文[3]．其中给出了一种复杂度为 $O(m+n)$ 的做法．

实现

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

class Edge {
 public:
  int toVertex, fromVertex;
  int next;
  int LCA;
  Edge() : toVertex(-1), fromVertex(-1), next(-1), LCA(-1) {};
  Edge(int u, int v, int n) : fromVertex(u), toVertex(v), next(n), LCA(-1)
{};
};

```

```
constexpr int MAX = 100;
int head[MAX], queryHead[MAX];
Edge edge[MAX], queryEdge[MAX];
int parent[MAX], visited[MAX];
int vertexCount, queryCount;

int find(int x) {
  if (parent[x] == x) {
    return x;
  } else {
    return parent[x] = find(parent[x]);
  }
}

void tarjan(int u) {
  parent[u] = u;
  visited[u] = 1;

  for (int i = head[u]; i != -1; i = edge[i].next) {
    Edge& e = edge[i];
    if (!visited[e.toVertex]) {
      tarjan(e.toVertex);
      parent[e.toVertex] = u;
    }
  }

  for (int i = queryHead[u]; i != -1; i = queryEdge[i].next) {
    Edge& e = queryEdge[i];
    if (visited[e.toVertex]) {
      queryEdge[i ^ 1].LCA = e.LCA = find(e.toVertex);
    }
  }
}

int main() {
  memset(head, 0xff, sizeof(head));
  memset(queryHead, 0xff, sizeof(queryHead));

  cin >> vertexCount >> queryCount;
  int count = 0;
  for (int i = 0; i < vertexCount - 1; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    edge[count] = Edge(start, end, head[start]);
    head[start] = count;
    count++;

    edge[count] = Edge(end, start, head[end]);
    head[end] = count;
    count++;
  }
```

```
67
68    count = 0;
69    for (int i = 0; i < queryCount; i++) {
70      int start = 0, end = 0;
71      cin >> start >> end;
72
73      queryEdge[count] = Edge(start, end, queryHead[start]);
74      queryHead[start] = count;
75      count++;
76
77      queryEdge[count] = Edge(end, start, queryHead[end]);
78      queryHead[end] = count;
79      count++;
80    }
81
82    tarjan(1);
83
84    for (int i = 0; i < queryCount; i++) {
85      Edge& e = queryEdge[i * 2];
86      cout << "(" << e.fromVertex << "," << e.toVertex << ") " << e.LCA
<< endl;
87    }
88
89    return 0;
90 }
91
```

用欧拉序列转化为 RMQ 问题

定义

对一棵树进行 DFS，无论是第一次访问还是回溯，每次到达一个结点时都将编号记录下来，可以得到一个长度为 $2n-1$ 的序列，这个序列被称作这棵树的欧拉序列.

在下文中，把结点 u 在欧拉序列中第一次出现的位置编号记为 $pos(u)$（也称作节点 u 的欧拉序），把欧拉序列本身记作 $E[1..2n-1]$.

过程

有了欧拉序列，LCA 问题可以在线性时间内转化为 RMQ 问题，即 $pos(LCA(u,v)) = \min\{pos(k)|\ k \in E[pos(u)..pos(v)]\}$.

这个等式不难理解：从 u 走到 v 的过程中一定会经过 $LCA(u,v)$，但不会经过 $LCA(u,v)$ 的祖先. 因此，从 u 走到 v 的过程中经过的欧拉序最小的结点就是 $LCA(u,v)$.

用 DFS 计算欧拉序列的时间复杂度是 $O(n)$，且欧拉序列的长度也是 $O(n)$，所以 LCA 问题可以在 $O(n)$ 的时间内转化成等规模的 RMQ 问题.

实现

参考代码

```
1 int dfn[N << 1], pos[N], tot, st[30][(N << 1) + 2],
2     rev[30][(N << 1) + 2];  // rev表示最小深度对应的节点编号
```

```
3
4  void dfs(int cur, int dep) {
5    dfn[++tot] = cur;
6    depth[tot] = dep;
7    pos[cur] = tot;
8    for (int i = head[t]; i; i = side[i].next) {
9      int v = side[i].to;
10     if (!pos[v]) {
11       dfs(v, dep + 1);
12       dfn[++tot] = cur, depth[tot] = dep;
13     }
14   }
15 }
16
17 void init() {
18   for (int i = 2; i <= tot + 1; ++i)
19     lg[i] = lg[i >> 1] + 1;  // 预处理 lg 代替库函数 log2 来优化常数
20   for (int i = 1; i <= tot; i++) st[0][i] = depth[i], rev[0][i] = dfn[i];
21   for (int i = 1; i <= lg[tot]; i++)
22     for (int j = 1; j + (1 << i) - 1 <= tot; j++)
23       if (st[i - 1][j] < st[i - 1][j + (1 << i - 1)])
24         st[i][j] = st[i - 1][j], rev[i][j] = rev[i - 1][j];
25       else
26         st[i][j] = st[i - 1][j + (1 << i - 1)],
27         rev[i][j] = rev[i - 1][j + (1 << i - 1)];
28 }
29
30 int query(int l, int r) {
31   int k = lg[r - l + 1];
32   return st[k][l] < st[k][r + 1 - (1 << k)] ? rev[k][l]
33                                             : rev[k][r + 1 - (1 << k)];
34 }
```

当我们需要查询某点对 (u, v) 的 LCA 时，查询区间 $[\min\{pos[u], pos[v]\}, \max\{pos[u], pos[v]\}]$ 上最小值的所代表的节点即可.

若使用 ST 表来解决 RMQ 问题，那么该算法不支持在线修改，预处理的时间复杂度为 $O(n \log n)$，每次查询 LCA 的时间复杂度为 $O(1)$.

利用 DFS 序转化为 RMQ 问题

欧拉序列的长度是 $2n - 1$，时空常数略大. 实际上，可以直接利用 DFS 时间戳 dfn 求 LCA.

考虑点对 (u, v) 的 LCA，记 $d = \mathrm{LCA}(u, v)$. 若 $u = v$，则 $d = u$，需要特判. 否则不妨设 $\mathrm{dfn}(u) < \mathrm{dfn}(v)$，此时，$v$ 一定不是 u 的祖先结点. 和欧拉序列不同，在 DFS 序中，区间 $(\mathrm{dfn}(u), \mathrm{dfn}(v)]$ 内不会出现 d. 但是，它一定包含位于自 d 到 v 的路径上 d 的那个子结点. 无论 u 是不是 v 的祖先结点，这一点总是成立的. 所以，只要找到区间 $[\mathrm{dfn}(u) + 1, \mathrm{dfn}(v)]$ 中深度最小的那个结点（未必唯一），它的父结点一定是 LCA. 其中，区间开始是 $\mathrm{dfn}(u) + 1$ 的原因是：若 u 是 v 的祖先，区间包含 u 时深度最小的结点将是 u 本身，而它的父结点并非 LCA.

由此，求 LCA 就又变成了一个 RMQ 问题.

如果不想存储额外的深度和父结点信息，可以直接在 DFS 序中 $\mathrm{dfn}(u)$ 位置处存储 $\mathrm{fa}(u)$. 然后，在这个序列上，利用时间戳比较两个值. 那么，区间 $[\mathrm{dfn}(u) + 1, \mathrm{dfn}(v)]$ 中该序列时间戳最小的就是 $\mathrm{LCA}(u, v)$.

这是因为，区间 $[\mathrm{dfn}(u)+1, \mathrm{dfn}(v)]$ 中结点一定是 d 的真子孙结点，这些结点父结点的 DFS 时间戳不会小于 $\mathrm{dfn}(d)$；而在区间内 d 的子结点处恰取得最小值 $\mathrm{dfn}(d)$，按照前文讨论，这样的子结点一定存在.

利用 DFS 序将 LCA 转化为 RMQ 问题的过程是 $O(n)$ 的，总复杂度取决于所用的 RMQ 方法. 利用 ST 表 $O(n \log n)$ 预处理、$O(1)$ 查询的参考实现如下：

参考实现

```cpp
class LcaSolver {
  std::array<std::vector<int>, 30> mi;
  std::vector<int> dfn, log;

  auto get(int u, int v) const { return dfn[u] < dfn[v] ? u : v; }

 public:
  // Initialize.
  // n - tree size; fa - the id of parent node, 0-indexed, 0 is the root.
  LcaSolver(int n, const std::vector<int>& fa) : dfn(n), log(n + 1) {
    // Build the tree.
    std::vector<std::vector<int>> gr(n);
    for (int i = 1; i < n; ++i) gr[fa[i]].push_back(i);
    // DFS.
    auto id = 0;
    auto dfs = [&](auto&& dfs, int cr) -> void {
      dfn[cr] = id++;
      for (int nt : gr[cr]) dfs(dfs, nt);
    };
    dfs(dfs, 0);
    // Build ST table.
    for (int i = 2; i <= n; ++i) log[i] = log[i >> 1] + 1;
    mi[0].assign(n, 0);
    for (int i = 0; i < n; ++i) mi[0][dfn[i]] = fa[i];
    for (int k = 1; k <= log[n]; ++k) {
      mi[k].assign(n - (1 << k) + 1, 0);
      for (int i = 0; i + (1 << k) <= n; ++i)
        mi[k][i] = get(mi[k - 1][i], mi[k - 1][i + (1 << (k - 1))]);
    }
  }

  // Find the LCA.
  int lca(int u, int v) const {
    if (u == v) return u;
    u = dfn[u], v = dfn[v];
    if (u > v) std::swap(u, v);
    auto k = log[v - u];
    return get(mi[k][u + 1], mi[k][v - (1 << k) + 1]);
  }
};

```

树链剖分

LCA 为两个游标跳转到同一条重链上时深度较小的那个游标所指向的点.

树链剖分的预处理时间复杂度为 $O(n)$，单次查询的时间复杂度为 $O(\log n)$，并且常数较小.

Link Cut Tree

在 Link Cut Tree→10.20.1@p2470 中，设连续两次 access→10.20.1@p2470 操作的点分别为 u 和 v，则第二次 access→10.20.1@p2470 操作返回的点即为 u 和 v 的 LCA.

在无 link 和 cut 等操作的情况下，使用 Link Cut Tree 单次查询的时间复杂度为 $O(\log n)$.

标准 RMQ

前面讲到了借助欧拉序将 LCA 问题转化为 RMQ 问题，其瓶颈在于 RMQ. 如果能做到 $O(n) \sim O(1)$ 求解 RMQ，那么也就能做到 $O(n) \sim O(1)$ 求解 LCA.

注意到欧拉序满足相邻两数之差为 1 或者 -1，所以可以使用 $O(n) \sim O(1)$ 的 加减 1RMQ→14.1@p3342 来做.

时间复杂度 $O(n) \sim O(1)$，空间复杂度 $O(n)$，支持在线查询，常数较大.

例题 Luogu P3379【模板】最近公共祖先（LCA） [4]

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

constexpr int N = 5e5 + 5;

struct PlusMinusOneRMQ {  // RMQ
  // Copyright (C) 2018 Skqliao. All rights served.
  constexpr static int M = 9;

  int blocklen, block, Minv[N], F[N / M * 2 + 5][M << 1], T[N], f[1 << M]
[M][M],
      S[N];

  void init(int n) {  // 初始化
    blocklen = std::max(1, (int)(log(n * 1.0) / log(2.0)) / 2);
    block = n / blocklen + (n % blocklen > 0);
    int total = 1 << (blocklen - 1);
    for (int i = 0; i < total; i++) {
      for (int l = 0; l < blocklen; l++) {
        f[i][l][l] = l;
        int now = 0, minv = 0;
        for (int r = l + 1; r < blocklen; r++) {
          f[i][l][r] = f[i][l][r - 1];
          if ((1 << (r - 1)) & i) {
            now++;
          } else {
            now--;
            if (now < minv) {
              minv = now;
              f[i][l][r] = r;
            }
          }
        }
```

```
      }
    }
    T[1] = 0;
    for (int i = 2; i < N; i++) {
      T[i] = T[i - 1];
      if (!(i & (i - 1))) {
        T[i]++;
      }
    }
  }

  void initmin(int a[], int n) {
    for (int i = 0; i < n; i++) {
      if (i % blocklen == 0) {
        Minv[i / blocklen] = i;
        S[i / blocklen] = 0;
      } else {
        if (a[i] < a[Minv[i / blocklen]]) {
          Minv[i / blocklen] = i;
        }
        if (a[i] > a[i - 1]) {
          S[i / blocklen] |= 1 << (i % blocklen - 1);
        }
      }
    }
    for (int i = 0; i < block; i++) {
      F[i][0] = Minv[i];
    }
    for (int j = 1; (1 << j) <= block; j++) {
      for (int i = 0; i + (1 << j) - 1 < block; i++) {
        int b1 = F[i][j - 1], b2 = F[i + (1 << (j - 1))][j - 1];
        F[i][j] = a[b1] < a[b2] ? b1 : b2;
      }
    }
  }

  int querymin(int a[], int L, int R) {
    int idl = L / blocklen, idr = R / blocklen;
    if (idl == idr)
      return idl * blocklen + f[S[idl]][L % blocklen][R % blocklen];
    else {
      int b1 = idl * blocklen + f[S[idl]][L % blocklen][blocklen - 1];
      int b2 = idr * blocklen + f[S[idr]][0][R % blocklen];
      int buf = a[b1] < a[b2] ? b1 : b2;
      int c = T[idr - idl - 1];
      if (idr - idl - 1) {
        int b1 = F[idl + 1][c];
        int b2 = F[idr - 1 - (1 << c) + 1][c];
        int b = a[b1] < a[b2] ? b1 : b2;
        return a[buf] < a[b] ? buf : b;
      }
      return buf;
```

```
    }
  }
} rmq;

int n, m, s;

struct Edge {
  int v, nxt;
} e[N * 2];

int tot, head[N];

void init(int n) {
  tot = 0;
  fill(head, head + n + 1, 0);
}

void addedge(int u, int v) {  // 加边
  ++tot;
  e[tot] = Edge{v, head[u]};
  head[u] = tot;

  ++tot;
  e[tot] = Edge{u, head[v]};
  head[v] = tot;
}

int dfs_clock, dfn[N * 2], dep[N * 2], st[N];

void dfs(int u, int fa, int d) {
  st[u] = dfs_clock;

  dfn[dfs_clock] = u;
  dep[dfs_clock] = d;
  ++dfs_clock;

  int v;
  for (int i = head[u]; i; i = e[i].nxt) {
    v = e[i].v;
    if (v == fa) continue;
    dfs(v, u, d + 1);
    dfn[dfs_clock] = u;
    dep[dfs_clock] = d;
    ++dfs_clock;
  }
}

void build_lca() {  // like init
  rmq.init(dfs_clock);
  rmq.initmin(dep, dfs_clock);
}

```

```cpp
int LCA(int u, int v) {  // 求解LCA，看题解用RMQ的方法
  int l = st[u], r = st[v];
  if (l > r) swap(l, r);
  return dfn[rmq.querymin(dep, l, r)];
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m >> s;

  init(n);
  int u, v;
  for (int i = 1; i <= n - 1; ++i) {
    cin >> u >> v;
    addedge(u, v);
  }

  dfs_clock = 0;
  dfs(s, s, 0);

  build_lca();

  for (int i = 1; i <= m; ++i) {
    cin >> u >> v;
    cout << LCA(u, v) << '\n';
  }

  return 0;
}
```

习题

- 祖孙询问[5]
- 货车运输[6]
- 点的距离[7]

参考资料

- 冷门科技——DFS 序求 LCA by Alex_Wei - 洛谷[8]

外部参考资料

[1] wcipeg

[2] HDU 2586 How far away?

[3] Gabow 和 Tarjan 于 1983 年的论文

[4] Luogu P3379【模板】最近公共祖先（LCA）

[5] 祖孙询问

[6] 货车运输

[7] 点的距离

[8] 冷门科技——DFS 序求 LCA by Alex_Wei - 洛谷

11.6.6 树链剖分

> **Authors:** GoodCoder666, Ir1d, Marcythm, ouuan, hsfzLZH1, Xeonacid, greyqz, Chrogeek, ftxj, sshwy, LuoshuiTianyi, hyp1231, sun2snow, Desmos666-xh

引入

树链剖分用于将树分割成若干条链的形式，以维护树上路径的信息.

具体来说，将整棵树剖分为若干条链，使它组合成线性结构，然后用其他的数据结构维护信息.

树链剖分（树剖/链剖）有多种形式，如 **重链剖分**，**长链剖分** 和用于 Link/cut Tree 的剖分（有时被称作「实链剖分」）. 大多数情况下（没有特别说明时），「树链剖分」都指「重链剖分」.

重链剖分可以将树上的任意一条路径划分成不超过 $O(\log n)$ 条连续的链，每条链上的点深度互不相同（即是自底向上的一条链，链上所有点的 LCA 为链的一个端点）.

重链剖分还能保证划分出的每条链上的结点 DFS 序连续，因此可以方便地用一些维护序列的数据结构（如线段树）来维护树上路径的信息. 例如：

1. 修改 **树上两点之间的路径上** 所有点的值.
2. 查询 **树上两点之间的路径上** 结点权值的 **和/极值/其它（在序列上可以用数据结构维护，便于合并的信息）**.

除了配合数据结构来维护树上路径信息，树剖还可以用来 $O(\log n)$（且常数较小）地求 LCA. 在某些题目中，还可以利用其性质来灵活地运用树剖.

重链剖分

我们给出一些定义：

定义 **重子结点** 表示其子结点中子树最大的子结点. 如果有多个子树最大的子结点，取其一. 如果没有子结点，就无重子结点.

定义 **轻子结点** 表示剩余的所有子结点.

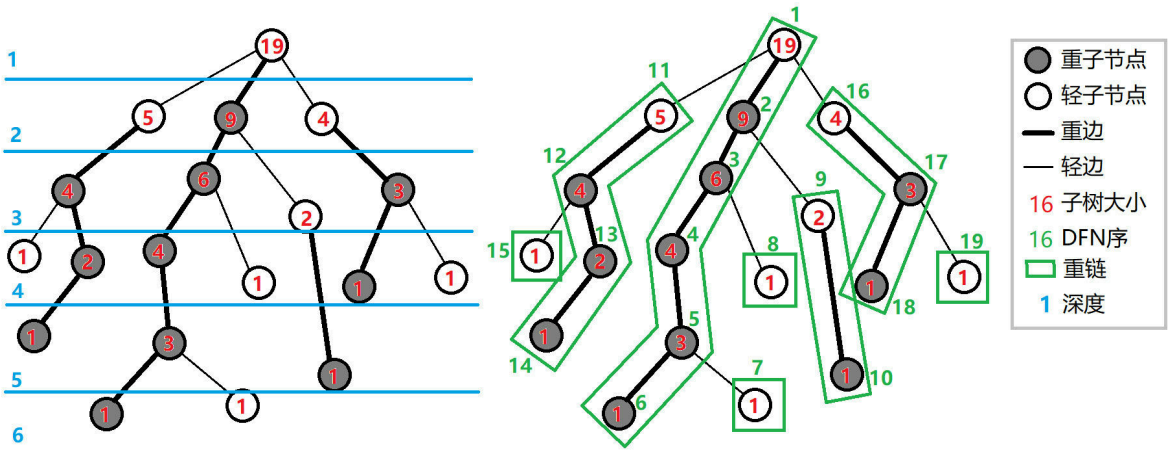
从这个结点到重子结点的边为 **重边**.

到其他轻子结点的边为 **轻边**.

若干条首尾衔接的重边构成 **重链**.

把落单的结点也当作重链，那么整棵树就被剖分成若干条重链.

如图：



实现

树剖的实现分两个 DFS 的过程. 伪代码如下:

第一个 DFS 记录每个结点的父结点（$father$）、深度（$depth$）、子树大小（$size$）、重子结点（$hson$）.

```
TREE-BUILD (u, dep)
1  u.hson ← 0
2  u.hson.size ← 0
3  u.depth ← dep
4  u.size ← 1
5  for each son v of u
6      u.size ← u.size + TREE-BUILD (v, dep + 1)
7      v.father ← u
8      if v.size > u.hson.size
9          u.hson ← v
10 return u.size
```

第二个 DFS 记录所在链的链顶（top，应初始化为结点本身）、重边优先遍历时的 DFS 序（dfn）、DFS 序对应的结点编号（$rank$）.

```
TREE-DECOMPOSITION (u, top)
1 u.top ← top
2 tot ← tot + 1
3 u.dfn ← tot
4 rank(tot) ← u
5 if u.hson is not 0
6     TREE-DECOMPOSITION (u.hson, top)
7     for each son v of u
8         if v is not u.hson
9             TREE-DECOMPOSITION (v, v)
```

以下为代码实现.

我们先给出一些定义:

- $\mathrm{fa}(x)$ 表示结点 x 在树上的父亲.
- $\mathrm{dep}(x)$ 表示结点 x 在树上的深度.
- $\mathrm{siz}(x)$ 表示结点 x 的子树的结点个数.
- $\mathrm{son}(x)$ 表示结点 x 的 **重儿子**.
- $\mathrm{top}(x)$ 表示结点 x 所在 **重链** 的顶部结点（深度最小）.
- $\mathrm{dfn}(x)$ 表示结点 x 的 **DFS 序**，也是其在线段树中的编号.
- $\mathrm{rnk}(x)$ 表示 DFS 序所对应的结点编号，有 $\mathrm{rnk}(\mathrm{dfn}(x)) = x$.

我们进行两遍 DFS 预处理出这些值，其中第一次 DFS 求出 $\mathrm{fa}(x)$，$\mathrm{dep}(x)$，$\mathrm{siz}(x)$，$\mathrm{son}(x)$，第二次 DFS 求出 $\mathrm{top}(x)$，$\mathrm{dfn}(x)$，$\mathrm{rnk}(x)$.

```cpp
void dfs1(int u, int f) {
  fa[u] = f, dep[u] = dep[f] + 1, siz[u] = 1;
  for (auto v : G[u]) {
    if (v == f) continue;
    dfs1(v, u);
    siz[u] += siz[v];
    if (siz[v] > siz[son[u]]) son[u] = v;
  }
}

void dfs2(int u, int ftop) {
  top[u] = ftop, dfn[u] = ++idx, rnk[idx] = u;
  if (son[u]) dfs2(son[u], ftop);
  for (auto v : G[u])
```

```
15     if (v != son[u] && v != fa[u]) dfs2(v, v);
16 }
```

重链剖分的性质

树上每个结点都属于且仅属于一条重链.

重链开头的结点一定不是重子结点（因为重链开头的结点要么是根，要么是其父亲结点的轻子结点）.

所有的重链将整棵树 **完全剖分**.

在剖分时 **重边优先遍历**，最后树的 DFS 序上，重链内的 DFS 序是连续的. 按 DFN 排序后的序列即为剖分后的链.

一棵子树内的 DFS 序是连续的.

可以发现，当我们向下经过一条 **轻边** 时，所在子树的大小至少会除以二.

因此，对于树上的任意一条路径，把它拆分成从 LCA→11.6.5@p2646 分别向两边往下走，分别最多走 $O(\log n)$ 次，因此，树上的每条路径都可以被拆分成不超过 $O(\log n)$ 条重链.

怎么有理有据地卡树剖

一般情况下树剖的 $O(\log n)$ 常数不满很难卡，如果要卡只能建立二叉树深度低.

于是我们可以考虑折中方案.

我们建立一棵 $\sqrt{n}$ 个结点的二叉树. 对于每个结点到其儿子的边，我们将其替换成一条长度为 $\sqrt{n}$ 的链.

这样子我们可以将随机询问轻重链切换次数卡到平均 $\frac{\log n}{2}$ 次，同时有 $O(\sqrt{n}\log n)$ 的深度.

加上若干随机叶子看上去可以卡树剖. 但是树剖常数小有可能卡不掉.

常见应用

路径上维护

用树链剖分求树上两点路径权值和，伪代码如下：

```
TREE-PATH-SUM (u, v)
1 tot ← 0
2 while u. top is not v. top
3       if u. top. depth < v. top. depth
4             SWAP(u, v)
5       tot ← tot + sum of values between u and u. top
6       u ← u. top. father
7 tot ← tot + sum of values between u and v
8 return tot
```

链上的 DFS 序是连续的，可以使用线段树、树状数组维护.

每次选择深度较大的链往上跳，直到两点在同一条链上.

同样的跳链结构适用于维护、统计路径上的其他信息.

子树维护

有时会要求，维护子树上的信息，譬如将以 x 为根的子树的所有结点的权值增加 v.

在 DFS 搜索的时候，子树中的结点的 DFS 序是连续的.

每一个结点记录 bottom 表示所在子树连续区间末端的结点.

这样就把子树信息转化为连续的一段区间信息.

求最近公共祖先

不断向上跳重链，当跳到同一条重链上时，深度较小的结点即为 LCA.

向上跳重链时需要先跳所在重链顶端深度较大的那个.

参考代码：

```
int lca(int u, int v) {
  while (top[u] != top[v]) {
    if (dep[top[u]] > dep[top[v]])
      u = fa[top[u]];
    else
      v = fa[top[v]];
  }
  return dep[u] > dep[v] ? v : u;
}
```

换根操作

考虑一类新的问题：除了树链剖分支持的基本操作外，加上了换根操作.

由于树链剖分维护的信息是静态的，不支持动态修改. 同时，不可能每次换根后重新预处理信息，复杂度过高. 那么，需要充分利用之前得到的信息来帮助解决换根操作.

对于路径修改和查询操作，由于树上两点之间的简单路径唯一，所以不会发生变化，与正常的处理方式相同.

对于子树修改和查询操作，一般的思路就是将换根后的子树映射到原来的子树. 这需要分类讨论操作子树的根结点、换根后的整个树的根结点，以及原来树的根结点的相对位置关系. 具体细节详见 后文例题「LOJ 139. 树链剖分」→11@p2597.

例题

本文通过例题展示如何应用重链剖分. 首先是一道模板题.

「ZJOI2008」树的统计[1]

对一棵有 n 个结点，结点带权值的静态树，进行三种操作共 q 次：

1. 修改单个结点的权值；
2. 查询 u 到 v 的路径上的最大权值；
3. 查询 u 到 v 的路径上的权值之和.

保证 $1 \le n \le 30000$，$0 \le q \le 200000$.

解答

根据题面以及前文所述性质，线段树需要维护三种操作：

1. 单点修改；
2. 区间查询最大值；
3. 区间查询和.

单点修改很容易实现.

由于子树的 DFS 序连续（无论是否树剖都是如此），修改一个结点的子树只用修改这一段连续的 DFS 序区间.

问题是如何修改/查询两个结点之间的路径.

考虑我们是如何用 **倍增法求解 LCA** 的. 首先我们 **将两个结点提到同一高度，然后将两个结点一起向上跳**. 对于树链剖分也可以使用这样的思想.

在向上跳的过程中，如果当前结点在重链上，向上跳到重链顶端，如果当前结点不在重链上，向上跳一个结点. 如此直到两结点相同. 沿途更新/查询区间信息.

对于每个询问，最多经过 $O(\log n)$ 条重链，每条重链上线段树的复杂度为 $O(\log n)$，因此总时间复杂度为 $O(n \log n + q \log^2 n)$. 实际上重链个数很难达到 $O(\log n)$（可以用完全二叉树卡满），所以树剖在一般情况下常数较小.

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#define lc o << 1
#define rc o << 1 | 1
constexpr int MAXN = 60010;
constexpr int inf = 2e9;
int n, a, b, w[MAXN], q, u, v;
int cur, h[MAXN], nxt[MAXN], p[MAXN];
int siz[MAXN], top[MAXN], son[MAXN], dep[MAXN], fa[MAXN], dfn[MAXN],
rnk[MAXN],
    cnt;
char op[10];

void add_edge(int x, int y) {  // 加边
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

struct SegTree {
  int sum[MAXN * 4], maxx[MAXN * 4];

  void build(int o, int l, int r) {
    if (l == r) {
      sum[o] = maxx[o] = w[rnk[l]];
      return;
    }
    int mid = (l + r) >> 1;
    build(lc, l, mid);
    build(rc, mid + 1, r);
    sum[o] = sum[lc] + sum[rc];
    maxx[o] = std::max(maxx[lc], maxx[rc]);
  }

  int query1(int o, int l, int r, int ql, int qr) {  // 查询 max
    if (l > qr || r < ql) return -inf;
    if (ql <= l && r <= qr) return maxx[o];
    int mid = (l + r) >> 1;
```

```
40      return std::max(query1(lc, l, mid, ql, qr), query1(rc, mid + 1, r,
    ql, qr));
41    }
42
43    int query2(int o, int l, int r, int ql, int qr) {  // 查询 sum
44      if (l > qr || r < ql) return 0;
45      if (ql <= l && r <= qr) return sum[o];
46      int mid = (l + r) >> 1;
47      return query2(lc, l, mid, ql, qr) + query2(rc, mid + 1, r, ql, qr);
48    }
49
50    void update(int o, int l, int r, int x, int t) {  // 更新
51      if (l == r) {
52        maxx[o] = sum[o] = t;
53        return;
54      }
55      int mid = (l + r) >> 1;
56      if (x <= mid)
57        update(lc, l, mid, x, t);  // 左右分别更新
58      else
59        update(rc, mid + 1, r, x, t);
60      sum[o] = sum[lc] + sum[rc];
61      maxx[o] = std::max(maxx[lc], maxx[rc]);
62    }
63  } st;
64
65  void dfs1(int o) {
66    son[o] = -1;
67    siz[o] = 1;
68    for (int j = h[o]; j; j = nxt[j])
69      if (!dep[p[j]]) {
70        dep[p[j]] = dep[o] + 1;
71        fa[p[j]] = o;
72        dfs1(p[j]);
73        siz[o] += siz[p[j]];
74        if (son[o] == -1 || siz[p[j]] > siz[son[o]]) son[o] = p[j];
75      }
76  }
77
78  void dfs2(int o, int t) {
79    top[o] = t;
80    cnt++;
81    dfn[o] = cnt;
82    rnk[cnt] = o;
83    if (son[o] == -1) return;
84    dfs2(son[o], t);
85    for (int j = h[o]; j; j = nxt[j])
86      if (p[j] != son[o] && p[j] != fa[o]) dfs2(p[j], p[j]);
87  }
88
89  int querymax(int x, int y) {  // 查询，看main函数理解一下
90    int ret = -inf, fx = top[x], fy = top[y];
```

```
  while (fx != fy) {
    if (dep[fx] >= dep[fy])
      ret = std::max(ret, st.query1(1, 1, n, dfn[fx], dfn[x])), x = fa[fx];
    else
      ret = std::max(ret, st.query1(1, 1, n, dfn[fy], dfn[y])), y = fa[fy];
    fx = top[x];
    fy = top[y];
  }
  if (dfn[x] < dfn[y])
    ret = std::max(ret, st.query1(1, 1, n, dfn[x], dfn[y]));
  else
    ret = std::max(ret, st.query1(1, 1, n, dfn[y], dfn[x]));
  return ret;
}

int querysum(int x, int y) {
  int ret = 0, fx = top[x], fy = top[y];
  while (fx != fy) {
    if (dep[fx] >= dep[fy])
      ret += st.query2(1, 1, n, dfn[fx], dfn[x]), x = fa[fx];
    else
      ret += st.query2(1, 1, n, dfn[fy], dfn[y]), y = fa[fy];
    fx = top[x];
    fy = top[y];
  }
  if (dfn[x] < dfn[y])
    ret += st.query2(1, 1, n, dfn[x], dfn[y]);
  else
    ret += st.query2(1, 1, n, dfn[y], dfn[x]);
  return ret;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i < n; i++) cin >> a >> b, add_edge(a, b), add_edge(b, a);
  for (int i = 1; i <= n; i++) cin >> w[i];
  dep[1] = 1;
  dfs1(1);
  dfs2(1, 1);
  st.build(1, 1, n);
  cin >> q;
  while (q--) {
    cin >> op >> u >> v;
    if (!strcmp(op, "CHANGE")) st.update(1, 1, n, dfn[u], v);
    if (!strcmp(op, "QMAX")) cout << querymax(u, v) << '\n';
    if (!strcmp(op, "QSUM")) cout << querysum(u, v) << '\n';
  }
  return 0;
```

```
143 }
```

然后是一道带换根操作的重链剖分模板题.

LOJ 139. 树链剖分[2]

给定一棵 n 个结点的树（初始根结点为 1），要求支持如下的 m 次操作：

- 换根，将结点 u 设为新的树根.
- 修改路径上结点权值，将结点 u 和结点 v 之间路径上的所有结点（包括这两个结点）的权值增加 w.
- 修改子树上结点权值，将以结点 u 为根的子树上的所有结点的权值增加 w.
- 询问路径，询问结点 u 和结点 v 之间路径上的所有结点（包括这两个结点）的权值和.
- 询问子树，询问以结点 u 为根的子树上的所有结点的权值和.

$1 \leq n, m \leq 10^5$.

解法

先以 1 作为根结点跑 DFS，预处理出树链剖分所必需的信息. 为方便表述，称以 1 为根结点的树为「原始树」，而称经历了若干次换根操作之后的树为「当前树」. 在操作过程中，需要维护 $root$ 为当前树的根结点. 由于线段树中存储的是原始树的 DFS 序的信息，所以每次查询和修改时，都需要将当前树的查询和修改操作转换到原始树上.

对于换根操作，我们直接令 $root \leftarrow u$. 对于路径操作，由于换根不影响路径，所以直接在原始树上做对应操作即可.

重点考虑操作子树的问题. 我们根据 u 和 $root$ 的相对位置关系做分类讨论：

- $u = root$：这是最特殊的情况，相当于对整棵树做操作. 为此，直接对线段树的根结点打上标记或查询答案即可.
- u 是 $root$ 在原始树上的祖先，即 u 位于 1 到 $root$ 的简单路径上.

  这是最值得注意的情况. 定义 v 为原始树上 u 到 $root$ 的简单路径上除 u 以外的深度最小的点，可以发现原始树上 v 及其子树以外的部分恰好是当前树上 u 及其子树.

  考虑如何高效找到 v. 我们先令 $v \leftarrow root$, 然后沿着重链往上跳直到 $\mathrm{dep}(\mathrm{top}(v)) \leq \mathrm{dep}(u) + 1$.
  - 若 $\mathrm{dep}(\mathrm{top}(v)) = \mathrm{dep}(u) + 1$，令 $v \leftarrow \mathrm{top}(v)$. 此时，$v$ 是 u 的一个轻儿子.
  - 若 $\mathrm{dep}(\mathrm{top}(v)) < \mathrm{dep}(u) + 1$，亦即 $\mathrm{dep}(\mathrm{top}(v)) \leq \mathrm{dep}(u)$，这说明 u, v 处在同一条重链上. 根据同一条重链上 DFS 序连续的性质，所求的 v 必然满足 $\mathrm{dfn}(v) = \mathrm{dfn}(u) + 1$. 所以，可以令 $v \leftarrow \mathrm{rnk}(\mathrm{dfn}(u) + 1)$.

  注意，这两种情形中可以合并：在跳完之后可以直接令

  $v \leftarrow \mathrm{rnk}(\mathrm{dfn}(\mathrm{top}(v)) + \mathrm{dep}(u) + 1 - \mathrm{dep}(\mathrm{top}(v)))$.

  容易验证，利用这一表达式找到的 v，和分类讨论找到的 v 是等价的. 参考实现中就用到了这一表达式.

  由于 v 子树覆盖的区间为 $[\mathrm{dfn}(v), \mathrm{dfn}(v) + \mathrm{siz}(v))$，所以只需要对 $[1, \mathrm{dfn}(v)) \cup [\mathrm{dfn}(v) + \mathrm{siz}(v), n]$ 操作即可.
- 其它情况. 可以发现换根操作不会影响 u 的子树，用正常的方式维护即可.

这样做的复杂度与不带换根的做法相同，均为 $O(n \log^2 n)$.

参考代码

```cpp
#include <iostream>
#include <vector>

using lint = long long;
constexpr int N = 1e5 + 10;

int n, m, root, a[N];
std::vector<int> G[N];

int idx, fa[N], dep[N], siz[N], son[N], top[N], dfn[N], rnk[N];

struct SegmentTree {  // 区间加，区间求和的线段树实现
  lint sum[N << 2], lzy[N << 2];

  void maketag(int u, int l, int r, lint x) {
    sum[u] += (r - l + 1) * x;
    lzy[u] += x;
  }

  void pushup(int u) { sum[u] = sum[u << 1] + sum[u << 1 | 1]; }

  void pushdown(int u, int l, int r) {
    int mid = l + r >> 1;
    maketag(u << 1, l, mid, lzy[u]);
    maketag(u << 1 | 1, mid + 1, r, lzy[u]);
    lzy[u] = 0;
  }

  void build(int u, int l, int r) {
    if (l == r) {
      sum[u] = a[rnk[l]];
      return;
    }
    int mid = l + r >> 1;
    build(u << 1, l, mid), build(u << 1 | 1, mid + 1, r);
    pushup(u);
  }

  void upd(int u, int l, int r, int ql, int qr, lint x) {
    if (ql <= l && r <= qr) {
      maketag(u, l, r, x);
      return;
    }
    int mid = l + r >> 1;
    pushdown(u, l, r);
    if (ql <= mid) upd(u << 1, l, mid, ql, qr, x);
    if (qr > mid) upd(u << 1 | 1, mid + 1, r, ql, qr, x);
    pushup(u);
  }

  lint ask(int u, int l, int r, int ql, int qr) {
```

```
52      if (ql <= l && r <= qr) return sum[u];
53      int mid = l + r >> 1;
54      pushdown(u, l, r);
55      if (qr <= mid) return ask(u << 1, l, mid, ql, qr);
56      if (ql > mid) return ask(u << 1 | 1, mid + 1, r, ql, qr);
57      return ask(u << 1, l, mid, ql, qr) + ask(u << 1 | 1, mid + 1, r,
    ql, qr);
58    }
59  } T;
60
61  void dfs1(int u, int f) {  // 树剖预处理 1
62    fa[u] = f, dep[u] = dep[f] + 1, siz[u] = 1;
63    for (auto v : G[u]) {
64      if (v == f) continue;
65      dfs1(v, u);
66      siz[u] += siz[v];
67      if (siz[v] > siz[son[u]]) son[u] = v;
68    }
69  }
70
71  void dfs2(int u, int tp) {  // 树剖预处理 2
72    top[u] = tp, dfn[u] = ++idx, rnk[idx] = u;
73    if (son[u]) dfs2(son[u], tp);
74    for (auto v : G[u])
75      if (v != fa[u] && v != son[u]) dfs2(v, v);
76  }
77
78  void modify_path(int u, int v, int w) {  // 正常树剖
79    while (top[u] != top[v]) {
80      if (dep[top[u]] < dep[top[v]]) std::swap(u, v);
81      T.upd(1, 1, n, dfn[top[u]], dfn[u], w);
82      u = fa[top[u]];
83    }
84    if (dep[u] < dep[v]) std::swap(u, v);
85    T.upd(1, 1, n, dfn[v], dfn[u], w);
86  }
87
88  void modify_subtree(int u, int w) {
89    if (u == root)
90      T.maketag(1, 1, n, w);
91    else if (dfn[u] <= dfn[root] && dfn[root] <= dfn[u] + siz[u] - 1) {
92      int v = root;
93      while (dep[top[v]] > dep[u] + 1) v = fa[top[v]];  // 向上跳
94      v = rnk[dfn[top[v]] + dep[u] + 1 - dep[top[v]]];  // 计算 v
95      // 上面这一句可以替换为如下两句:
96      // if (dep[top[v]] == dep[u] + 1) v = top[v];
97      // else if (dep[top[v]] < dep[u] + 1) v = rnk[dfn[u] + 1];
98      if (1 <= dfn[v] - 1) T.upd(1, 1, n, 1, dfn[v] - 1, w);
99      if (dfn[v] + siz[v] <= n) T.upd(1, 1, n, dfn[v] + siz[v], n, w);
100   } else
101     T.upd(1, 1, n, dfn[u], dfn[u] + siz[u] - 1, w);
102 }
```

```
103
104 lint query_path(int u, int v) {  // 正常树剖
105   lint res = 0;
106   while (top[u] != top[v]) {
107     if (dep[top[u]] < dep[top[v]]) std::swap(u, v);
108     res += T.ask(1, 1, n, dfn[top[u]], dfn[u]);
109     u = fa[top[u]];
110   }
111   if (dep[u] < dep[v]) std::swap(u, v);
112   res += T.ask(1, 1, n, dfn[v], dfn[u]);
113   return res;
114 }
115
116 lint query_subtree(int u) {
117   if (u == root) return T.sum[1];
118   if (dfn[u] <= dfn[root] && dfn[root] <= dfn[u] + siz[u] - 1) {
119     int v = root;
120     while (dep[top[v]] > dep[u] + 1) v = fa[top[v]];
121     v = rnk[dfn[top[v]] + dep[u] + 1 - dep[top[v]]];
122     lint res = 0;
123     if (1 <= dfn[v] - 1) res += T.ask(1, 1, n, 1, dfn[v] - 1);
124     if (dfn[v] + siz[v] <= n) res += T.ask(1, 1, n, dfn[v] + siz[v], n);
125     return res;
126   }
127   return T.ask(1, 1, n, dfn[u], dfn[u] + siz[u] - 1);
128 }
129
130 int main() {
131   std::cin.tie(nullptr)->sync_with_stdio(false);
132   std::cin >> n, root = 1;
133   for (int i = 1; i <= n; ++i) std::cin >> a[i];
134   for (int i = 2, f; i <= n; ++i) {
135     std::cin >> f;
136     G[i].emplace_back(f);
137     G[f].emplace_back(i);
138   }
139   dfs1(1, 0), dfs2(1, 1), T.build(1, 1, n);  // 预处理
140   for (std::cin >> m; m; --m) {
141     int op, u, v, w;
142     std::cin >> op >> u;
143     if (op == 1)
144       root = u;  // 换根
145     else if (op == 2)
146       std::cin >> v >> w, modify_path(u, v, w);  // 修改路径
147     else if (op == 3)
148       std::cin >> w, modify_subtree(u, w);  // 修改子树
149     else if (op == 4)
150       std::cin >> v, std::cout << query_path(u, v) << "\n";  // 查询路径
151     else
152       std::cout << query_subtree(u) << "\n";  // 查询子树
153   }
154   std::cout.flush();
```

```
155   return 0;
156 }
157
```

最后是一道交互题，也是树剖的非传统应用.

Nauuo and Binary Tree[3]

有一棵以 1 为根的二叉树，你可以询问任意两点之间的距离，求出每个点的父亲.

结点数不超过 3000，你最多可以进行 30000 次询问.

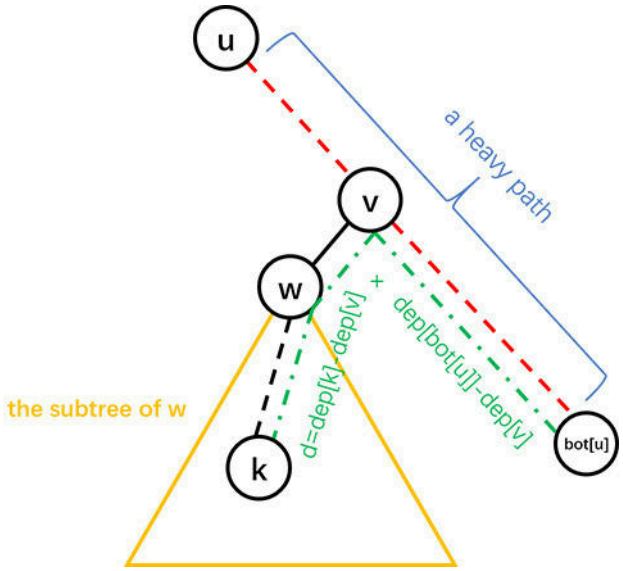
解法

首先可以通过 $n-1$ 次询问确定每个结点的深度.

然后考虑按深度从小到大确定每个结点的父亲，这样的话确定一个结点的父亲时其所有祖先一定都是已知的.

确定一个结点的父亲之前，先对树已知的部分进行重链剖分.

假设我们需要在子树 u 中找结点 k 所在的位置，我们可以询问 k 与 u 所在重链的尾端的距离，就可以进一步确定 k 的位置，具体见图:



其中红色虚线是一条重链，d 是询问的结果即 $dis(k, bot(u))$，v 的深度为 $(dep(k)+dep(bot(u))-d)/2$.

这样的话，如果 v 只有一个儿子，k 的父亲就是 v，否则可以递归地在 w 的子树中找 k 的父亲.

时间复杂度 $O(n^2)$，询问复杂度 $O(n\log n)$.

具体地，设 $T(n)$ 为最坏情况下在一棵大小为 n 的树中找到一个新结点的位置所需的询问次数，可以得到:

$$T(n)\le\begin{cases}0 & n=1\\ T\left(\left\lfloor\frac{n-1}{2}\right\rfloor\right)+1 & n\ge 2\end{cases}$$

$2999+\sum_{i=1}^{2999}T(i)\le 29940$，事实上这个上界是可以通过构造数据达到的，然而只要进行一些随机扰动（如对深度进行排序时使用不稳定的排序算法），询问次数很难超过 21000 次.

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <iostream>

using namespace std;

constexpr int N = 3010;

int n, fa[N], ch[N][2], dep[N], siz[N], son[N], bot[N], id[N];

int query(int u, int v) {
  printf("? %d %d\n", u, v);
  fflush(stdout);
  int d;
  scanf("%d", &d);
  return d;
}

void setFather(int u, int v) {
  fa[v] = u;
  if (ch[u][0])
    ch[u][1] = v;
  else
    ch[u][0] = v;
}

void dfs(int u) {
  if (ch[u][0]) dfs(ch[u][0]);
  if (ch[u][1]) dfs(ch[u][1]);

  siz[u] = siz[ch[u][0]] + siz[ch[u][1]] + 1;

  if (ch[u][1])
    son[u] = int(siz[ch[u][0]] < siz[ch[u][1]]);
  else
    son[u] = 0;

  if (ch[u][son[u]])
    bot[u] = bot[ch[u][son[u]]];
  else
    bot[u] = u;
}

void solve(int u, int k) {
  if (!ch[u][0]) {
    setFather(u, k);
    return;
```

```
  }
  int d = query(k, bot[u]);
  int v = bot[u];
  while (dep[v] > (dep[k] + dep[bot[u]] - d) / 2) v = fa[v];
  int w = ch[v][son[v] ^ 1];
  if (w)
    solve(w, k);
  else
    setFather(v, k);
}

int main() {
  int i;

  scanf("%d", &n);

  for (i = 2; i <= n; ++i) {
    id[i] = i;
    dep[i] = query(1, i);
  }

  sort(id + 2, id + n + 1, [](int x, int y) { return dep[x] < dep[y]; });

  for (i = 2; i <= n; ++i) {
    dfs(1);
    solve(1, id[i]);
  }

  printf("!");
  for (i = 2; i <= n; ++i) printf(" %d", fa[i]);
  printf("\n");
  fflush(stdout);

  return 0;
}

```

长链剖分

长链剖分本质上就是另外一种链剖分方式.

定义 **重子结点** 表示其子结点中子树深度最大的子结点. 如果有多个子树深度最大的子结点, 取其一. 如果没有子结点, 就无重子结点.

定义 **轻子结点** 表示剩余的子结点.

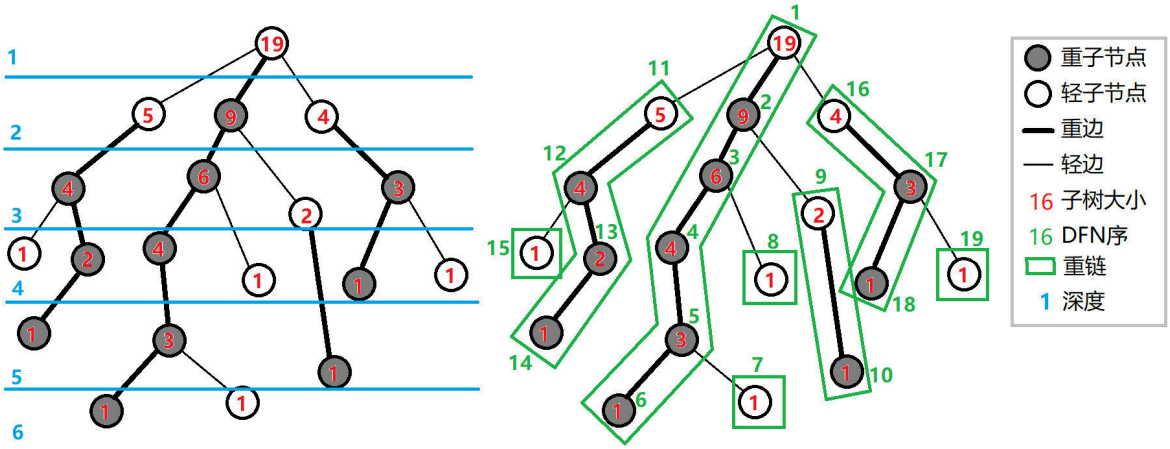
从这个结点到重子结点的边为 **重边**.

到其他轻子结点的边为 **轻边**.

若干条首尾衔接的重边构成 **重链**.

把落单的结点也当作重链, 那么整棵树就被剖分成若干条重链.

如图（这种剖分方式既可以看成重链剖分也可以看成长链剖分）:



长链剖分实现方式和重链剖分类似，这里就不再展开.

常见应用

首先，我们发现长链剖分从一个结点到根的路径的轻边切换条数是 $\sqrt{n}$ 级别的.

如何构造数据将轻重边切换次数卡满

我们可以构造这么一棵二叉树 T:

假设构造的二叉树参数为 D.

若 $D \neq 0$, 则在左儿子构造一棵参数为 $D-1$ 的二叉树，在右儿子构造一个长度为 $2D-1$ 的链.

若 $D = 0$, 则我们可以直接构造一个单独叶结点，并且结束调用.

这样子构造一定可以将单独叶结点到根的路径全部为轻边且需要 D^2 级别的结点数.

取 $D = \sqrt{n}$ 即可.

长链剖分优化 DP

一般情况下可以使用长链剖分来优化的 DP 会有一维状态为深度维.

我们可以考虑使用长链剖分优化树上 DP.

具体的，我们每个结点的状态直接继承其重儿子的结点状态，同时将轻儿子的 DP 状态暴力合并.

Codeforces 1009 F. Dominant Indices[4]

给定一棵有 n 个顶点的有根树，以顶点 1 作为根.

定义顶点 x 的深度数组为一个无限序列 $[d_{x,0}, d_{x,1}, d_{x,2}, ...]$，其中 $d_{x,i}$ 表示满足以下两个条件的顶点 y 的数量:

- x 是 y 的祖先;
- 从 x 到 y 的简单路径恰好经过 i 条边.

顶点 x 的深度数组的主导下标（dominant index）（简称顶点 x 的主导下标）定义为一个下标 j, 满足:

- 对于所有 $k < j$，都有 $d_{x,k} < d_{x,j}$;
- 对于所有 $k > j$，都有 $d_{x,k} \leq d_{x,j}$.

请计算树中每个顶点的主导下标.

解答

我们设 $f_{i,j}$ 表示在子树 i 内，和 i 距离为 j 的点数.

直接暴力转移时间复杂度为 $O(n^2)$

我们考虑每次转移我们直接继承重儿子的 DP 数组和答案，并且考虑在此基础上进行更新.

首先我们需要将重儿子的 DP 数组前面插入一个元素 1, 这代表着当前结点.

然后我们将所有轻儿子的 DP 数组暴力和当前结点的 DP 数组合并.

注意到因为轻儿子的 DP 数组长度为轻儿子所在重链长度，而所有重链长度和为 n.

也就是说，我们直接暴力合并轻儿子的总时间复杂度为 $O(n)$.

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int N = 1000005;

struct edge {
  int to, next;
} e[N * 2];

int head[N], tot, n;
int d[N], fa[N], mx[N];
int *f[N], g[N], mxp[N];
int dfn[N];

void add(int x, int y) {
  e[++tot] = edge{y, head[x]};
  head[x] = tot;
}

void dfs1(int x) {  // 第一次插入一个1
  d[x] = 1;
  for (int i = head[x]; i; i = e[i].next)
    if (e[i].to != fa[x]) {
      fa[e[i].to] = x;
      dfs1(e[i].to);
      d[x] = max(d[x], d[e[i].to] + 1);
      if (d[e[i].to] > d[mx[x]]) mx[x] = e[i].to;
    }
}

void dfs2(int x) {  // 第二次合并
  dfn[x] = ++*dfn;
  f[x] = g + dfn[x];
  if (mx[x]) dfs2(mx[x]);
  for (int i = head[x]; i; i = e[i].next)
    if (e[i].to != fa[x] && e[i].to != mx[x]) dfs2(e[i].to);
}
```

```
38
39 void getans(int x) {  // 暴力合并算答案
40   if (mx[x]) {
41     getans(mx[x]);
42     mxp[x] = mxp[mx[x]] + 1;
43   }
44   f[x][0] = 1;
45   if (f[x][mxp[x]] <= 1) mxp[x] = 0;
46   for (int i = head[x]; i; i = e[i].next)
47     if (e[i].to != fa[x] && e[i].to != mx[x]) {
48       getans(e[i].to);
49       int len = d[e[i].to];
50       for (int j = 0; j <= len - 1; j++) {
51         f[x][j + 1] += f[e[i].to][j];
52         if (f[x][j + 1] > f[x][mxp[x]]) mxp[x] = j + 1;
53         if (f[x][j + 1] == f[x][mxp[x]] && j + 1 < mxp[x]) mxp[x] = j + 1;
54       }
55     }
56 }
57
58 int main() {
59   cin.tie(nullptr)->sync_with_stdio(false);
60   cin >> n;
61   for (int i = 1; i < n; i++) {
62     int x, y;
63     cin >> x >> y;
64     add(x, y);
65     add(y, x);
66   }
67   dfs1(1);
68   dfs2(1);
69   getans(1);
70   for (int i = 1; i <= n; i++) cout << mxp[i] << '\n';
71 }
72
```

注意，一般情况下 DP 数组的内存分配为一条重链整体分配内存，链上不同的结点有不同的首位置指针.

DP 数组的长度我们可以根据子树最深结点算出.

当然长链剖分优化 DP 技巧非常多，包括但是不仅限于打标记等等. 这里不再展开.

参考 租酥雨的博客[5].

长链剖分求 k 级祖先

即询问一个点向父亲跳 k 次跳到的结点.

首先我们假设我们已经预处理了每一个结点的 2^i 级祖先.

现在我们假设我们找到了询问结点的 2^i 级祖先满足 $2^i \leq k < 2^{i+1}$.

我们考虑求出其所在重链的结点并且按照深度列入表格. 假设重链长度为 d.

同时我们在预处理的时候找到每条重链的根结点的 1 到 d 级祖先，同样放入表格.

根据长链剖分的性质，$k-2^i \le 2^i \le d$，也就是说，我们可以 $O(1)$ 在这条重链的表格上求出的这个结点的 k 级祖先.

预处理需要倍增出 2^i 次级祖先，同时需要预处理每条重链对应的表格.

预处理复杂度 $O(n \log n)$，询问复杂度 $O(1)$.

习题

- 「洛谷 P3379」【模板】最近公共祖先（LCA）[6]（树剖求 LCA 无需数据结构，可以用作练习）
- 「JLOI2014」松鼠的新家[7]（当然也可以用树上差分）
- 「HAOI2015」树上操作[8]
- 「洛谷 P3384」【模板】重链剖分/树链剖分[9]
- 「洛谷 P1505」[国家集训队] 旅游[10]
- 「NOI2015」软件包管理器[11]
- 「SDOI2011」染色[12]
- 「SDOI2014」旅行[13]
- 「洛谷 P3979」遥远的国度[14]
- 「POI2014」Hotel 加强版[15]（长链剖分优化 DP）
- 攻略[16]（长链剖分优化贪心）

外部参考资料

[1] 「ZJOI2008」树的统计

[2] LOJ 139. 树链剖分

[3] Nauuo and Binary Tree

[4] Codeforces 1009 F. Dominant Indices

[5] 租酥雨的博客

[6] 「洛谷 P3379」【模板】最近公共祖先（LCA）

[7] 「JLOI2014」松鼠的新家

[8] 「HAOI2015」树上操作

[9] 「洛谷 P3384」【模板】重链剖分/树链剖分

[10] 「洛谷 P1505」[国家集训队] 旅游

[11] 「NOI2015」软件包管理器

[12] 「SDOI2011」染色

[13] 「SDOI2014」旅行

[14] 「洛谷 P3979」遥远的国度

[15] 「POI2014」Hotel 加强版

[16] 攻略

11.6.7 树上启发式合并

Authors: abc1763613206, cesonic, Ir1d, MingqiHuang, xinchengo, xiaofu-15191, hsefz-ChenJunJie

引入

启发式算法是什么呢?

启发式算法是基于人类的经验和直观感觉，对一些算法的优化.

举个例子，最常见的就是并查集的启发式合并了，代码是这样的:

```
1 void merge(int x, int y) {
2   int xx = find(x), yy = find(y);
3   if (size[xx] < size[yy]) swap(xx, yy);
4   fa[yy] = xx;
5   size[xx] += size[yy];
```

```
6 }
```

在这里，对于两个大小不一样的集合，我们将小的集合合并到大的集合中，而不是将大的集合合并到小的集合中.

为什么呢？这个集合的大小可以认为是集合的高度（在正常情况下），而我们将集合高度小的并到高度大的显然有助于我们找到父亲.

让高度小的树成为高度较大的树的子树，这个优化可以称为启发式合并算法.

算法内容

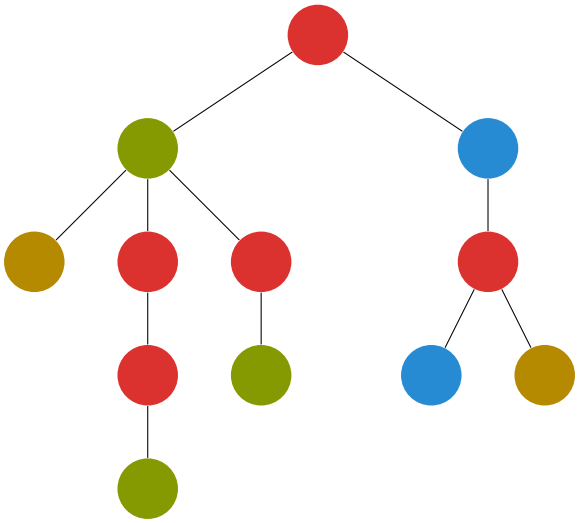
树上启发式合并（dsu on tree）对于某些树上离线问题可以速度大于等于大部分算法且更易于理解和实现的算法.

考虑下面的问题：树上数颜色[1].

例题引入

给出一棵 n 个节点以 1 为根的树，节点 u 的颜色为 c_u，现在对于每个结点 u 询问以 u 为根的子树里一共出现了多少种不同的颜色.

$n \leq 2 \times 10^5$.



对于这种问题解决方式大多是运用大量的数据结构（树套树等），如果可以离线，是不是有更简单的方法?

过程

既然支持离线，考虑预处理后 $O(1)$ 输出答案.

直接暴力预处理的时间复杂度为 $O(n^2)$，即对每一个子节点进行一次遍历，每次遍历的复杂度显然与 n 同阶，有 n 个节点，故复杂度为 $O(n^2)$.

可以发现，每个节点的答案由其子树和其本身得到，考虑利用这个性质处理问题.

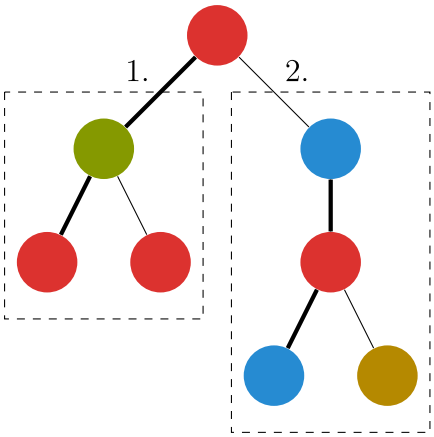
我们可以先预处理出每个节点子树的大小和它的重儿子，重儿子同树链剖分一样，是拥有节点最多子树的儿子，这个过程显然可以 $O(n)$ 完成.

我们用 cnt_i 表示颜色 i 的出现次数，ans_u 表示结点 u 的答案.

遍历一个节点 u，我们按以下的步骤进行遍历：

1. 先遍历 u 的轻（非重）儿子，并计算答案，但 **不保留遍历后它对 cnt 数组的影响**；

2. 遍历它的重儿子，**保留它对 cnt 数组的影响**；
3. 再次遍历 u 的轻儿子的子树结点，加入这些结点的贡献，以得到 u 的答案.



上图是一个例子.

这样，对于一个节点，我们遍历了一次重子树，两次非重子树，显然是最划算的.

通过执行这个过程，我们获得了这个节点所有子树的答案.

为什么不合并第一步和第三步呢？因为 cnt 数组不能重复使用，否则空间会太大，需要在 $O(n)$ 的空间内完成.

显然若一个节点 u 被遍历了 x 次，则其重儿子会被遍历 x 次，轻儿子（如果有的话）会被遍历 $2x$ 次.

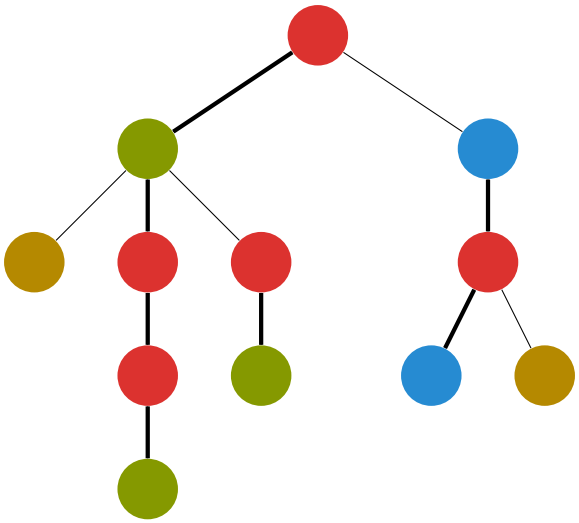
注意除了重儿子，每次遍历完 cnt 要清零.

证明

我们像树链剖分一样定义重边和轻边（连向重儿子的为重边，其余为轻边）. 关于重儿子和重边的定义，可以见下图，对于一棵有 n 个节点的树：

根节点到树上任意节点的轻边数不超过 $\log n$ 条. 我们设根到该节点有 x 条轻边该节点的子树大小为 y, 显然轻边连接的子节点的子树大小小于父亲的一半（若大于一半就不是轻边了），则 $y < n/2^x$, 显然 $n > 2^x$，所以 $x < \log n$.

又因为如果一个节点是其父亲的重儿子, 则它的子树必定在它的兄弟之中最多, 所以任意节点到根的路径上所有重边连接的父节点在计算答案时必定不会遍历到这个节点，所以一个节点的被遍历的次数等于它到根节点路径上的轻边数 $+1$（之所以要 $+1$ 是因为它本身要被遍历到），所以一个节点的被遍历次数 $= \log n + 1$, 总时间复杂度则为 $O(n(\log n + 1)) = O(n \log n)$，输出答案花费 $O(m)$.



图中标粗的即为重边，重边连向的子节点为重儿子

优化

在证明过程中提到，dsu on tree 利用了重链剖分中的轻重儿子概念加速合并．既然如此，我们也可以直接利用重链剖分得到的 dfs 序，化递归为迭代，进一步优化 dsu on tree 的常数．

dfs 序本身就有如下的性质：一个节点的子树在 dfs 序上一定连续．因此，可以倒序遍历 dfs 序数组．这样保证了在遍历到一个节点时，它的子树中的其他节点一定已经得到了处理．

重链剖分得到的 dfs 序有着如下优良的性质：一条重链在 dfs 序上一定是连续的．因此，当按照 dfs 序倒序遍历节点时，对于一条重链顶端的节点，要遍历的下一个节点一定不是该节点的父亲，所以要清空它的影响；除此之外，对于不在重链顶端的节点，遍历的前一个节点要么是自己的重儿子，要么是已经清除了影响的其他分支的节点，所以可以直接继承其影响．在此基础上，再利用 dfs 序快速统计所有轻儿子的影响，记录答案．

以上过程称为 dsu on tree 的非递归/迭代实现（也被称为 dsu on tree 的 dfs 序实现）．相较于原本递归实现，它减小了递归调用函数的时空开销，获得了不小的常数优化，**尤其在处理包含大量链状结构的树时，拥有显著的栈空间优势．**

实现

参考实现

- 递归实现

```cpp
#include <cstdio>
#include <vector>
using namespace std;

constexpr int N = 2e5 + 5;

int n, m;

// g[u]: 存储与 u 相邻的结点
vector<int> g[N];

// sz: 子树大小
// big: 重儿子
```

```cpp
// col: 结点颜色
// L[u]: 结点 u 的 DFS 序
// R[u]: 结点 u 子树中结点的 DFS 序的最大值
// Node[i]: DFS 序为 i 的结点
// totdfn: 节点计数器，也是当前遍历过节点的 DFS 序最大值
// ans: 存答案
// cnt[i]: 颜色为 i 的结点个数
// totColor: 目前出现过的颜色个数
int sz[N], big[N], col[N], L[N], R[N], Node[N], totdfn;
int ans[N], cnt[N], totColor;

void add(int u) {
  if (cnt[col[u]] == 0) ++totColor;
  cnt[col[u]]++;
}

void del(int u) {
  cnt[col[u]]--;
  if (cnt[col[u]] == 0) --totColor;
}

int getAns() { return totColor; }

void dfs0(int u, int p) {
  L[u] = ++totdfn;
  Node[totdfn] = u;
  sz[u] = 1;
  for (int v : g[u])
    if (v != p) {
      dfs0(v, u);
      sz[u] += sz[v];
      if (!big[u] || sz[big[u]] < sz[v]) big[u] = v;
    }
  R[u] = totdfn;
}

void dfs1(int u, int p, bool keep) {
  // 计算轻儿子的答案
  for (int v : g[u])
    if (v != p && v != big[u]) {
      dfs1(v, u, false);
    }
  // 计算重儿子答案并保留计算过程中的数据（用于继承）
  if (big[u]) {
    dfs1(big[u], u, true);
  }
  for (int v : g[u])
    if (v != p && v != big[u]) {
      // 子树结点的 DFS 序构成一段连续区间，可以直接遍历
      for (int i = L[v]; i <= R[v]; i++) {
        add(Node[i]);
      }
```

```
    }
  add(u);
  ans[u] = getAns();
  if (!keep) {
    for (int i = L[u]; i <= R[u]; i++) {
      del(Node[i]);
    }
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    g[u].push_back(v);
    g[v].push_back(u);
  }
  for (int i = 1; i <= n; i++) scanf("%d", &col[i]);
  dfs0(1, 0);
  dfs1(1, 0, false);
  scanf("%d", &m);
  for (int i = 1; i <= m; i++) {
    int q;
    scanf("%d", &q);
    printf("%d\n", ans[q]);
  }
  return 0;
}

```

- 非递归实现

```
#include <cstdio>
#include <vector>
using namespace std;

constexpr int N = 2e5 + 5;

int n, m;

// g[u]: 存储与 u 相邻的节点
vector<int> g[N];

// sz: 子树大小
// son[u]: 节点u的重儿子的编号
// col: 节点颜色
// dfn[u]: 节点 u 的 DFS 序
// bottom[u]: 节点 u 子树中节点的 DFS 序的最大值
// totdfn: 节点计数器，也是当前遍历过节点的 DFS 序最大值
// fa[u]: 节点u的父亲编号
// top[u]:节点u所在重链的顶端节点编号
// rnk[i]: DFS 序为 i 的节点
```

```cpp
// ans[u]: 存答案
// cnt[i]: 颜色为 i 的节点个数
// totColor: 目前出现过的颜色个数
int sz[N], son[N], col[N], dfn[N], fa[N], bottom[N], totdfn;
int top[N], rnk[N];
int ans[N], cnt[N], totColor;

void add(int u) {
  if (cnt[col[u]] == 0) ++totColor;
  cnt[col[u]]++;
}

void del(int u) {
  cnt[col[u]]--;
  if (cnt[col[u]] == 0) --totColor;
}

int getAns() { return totColor; }

void dfs0(int u, int p) {
  sz[u] = 1;
  fa[u] = p;
  for (int v : g[u])
    if (v != p) {
      dfs0(v, u);
      sz[u] += sz[v];
      if (!son[u] || sz[son[u]] < sz[v]) son[u] = v;
    }
}

void dfs1(int u, int t) {
  rnk[bottom[u] = dfn[u] = ++totdfn] = u;
  top[u] = t;
  if (!son[u]) return;
  dfs1(son[u], t);
  for (int v : g[u])
    if (v != fa[u] && v != son[u]) dfs1(v, v);
  bottom[u] = totdfn;
}

void dsu_on_tree() {
  for (int i = totdfn; i >= 1; i--) {
    int u = rnk[i];
    for (int v : g[u])
      if (v != fa[u] && v != son[u]) {
        for (int i = dfn[v]; i <= bottom[v]; i++) {
          add(rnk[i]);
        }
      }
    add(u);
    ans[u] = getAns();
    if (u == top[u]) {
```

```
      for (int i = dfn[u]; i <= bottom[u]; i++) {
        del(rnk[i]);
      }
    }
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d%d", &u, &v);
    g[u].push_back(v);
    g[v].push_back(u);
  }
  for (int i = 1; i <= n; i++) scanf("%d", &col[i]);
  dfs0(1, 1);
  dfs1(1, 1);
  dsu_on_tree();
  scanf("%d", &m);
  for (int i = 1; i <= m; i++) {
    int q;
    scanf("%d", &q);
    printf("%d\n", ans[q]);
  }
  return 0;
}

```

运用

1. 某些出题人设置的正解是 dsu on tree 的题

   如 CF741D[2]. 给一棵树，每个节点的权值是'a' 到'v' 的字母，每次询问要求在一个子树找一条路径，使该路径包含的字符排序后成为回文串.

   因为是排列后成为回文串，所以一个字符出现了两次相当于没出现，也就是说，这条路径满足 **最多有一个字符出现奇数次**.

   正常做法是对每一个节点 dfs，每到一个节点就强行枚举所有字母找到和它异或后结果为 1 的个数大于 1 的路径，再取最长值，这样是 $O(n^2 \log n)$ 的，可以用 dsu on tree 优化到 $O(n \log^2 n)$. 关于具体做法，可以参考下面的扩展阅读.
2. 可以用 dsu 乱搞的题

   可以水一些树套树的部分分（没有修改操作），而且 dsu 的复杂度优于树上莫队的 $O(n\sqrt{m})$.

练习题

CF600E Lomsat gelral[3]

题意翻译：树的节点有颜色，一种颜色占领了一个子树，当且仅当没有其他颜色在这个子树中出现得比它多. 求占领每个子树的所有颜色之和.

UOJ284 快乐游戏鸡[4]

CF1709E XOR Tree[5]

参考资料/扩展阅读

CF741D 作者介绍的 dsu on tree[6]

这位作者的题解[7]

外部参考资料

[1] 树上数颜色

[2] CF741D

[3] CF600E Lomsat gelral

[4] UOJ284 快乐游戏鸡

[5] CF1709E XOR Tree

[6] CF741D 作者介绍的 dsu on tree

[7] 这位作者的题解

11.6.8 虚树

Authors: HeRaNO, Ir1d, konnyakuxzy, ksyx, Xeonacid, konnyakuxzy, greyqz, sshwy, y-kx-b

引入

「SDOI2011」消耗战[1]

在一场战争中，战场由 n 个岛屿和 $n-1$ 个桥梁组成，保证每两个岛屿间有且仅有一条路径可达．现在，我军已经侦查到敌军的总部在编号为 1 的岛屿，而且他们已经没有足够多的能源维系战斗，我军胜利在望．已知在其他 k 个岛屿上有丰富能源，为了防止敌军获取能源，我军的任务是炸毁一些桥梁，使得敌军不能到达任何能源丰富的岛屿．由于不同桥梁的材质和结构不同，所以炸毁不同的桥梁有不同的代价，我军希望在满足目标的同时使得总代价最小．

侦查部门还发现，敌军有一台神秘机器．即使我军切断所有能源之后，他们也可以用那台机器．机器产生的效果不仅仅会修复所有我军炸毁的桥梁，而且会重新随机资源分布（但可以保证的是，资源不会分布到 1 号岛屿上）．不过侦查部门还发现了这台机器只能够使用 m 次，所以我们只需要把每次任务完成即可．

对于所有数据，$2 \le n \le 2.5 \times 10^5, 1 \le m \le 5 \times 10^5, \sum k_i \le 5 \times 10^5, 1 \le k_i \le n-1$．

朴素做法

对于上面那题，我们不难发现——如果树的点数很少，那么我们可以直接跑 DP．

首先我们称某次询问中被选中的点为——**「关键点」**．

设 $Dp(i)$ 表示——使 i 不与其子树中任意一个关键点连通的 **最小代价**．

设 $w(a,b)$ 表示 a 与 b 之间的边的权值．

则枚举 i 的儿子 v：

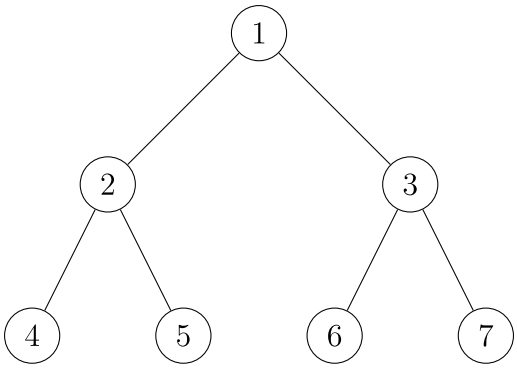
- 若 v 不是关键点：$Dp(i) = Dp(i) + \min\{Dp(v), w(i,v)\}$；
- 若 v 是关键点：$Dp(i) = Dp(i) + w(i,v)$．

很好，这样我们得到了一份 $O(nq)$ 的代码．

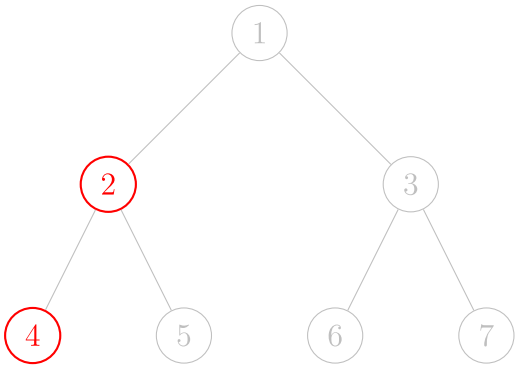
听起来很有意思．

优化做法

我们不难发现——其实很多点是没有用的．以下图为例：



如果我们选取的关键点是：



图中只有两个红色的点是 **关键点**，而别的点全都是「非关键点」.

对于这题来说，我们只需要保证红色的点无法到达 1 号节点就行了.

通过肉眼观察可以得出结论——1 号节点的右子树（虽然实际上可能有多个子树，但这里只有两个子树，所以暂时这么称呼了）一个红色节点都没有，**所以没必要去 DP 它**.

观察题目给出的条件，红色点（关键点）的总数是与 n 同阶的，也就是说实际上一次询问中红色的点对于整棵树来说是很稀疏的，所以如果我们能让复杂度由红色点的总数来决定就好了.

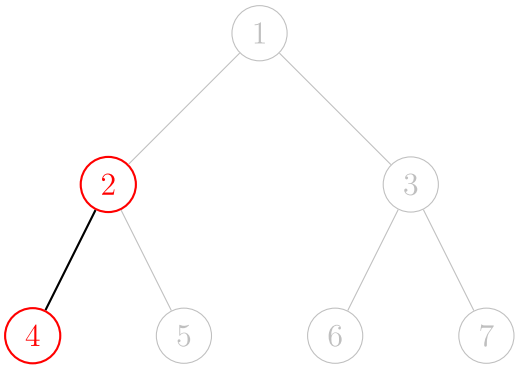
因此我们需要 **浓缩信息，把一整颗大树浓缩成一颗小树**.

虚树 Virtual Tree

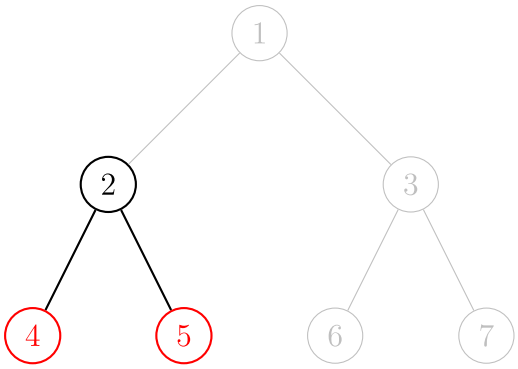
由此我们引出了 **「虚树」** 这个概念.

我们先直观地来看看虚树的样子.

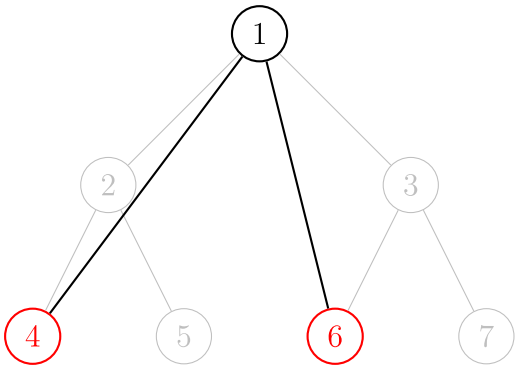
下图中，红色结点是我们选择的关键点. 红色和黑色结点都是虚树中的点. 黑色的边是虚树中的边.



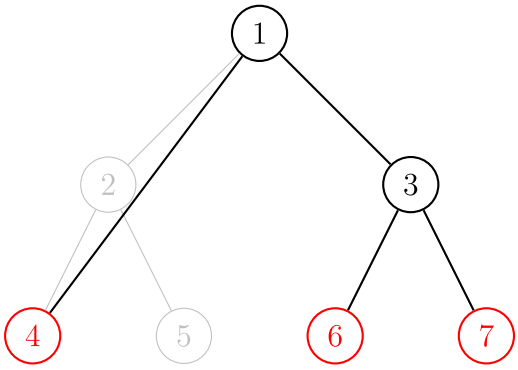
$\{2,4\}$ 的虚树



$\{4,5\}$ 的虚树



$\{4,6\}$ 的虚树



$\{4,6,7\}$ 的虚树

因为任意两个关键点的 LCA 也是需要保存重要信息的，所以我们需要保存它们的 LCA，因此虚树中不一定只有关键点.

不难发现虚树中祖先后代的关系并不会改变.（就是不会出现原本 a 是 b 的祖先结果后面 a 变成 b 的后代了之类的鬼事）

但我们不可能 $O(k^2)$ 暴力枚举 LCA，所以我们不难想到——首先将关键点按 DFS 序排序，然后排完序以后相邻的两个关键点（相邻指的是在排序后的序列中下标差值的绝对值等于 1）求一下 LCA，并把它加入虚树.

我们的当务之急就是如何构造虚树.

在提出方案之前，我们先确认一个事实——在虚树里，只要保证祖先后代的关系没有改变，就可以随意添加节点.

也就是，如果我们乐意，我们可以把原树中所有的点都加入虚树中，也不会导致 WA（虽然会导致 TLE）.

因此，我们为了方便，可以首先将 1 号节点加入虚树中，并且并不会影响答案.

第一种构造过程：二次排序 + LCA 连边

因为多个节点的 LCA 可能是同一个，所以我们不能多次将它加入虚树.

非常直观的一个方法是：

- 将关键点按 DFS 序排序；
- 遍历一遍，任意两个相邻的关键点求一下 LCA，并且判重；
- 然后根据原树中的祖先后代关系建树.

具体实现上，在 **关键点序列** 上，枚举 **相邻的两个数**，两两求得 LCA 并且加入序列 A 中.

因为 DFS 序的性质，此时的序列 A 已经包含了 **虚树中的所有点**，但是可能有重复.

所以我们把序列 A 按照 DFS 序 **从小到大排序并去重**.

最后，在序列 A 上，枚举 **相邻** 的两个 **点编号** x, y，求得它们的 LCA 并且连接 $\mathrm{LCA}(x, y), y$，虚树就构造完成了.

为什么连接 $\mathrm{LCA}(x, y)$ 和 y 可以做到不重不漏呢?

证明

如果 x 是 y 的祖先，那么 x 直接到 y 连边. 因为 DFS 序保证了 x 和 y 的 DFS 序是相邻的，所以 x 到 y 的路径上面没有关键点.

如果 x 不是 y 的祖先, 那么就把 $\mathrm{LCA}(x, y)$ 当作 y 的祖先, 根据上一种情况也可以证明 $\mathrm{LCA}(x, y)$ 到 y 点的路径上不会有关键点.

所以连接 $\mathrm{LCA}(x, y)$ 和 y，不会遗漏，也不会重复.

另外第一个点没有被一个节点连接会不会有影响呢? 因为第一个点一定是这棵树的根, 所以不会有影响，所以总边数就是 $m - 1$ 条.

因为至少要两个实点才能够召唤出来一个虚点, 再加上一个根节点, 所以虚树的点数就是实点数量的两倍.

时间复杂度 $O(m \log n)$，其中 m 为关键点数，n 为总点数.

实现

```
int dfn[MAXN];
int h[MAXN], m, a[MAXN], len;  // 存储关键点

bool cmp(int x, int y) {
  return dfn[x] < dfn[y];  // 按照 dfs 序排序
}

void build_virtual_tree() {
  sort(h + 1, h + m + 1, cmp);  // 把关键点按照 dfs 序排序
  for (int i = 1; i < m; ++i) {
    a[++len] = h[i];
    a[++len] = lca(h[i], h[i + 1]);  // 插入 lca
  }
  a[++len] = h[m];
  sort(a + 1, a + len + 1, cmp);  // 把所有虚树上的点按照 dfs 序排序
```

```
16    len = unique(a + 1, a + len + 1) - a - 1;  // 去重
17    for (int i = 1, lc; i < len; ++i) {
18      lc = lca(a[i], a[i + 1]);
19      conn(lc, a[i + 1]);  // 连边，如有边权 就是 distance(lc,a[i+1])
20    }
21  }
```

其实这样就足以构造一棵虚树了.

第二种构造过程：使用单调栈

如何使用单调栈构造虚树?

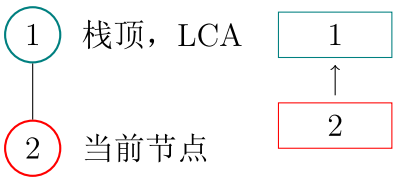
首先我们要明确一个目的——我们要用单调栈来维护一条虚树上的链.

也就是一个栈里相邻的两个节点在虚树上也是相邻的，而且栈是从底部到栈首单调递增的（指的是栈中节点 DFS 序单调递增），说白了就是某个节点的父亲就是栈中它下面的那个节点.

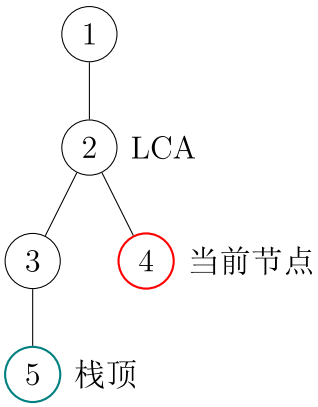
首先我们在栈中添加节点 1.

然后接下来按照 DFS 序从小到大添加关键节点.

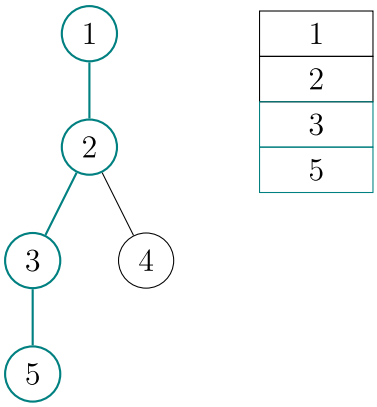
假如当前的节点与栈顶节点的 LCA 就是栈顶节点的话，则说明它们是在一条链上的. 所以直接把当前节点入栈就行了.



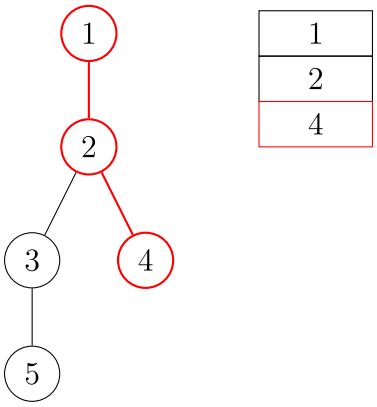
假如当前节点与栈顶节点的 LCA 不是栈顶节点的话:



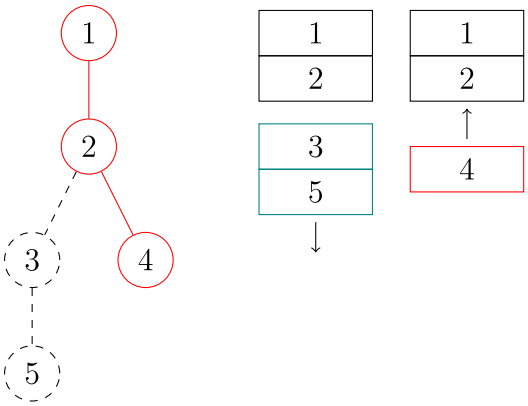
这时，当前单调栈维护的链是:



而我们需要把链变成：



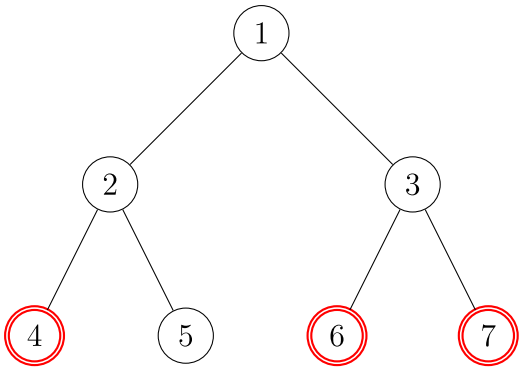
那么我们就把用虚线标出的结点弹栈即可，在弹栈前别忘了向它在虚树中的父亲连边.



假如弹出以后发现栈首不是 LCA 的话要让 LCA 入栈.

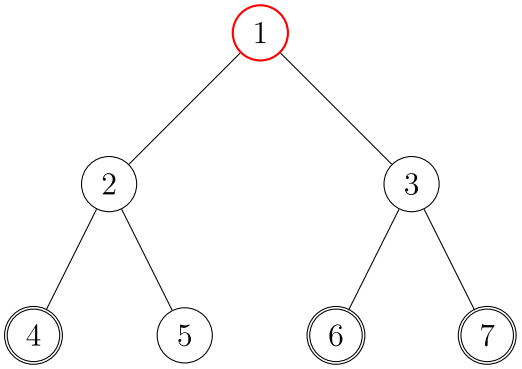
再把当前节点入栈就行了.

下面给出一个具体的例子. 假设我们要对下面这棵树的 4，6 和 7 号结点建立虚树：



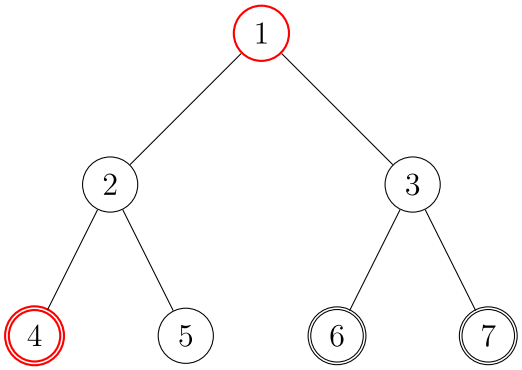
那么步骤是这样的：

- 将 3 个关键点 $6, 4, 7$ 按照 DFS 序排序，得到序列 $[4, 6, 7]$.
- 将 1 入栈.

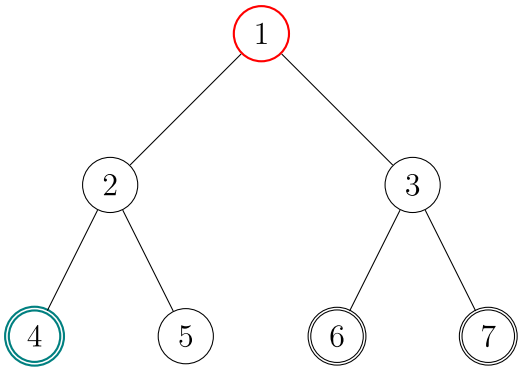


我们用红色的点代表在栈内的点，青色的点代表从栈中弹出的点.

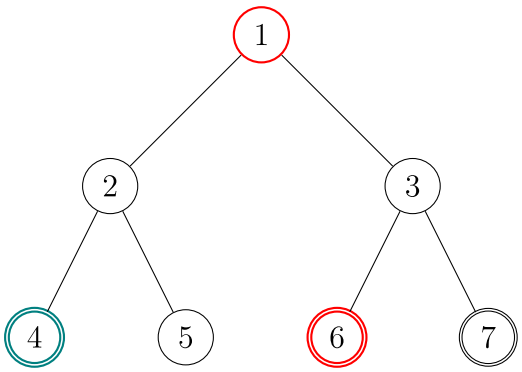
- 取序列中第一个作为当前节点，也就是 4. 再取栈顶元素，为 1. 求 1 和 4 的 LCA：$LCA(1, 4) = 1$.
- 发现 $LCA(1, 4) =$ 栈顶元素，说明它们在虚树的一条链上，所以直接把当前节点 4 入栈，当前栈为 $4, 1$.



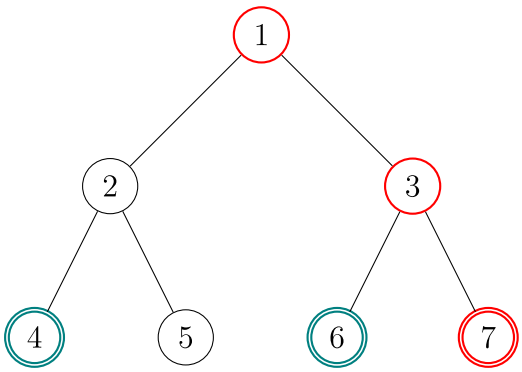
- 取序列第二个作为当前节点，为 6. 再取栈顶元素，为 4. 求 6 和 4 的 LCA：$LCA(6, 4) = 1$.
- 发现 $LCA(6, 4) \neq$ 栈顶元素，进入判断阶段.
- 判断阶段：发现栈顶节点 4 的 DFS 序是大于 $LCA(6, 4)$ 的，但是次大节点（栈顶节点下面的那个节点）1 的 DFS 序是等于 LCA 的（其实 DFS 序相等说明节点也相等），说明 LCA 已经入栈了，所以直接连接 $1 \to 4$ 的边，也就是 LCA 到栈顶元素的边. 并把 4 从栈中弹出.



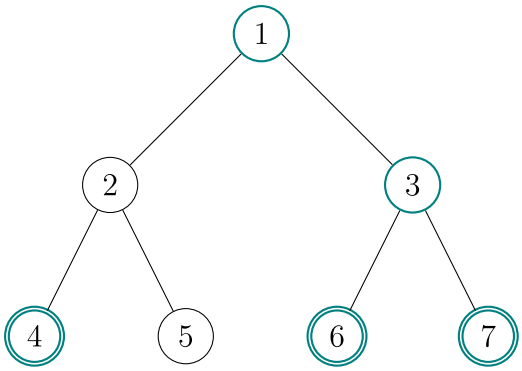
- 结束了判断阶段，将 6 入栈，当前栈为 $6, 1$.



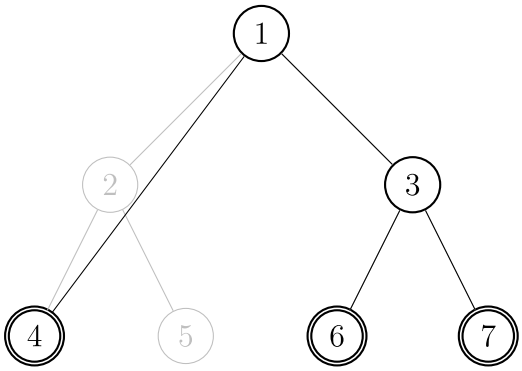
- 取序列第三个作为当前节点，为 7．再取栈顶元素，为 6．求 7 和 6 的 LCA：$LCA(7,6)=3$.
- 发现 $LCA(7,6)\neq$ 栈顶元素，进入判断阶段.
- 判断阶段：发现栈顶节点 6 的 DFS 序是大于 $LCA(7,6)$ 的，但是次大节点（栈顶节点下面的那个节点）1 的 DFS 序是小于 LCA 的，说明 LCA 还没有入过栈，所以直接连接 $3\to 6$ 的边，也就是 LCA 到栈顶元素的边．把 6 从栈中弹出，并且把 $LCA(6,7)$ 入栈.
- 结束了判断阶段，将 7 入栈，当前栈为 $1, 3, 7$.



- 发现序列里的 3 个节点已经全部加入过栈了，退出循环.
- 此时栈中还有 3 个节点：$1, 3, 7$，很明显它们是一条链上的，所以直接链接：$1\to 3$ 和 $3\to 7$ 的边.
- 虚树就建完啦!



我们接下来将那些没入过栈的点（非青色的点）删掉，对应的虚树长这个样子：



其中有很多细节，比如用邻接表存图的方式存虚树的话，需要清空邻接表．但是直接清空整个邻接表是很慢的，所以我们在 **有一个从未入栈的元素入栈的时候清空该元素对应的邻接表** 即可．

时间复杂度同样为 $O(m \log n)$（因为有排序），其中 m 为关键点数，n 为总点数．

实现

建立虚树的 C++ 代码大概长这样：

代码实现

```cpp
bool cmp(const int x, const int y) { return id[x] < id[y]; }

void build() {
  sort(h + 1, h + k + 1, cmp);
  sta[top = 1] = 1, g.sz = 0, g.head[1] = -1;
  // 1 号节点入栈，清空 1 号节点对应的邻接表，设置邻接表边数为 0
  for (int i = 1, l; i <= k; ++i)
    if (h[i] != 1) {
      // 如果 1 号节点是关键节点就不要重复添加
      l = lca(h[i], sta[top]);
      // 计算当前节点与栈顶节点的 LCA
      if (l != sta[top]) {
        // 如果 LCA 和栈顶元素不同，则说明当前节点不在当前栈所存的链上
        while (id[l] < id[sta[top - 1]])
          // 当次大节点的 Dfs 序大于 LCA 的 Dfs 序
          g.push(sta[top - 1], sta[top]), top--;
        // 把与当前节点所在的链不重合的链连接掉并且弹出
        if (id[l] > id[sta[top - 1]])
          // 如果 LCA 不等于次大节点（这里的大于其实和不等于没有区别）
```

```
20            g.head[l] = -1, g.push(l, sta[top]), sta[top] = l;
21          // 说明 LCA 是第一次入栈，清空其邻接表，连边后弹出栈顶元素，并将 LCA
22          // 入栈
23          else
24            g.push(l, sta[top--]);
25          // 说明 LCA 就是次大节点，直接弹出栈顶元素
26        }
27        g.head[h[i]] = -1, sta[++top] = h[i];
28        // 当前节点必然是第一次入栈，清空邻接表并入栈
29      }
30    for (int i = 1; i < top; ++i)
31      g.push(sta[i], sta[i + 1]);  // 剩余的最后一条链连接一下
32    return;
33  }
```

于是我们就学会了虚树的建立了！

对于消耗战这题，直接在虚树上跑最开始讲的那个 DP 就行了，我们等于利用了虚树排除了那些没用的非关键节点！仍然考虑 i 的所有儿子 v：

- 若 v 不是关键点：$Dp(i) = Dp(i) + \min\{Dp(v), w(i,v)\}$
- 若 v 是关键点：$Dp(i) = Dp(i) + w(i,v)$

于是这题很简单就过了.

推荐习题

- 「SDOI2011」消耗战[2]
- 「HEOI2014」大工程[3]
- CF613D Kingdom and its Cities[4]
- 「HNOI2014」世界树[5]

外部参考资料

[1] 「SDOI2011」消耗战 

[2] 「SDOI2011」消耗战 

[3] 「HEOI2014」大工程 

[4] CF613D Kingdom and its Cities 

[5] 「HNOI2014」世界树 

11.6.9 树分治

点分治

点分治适合处理大规模的树上路径信息问题.

例题 1 Luogu P3806【模板】点分治 1[1]

给定一棵有 n 个点的带边权树，m 次询问，每次询问给出 k，询问树上距离为 k 的点对是否存在.

$n \leq 10000, m \leq 100, k \leq 10000000$

我们先随意选择一个节点作为根节点 rt，所有完全位于其子树中的路径可以分为两种，一种是经过当前根节点的路径，一种是不经过当前根节点的路径. 对于经过当前根节点的路径，又可以分为两种，一种是以根节点为一个端点的路径，另一种是两个端点都不为根节点的路径. 而后者又可以由两条属于前者链合并得到. 所以，对于枚举的根节点 rt，我们先计算在其子树中且经过该节点的路径对答案的贡献，再递归其子树对不经过该节点的路径进行求解.

在本题中，对于经过根节点 rt 的路径，我们先枚举其所有子节点 ch，以 ch 为根计算 ch 子树中所有节点到 rt 的距离．记节点 i 到当前根节点 rt 的距离为 $dist_i$，tf_d 表示之前处理过的子树中是否存在一个节点 v 使得 $dist_v = d$．若一个询问的 k 满足 $tf_{k-dist_i} = true$，则存在一条长度为 k 的路径．在计算完 ch 子树中所连的边能否成为答案后，我们将这些新的距离加入 tf 数组中．

注意在清空 tf 数组的时候不能直接用 `memset`，而应将之前占用过的 tf 位置加入一个队列中，进行清空，这样才能保证时间复杂度．

点分治过程中，每一层的所有递归过程合计对每个点处理一次，假设共递归 h 层，则总时间复杂度为 $O(hn)$．

若我们每次选择子树的 重心→11.6.4@p2639 作为根节点，可以保证递归层数最少，时间复杂度为 $O(n \log n)$．因此，点分治在国外竞赛圈也常称为树的 **重心分解**（centroid decomposition）．

请注意在重新选择根节点之后一定要重新计算子树的大小，否则一点看似微小的改动就可能会使时间复杂度错误或正确性难以保证．

参考代码

```cpp
/* size 处理子树 d[], 连通块大小 cnt
   dp 最大子树 f[], 树的重心 rot
   get 计算出点到重心的距离 t[], top
   calc 点分治 bu[] 长度桶
   hd to nx wg 链式前向星存图
   ak[] as[] 离线处理询问
   ok[] 点分治中已成为重心的点
 */
#include <iostream>
const int N = 1e4 + 4, M = 105, Q = 1e7 + 7;
int n, m, hd[N], to[N * 2], nx[N * 2], wg[N * 2];
int ak[M], d[N], f[N], t[N], top, cnt, rot;
bool as[M], ok[N], bu[Q];

int size(int u, int pa) {
  cnt++, d[u] = 1;
  for (int p = hd[u]; ~p; p = nx[p])
    if (to[p] != pa && !ok[to[p]]) d[u] += size(to[p], u);
  return d[u];
}

void dp(int u, int pa) {
  f[u] = cnt - d[u];
  for (int p = hd[u]; ~p; p = nx[p])
    if (to[p] != pa && !ok[to[p]]) {
      f[u] = std::max(f[u], d[to[p]]);
      dp(to[p], u);
    }
  if (f[u] < f[rot]) rot = u;
}

void get(int u, int pa, int dis) {
  t[top++] = dis;
  for (int p = hd[u]; ~p; p = nx[p])
    if (to[p] != pa && !ok[to[p]]) get(to[p], u, dis + wg[p]);
```

```
}

void calc(int u) {
  cnt = 0, size(u, u);
  rot = u, dp(u, u);
  bu[0] = true, t[0] = 0, top = 1;
  for (int p = hd[rot], i; ~p; p = nx[p])
    if (!ok[to[p]]) {
      i = top, get(to[p], rot, wg[p]);
      for (int q = 0; q < m; q++)
        for (int j = i; j < top && !as[q]; j++)
          if (ak[q] >= t[j]) as[q] = bu[ak[q] - t[j]];
      --i;
      while (++i < top)
        if (t[i] < Q) bu[t[i]] = true;
    }
  while (top--)
    if (t[top] < Q) bu[t[top]] = false;
  ok[rot] = true;
  for (int p = hd[rot]; ~p; p = nx[p])
    if (!ok[to[p]]) calc(to[p]);
}

int main() {
  std::cin >> n >> m;
  for (int i = 1; i <= n; i++) hd[i] = -1;
  for (int i = 0, u, v; i + 2 < n * 2;) {
    std::cin >> u >> v >> wg[i];
    wg[i + 1] = wg[i];
    to[i] = v, nx[i] = hd[u], hd[u] = i++;
    to[i] = u, nx[i] = hd[v], hd[v] = i++;
  }
  for (int i = 0; i < m; i++) std::cin >> ak[i];
  calc(1);
  for (int i = 0; i < m; i++) std::cout << (as[i] ? "AYE\n" : "NAY\n");
}

```

例题 2 Luogu P4178 Tree[2]

给定一棵有 n 个点的带权树，给出 k，询问树上距离小于等于 k 的点对数量.

$n \leq 40000, k \leq 20000, w_i \leq 1000$

由于这里查询的是树上距离为 $[0, k]$ 的点对数量，所以我们用线段树来支持维护和查询.

参考代码

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
```

```
using namespace std;
constexpr long long MAXN = 2000010;
constexpr long long inf = 2e9;
long long n, a, b, c, q, rt, siz[MAXN], maxx[MAXN], dist[MAXN];
long long cur, h[MAXN], nxt[MAXN], p[MAXN], w[MAXN], ret;
bool vis[MAXN];

void add_edge(long long x, long long y, long long z) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
  w[cur] = z;
}

long long sum;

void calcsiz(long long x, long long fa) {
  siz[x] = 1;
  maxx[x] = 0;
  for (long long j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      calcsiz(p[j], x);
      maxx[x] = max(maxx[x], siz[p[j]]);
      siz[x] += siz[p[j]];
    }
  maxx[x] = max(maxx[x], sum - siz[x]);
  if (maxx[x] < maxx[rt]) rt = x;
}

long long dd[MAXN], cnt;

void calcdist(long long x, long long fa) {
  dd[++cnt] = dist[x];
  for (long long j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]])
      dist[p[j]] = dist[x] + w[j], calcdist(p[j], x);
}

queue<long long> tag;

struct segtree {
  long long cnt, rt, lc[MAXN], rc[MAXN], sum[MAXN];

  void clear() {
    while (!tag.empty()) update(rt, 1, 20000000, tag.front(), -1),
tag.pop();
    cnt = 0;
  }

  void print(long long o, long long l, long long r) {
    if (!o || !sum[o]) return;
```

```
    if (l == r) {
      cout << l << ' ' << sum[o] << '\n';
      return;
    }
    long long mid = (l + r) >> 1;
    print(lc[o], l, mid);
    print(rc[o], mid + 1, r);
  }

  void update(long long& o, long long l, long long r, long long x,
              long long v) {
    if (!o) o = ++cnt;
    if (l == r) {
      sum[o] += v;
      if (!sum[o]) o = 0;
      return;
    }
    long long mid = (l + r) >> 1;
    if (x <= mid)
      update(lc[o], l, mid, x, v);
    else
      update(rc[o], mid + 1, r, x, v);
    sum[o] = sum[lc[o]] + sum[rc[o]];
    if (!sum[o]) o = 0;
  }

  long long query(long long o, long long l, long long r, long long ql,
                  long long qr) {
    if (!o) return 0;
    if (r < ql || l > qr) return 0;
    if (ql <= l && r <= qr) return sum[o];
    long long mid = (l + r) >> 1;
    return query(lc[o], l, mid, ql, qr) + query(rc[o], mid + 1, r, ql, qr);
  }
} st;

void dfz(long long x, long long fa) {
  // tf[0]=true;tag.push(0);
  st.update(st.rt, 1, 20000000, 1, 1);
  tag.push(1);
  vis[x] = true;
  for (long long j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      dist[p[j]] = w[j];
      calcdist(p[j], x);
      for (long long k = 1; k <= cnt; k++)
        if (q - dd[k] >= 0)
          ret += st.query(st.rt, 1, 20000000, max(0ll, 1 - dd[k]) + 1,
                          max(0ll, q - dd[k]) + 1);
      for (long long k = 1; k <= cnt; k++)
        st.update(st.rt, 1, 20000000, dd[k] + 1, 1), tag.push(dd[k] + 1);
      cnt = 0;
```

```
    }
  st.clear();
  for (long long j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      sum = siz[p[j]];
      rt = 0;
      maxx[rt] = inf;
      calcsiz(p[j], x);
      calcsiz(rt, -1);
      dfz(rt, x);
    }
}

signed main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (long long i = 1; i < n; i++)
    cin >> a >> b >> c, add_edge(a, b, c), add_edge(b, a, c);
  cin >> q;
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  dfz(rt, -1);
  cout << ret << '\n';
  return 0;
}

```

例题 3 Luogu P2664 树上游戏[3]

一棵每个节点都给定颜色的树，定义 $s(i,j)$ 为 i 到 j 的颜色数量，$sum_i=\sum_{j=1}^{n}s(i,j)$. 对所有的 $1\le i\le n$，求 sum_i. $(1\le n,c_i\le 10^5)$

这道题很考验对点分治思想的理解和应用，适合作为点分治的难度较高的例题和练习题.

首先，我们需要想明白一个转化. 题目定义 sum_i 是 i 到所有节点路径上的颜色数量之和，可是如果用这个方法，在点分治中是不好统计答案的，因为这样很难合并从当前根出发的两棵子树的信息. 所以我们想到将 sum_i 的意义转化. 对于每个颜色 j, 其中一个端点为 i 且含有颜色 j 的路径数量记为 cnt_j，sum_i 其实就是 $\sum cnt_j$. 这一步转化其实就是换了个观察对象，考虑的是每个颜色对 sum_i 的贡献. 而 cnt_j 其实很好处理出来，只需要每遇到一个新颜色，就 $cnt_{col_u}+=size_u$ 即可，其中 $size_u$ 为 u 的子树大小，意味着这个子树里的所有节点都在这个颜色上对 u 的答案有一个贡献.

考虑到点分治过程中，我们只需要分别考虑统计：

1. 子树中以当前根节点为端点的路径对根的贡献
2. lca 为当前根节点的路径对子树内每个点的贡献

1 部分比较好办，由于点分治中，递归层数不超过 $\log n$，每一层我们都可以遍历全部子树，这个时候就可以使用 sum_i 的定义式来在遍历子树的过程中顺便统计了.

而针对 2 部分，设当前根节点 u 的一个子节点为 d,d 的子树里任取一个点为 v，那么 v 的答案可以分为两部分：

1. (u,v) 路径上出现过的颜色，数量设为 num，u 除了 d 以外的其他所有子树的总大小设为 $siz1$，那么这些出现过的颜色对 v 的答案贡献为 $num \times siz1$.
2. (u,v) 路径上没有出现过的颜色 j，它们的贡献来自于 u 除了 d 以外的其他所有子树的 cnt_j，这部分答案为 $\sum_{j \notin (u,v)} cnt_j$.

以上是全部统计思路，实现细节详见参考代码.

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
#define rep(i, a, b) for (int i = (a); i <= (b); ++i)
constexpr int N = 200005;
int h[N], nxt[N * 2], to[N * 2], c[N], gr;

void tu(int x, int y) { to[++gr] = y, nxt[gr] = h[x], h[x] = gr; }

using ll = long long;
int n, nn, siz[N], mn, rt;
bool vis[N];

void get_root(int u, int f) {
  siz[u] = 1;
  int mx = 0;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    get_root(d, u);
    siz[u] += siz[d];
    mx = max(mx, siz[d]);
  }
  mx = max(mx, nn - siz[u]);
  if (mx < mn) mn = mx, rt = u;
}

ll ans[N], sum;
int cnt[N], v[N];
// sum实时统计的是cnt[i]的和
int nowrt;

void get_dis(int u, int f, int now) {  // now为当前树链上的颜色数量(不含u)
  siz[u] = 1;
  if (!v[c[u]]) {
    sum -= cnt[c[u]];  // 减去在之前子树中已经出现过的颜色信息
    now++;
  }
  v[c[u]]++;
  ans[u] += sum + now * siz[nowrt];  // 统计过u点的路径对u的贡献
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (d == f || vis[d]) continue;
    get_dis(d, u, now);
    siz[u] += siz[d];
```

```
  }
  v[c[u]]--;
  if (!v[c[u]]) {
    sum += cnt[c[u]];  // 回溯
  }
}

void get_cnt(int u, int f) {
  if (!v[c[u]]) {
    cnt[c[u]] += siz[u];
    sum += siz[u];  // 将刚遍历过的子树的信息整合到cnt[i]和sum上去
  }
  v[c[u]]++;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    get_cnt(d, u);
  }
  v[c[u]]--;
}

void clear(int u, int f, int now) {
  if (!v[c[u]]) now++;
  v[c[u]]++;
  ans[u] -= now;
  ans[nowrt] += now;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    clear(d, u, now);
  }
  v[c[u]]--;
  cnt[c[u]] = 0;
}

void clear2(int u, int f) {
  cnt[c[u]] = 0;
  for (int i = h[u]; i; i = nxt[i]) {
    int d = to[i];
    if (vis[d] || d == f) continue;
    clear2(d, u);
  }
}

int son[N];

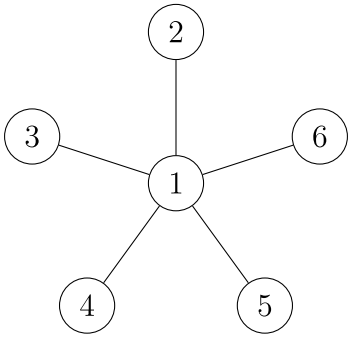
void divid(int u) {
  vis[u] = true;
  int tot = 0;
  nowrt = u;
  ans[u]++;
  for (int i = h[u]; i; i = nxt[i]) {
```

```
98      if (vis[to[i]]) continue;
99      son[++tot] = to[i];
100   }
101   siz[u] = sum = cnt[c[u]] = 1;
102   v[c[u]]++;
103   rep(i, 1, tot) {  // 统计每个子树和它之前的所有子树中节点组合产生的贡献
104     int d = son[i];
105     get_dis(d, u, 0);
106     get_cnt(d, u);
107     siz[u] += siz[d];
108     cnt[c[u]] += siz[d];
109     sum += siz[d];
110   }
111   clear2(u, 0);  // 清空数组，记得不可以用memset
112   siz[u] = sum = cnt[c[u]] = 1;
113   for (int i = tot; i >= 1;
114         --i) {  // 统计每个子树和它之后的所有子树中节点组合产生的贡献
115     int d = son[i];
116     get_dis(d, u, 0);
117     get_cnt(d, u);
118     siz[u] += siz[d];
119     cnt[c[u]] += siz[d];
120     sum += siz[d];
121   }
122   v[c[u]]--;
123   clear(u, 0, 0);                        // 清空的同时统计答案
124   for (int i = h[u]; i; i = nxt[i]) {  // 继续向下进行点分治
125     int d = to[i];
126     if (vis[d]) continue;
127     nn = siz[d], mn = n + 1, rt = 0;
128     get_root(d, u);
129     divid(rt);
130   }
131 }
132
133 int main() {
134   cin.tie(nullptr)->sync_with_stdio(false);
135   cin >> n;
136   int u, v;
137   rep(i, 1, n) cin >> c[i];
138   rep(i, 2, n) cin >> u >> v, tu(u, v), tu(v, u);
139   rt = 0, nn = n, mn = n + 1;
140   get_root(1, 0);
141   divid(rt);
142   rep(i, 1, n) cout << ans[i] << '\n';
143   return 0;
144 }
```

边分治

与上面的点分治类似，我们选取一条边，把树尽量均匀地分成两部分（使边连接的两个子树的 $size$ 尽量接近）．然后递归处理左右子树，统计信息．

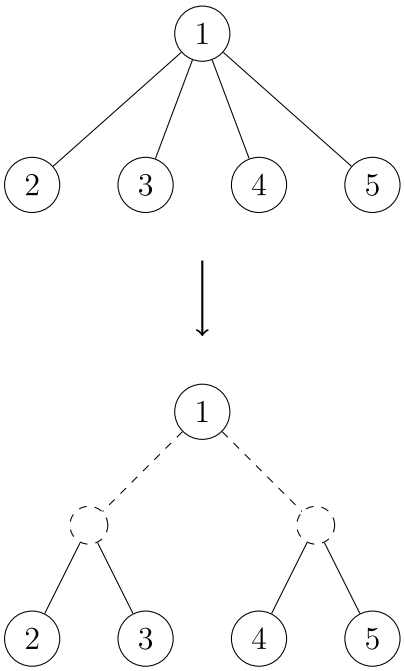
但是这是不行的，考虑一个菊花图：



我们发现当一个点下有多个 $size$ 接近的儿子时，应用边分治的时间复杂度是无法接受的.

如果这个图是个二叉树, 就可以避免上面菊花图中应用边分治的弊端. 因此我们考虑把一个多叉树转化成二叉树.

显然，我们只需像线段树那样建树就可以了．就像这样



新建出来的点根据题目要求给予恰当的信息即可．例如：统计路径长度时，将原边边权赋为 1, 将新建的边边权赋为 0 即可.

分析复杂度，发现最多会增加 $O(n)$ 个点，则总复杂度为 $O(n \log n)$

几乎所有点分治的题边分都能做（常数上有差距，但是不卡），所以就不放例题了.

点分树

点分树是通过更改原树形态使树的层数变为稳定 $\log n$ 的一种重构树.

常用于解决与树原形态无关的带修改问题.

算法分析

我们通过点分治每次找重心的方式来对原树进行重构.

将每次找到的重心与上一层的重心缔结父子关系，这样就可以形成一棵 $\log n$ 层的树.

由于树是 $\log n$ 层的，很多原来并不对劲的暴力在点分树上均有正确的复杂度.

代码实现

有一个小技巧：每次用递归上一层的总大小 tot 减去上一层的点的重儿子大小，得到的就是这一层的总大小．这样求重心就只需一次 DFS 了．

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

using IT = vector<int>::iterator;

struct Edge {
  int to, nxt, val;

  Edge() {}

  Edge(int to, int nxt, int val) : to(to), nxt(nxt), val(val) {}
} e[300010];

int head[150010], cnt;

void addedge(int u, int v, int val) {
  e[++cnt] = Edge(v, head[u], val);
  head[u] = cnt;
}

int siz[150010], son[150010];
bool vis[150010];

int tot, lasttot;
int maxp, root;

void getG(int now, int fa) {
  siz[now] = 1;
  son[now] = 0;
  for (int i = head[now]; i; i = e[i].nxt) {
    int vs = e[i].to;
    if (vs == fa || vis[vs]) continue;
    getG(vs, now);
    siz[now] += siz[vs];
    son[now] = max(son[now], siz[vs]);
  }
  son[now] = max(son[now], tot - siz[now]);
  if (son[now] < maxp) {
    maxp = son[now];
    root = now;
  }
}

struct Node {
  int fa;
```

```cpp
  vector<int> anc;
  vector<int> child;
} nd[150010];

int build(int now, int ntot) {
  tot = ntot;
  maxp = 0x7f7f7f7f;
  getG(now, 0);
  int g = root;
  vis[g] = true;
  for (int i = head[g]; i; i = e[i].nxt) {
    int vs = e[i].to;
    if (vis[vs]) continue;
    int tmp = build(vs, ntot - son[vs]);
    nd[tmp].fa = now;
    nd[now].child.push_back(tmp);
  }
  return g;
}

int virtroot;

int main() {
  int n;
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v, val;
    cin >> u >> v >> val;
    addedge(u, v, val);
    addedge(v, u, val);
  }
  virtroot = build(1, n);
}
```

外部参考资料

[1] Luogu P3806【模板】点分治 1  [2] Luogu P4178 Tree  [3] Luogu P2664 树上游戏 

11.6.10 动态树分治

动态点分治

动态点分治用来解决 **带点权/边权修改** 的树上路径信息统计问题.

点分树

回顾点分治的计算过程.

对于一个结点 x 来说，其子树中的简单路径包括两种：经过结点 x 的，由一条或两条从 x 出发的路径组成的；和不经过结点 x 的，即已经包含在其所有儿子结点子树中的路径.

对于一个子树中简单路径的计算，我们选择一个分治中心 rt，计算经过该节点的子树中路径的信息，然后对于其每个儿子结点，将删去 rt 后该点所在连通块作为一个子树，递归计算. 选择的分治中心点可以构成一个树形结构，称为 **点分树**. 我们发现，计算点分树中同一层的结点所代表的连通块（即以该结点

为分治中心的连通块）的大小总和是 $O(n)$ 的．这意味着，点分治的时间复杂度是与点分树的深度相关的，若点分树的深度为 h，则点分治的复杂度为 $O(nh)$．

可以证明，当我们每次选择连通块的重心作为分治中心的时候，点分树的深度最小，为 $O(\log n)$ 的．这样，我们就可以在 $O(n \log n)$ 的时间复杂度内统计树上 $O(n^2)$ 条路径的信息了．

由于树的形态在动态点分治的过程中不会改变，所以点分树的形态在动态点分治的过程中也不会改变．

下面给出求点分树的参考代码：

```
void calcsiz(int x, int f) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
  if (maxx[x] < maxx[rt])
    rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

void pre(int x) {
  vis[x] = true;  // 表示在之后的过程中不考虑 x 这个点
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
      sum = siz[p[j]];
      rt = 0;
      maxx[rt] = inf;
      calcsiz(p[j], -1);
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      fa[rt] = x;
      pre(rt);  // 记录点分树上的父亲
    }
}

int main() {
  sum = n;
  rt = 0;
  maxx[rt] = inf;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
}
```

实现修改

在查询和修改的时候，我们在点分树上暴力跳父亲修改．由于点分树的深度最多是 $O(\log n)$ 的，所以这样做复杂度能得到保证．

在动态点分治的过程中，需要一个结点到其点分树上的祖先的距离等其他信息，由于一个点最多有 $O(\log n)$ 个祖先，我们可以在计算点分树时额外计算深度 $dep[x]$ 或使用 LCA，预处理出这些距离或实现实时查询．**注意**：一个结点到其点分树上的祖先的距离不一定递增，不能累加！

在动态点分治的过程中，一个结点在其点分树上的祖先结点的信息中可能会被重复计算，这是我们需要消去重复部分的影响．一般的方法是对于一个连通块用两种方式记录：一个是其到分治中心的距离信息，另一个是其到点分树上分治中心父亲的距离信息．这一部分内容将在例题中得到展现．

例题「ZJOI2007」捉迷藏[1]

给定一棵有 n 个结点的树，初始时所有结点都是黑色的．你需要实现以下两种操作：

1. 反转一个结点的颜色（白变黑，黑变白）；
2. 询问树上两个最远的黑点的距离．

$n \leq 10^5, m \leq 5 \times 10^5$

求出点分树，对于每个结点 x 维护两个 **可删堆**．$dist[x]$ 存储结点 x 代表的连通块中的所有黑点到 x 的距离信息，$ch[x]$ 表示结点 x 在点分树上的所有儿子和它自己中的黑点到 x 的距离信息，由于本题贪心的求答案方法，且两个来自于同一子树的路径不能成为一条完成的路径，我们只在这个堆中插入其自己的值和其每个子树中的最大值．我们发现，$ch[x]$ 中最大的两个值（如果没有两个就是所有值）的和就是分治时分支中心为 x 时经过结点 x 的最长黑端点路径．我们可以用可删堆 ans 存储所有结点的答案，这个堆中的最大值就是我们所求的答案．

我们可以根据上面的定义维护 $dist[x], ch[x], ans$ 这些可删堆．当 $dist[x]$ 中的值发生变化时，我们也可以在 $O(\log n)$ 的时间复杂度内维护 $ch[x], ans$．

现在我们来看一下，当我们反转一个点的颜色时，$dist[x]$ 值会发生怎样的改变．当结点原来是黑色时，我们要进行的是删除操作；当结点原来是白色时，我们要进行的是插入操作．

假如我们要反转结点 x 的颜色．对于其所有祖先 u，我们在 $dist[u]$ 中插入或删除 $dist(x, u)$，并同时维护 $ch[x], ans$ 的值．特别的，我们要在 $ch[x]$ 中插入或删除值 0．

参考代码：

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
constexpr int MAXN = 100010;
constexpr int inf = 2e9;
int n, a, b, m, x, col[MAXN];
// 0 off 1 on
char op;
int cur, h[MAXN * 2], nxt[MAXN * 2], p[MAXN * 2];

void add_edge(int x, int y) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

bool vis[MAXN];
int rt, sum, siz[MAXN], maxx[MAXN], fa[MAXN], dep[MAXN];

void calcsiz(int x, int f) {
  siz[x] = 1;
  maxx[x] = 0;
```

```
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
  if (maxx[x] < maxx[rt])
    rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

struct heap {
  priority_queue<int> A, B;  // heap=A-B

  void insert(int x) { A.push(x); }

  void erase(int x) { B.push(x); }

  int top() {
    while (!B.empty() && A.top() == B.top()) A.pop(), B.pop();
    return A.top();
  }

  void pop() {
    while (!B.empty() && A.top() == B.top()) A.pop(), B.pop();
    A.pop();
  }

  int top2() {
    int t = top(), ret;
    pop();
    ret = top();
    A.push(t);
    return ret;
  }

  int size() { return A.size() - B.size(); }
} dist[MAXN], ch[MAXN], ans;

void dfs(int x, int f, int d, heap& y) {
  y.insert(d);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != f && !vis[p[j]]) dfs(p[j], x, d + 1, y);
}

void pre(int x) {
  vis[x] = true;  // 不考虑 x
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
      rt = 0;
      maxx[rt] = inf;
      sum = siz[p[j]];
      calcsiz(p[j], -1);
```

```
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      fa[rt] = x;
      dfs(p[j], -1, 1, dist[rt]);
      ch[x].insert(dist[rt].top());
      dep[rt] = dep[x] + 1;
      pre(rt);  // 记录点分树上的父亲
    }
  ch[x].insert(0);
  if (ch[x].size() >= 2)
    ans.insert(ch[x].top() + ch[x].top2());
  else if (ch[x].size())
    ans.insert(ch[x].top());
}

struct LCA {
  int dep[MAXN], lg[MAXN], fa[MAXN][20];

  void dfs(int x, int f) {
    for (int j = h[x]; j; j = nxt[j])
      if (p[j] != f) dep[p[j]] = dep[x] + 1, fa[p[j]][0] = x, dfs(p[j], x);
  }

  void init() {
    dfs(1, -1);
    for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
    for (int j = 1; j <= lg[n]; j++)
      for (int i = 1; i <= n; i++) fa[i][j] = fa[fa[i][j - 1]][j - 1];
  }

  int query(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int k = dep[y] - dep[x];
    for (int i = 0; k; k = k / 2, i++)
      if (k & 1) y = fa[y][i];
    if (x == y) return x;
    k = dep[x];
    for (int i = lg[k]; i >= 0; i--)
      if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
    return fa[x][0];
  }

  int dist(int x, int y) { return dep[x] + dep[y] - 2 * dep[query(x, y)]; }
} lca;

int d[MAXN][20];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i < n; i++) cin >> a >> b, add_edge(a, b), add_edge(b, a);
  lca.init();
  rt = 0;
  maxx[rt] = inf;
  sum = n;
```

```
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
  for (int i = 1; i <= n; i++)
    for (int j = i; j; j = fa[j]) d[i][dep[i] - dep[j]] = lca.dist(i, j);
  cin >> m;
  int cnt = n;
  while (m--) {
    cin >> op;
    if (op == 'G') {
      if (cnt)
        cout << ans.top() << '\n';
      else
        cout << "-1\n";
    } else {
      cin >> x;
      if (!col[x]) {
        if (ch[x].size() >= 2) ans.erase(ch[x].top() + ch[x].top2());
        ch[x].erase(0);
        if (ch[x].size() >= 2) ans.insert(ch[x].top() + ch[x].top2());
        for (int i = x; fa[i]; i = fa[i]) {
          if (ch[fa[i]].size() >= 2)
            ans.erase(ch[fa[i]].top() + ch[fa[i]].top2());
          ch[fa[i]].erase(dist[i].top());
          dist[i].erase(d[x][dep[x] - dep[fa[i]]]);
          if (dist[i].size()) ch[fa[i]].insert(dist[i].top());
          if (ch[fa[i]].size() >= 2)
            ans.insert(ch[fa[i]].top() + ch[fa[i]].top2());
        }
        cnt--;
      } else {
        if (ch[x].size() >= 2) ans.erase(ch[x].top() + ch[x].top2());
        ch[x].insert(0);
        if (ch[x].size() >= 2) ans.insert(ch[x].top() + ch[x].top2());
        for (int i = x; fa[i]; i = fa[i]) {
          if (ch[fa[i]].size() >= 2)
            ans.erase(ch[fa[i]].top() + ch[fa[i]].top2());
          if (dist[i].size()) ch[fa[i]].erase(dist[i].top());
          dist[i].insert(d[x][dep[x] - dep[fa[i]]]);
          ch[fa[i]].insert(dist[i].top());
          if (ch[fa[i]].size() >= 2)
            ans.insert(ch[fa[i]].top() + ch[fa[i]].top2());
        }
        cnt++;
      }
      col[x] ^= 1;
    }
  }
  return 0;
}

```

例题 Luogu P6329【模板】点分树 | 震波[2]

给定一棵有 n 个结点的树，树上每个结点都有一个权值 $v[x]$. 实现以下两种操作：

1. 询问与结点 x 距离不超过 y 的结点权值和；
2. 修改结点 x 的点权为 y，即 $v[x]=y$.

我们用动态开点权值线段树记录距离信息.

类似于上题的思路，对于每个结点，我们维护线段树 $dist[x]$，表示分治块 x 中的所有结点到结点 x 的距离信息，下标为距离，权值加上点权. 线段树 $ch[x]$ 表示分治块 x 中所有结点到结点 x 在分治树上的父亲结点的距离信息.

在本题中，所有查询和修改都需要在点分树上对所有祖先进行修改.

以查询操作为例，如果我们要查询距离结点 x 不超过 y 的结点的权值和，我们要先将答案加上线段树 $dist[x]$ 中下标从 0 到 y 的权值和，然后我们遍历 x 的所有祖先 u，设其低一级祖先为 v，令 $d=dist(x,u)$，如果我们不进入包含 x 的子树，即以 v 为根的子树，那么我们要将答案加上线段树 $dist[u]$ 中下标从 0 到 $y-d$ 的权值和. 由于我们重复计算了以 v 为根的部分，我们要将答案减去线段树 $ch[v]$ 中下标从 0 到 $y-d$ 的权值和.

在进行修改操作时，我们要同时维护 $dist[x]$ 和 $ch[x]$.

参考代码：

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;
constexpr int MAXN = 100010;
constexpr int inf = 2e9;
constexpr int ddd = 6000010;

struct Segtree {
  int cnt, rt[MAXN], sum[ddd], lc[ddd], rc[ddd];

  void update(int& o, int l, int r, int x, int v) {
    if (!o) o = ++cnt;
    if (l == r) {
      sum[o] += v;
      return;
    }
    int mid = (l + r) >> 1;
    if (x <= mid)
      update(lc[o], l, mid, x, v);
    else
      update(rc[o], mid + 1, r, x, v);
    sum[o] = sum[lc[o]] + sum[rc[o]];
  }

  int query(int o, int l, int r, int ql, int qr) {
    if (!o || r < ql || l > qr) return 0;
    if (ql <= l && r <= qr) return sum[o];
    int mid = (l + r) >> 1;
    return query(lc[o], l, mid, ql, qr) + query(rc[o], mid + 1, r, ql, qr);
  }
```

```
} dist, ch;

int n, m, val[MAXN], u, v, op, x, y, lstans;
int cur, h[MAXN * 2], nxt[MAXN * 2], p[MAXN * 2];

void add_edge(int x, int y) {
  cur++;
  nxt[cur] = h[x];
  h[x] = cur;
  p[cur] = y;
}

struct LCA {
  int dep[MAXN], lg[MAXN], fa[MAXN][20];

  void dfs(int x, int f) {
    for (int j = h[x]; j; j = nxt[j])
      if (p[j] != f) dep[p[j]] = dep[x] + 1, fa[p[j]][0] = x, dfs(p[j], x);
  }

  void init() {
    dep[1] = 1;
    dfs(1, -1);
    for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
    for (int j = 1; j <= lg[n]; j++)
      for (int i = 1; i <= n; i++) fa[i][j] = fa[fa[i][j - 1]][j - 1];
  }

  int query(int x, int y) {
    if (dep[x] > dep[y]) swap(x, y);
    int k = dep[y] - dep[x];
    for (int i = 0; k; k = k / 2, i++)
      if (k & 1) y = fa[y][i];
    if (x == y) return x;
    k = dep[x];
    for (int i = lg[k]; i >= 0; i--)
      if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
    return fa[x][0];
  }

  int dist(int x, int y) { return dep[x] + dep[y] - 2 * dep[query(x, y)]; }
} lca;

int rt, sum, siz[MAXN], maxx[MAXN], fa[MAXN];
int d[MAXN][20], dep[MAXN];
bool vis[MAXN];

void calcsiz(int x, int fa) {
  siz[x] = 1;
  maxx[x] = 0;
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) {
      calcsiz(p[j], x);
      siz[x] += siz[p[j]];
```

```
      maxx[x] = max(maxx[x], siz[p[j]]);
    }
  maxx[x] =
      max(maxx[x], sum - siz[x]);  // maxx[x] 表示以 x 为根时的最大子树大小
  if (maxx[x] < maxx[rt])
    rt = x;  // 这里不能写 <= ，保证在第二次 calcsiz 时 rt 不改变
}

void dfs1(int x, int fa, int y, int d) {
  ch.update(ch.rt[y], 0, n, d, val[x]);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) dfs1(p[j], x, y, d + 1);
}

void dfs2(int x, int fa, int y, int d) {
  dist.update(dist.rt[y], 0, n, d, val[x]);
  for (int j = h[x]; j; j = nxt[j])
    if (p[j] != fa && !vis[p[j]]) dfs2(p[j], x, y, d + 1);
}

void pre(int x) {
  vis[x] = true;  // 表示在之后的过程中不考虑 x 这个点
  dfs2(x, -1, x, 0);
  for (int j = h[x]; j; j = nxt[j])
    if (!vis[p[j]]) {
      rt = 0;
      maxx[rt] = inf;
      sum = siz[p[j]];
      calcsiz(p[j], -1);
      calcsiz(rt, -1);  // 计算两次，第二次求出以 rt 为根时的各子树大小
      dfs1(p[j], -1, rt, 1);
      fa[rt] = x;
      dep[rt] = dep[x] + 1;
      pre(rt);  // 记录点分树上的父亲
    }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> val[i];
  for (int i = 1; i < n; i++) cin >> u >> v, add_edge(u, v), add_edge(v, u);
  lca.init();
  rt = 0;
  maxx[rt] = inf;
  sum = n;
  calcsiz(1, -1);
  calcsiz(rt, -1);
  pre(rt);
  for (int i = 1; i <= n; i++)
    for (int j = i; j; j = fa[j]) d[i][dep[i] - dep[j]] = lca.dist(i, j);
  while (m--) {
    cin >> op >> x >> y;
    x ^= lstans;
```

```
140     y ^= lstans;
141     if (op == 0) {
142       lstans = dist.query(dist.rt[x], 0, n, 0, y);
143       int nww = 0;
144       for (int i = x; fa[i]; i = fa[i]) {
145         nww = d[x][dep[x] - dep[fa[i]]];  // lca.dist(x,fa[i]);
146         lstans += dist.query(dist.rt[fa[i]], 0, n, 0, y - nww);
147         lstans -= ch.query(ch.rt[i], 0, n, 0, y - nww);
148       }
149       cout << lstans << '\n';
150     }
151     if (op == 1) {
152       int nww = 0;
153       dist.update(dist.rt[x], 0, n, 0, y - val[x]);
154       for (int i = x; fa[i]; i = fa[i]) {
155         nww = d[x][dep[x] - dep[fa[i]]];  // lca.dist(x,fa[i]);
156         dist.update(dist.rt[fa[i]], 0, n, nww, y - val[x]);
157         ch.update(ch.rt[i], 0, n, nww, y - val[x]);
158       }
159       val[x] = y;
160     }
161   }
162   return 0;
163 }
```

外部参考资料

[1] 「ZJOI2007」捉迷藏



[2] Luogu P6329【模板】点分树 | 震波



11.6.11 AHU 算法

Authors: Backl1ght

AHU 算法用于判断两棵有根树是否同构.

判断树同构外还有一种常见的做法是 树哈希→11.6.12@p2717.

前置知识：树基础→11.6.1@p2621，树的重心→11.6.4@p2639

建议配合参考资料里给的例子观看.

树同构的定义

有根树同构

对于两棵有根树 $T_1(V_1, E_1, r_1)$ 和 $T_2(V_2, E_2, r_2)$，如果存在一个双射 $\varphi : V_1 \to V_2$，使得

$$\forall u, v \in V_1, (u, v) \in E_1 \iff (\varphi(u), \varphi(v)) \in E_2$$

且 $\varphi(r_1) = r_2$ 成立，那么称有根树 $T_1(V_1, E_1, r_1)$ 和 $T_2(V_2, E_2, r_2)$ 同构.

无根树同构

对于两棵无根树 $T_1(V_1, E_1)$ 和 $T_2(V_2, E_2)$，如果存在一个双射 $\varphi : V_1 \to V_2$，使得

$$\forall u, v \in V_1, (u, v) \in E_1 \iff (\varphi(u), \varphi(v)) \in E_2$$

成立，那么称无根树 $T_1(V_1, E_1)$ 和 $T_2(V_2, E_2)$ 同构.

简单的说就是，如果能够通过把树 T_1 的所有节点重新标号，使得树 T_1 和树 T_2 **完全相同**，那么称这两棵树同构.

问题的转化

无根树同构问题可以转化为有根树同构问题．具体方法如下：

对于无根树 $T_1(V_1, E_1)$ 和 $T_2(V_2, E_2)$，先分别找出它们的 **所有** 重心.

- 如果这两棵无根树重心数量不同，那么这两棵树不同构.
- 如果这两颗无根树重心数量都为 1，分别记为 c_1 和 c_2，那么如果有根树 $T_1(V_1, E_1, c_1)$ 和有根树 $T_2(V_2, E_2, c_2)$ 同构，那么无根树 $T_1(V_1, E_1)$ 和 $T_2(V_2, E_2)$ 同构，反之则不同构.
- 如果这两颗无根树重心数量都为 2，分别记为 $c_1, {c'}_1$ 和 $c_2, {c'}_2$，那么如果有根树 $T_1(V_1, E_1, c_1)$ 和有根树 $T_2(V_2, E_2, c_2)$ 同构 **或者** 有根树 $T_1(V_1, E_1, {c'}_1)$ 和 $T_2(V_2, E_2, c_2)$ 同构，那么无根树 $T_1(V_1, E_1)$ 和 $T_2(V_2, E_2)$ 同构，反之则不同构.

所以，只要解决了有根树同构问题，我们就可以把无根树同构问题根据上述方法转化成有根树同构的问题，进而解决无根树同构的问题.

假设有一个可以 $O(|V|)$ 解决有根树同构问题的算法，那么根据上述方法我们也可以在 $O(|V|)$ 的时间内解决无根树同构问题.

朴素的 AHU 算法

朴素的 AHU 算法是基于括号序的.

原理 1

我们知道一段合法的括号序和一棵有根树唯一对应，而且一棵树的括号序是由它的子树的括号序拼接而成的．如果我们通过改变子树括号序拼接的顺序，从而获得了一段新的括号序，那么新括号序对应的树和原括号序对应的树同构.

原理 2

树的同构关系是传递的．既如果 T_1 和 T_2 同构，T_2 和 T_3 同构，那么 T_1 和 T_3 同构.

推论

考虑求树括号序的递归算法，我们在回溯时拼接子树的括号序．如果在拼接的时候将字典序小的序列先拼接，并将最后的结果记为 $NAME$.

将以节点 r 为根的子树的 $NAME$ 作为节点 r 的 $NAME$，记为 $NAME(r)$，那么对于有根树 $T_1(V_1, E_1, r_1)$ 和 $T_2(V_2, E_2, r_2)$，如果 $NAME(r_1) = NAME(r_2)$，那么 T_1 和 T_2 同构.

命名算法

实现

```
1   Input. A rooted tree T
2   Output. The name of rooted tree T
3   ASSIGN-NAME(u)
4       if u is a leaf
5           NAME(u)=(0)
6       else
7           for all child v of u
8               ASSIGN-NAME(v)
9       sort the names of the children of u
10      concatenate the names of all children u to temp
11      NAME(u)=(temp)
```

AHU 算法

实现

```
1  Input. Two rooted trees T1(V1, E1, r1) and T2(V2, E2, r2)
2  Output. Whether these two trees are isomorphic
3  AHU(T1(V1, E1, r1), T2(V2, E2, r2))
4      ASSIGN-NAME(r1)
5      ASSIGN-NAME(r2)
6      if NAME(r1) = NAME(r2)
7          return true
8      else
10         return false
```

复杂度证明

对于一颗有 n 个节点的有根树，假设他是链状的，那么节点名字长度最长可以是 n，那么 ASSIGN-NAME 算法的复杂度是 $1+2+\cdots+n$ 的常数倍，即 $\Theta(n^2)$. 由此，朴素 AHU 算法的复杂度为 $O(n^2)$.

优化的 AHU 算法

朴素的 AHU 算法的缺点是树的 $NAME$ 的长度可能会过长，我们可以针对这一点做一些优化.

原理 1

对树进行层次划分，第 i 层的节点到根的最短距离为 i. 位于第 i 层的节点的 $NAME$ 可以 **只** 由位于第 $i+1$ 层的节点的 $NAME$ 拼接得到.

原理 2

在同一层内，节点的 $NAME$ 可以由其在层内的排名唯一标识.

注意，这里的排名是对两棵树而言的，假设节点 u 位于第 i 层，那么节点 u 的排名等于所有 T_1 和 T_2 第 i 层的节点中 $NAME$ 比 $NAME(u)$ 小的节点的个数.

推论

我们可以将节点原来的 $NAME$ 用其在层内的排名代替, 然后把原来拼接节点 $NAME$ 用向数组加入元素代替.

这样用整数和数组来代替字符串，既不会影响算法的正确性，又很大的降低了算法的复杂度.

复杂度证明

首先注意到第 i 层由拼接得到的 $NAME$ 的总长度为第 i 层点的度数之和，即第 $i+1$ 层的总点数，以下用 L_i 表示. 算法的下一步会将这些 $NAME$ 看成字符串（数组）并排序，然后将它们替换为其在层内的排名（即重新映射为一个数）. 以下引理表明了对总长为 L 的 m 个字符串排序的复杂度:

1. 我们可以使用基数排序在 $O(L+|\Sigma|)$ 的时间内完成排序，其中 $|\Sigma|$ 为字符集的大小.（有一些实现细节，参见参考资料）
2. 我们可以使用快速排序在 $O(L\log m)$ 的时间内完成排序. 证明的大致思路为快排递归树的高度为 $O(\log m)$，且暴力比较长度为 ℓ_1 和 ℓ_2 的两个字符串的复杂度为 $O(\min\{\ell_1,\ell_2\})$.

在 AHU 算法中，第 i 层字符串的字符集大小最多为第 $i+1$ 层的点数，即 L_i，所以基数排序的复杂度是线性的. 根据 $\sum_i L_i = O(n)$，并将每层的复杂度相加后可以看出，若使用字符串的基数排序，则算法的总复杂度为 $T(n)=O(n)$. 同理，如果使用快排排序字符串，那么 $T(n)=O(n\log n)$.

例题

SPOJ-TREEISO[1]

题意翻译：给你两颗无根树，判断两棵树是否同构.

参考代码

```cpp
// Tree Isomorphism, O(nlogn)
// replace quick sort with radix sort ==> O(n)
// Author: _Backl1ght
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
using ll = long long;
constexpr int N = 1e5 + 5;
constexpr int MAXN = N << 1;

int n;

struct Edge {
  int v, nxt;
} e[MAXN << 1];

int head[MAXN], sz[MAXN], f[MAXN], maxv[MAXN], tag[MAXN], tot, Max;
vector<int> center[2], L[MAXN], subtree_tags[MAXN];

void addedge(int u, int v) {  // 建图
  e[tot].v = v;
  e[tot].nxt = head[u];
  head[u] = tot++;
  e[tot].v = u;
  e[tot].nxt = head[v];
  head[v] = tot++;
}

void dfs_size(int u, int fa) {  // 找到 size 值
  sz[u] = 1;
  maxv[u] = 0;
  for (int i = head[u]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (v == fa) continue;
    dfs_size(v, u);
    sz[u] += sz[v];
    maxv[u] = max(maxv[u], sz[v]);
  }
}

void dfs_center(int rt, int u, int fa, int id) {
  maxv[u] = max(maxv[u], sz[rt] - sz[u]);
  if (Max > maxv[u]) {
    center[id].clear();
    Max = maxv[u];
  }
  if (Max == maxv[u]) center[id].push_back(u);  // 如果相等就 push_back
  for (int i = head[u]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (v == fa) continue;
```

```
52      dfs_center(rt, v, u, id);
53    }
54  }
55
56  int dfs_height(int u, int fa, int depth) {  // 递归查找 height
57    L[depth].push_back(u);
58    f[u] = fa;
59    int h = 0;
60    for (int i = head[u]; i; i = e[i].nxt) {
61      int v = e[i].v;
62      if (v == fa) continue;
63      h = max(h, dfs_height(v, u, depth + 1));
64    }
65    return h + 1;
66  }
67
68  void init(int n) {  // 一开始的处理
69    for (int i = 1; i <= 2 * n; i++) head[i] = 0;
70    tot = 1;
71    center[0].clear();
72    center[1].clear();
73
74    int u, v;
75    for (int i = 1; i <= n - 1; i++) {
76      cin >> u >> v;
77      addedge(u, v);
78    }
79    dfs_size(1, -1);
80    Max = n;
81    dfs_center(1, 1, -1, 0);
82
83    for (int i = 1; i <= n - 1; i++) {
84      cin >> u >> v;
85      addedge(u + n, v + n);
86    }
87    dfs_size(1 + n, -1);
88    Max = n;
89    dfs_center(1 + n, 1 + n, -1, 1);
90  }
91
92  bool cmp(int u, int v) { return subtree_tags[u] < subtree_tags[v]; }
93
94  bool rootedTreeIsomorphism(int rt1, int rt2) {
95    for (int i = 0; i <= 2 * n + 1; i++) L[i].clear(),
  subtree_tags[i].clear();
96    int h1 = dfs_height(rt1, -1, 0);
97    int h2 = dfs_height(rt2, -1, 0);
98    if (h1 != h2) return false;
99    int h = h1 - 1;
100   for (int j = 0; j < (int)L[h].size(); j++) tag[L[h][j]] = 0;
101   for (int i = h - 1; i >= 0; i--) {
102     for (int j = 0; j < (int)L[i + 1].size(); j++) {
```

```
103       int v = L[i + 1][j];
104       subtree_tags[f[v]].push_back(tag[v]);
105     }
106
107     sort(L[i].begin(), L[i].end(), cmp);
108
109     for (int j = 0, cnt = 0; j < (int)L[i].size(); j++) {
110       if (j && subtree_tags[L[i][j]] != subtree_tags[L[i][j - 1]]) ++cnt;
111       tag[L[i][j]] = cnt;
112     }
113   }
114   return subtree_tags[rt1] == subtree_tags[rt2];
115 }
116
117 bool treeIsomorphism() {
118   if (center[0].size() == center[1].size()) {
119     if (rootedTreeIsomorphism(center[0][0], center[1][0])) return true;
120     if (center[0].size() > 1)
121       return rootedTreeIsomorphism(center[0][0], center[1][1]);
122   }
123   return false;
124 }
125
126 int main() {
127   cin.tie(nullptr)->sync_with_stdio(false);
128   int T;
129   cin >> T;
130   while (T--) {
131     cin >> n;
132     init(n);
133     cout << (treeIsomorphism() ? "YES" : "NO") << '\n';
134   }
135   return 0;
136 }
```

参考资料

本文大部分内容译自 Paper[2] 和 Slide[3]. 参考资料里的证明会更加全面和严谨，本文做了一定的简化.

对 AHU 算法的复杂度分析，以及字符串的线性时间基数排序算法可以参见 The Design and Analysis of Computer Algorithms 的 3.2 节 Radix sorting，以及其中的 Example 3.2.

外部参考资料

[1] SPOJ-TREEISO  [2] Paper  [3] Slide 

11.6.12 树哈希

判断一些树是否同构的时，我们常常把这些树转成哈希值储存起来，以降低复杂度.

树哈希是很灵活的，可以设计出各种各样的哈希方式；但是如果随意设计，很有可能是错误的，可能被卡. 以下介绍一类容易实现且不易被卡的方法.

方法

这类方法需要一个多重集的哈希函数. 以某个结点为根的子树的哈希值, 就是以它的所有儿子为根的子树的哈希值构成的多重集的哈希值，即:

$$h_x = f(\{h_i \mid i \in son(x)\})$$

其中 h_x 表示以 x 为根的子树的哈希值，f 是多重集的哈希函数.

以代码中使用的哈希函数为例:

$$f(S) = \left(c + \sum_{x \in S} g(x)\right) \bmod m$$

其中 c 为常数，一般使用 1 即可. m 为模数，一般使用 2^{32} 或 2^{64} 进行自然溢出，也可使用大素数. g 为整数到整数的映射，代码中使用 xor shift，也可以选用其他的函数，但是不建议使用多项式. 为了预防出题人对着 xor hash 卡，还可以在映射前后异或一个随机常数.

这种哈希十分好写. 如果需要换根，第二次 DP 时只需把子树哈希减掉即可.

例题

UOJ #763. 树哈希[1]

这是一道模板题. 不用多说，以 1 为根跑一遍 DFS 就好了.

参考代码

```cpp
#include <cctype>
#include <iostream>
#include <random>
#include <set>
#include <vector>

using ull = unsigned long long;

const ull mask = std::mt19937_64(time(nullptr))();

ull shift(ull x) {
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}

constexpr int N = 1e6 + 10;

int n;
ull hash[N];
std::vector<int> edge[N];
std::set<ull> trees;

void getHash(int x, int p) {
  hash[x] = 1;
  for (int i : edge[x]) {
    if (i == p) {
```

```
      continue;
    }
    getHash(i, x);
    hash[x] += shift(hash[i]);
  }
  trees.insert(hash[x]);
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n;
  for (int i = 1; i < n; i++) {
    int u, v;
    cin >> u >> v;
    edge[u].push_back(v);
    edge[v].push_back(u);
  }
  getHash(1, 0);
  cout << trees.size();
}

```

[BJOI2015] 树的同构[2]

这道题所说的同构是指无根树的，而上面所介绍的方法是针对有根树的．因此只有当根一样时，同构的两棵无根树哈希值才相同．由于数据范围较小，我们可以暴力求出以每个点为根时的哈希值，排序后比较．

如果数据范围较大，我们也可以使用换根 DP，遍历树两遍，求出以每个点为根时的哈希值．我们还可以利用上面的多重集哈希函数：把以每个结点为根时的哈希值都存进多重集，再把多重集的哈希值算出来，进行比较（做法一）．

还可以通过找重心的方式来优化复杂度．一棵树的重心最多只有两个，只需把以它（们）为根时的哈希值求出来即可．接下来，既可以分别比较这些哈希值（做法二），也可以在有一个重心时取它的哈希值作为整棵树的哈希值，有两个时则取其中较小（大）的．

做法一

```
#include <iostream>
#include <map>
#include <random>
#include <vector>

using ull = unsigned long long;

constexpr int N = 60, M = 998244353;
const ull mask = std::mt19937_64(time(nullptr))();

ull shift(ull x) {
```

```
  x ^= mask;
  x ^= x << 13;
  x ^= x >> 7;
  x ^= x << 17;
  x ^= mask;
  return x;
}

std::vector<int> edge[N];
ull sub[N], root[N];
std::map<ull, int> trees;

void getSub(int x) {
  sub[x] = 1;
  for (int i : edge[x]) {
    getSub(i);
    sub[x] += shift(sub[i]);
  }
}

void getRoot(int x) {
  for (int i : edge[x]) {
    root[i] = sub[i] + shift(root[x] - shift(sub[i]));
    getRoot(i);
  }
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int m;
  cin >> m;
  for (int t = 1; t <= m; t++) {
    int n, rt = 0;
    cin >> n;
    for (int i = 1; i <= n; i++) {
      int fa;
      cin >> fa;
      if (fa) {
        edge[fa].push_back(i);
      } else {
        rt = i;
      }
    }
    getSub(rt);
    root[rt] = sub[rt];
    getRoot(rt);
    ull hash = 1;
    for (int i = 1; i <= n; i++) {
      hash += shift(root[i]);
```

```
64      }
65      if (!trees.count(hash)) {
66        trees[hash] = t;
67      }
68      cout << trees[hash] << '\n';
69      for (int i = 1; i <= n; i++) {
70        edge[i].clear();
71      }
72    }
73  }
74
```

做法二

```
1  #include <iostream>
2  #include <map>
3  #include <random>
4  #include <vector>
5
6  using ull = unsigned long long;
7  using Hash2 = std::pair<ull, ull>;
8
9  constexpr int N = 60, M = 998244353;
10 const ull mask = std::mt19937_64(time(nullptr))();
11
12 ull shift(ull x) {
13   x ^= mask;
14   x ^= x << 13;
15   x ^= x >> 7;
16   x ^= x << 17;
17   x ^= mask;
18   return x;
19 }
20
21 int n;
22 int size[N], weight[N], centroid[2];
23 std::vector<int> edge[N];
24 std::map<Hash2, int> trees;
25
26 void getCentroid(int x, int fa) {
27   size[x] = 1;
28   weight[x] = 0;
29   for (int i : edge[x]) {
30     if (i == fa) {
31       continue;
32     }
33     getCentroid(i, x);
34     size[x] += size[i];
35     weight[x] = std::max(weight[x], size[i]);
36   }
37   weight[x] = std::max(weight[x], n - size[x]);
```

```
  if (weight[x] <= n / 2) {
    int index = centroid[0] != 0;
    centroid[index] = x;
  }
}

ull getHash(int x, int fa) {
  ull hash = 1;
  for (int i : edge[x]) {
    if (i == fa) {
      continue;
    }
    hash += shift(getHash(i, x));
  }
  return hash;
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int m;
  cin >> m;
  for (int t = 1; t <= m; t++) {
    cin >> n;
    for (int i = 1; i <= n; i++) {
      int fa;
      cin >> fa;
      if (fa) {
        edge[fa].push_back(i);
        edge[i].push_back(fa);
      }
    }
    getCentroid(1, 0);
    Hash2 hash;
    hash.first = getHash(centroid[0], 0);
    if (centroid[1]) {
      hash.second = getHash(centroid[1], 0);
      if (hash.first > hash.second) {
        std::swap(hash.first, hash.second);
      }
    } else {
      hash.second = hash.first;
    }
    if (!trees.count(hash)) {
      trees[hash] = t;
    }
    cout << trees[hash] << '\n';
    for (int i = 1; i <= n; i++) {
      edge[i].clear();
    }
```

```
    centroid[0] = centroid[1] = 0;
  }
}

```

HDU 6647 Bracket Sequences on Tree[3]

题目要求遍历一棵无根树产生的本质不同括号序列方案数.

首先可以注意到, 两棵不同构的有根树一定不会生成相同的括号序列. 我们先考虑遍历有根树能够产生的本质不同括号序列方案数, 假设我们当前考虑的子树根节点为 u, 记 $f(u)$ 表示这棵子树的方案数, 从 u 开始往下遍历, 顺序可以随意选择, 产生 $|son(u)|!$ 种排列, 遍历每个儿子节点 v, v 的子树内有 $f(v)$ 种方案, 因此有 $f(u) = |son(u)|! \cdot \prod_{v \in son(u)} f(v)$. 但是, 同构的子树之间会产生重复, $f(u)$ 需要除掉每种本质不同子树出现次数阶乘的乘积, 类似于多重集合的排列.

通过上述 DP, 可以求出根节点的方案数. 再通过换根 DP, 将父亲节点的哈希值和方案信息转移给儿子, 可以求出以每个节点为根时的哈希值和方案数. 每种不同的子树只需要计数一次即可.

参考代码

```cpp
#include <iostream>
#include <map>
#include <random>
#include <vector>

using ull = unsigned long long;

constexpr int N = 1e5 + 10, M = 998244353;
const ull mask = std::mt19937_64(time(nullptr))();

struct Tree {
  ull hash, deg, ans;
  std::map<ull, ull> son;

  Tree() { clear(); }

  void add(Tree& o);
  void remove(Tree& o);
  void clear();
};

ull inv(ull x) {
  ull y = M - 2, z = 1;
  while (y) {
    if (y & 1) {
      z = z * x % M;
    }
    x = x * x % M;
    y >>= 1;
  }
  return z;
}
```

```
33
34 ull shift(ull x) {
35   x ^= mask;
36   x ^= x << 13;
37   x ^= x >> 7;
38   x ^= x << 17;
39   x ^= mask;
40   return x;
41 }
42
43 void Tree::add(Tree& o) {
44   ull temp = shift(o.hash);
45   hash += temp;
46   ans = ans * ++deg % M * inv(++son[temp]) % M * o.ans % M;
47 }
48
49 void Tree::remove(Tree& o) {
50   ull temp = shift(o.hash);
51   hash -= temp;
52   ans = ans * inv(deg--) % M * son[temp]-- % M * inv(o.ans) % M;
53 }
54
55 void Tree::clear() {
56   hash = 1;
57   deg = 0;
58   ans = 1;
59   son.clear();
60 }
61
62 std::vector<int> edge[N];
63 Tree sub[N], root[N];
64 std::map<ull, ull> trees;
65
66 void getSub(int x, int fa) {
67   for (int i : edge[x]) {
68     if (i == fa) {
69       continue;
70     }
71     getSub(i, x);
72     sub[x].add(sub[i]);
73   }
74 }
75
76 void getRoot(int x, int fa) {
77   for (int i : edge[x]) {
78     if (i == fa) {
79       continue;
80     }
81     root[x].remove(sub[i]);
82     root[i] = sub[i];
83     root[i].add(root[x]);
84     root[x].add(sub[i]);
```

```
85      getRoot(i, x);
86    }
87    trees[root[x].hash] = root[x].ans;
88  }
89
90  using std::cin;
91  using std::cout;
92
93  int main() {
94    cin.tie(nullptr)->sync_with_stdio(false);
95    int t, n;
96    cin >> t;
97    while (t--) {
98      cin >> n;
99      for (int i = 1; i < n; i++) {
100       int u, v;
101       cin >> u >> v;
102       edge[u].push_back(v);
103       edge[v].push_back(u);
104     }
105     getSub(1, 0);
106     root[1] = sub[1];
107     getRoot(1, 0);
108     ull tot = 0;
109     for (auto p : trees) {
110       tot = (tot + p.second) % M;
111     }
112     cout << tot << '\n';
113     for (int i = 1; i <= n; i++) {
114       edge[i].clear();
115       sub[i].clear();
116       root[i].clear();
117     }
118     trees.clear();
119   }
120 }
121
```

参考资料

文中的哈希方法参考并拓展自博客 一种好写且卡不掉的树哈希[4].

外部参考资料

[1] UOJ #763. 树哈希

[2] [BJOI2015] 树的同构

[3] HDU 6647 Bracket Sequences on Tree

[4] 一种好写且卡不掉的树哈希

11.6.13 树上随机游走

给定一棵有根树，树的某个结点上有一个硬币，在某一时刻硬币会等概率地移动到邻接结点上，问硬币移动到邻接结点上的期望距离.

需要用到的定义

- $T=(V,E)$: 所讨论的树
- $d(u)$: 结点 u 的度数
- $w(u,v)$: 结点 u 与结点 v 之间的边的边权
- p_u: 结点 u 的父结点
- $root$: 树的根结点
- son_u: 结点 u 的子结点集合
- $sibling_u$: 结点 u 的兄弟结点集合

向父结点走的期望距离

设 $f(u)$ 代表 u 结点走到其父结点 p_u 的期望距离，则有:

$$f(u)=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v)+f(u))}{d(u)}$$

分子中的前半部分代表直接走向了父结点，后半部分代表先走向了子结点再由子结点走回来然后再向父结点走；分母 $d(u)$ 代表从 u 结点走向其任何邻接点的概率相同.

化简如下:

$$\begin{aligned}f(u)&=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v)+f(u))}{d(u)}\\&=\frac{w(u,p_u)+\sum\limits_{v\in son_u}(w(u,v)+f(v))+(d(u)-1)f(u)}{d(u)}\\&=w(u,p_u)+\sum_{v\in son_u}(w(u,v)+f(v))\\&=\sum_{(u,t)\in E}w(u,t)+\sum_{v\in son_u}f(v)\end{aligned}$$

对于叶子结点 l，初始状态为 $f(l)=w(p_l,l)$.

当树上所有边的边权都为 1 时，上式可化为:

$$f(u)=d(u)+\sum_{v\in son_u}f(v)$$

即 u 子树的所有结点的度数和，也即 u 子树大小的两倍 -1（每个结点连向其父亲的边都有且只有一条，除 u 与 p_u 之间的边只有 1 点度数的贡献外，每条边会产生 2 点度数的贡献）.

向子结点走的期望距离

设 $g(u)$ 代表 p_u 结点走到其子结点 u 的期望距离，则有:

$$g(u)=\frac{w(p_u,u)+\Big(w(p_u,p_{p_u})+g(p_u)+g(u)\Big)+\sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)+g(u))}{d(p_u)}$$

分子中的第一部分代表直接走向了子结点 u, 第二部分代表先走向了父结点再由父结点走回来然后再向 u 结点走，第三部分代表先走向 u 结点的兄弟结点再由其走回来然后再向 u 结点走；分母 $d(p_u)$ 代表从 p_u 结点走向其任何邻接点的概率相同.

化简如下:

$$
\begin{aligned}
g(u) &= \frac{w(p_u,u) + \Big(w(p_u,p_{p_u}) + g(p_u) + g(u)\Big) + \sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)+g(u))}{d(p_u)} \\
&= \frac{w(p_u,u) + w(p_u,p_{p_u}) + g(p_u) + \sum\limits_{s\in sibling_u}(w(p_u,s)+f(s)) + (d(p_u)-1)g(u)}{d(p_u)} \\
&= w(p_u,u) + w(p_u,p_{p_u}) + g(p_u) + \sum_{s\in sibling_u}(w(p_u,s)+f(s)) \\
&= \sum_{(p_u,t)\in E} w(p_u,t) + g(p_u) + \sum_{s\in sibling_u} f(s) \\
&= \sum_{(p_u,t)\in E} w(p_u,t) + g(p_u) + \left(f(p_u) - \sum_{(p_u,t)\in E} w(p_u,t) - f(u)\right) \\
&= g(p_u) + f(p_u) - f(u)
\end{aligned}
$$

初始状态为 $g(\text{root}) = 0$.

代码实现（以无权树为例）

```cpp
vector<int> G[MAXN];

void dfs1(int u, int p) {
  f[u] = G[u].size();
  for (auto v : G[u]) {
    if (v == p) continue;
    dfs1(v, u);
    f[u] += f[v];
  }
}

void dfs2(int u, int p) {
  if (u != root) g[u] = g[p] + f[p] - f[u];
  for (auto v : G[u]) {
    if (v == p) continue;
    dfs2(v, u);
  }
}
```

11.7 有向无环图

定义

边有向，无环.

英文名叫 Directed Acyclic Graph，缩写是 DAG.

性质

- 能 拓扑排序→11.8@p2729 的图，一定是有向无环图；

  如果有环，那么环上的任意两个节点在任意序列中都不满足条件了.
- 有向无环图，一定能拓扑排序；

  （归纳法）假设节点数不超过 k 的 有向无环图都能拓扑排序，那么对于节点数等于 k 的，考虑执行拓扑排序第一步之后的情形即可.

判定

如何判定一个图是否是有向无环图呢?

检验它是否可以进行 拓扑排序→11.8@p2729 即可.

当然也有另外的方法，可以对图进行一遍 DFS→6.2@p654，在得到的 DFS 树上看看有没有连向祖先的非树边（返祖边）. 如果有的话，那就有环了.

应用

DP 求最长（短）路

在一般图上，求单源最长（短）路径的最优时间复杂度为 $O(nm)$（Bellman–Ford 算法→11.9.1@p2735，适用于有负权图）或 $O(m \log m)$（Dijkstra 算法→11.9.1@p2735，适用于无负权图）.

但在 DAG 上，我们可以使用 DP 求最长（短）路，使时间复杂度优化到 $O(n+m)$. 状态转移方程为 $dis_v = min(dis_v, dis_u + w_{u,v})$ 或 $dis_v = max(dis_v, dis_u + w_{u,v})$.

拓扑排序后，按照拓扑序遍历每个节点，用当前节点来更新之后的节点.

```
struct edge {
  int v, w;
};

int n, m;
vector<edge> e[MAXN];
vector<int> L;                                  // 存储拓扑排序结果
int max_dis[MAXN], min_dis[MAXN], in[MAXN];  // in 存储每个节点的入度

void toposort() {  // 拓扑排序
  queue<int> S;
  memset(in, 0, sizeof(in));
  for (int i = 1; i <= n; i++) {
    for (int j = 0; j < e[i].size(); j++) {
      in[e[i][j].v]++;
    }
  }
  for (int i = 1; i <= n; i++)
    if (in[i] == 0) S.push(i);
  while (!S.empty()) {
    int u = S.front();
    S.pop();
    L.push_back(u);
    for (int i = 0; i < e[u].size(); i++) {
      if (--in[e[u][i].v] == 0) {
        S.push(e[u][i].v);
      }
    }
  }
}

void dp(int s) {  // 以 s 为起点求单源最长（短）路
  toposort();     // 先进行拓扑排序
  memset(min_dis, 0x3f, sizeof(min_dis));
  memset(max_dis, 0, sizeof(max_dis));
  min_dis[s] = 0;
```

```
37   for (int i = 0; i < L.size(); i++) {
38     int u = L[i];
39     for (int j = 0; j < e[u].size(); j++) {
40       min_dis[e[u][j].v] = min(min_dis[e[u][j].v], min_dis[u] + e[u][j].w);
41       max_dis[e[u][j].v] = max(max_dis[e[u][j].v], max_dis[u] + e[u][j].w);
42     }
43   }
44 }
```

参见：DAG 上的 DP→7.6@p735.

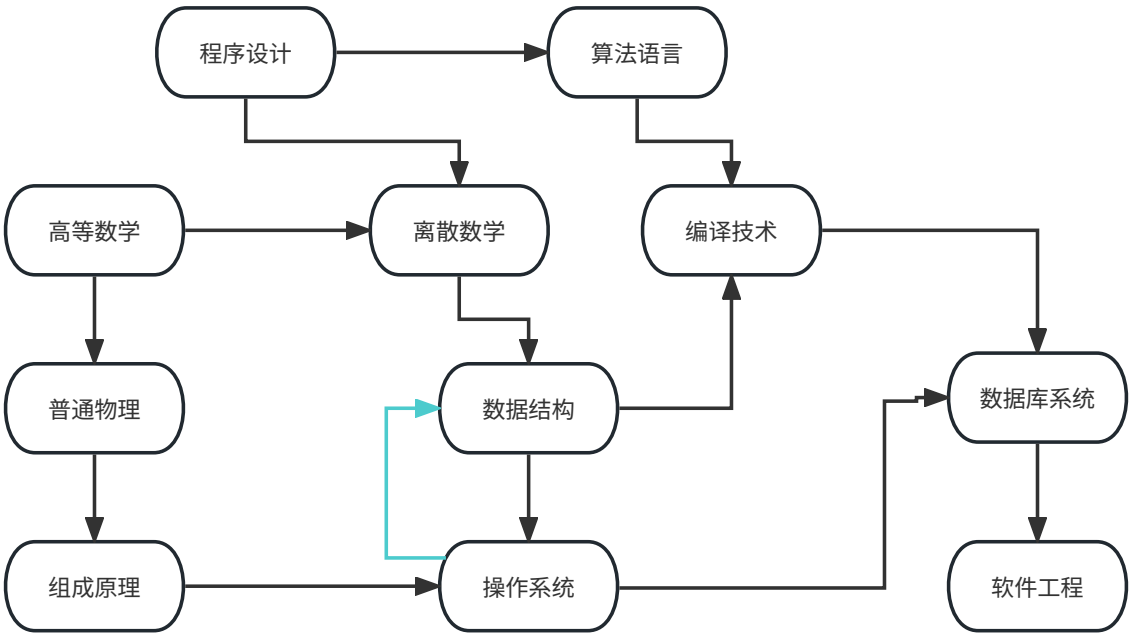
11.8 拓扑排序

Authors: marscheng1

定义

拓扑排序（Topological sorting）要解决的问题是如何给一个有向无环图的所有节点排序.

我们可以拿大学每学期排课的例子来描述这个过程，比如学习大学课程中有:「程序设计」,「算法语言」,「高等数学」,「离散数学」,「编译技术」,「普通物理」,「数据结构」,「数据库系统」等. 按照例子中的排课，当我们想要学习「数据结构」的时候，就必须先学会「离散数学」，学习完这门课后就获得了学习「编译技术」的前置条件. 当然,「编译技术」还有一个更加前的课程「算法语言」. 这些课程就相当于几个顶点 u, 顶点之间的有向边 (u,v) 就相当于学习课程的顺序. 教务处安排这些课程，使得在逻辑关系符合的情况下排出课表，就是拓扑排序的过程.



但是如果某一天排课的老师打瞌睡了，说想要学习 数据结构，还得先学 操作系统，而 操作系统 的前置课程又是 数据结构，那么到底应该先学哪一个（不考虑同时学习的情况)? 在这里，数据结构 和 操作系统 间就出现了一个环，显然同学们现在没办法弄清楚自己需要先学什么了，也就没办法进行拓扑排序了. 因为如果有向图中存在环路，那么我们就没办法进行拓扑排序.

因此我们可以说 在一个 DAG（有向无环图）→11.7@p2727 中，我们将图中的顶点以线性方式进行排序，使得对于任何的顶点 u 到 v 的有向边 (u,v), 都可以有 u 在 v 的前面.

还有给定一个 DAG，如果从 i 到 j 有边，则认为 j 依赖于 i．如果 i 到 j 有路径（i 可达 j），则称 j 间接依赖于 i.

拓扑排序的目标是将所有节点排序，使得排在前面的节点不能依赖于排在后面的节点.

AOV 网

日常生活中，一项大的工程可以看作是由若干个子工程组成的集合，这些子工程之间必定存在一定的先后顺序，即某些子工程必须在其他的一些子工程完成后才能开始.

我们用有向图来表现子工程之间的先后关系，子工程之间的先后关系为有向边，这种有向图称为顶点活动网络，即 **AOV 网 (Activity On Vertex Network)**．一个 AOV 网必定是一个有向无环图，即不带有回路．与 DAG 不同的是，AOV 的活动都表示在顶点上．（上面的例图即为一个 AOV 网）

在 AOV 网中，顶点表示活动，弧表示活动间的优先关系．AOV 网中不应该出现环，这样就能够找到一个顶点序列，使得每个顶点代表的活动的前驱活动都排在该顶点的前面，这样的序列称为拓扑序列（一个 AOV 网的拓扑序列不是唯一的），由 AOV 网构造拓扑序列的过程称为拓扑排序．因此，拓扑排序也可以解释为将 AOV 网中所有活动排成一个序列，使得每个活动的前驱活动都排在该活动的前面（一个 AOV 网中的拓扑排序也不是唯一的）.

- 前驱活动：有向边起点的活动称为终点的前驱活动（只有当一个活动的前驱全部都完成后，这个活动才能进行）.
- 后继活动：有向边终点的活动称为起点的后继活动.

检测 AOV 网中是否带环的方式是构造拓扑序列，看是否包含所有顶点.

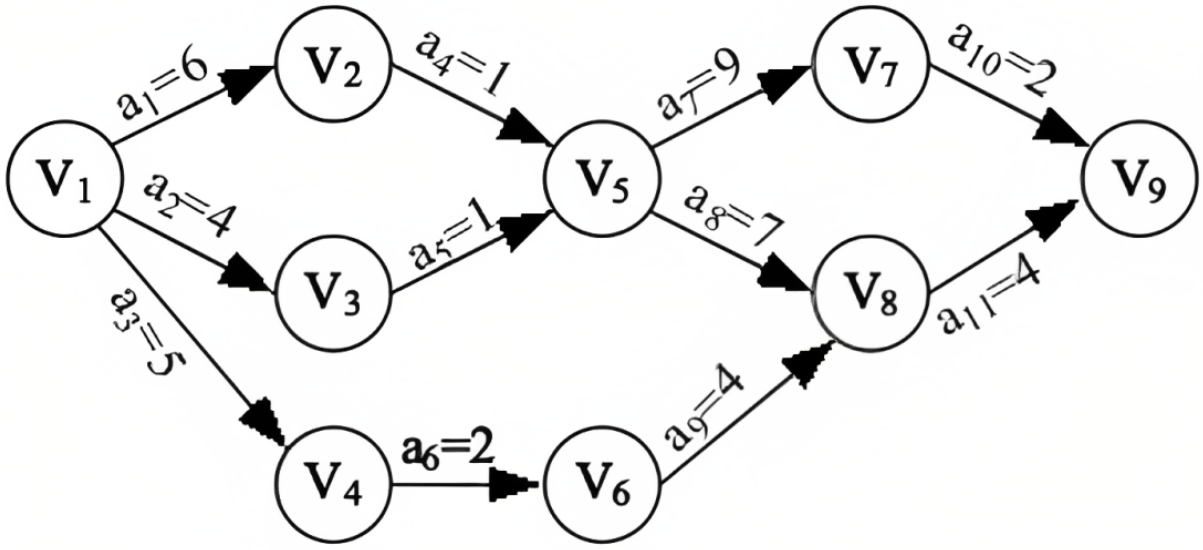
构造拓扑序列步骤

1. 从图中选择一个入度为零的点.
2. 输出该顶点，从图中删除此顶点及其所有的出边.

重复上面两步，直到所有顶点都输出，拓扑排序完成，或者图中不存在入度为零的点，此时说明图是有环图，拓扑排序无法完成，陷入死锁.

关键路径和 AOE 网

与 AOV 网对应的是 **AOE 网（Activity On Edge Network)** 即边表示活动的网．AOE 网是一个带权的有向无环图，其中，顶点表示事件，弧表示活动持续的时间．通常，AOE 网可以用来估算工程的完成时间．AOE 网应该是无环的，且存在唯一入度为零的起始顶点（源点），以及唯一出度为零的完成顶点（汇点）.



AOE 网中的有些活动是可以并行进行的，所以完成整个工程的最短时间是从开始点到完成点的最长活动路径长度（这里所说的路径长度是指路径上各活动的持续时间之和，即弧的权值之和，不是路径上弧的数目）．因为一项工程需要完成所有工程内的活动，所以最长的活动路径也是关键路径，它决定工程完成的总时间．

AOE 网的相关基本概念

- 活动：AOE 网中，弧表示活动．弧的权值表示活动持续的时间，活动在其前驱事件（即该弧的起点）被触发后开始．
- 事件：AOE 网中，顶点表示事件，事件在它的所有前驱活动（即指向该边的弧）全部完成被触发．
- 事件（顶点）v_i 的最早发生时间：该事件最早可能的发生时间，记为 $ve(i)$，它决定了以该顶点开始的活动的最早发生时间，显然源点的最早发生时间为 0．因为事件发生需要其所有前驱活动全部完成，所以它等于初始点到该顶点的路径长度的最大值，写成递推：$ve(i) = \max\{ve(j) + val_i^j \mid j \in pre_i\}$，其中 val_i^j 表示 j 到 i 的边的权值（即 j 到 i 的活动的持续时间），pre_i 表示 i 的所有前驱事件的集合．
- 事件（顶点）v_i 的最迟发生时间：在不推迟整个工期的前提下，该事件最晚能容忍的发生时间，记为 $vl(i)$，它决定了所有以该状态结束的活动的最迟发生时间，它等于事件的所有后继活动的最迟开始时间的最小值，即 $vl(i) = \min\{vl(j) - val_j^i \mid j \in nxt_i\}$，其中 val_j^i 表示 i 到 j 的边的权值（即 i 到 j 的活动的持续时间），nxt_i 表示 i 的所有后驱事件的集合．
- 活动（弧）(u, v) 的最早开始时间：该活动最早可能的发生时间，记为 $e(u, v)$，显然，它等于其前驱事件的最早发生时间，即 $e(u, v) = ve(u)$．
- 活动（弧）(u, v) 的最迟开始时间：在不推迟整个工期的前提下，活动开始最晚能容忍的时间，记为 $l(u, v)$，它等于其后继事件的最迟发生时间 - 该事件的持续时间（权值），即 $l(u, v) = vl(v) - val_v^u$，其中 val_v^u 表示 u 到 v 的边的权值（即 u 到 v 的活动的持续时间）．
- 关键路径：AOE 网中从源点到汇点的最长路径的长度．
- 关键活动：即关键路径上的活动，它的最早开始时间和最迟开始时间相等．

递推求最早和最迟发生时间

按拓扑顺序求，最早发生时间从前往后递推，最迟发生时间从后往前递推，递推公式如上 **AOE 网的相关基本概念** 所示．

Kahn 算法

过程

初始状态下，集合 S 装着所有入度为 0 的点，L 是一个空列表．

每次从 S 中取出一个点 u（可以随便取）放入 L，然后将 u 的所有边 $(u, v_1), (u, v_2), (u, v_3)\cdots$ 删除．对于边 (u, v)，若将该边删除后点 v 的入度变为 0，则将 v 放入 S 中．

不断重复以上过程，直到集合 S 为空．检查图中是否存在任何边，如果有，那么这个图一定有环路，否则返回 L，L 中顶点的顺序就是构造拓扑序列的结果．

首先看来自 Wikipedia[1] 的伪代码

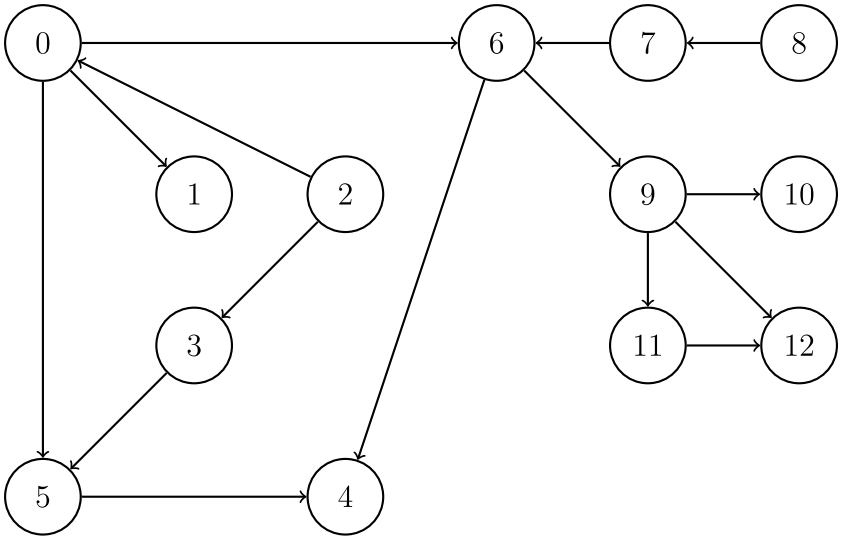
实现

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while S is not empty do
    remove a node n from S
    insert n into L
```

```
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph has at least one cycle)
else
    return L (a topologically sorted order)
```

代码的核心是维持一个入度为 0 的顶点的集合.

可以参考该图



对其排序的结果就是：2 -> 8 -> 0 -> 3 -> 7 -> 1 -> 5 -> 6 -> 9 -> 4 -> 11 -> 10 -> 12

时间复杂度

假设这个图 $G = (V, E)$ 在初始化入度为 0 的集合 S 的时候就需要遍历整个图，并检查每一条边，因而有 $O(E + V)$ 的复杂度. 然后对该集合进行操作，显然也是需要 $O(E + V)$ 的时间复杂度.

因而总的时间复杂度就有 $O(E + V)$

实现

- C++

```cpp
int n, m;
vector<int> G[MAXN];
int in[MAXN];  // 存储每个结点的入度

bool toposort() {
  vector<int> L;
  queue<int> S;
  for (int i = 1; i <= n; i++)
    if (in[i] == 0) S.push(i);
  while (!S.empty()) {
    int u = S.front();
    S.pop();
    L.push_back(u);
```

```
    for (auto v : G[u]) {
      if (--in[v] == 0) {
        S.push(v);
      }
    }
  }
  if (L.size() == n) {
    for (auto i : L) cout << i << ' ';
    return true;
  }
  return false;
}
```

- Python

```
from collections import defaultdict, deque


def topo_sort(graph):
    lst = []
    in_degree = defaultdict(int)
    for u in graph:
        for v in graph[u]:
            in_degree[v] += 1

    s = deque([u for u in graph if in_degree[u] == 0])
    while s:
        u = s.popleft()
        lst.append(u)
        for v in graph.get(u, []):
            in_degree[v] -= 1
            if in_degree[v] == 0:
                s.append(v)

    return None if any(in_degree.values()) else lst
```

DFS 算法

实现

- C++

```
using Graph = vector<vector<int>>;  // 邻接表

struct TopoSort {
  enum class Status : uint8_t { to_visit, visiting, visited };

  const Graph& graph;
  const int n;
  vector<Status> status;
  vector<int> order;
  vector<int>::reverse_iterator it;

  TopoSort(const Graph& graph)
      : graph(graph),
        n(graph.size()),
```

```
        status(n, Status::to_visit),
        order(n),
        it(order.rbegin()) {}

  bool sort() {
    for (int i = 0; i < n; ++i) {
      if (status[i] == Status::to_visit && !dfs(i)) return false;
    }
    return true;
  }

  bool dfs(const int u) {
    status[u] = Status::visiting;
    for (const int v : graph[u]) {
      if (status[v] == Status::visiting) return false;
      if (status[v] == Status::to_visit && !dfs(v)) return false;
    }
    status[u] = Status::visited;
    *it++ = u;
    return true;
  }
};
```

- Python

```
from enum import Enum, auto


class Status(Enum):
    to_visit = auto()
    visiting = auto()
    visited = auto()


def topo_sort(graph: list[list[int]]) -> list[int] | None:
    n = len(graph)
    status = [Status.to_visit] * n
    order = []

    def dfs(u: int) -> bool:
        status[u] = Status.visiting
        for v in graph[u]:
            if status[v] == Status.visiting:
                return False
            if status[v] == Status.to_visit and not dfs(v):
                return False
        status[u] = Status.visited
        order.append(u)
        return True

    for i in range(n):
        if status[i] == Status.to_visit and not dfs(i):
            return None

    return order[::-1]
```

时间复杂度：$O(E+V)$ 空间复杂度：$O(V)$

合理性证明

考虑一个图，删掉某个入度为 0 的节点之后，如果新图可以拓扑排序，那么原图一定也可以. 反过来，如果原图可以拓扑排序，那么删掉后也可以.

应用

拓扑排序可以判断图中是否有环，还可以用来判断图是否是一条链. 拓扑排序可以用来求 AOE 网中的关键路径，估算工程完成的最短时间.

求字典序最大/最小的拓扑排序

将 Kahn 算法中的队列替换成最大堆/最小堆实现的优先队列即可，此时总的时间复杂度为 $O(E+V\log V)$.

习题

CF 1385E[2]：需要通过拓扑排序构造.

Luogu P1347[3]: 拓扑排序模板.

参考

1. 离散数学及其应用. ISBN:9787111555391
2. Topological sorting - Wikipedia[4]
3. 数据结构第九讲（图：拓扑排序，关键路径，最短路径）- 知乎专栏[5]

外部参考资料

[1] Wikipedia

[2] CF 1385E

[3] Luogu P1347

[4] Topological sorting - Wikipedia

[5] 数据结构第九讲（图：拓扑排序，关键路径，最短路径）- 知乎专栏

11.9 最短路问题

11.9.1 最短路

Authors: du33169, lingkerio, Taoran-01

定义

（还记得这些定义吗？在阅读下列内容之前，请务必了解 图论相关概念→11.2@p2598 中的基础部分.）

- 路径
- 最短路
- 有向图中的最短路、无向图中的最短路
- 单源最短路、每对结点之间的最短路

记号

为了方便叙述，这里先给出下文将会用到的一些记号的含义.

- n 为图上点的数目，m 为图上边的数目；
- s 为最短路的源点；
- $D(u)$ 为 s 点到 u 点的 **实际** 最短路长度；

- $dis(u)$ 为 s 点到 u 点的 **估计** 最短路长度. 任何时候都有 $dis(u) \geq D(u)$. 特别地, 当最短路算法终止时, 应有 $dis(u) = D(u)$.
- $w(u, v)$ 为 (u, v) 这一条边的边权.

性质

对于边权为正的图, 任意两个结点之间的最短路, 不会经过重复的结点.

对于边权为正的图, 任意两个结点之间的最短路, 不会经过重复的边.

对于边权为正的图, 任意两个结点之间的最短路, 任意一条的结点数不会超过 n, 边数不会超过 $n - 1$.

Floyd 算法

是用来求任意两个结点之间的最短路的.

复杂度比较高, 但是常数小, 容易实现 (只有三个 `for`).

适用于任何图, 不管有向无向, 边权正负, 但是最短路必须存在. (不能有个负环)

实现

我们定义一个数组 `f[k][x][y]`, 表示只允许经过结点 1 到 k (也就是说, 在子图 $V' = 1, 2, ..., k$ 中的路径, 注意, x 与 y 不一定在这个子图中), 结点 x 到结点 y 的最短路长度.

很显然, `f[n][x][y]` 就是结点 x 到结点 y 的最短路长度 (因为 $V' = 1, 2, ..., n$ 即为 V 本身, 其表示的最短路径就是所求路径).

接下来考虑如何求出 `f` 数组的值.

`f[0][x][y]`: x 与 y 的边权, 或者 0, 或者 $+\infty$ (`f[0][x][y]` 什么时候应该是 $+\infty$? 当 x 与 y 间有直接相连的边的时候, 为它们的边权; 当 $x = y$ 的时候为零, 因为到本身的距离为零; 当 x 与 y 没有直接相连的边的时候, 为 $+\infty$).

`f[k][x][y] = min(f[k-1][x][y], f[k-1][x][k]+f[k-1][k][y])` (`f[k-1][x][y]`, 为不经过 k 点的最短路径, 而 `f[k-1][x][k]+f[k-1][k][y]`, 为经过了 k 点的最短路).

上面两行都显然是对的, 所以说这个做法空间是 $O(N^3)$, 我们需要依次增加问题规模 (k 从 1 到 n), 判断任意两点在当前问题规模下的最短路.

- C++

```cpp
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
    }
  }
}
```

- Python

```python
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1]
[k][y])
```

因为第一维对结果无影响, 我们可以发现数组的第一维是可以省略的, 于是可以直接改成 `f[x][y] = min(f[x][y], f[x][k]+f[k][y])`.

证明第一维对结果无影响

对于给定的 k，当更新 f[k][x][y] 时，涉及的元素总是来自 f[k-1] 数组的第 k 行和第 k 列. 然后我们可以发现，对于给定的 k，当更新 f[k][k][y] 或 f[k][x][k]，总是不会发生数值更新，因为按照公式 f[k][k][y] = min(f[k-1][k][y], f[k-1][k][k]+f[k-1][k][y]),f[k-1][k][k] 为 0，因此这个值总是 f[k-1][k][y]，对于 f[k][x][k] 的证明类似.

因此，如果省略第一维，在给定的 k 下，每个元素的更新中使用到的元素都没有在这次迭代中更新，因此第一维的省略并不会影响结果.

- C++

```cpp
for (k = 1; k <= n; k++) {
  for (x = 1; x <= n; x++) {
    for (y = 1; y <= n; y++) {
      f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
    }
  }
}
```

- Python

```python
for k in range(1, n + 1):
    for x in range(1, n + 1):
        for y in range(1, n + 1):
            f[x][y] = min(f[x][y], f[x][k] + f[k][y])
```

综上时间复杂度是 $O(N^3)$，空间复杂度是 $O(N^2)$.

应用

给一个正权无向图，找一个最小权值和的环.

首先这一定是一个简单环.

想一想这个环是怎么构成的.

考虑环上编号最大的结点 u.

f[u-1][x][y] 和 $(u,x),(u,y)$ 共同构成了环.

在 Floyd 的过程中枚举 u，计算这个和的最小值即可.

时间复杂度为 $O(n^3)$.

更多参见 最小环→11.15@p2854 部分内容.

已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通.

该问题即是求 **图的传递闭包**.

我们只需要按照 Floyd 的过程，逐个加入点判断一下.

只是此时的边的边权变为 1/0，而取 min 变成了 **或** 运算.

再进一步用 bitset 优化，复杂度可以到 $O(\frac{n^3}{w})$.

```cpp
// std::bitset<SIZE> f[SIZE];
for (k = 1; k <= n; k++)
```

```
  for (i = 1; i <= n; i++)
    if (f[i][k]) f[i] = f[i] | f[k];
```

Bellman–Ford 算法

Bellman–Ford 算法是一种基于松弛（relax）操作的最短路算法，可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断.

在国内 OI 界，你可能听说过的「SPFA」，就是 Bellman–Ford 算法的一种实现.

过程

先介绍 Bellman–Ford 算法要用到的松弛操作（Dijkstra 算法也会用到松弛操作）.

对于边 (u, v)，松弛操作对应下面的式子：$dis(v) = \min(dis(v), dis(u) + w(u, v))$.

这么做的含义是显然的：我们尝试用 $S \to u \to v$（其中 $S \to u$ 的路径取最短路）这条路径去更新 v 点最短路的长度，如果这条路径更优，就进行更新.

Bellman–Ford 算法所做的，就是不断尝试对图上每一条边进行松弛. 我们每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止.

每次循环是 $O(m)$ 的，那么最多会循环多少次呢?

在最短路存在的情况下，由于一次松弛操作会使最短路的边数至少 $+1$，而最短路的边数最多为 $n - 1$，因此整个算法最多执行 $n - 1$ 轮松弛操作. 故总时间复杂度为 $O(nm)$.

但还有一种情况，如果从 S 点出发，抵达一个负环时，松弛操作会无休止地进行下去. 注意到前面的论证中已经说明了，对于最短路存在的图，松弛操作最多只会执行 $n - 1$ 轮，因此如果第 n 轮循环时仍然存在能松弛的边，说明从 S 点出发，能够抵达一个负环.

⚠ 负环判断中存在的常见误区

需要注意的是，以 S 点为源点跑 Bellman–Ford 算法时，如果没有给出存在负环的结果，只能说明从 S 点出发不能抵达一个负环，而不能说明图上不存在负环.

因此如果需要判断整个图上是否存在负环, 最严谨的做法是建立一个超级源点, 向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman–Ford 算法.

实现

参考实现

- C++

```
struct Edge {
  int u, v, w;
};

vector<Edge> edge;

int dis[MAXN], u, v, w;
constexpr int INF = 0x3f3f3f3f;

bool bellmanford(int n, int s) {
```

```cpp
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
  bool flag = false;  // 判断一轮循环过程中是否发生松弛操作
  for (int i = 1; i <= n; i++) {
    flag = false;
    for (int j = 0; j < edge.size(); j++) {
      u = edge[j].u, v = edge[j].v, w = edge[j].w;
      if (dis[u] == INF) continue;
      // 无穷大与常数加减仍然为无穷大
      // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        flag = true;
      }
    }
    // 没有可以松弛的边时就停止算法
    if (!flag) {
      break;
    }
  }
  // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
  return flag;
}
```

- Python

```python
class Edge:
    def __init__(self, u=0, v=0, w=0):
        self.u = u
        self.v = v
        self.w = w


INF = 0x3F3F3F3F
edge = []


def bellmanford(n, s):
    dis = [INF] * (n + 1)
    dis[s] = 0
    for i in range(1, n + 1):
        flag = False
        for e in edge:
            u, v, w = e.u, e.v, e.w
            if dis[u] == INF:
                continue
            # 无穷大与常数加减仍然为无穷大
            # 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                flag = True
        # 没有可以松弛的边时就停止算法
        if not flag:
```

```
            break
    # 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环
    return flag
```

队列优化：SPFA

即 Shortest Path Faster Algorithm.

很多时候我们并不需要那么多无用的松弛操作.

很显然，只有上一次被松弛的结点，所连接的边，才有可能引起下一次的松弛操作.

那么我们用队列来维护「哪些结点可能会引起松弛操作」，就能只访问必要的边了.

SPFA 也可以用于判断 s 点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少 n 条边时，说明 s 点可以抵达一个负环.

实现

- C++

```cpp
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], cnt[MAXN], vis[MAXN];
queue<int> q;

bool spfa(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0, vis[s] = 1;
  q.push(s);
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = 0;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        cnt[v] = cnt[u] + 1;  // 记录最短路经过的边数
        if (cnt[v] >= n) return false;
        // 在不经过负环的情况下，最短路至多经过 n - 1 条边
        // 因此如果经过了多于 n 条边，一定说明经过了负环
        if (!vis[v]) q.push(v), vis[v] = 1;
      }
    }
  }
  return true;
}
```

- Python

```python
from collections import deque

```

```python

class Edge:
    def __init__(self, v=0, w=0):
        self.v = v
        self.w = w


e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
INF = 0x3F3F3F3F


def spfa(n, s):
    dis = [INF] * (n + 1)
    cnt = [0] * (n + 1)
    vis = [False] * (n + 1)
    q = deque()

    dis[s] = 0
    vis[s] = True
    q.append(s)
    while q:
        u = q.popleft()
        vis[u] = False
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                cnt[v] = cnt[u] + 1  # 记录最短路经过的边数
                if cnt[v] >= n:
                    return False
                # 在不经过负环的情况下，最短路至多经过 n - 1 条边
                # 因此如果经过了多于 n 条边，一定说明经过了负环
                if not vis[v]:
                    q.append(v)
                    vis[v] = True
```

虽然在大多数情况下 SPFA 跑得很快，但其最坏情况下的时间复杂度为 $O(nm)$，将其卡到这个复杂度也是不难的，所以考试时要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman–Ford 算法无法通过的数据范围）.

Bellman–Ford 的其他优化

除了队列优化（SPFA）之外，Bellman–Ford 还有其他形式的优化，这些优化在部分图上效果明显，但在某些特殊图上，最坏复杂度可能达到指数级.

- 堆优化：将队列换成堆，与 Dijkstra 的区别是允许一个点多次入队．在有负权边的图可能被卡成指数级复杂度.
- 栈优化：将队列换成栈（即将原来的 BFS 过程变成 DFS），在寻找负环时可能具有更高效率，但最坏时间复杂度仍然为指数级.
- LLL 优化：将普通队列换成双端队列，每次将入队结点距离和队内距离平均值比较，如果更大则插入至队尾，否则插入队首.

- SLF 优化：将普通队列换成双端队列，每次将入队结点距离和队首比较，如果更大则插入至队尾，否则插入队首.
- D'Esopo–Pape 算法：将普通队列换成双端队列，如果一个节点之前没有入队，则将其插入队尾，否则插入队首.

更多优化以及针对这些优化的 Hack 方法，可以看 fstqwq 在知乎上的回答[1].

Dijkstra 算法

Dijkstra（/'dikstrɑ/或/'dɛikstrɑ/）算法由荷兰计算机科学家 E. W. Dijkstra 于 1956 年发现，1959 年公开发表. 是一种求解 **非负权图** 上单源最短路径的算法.

过程

将结点分成两个集合：已确定最短路长度的点集（记为 S 集合）的和未确定最短路长度的点集（记为 T 集合）. 一开始所有的点都属于 T 集合.

初始化 $dis(s)=0$，其他点的 dis 均为 $+\infty$.

然后重复这些操作：

1. 从 T 集合中，选取一个最短路长度最小的结点，移到 S 集合中.
2. 对那些刚刚被加入 S 集合的结点的所有出边执行松弛操作.

直到 T 集合为空，算法结束.

时间复杂度

朴素的实现方法为每次 2 操作执行完毕后，直接在 T 集合中暴力寻找最短路长度最小的结点. 2 操作总时间复杂度为 $O(m)$，1 操作总时间复杂度为 $O(n^2)$，全过程的时间复杂度为 $O(n^2+m)=O(n^2)$.

可以用堆来优化这一过程：每成功松弛一条边 (u,v)，就将 v 插入堆中（如果 v 已经在堆中，直接执行 Decrease-key），1 操作直接取堆顶结点即可. 共计 $O(m)$ 次 Decrease-key，$O(n)$ 次 pop，选择不同堆可以取到不同的复杂度，参考 堆→10.7.1@p2025 页面. 堆优化能做到的最优复杂度为 $O(n\log n+m)$，能做到这一复杂度的有斐波那契堆等.

特别地，可以使用优先队列维护，此时无法执行 Decrease-key 操作，但可以通过每次松弛时重新插入该结点，且弹出时检查该结点是否已被松弛过，若是则跳过，复杂度 $O(m\log n)$，优点是实现较简单.

这里的堆也可以用线段树来实现，复杂度为 $O(m\log n)$，在一些特殊的非递归线段树实现下，该做法常数比堆更小. 并且线段树支持的操作更多，在一些特殊图问题上只能用线段树来维护.

在稀疏图中，$m=O(n)$，堆优化的 Dijkstra 算法具有较大的效率优势；而在稠密图中，$m=O(n^2)$，这时候使用朴素实现更优.

正确性证明

下面用数学归纳法证明，在 **所有边权值非负** 的前提下，Dijkstra 算法的正确性[1].

简单来说，我们要证明的，就是在执行 1 操作时，取出的结点 u 最短路均已经被确定，即满足 $D(u)=dis(u)$.

初始时 $S=\emptyset$，假设成立.

接下来用反证法.

设 u 点为算法中第一个在加入 S 集合时不满足 $D(u)=dis(u)$ 的点. 因为 s 点一定满足 $D(u)=dis(u)=0$，且它一定是第一个加入 S 集合的点，因此将 u 加入 S 集合前，$S\neq\emptyset$，如果不存在 s 到 u 的路径，则 $D(u)=dis(u)=+\infty$，与假设矛盾.

[1]《算法导论（第 3 版中译本）》，机械工业出版社，2013 年，第 384 - 385 页.

于是一定存在路径 $s \to x \to y \to u$，其中 y 为 $s \to u$ 路径上第一个属于 T 集合的点，而 x 为 y 的前驱结点（显然 $x \in S$）. 需要注意的是，可能存在 $s = x$ 或 $y = u$ 的情况，即 $s \to x$ 或 $y \to u$ 可能是空路径.

因为在 u 结点之前加入的结点都满足 $D(u) = dis(u)$，所以在 x 点加入到 S 集合时，有 $D(x) = dis(x)$，此时边 (x, y) 会被松弛，从而可以证明，将 u 加入到 S 时，一定有 $D(y) = dis(y)$.

下面证明 $D(u) = dis(u)$ 成立. 在路径 $s \to x \to y \to u$ 中，因为图上所有边边权非负，因此 $D(y) \leq D(u)$. 从而 $dis(y) = D(y) \leq D(u) \leq dis(u)$. 但是因为 u 结点在 1 过程中被取出 T 集合时，y 结点还没有被取出 T 集合，因此此时有 $dis(u) \leq dis(y)$，从而得到 $dis(y) = D(y) = D(u) = dis(u)$，这与 $D(u) \neq dis(u)$ 的假设矛盾，故假设不成立.

因此我们证明了，1 操作每次取出的点，其最短路均已经被确定. 命题得证.

注意到证明过程中的关键不等式 $D(y) \leq D(u)$ 是在图上所有边边权非负的情况下得出的. 当图上存在负权边时，这一不等式不再成立，Dijkstra 算法的正确性将无法得到保证，算法可能会给出错误的结果.

实现

这里同时给出 $O(n^2)$ 的暴力做法实现和 $O(m \log m)$ 的优先队列做法实现.

朴素实现

- C++

```cpp
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];

void dijkstra(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
  for (int i = 1; i <= n; i++) {
    int u = 0, mind = 0x3f3f3f3f;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && dis[j] < mind) u = j, mind = dis[j];
    vis[u] = true;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
    }
  }
}
```

- Python

```python
class Edge:
    def __init__(self, v=0, w=0):
        self.v = v
        self.w = w


e = [[Edge() for i in range(MAXN)] for j in range(MAXN)]
INF = 0x3F3F3F3F

```

```python

def dijkstra(n, s):
    dis = [INF] * (n + 1)
    vis = [0] * (n + 1)

    dis[s] = 0
    for i in range(1, n + 1):
        u = 0
        mind = INF
        for j in range(1, n + 1):
            if not vis[j] and dis[j] < mind:
                u = j
                mind = dis[j]
        vis[u] = True
        for ed in e[u]:
            v, w = ed.v, ed.w
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
```

优先队列实现

- C++

```cpp
struct edge {
  int v, w;
};

struct node {
  int dis, u;

  bool operator>(const node& a) const { return dis > a.dis; }
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];
priority_queue<node, vector<node>, greater<node>> q;

void dijkstra(int n, int s) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  memset(vis, 0, (n + 1) * sizeof(int));
  dis[s] = 0;
  q.push({0, s});
  while (!q.empty()) {
    int u = q.top().u;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.push({dis[v], v});
```

```
30        }
31      }
32    }
33  }
```

- Python

```python
def dijkstra(e, s):
    """
    输入:
    e:邻接表
    s:起点
    返回:
    dis:从s到每个顶点的最短路长度
    """
    dis = defaultdict(lambda: float("inf"))
    dis[s] = 0
    q = [(0, s)]
    vis = set()
    while q:
        _, u = heapq.heappop(q)
        if u in vis:
            continue
        vis.add(u)
        for v, w in e[u]:
            if dis[v] > dis[u] + w:
                dis[v] = dis[u] + w
                heapq.heappush(q, (dis[v], v))
    return dis
```

Johnson 全源最短路径算法

Johnson 和 Floyd 一样，是一种能求出无负环图上任意两点间最短路径的算法．该算法在 1977 年由 Donald B. Johnson 提出．

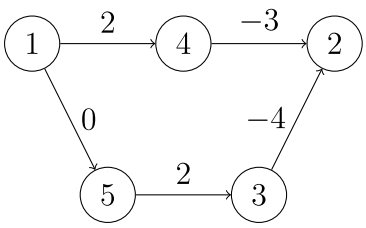
任意两点间的最短路可以通过枚举起点，跑 n 次 Bellman–Ford 算法解决，时间复杂度是 $O(n^2m)$ 的，也可以直接用 Floyd 算法解决，时间复杂度为 $O(n^3)$．

注意到堆优化的 Dijkstra 算法求单源最短路径的时间复杂度比 Bellman–Ford 更优，如果枚举起点，跑 n 次 Dijkstra 算法，就可以在 $O(nm\log m)$（取决于 Dijkstra 算法的实现）的时间复杂度内解决本问题，比上述跑 n 次 Bellman–Ford 算法的时间复杂度更优秀，在稀疏图上也比 Floyd 算法的时间复杂度更加优秀．

但 Dijkstra 算法不能正确求解带负权边的最短路，因此我们需要对原图上的边进行预处理，确保所有边的边权均非负．

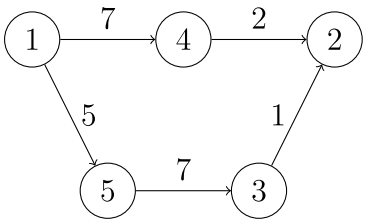
一种容易想到的方法是给所有边的边权同时加上一个正数 x，从而让所有边的边权均非负．如果新图上起点到终点的最短路经过了 k 条边，则将最短路减去 kx 即可得到实际最短路．

但这样的方法是错误的．考虑下图：



$1 \to 2$ 的最短路为 $1 \to 5 \to 3 \to 2$，长度为 -2.

但假如我们把每条边的边权加上 5 呢?



新图上 $1 \to 2$ 的最短路为 $1 \to 4 \to 2$，已经不是实际的最短路了.

Johnson 算法则通过另外一种方法来给每条边重新标注边权.

我们新建一个虚拟节点（在这里我们就设它的编号为 0）. 从这个点向其他所有点连一条边权为 0 的边.

接下来用 Bellman–Ford 算法求出从 0 号点到其他所有点的最短路，记为 h_i.

假如存在一条从 u 点到 v 点，边权为 w 的边，则我们将该边的边权重新设置为 $w + h_u - h_v$.

接下来以每个点为起点，跑 n 轮 Dijkstra 算法即可求出任意两点间的最短路了.

一开始的 Bellman–Ford 算法并不是时间上的瓶颈，若使用 `priority_queue` 实现 Dijkstra 算法，该算法的时间复杂度是 $O(nm \log m)$.

正确性证明

为什么这样重新标注边权的方式是正确的呢?

在讨论这个问题之前，我们先讨论一个物理概念——势能.

诸如重力势能，电势能这样的势能都有一个特点，势能的变化量只和起点和终点的相对位置有关，而与起点到终点所走的路径无关.

势能还有一个特点，势能的绝对值往往取决于设置的零势能点，但无论将零势能点设置在哪里，两点间势能的差值是一定的.

接下来回到正题.

在重新标记后的图上，从 s 点到 t 点的一条路径 $s \to p_1 \to p_2 \to ... \to p_k \to t$ 的长度表达式如下:

$(w(s, p_1) + h_s - h_{p_1}) + (w(p_1, p_2) + h_{p_1} - h_{p_2}) + ... + (w(p_k, t) + h_{p_k} - h_t)$

化简后得到:

$w(s, p_1) + w(p_1, p_2) + ... + w(p_k, t) + h_s - h_t$

无论我们从 s 到 t 走的是哪一条路径，$h_s - h_t$ 的值是不变的，这正与势能的性质相吻合!

为了方便，下面我们就把 h_i 称为 i 点的势能.

上面的新图中 $s \to t$ 的最短路的长度表达式由两部分组成，前面的边权和为原图中 $s \to t$ 的最短路，后面则是两点间的势能差. 因为两点间势能的差为定值，因此原图上 $s \to t$ 的最短路与新图上 $s \to t$ 的最短路相对应.

到这里我们的正确性证明已经解决了一半——我们证明了重新标注边权后图上的最短路径仍然是原来的最短路径．接下来我们需要证明新图中所有边的边权非负，因为在非负权图上，Dijkstra 算法能够保证得出正确的结果．

根据三角形不等式，图上任意一边 (u,v) 上两点满足：$h_v \le h_u + w(u,v)$．这条边重新标记后的边权为 $w'(u,v) = w(u,v) + h_u - h_v \ge 0$．这样我们证明了新图上的边权均非负．

这样，我们就证明了 Johnson 算法的正确性．

不同方法的比较

最短路算法	Floyd	Bellman–Ford	Dijkstra	Johnson
最短路类型	每对结点之间的最短路	单源最短路	单源最短路	每对结点之间的最短路
作用于	任意图	任意图	非负权图	任意图
能否检测负环?	能	能	不能	能
时间复杂度	$O(N^3)$	$O(NM)$	$O(M \log M)$	$O(NM \log M)$

注：表中的 Dijkstra 算法在计算复杂度时均用 `priority_queue` 实现．

输出方案

开一个 `pre` 数组，在更新距离的时候记录下来后面的点是如何转移过去的，算法结束前再递归地输出路径即可．

比如 Floyd 就要记录 `pre[i][j] = k;`，Bellman–Ford 和 Dijkstra 一般记录 `pre[v] = u`．

一些特殊情形

- 边权只由 0 和 1 组成的图上最短路：0-1 BFS→11.5@p2616；
- 允许至多 k 次改变路径成本等操作的最短路问题：分层图最短路→11.12@p2804．

参考资料与注释

[1] fstqwq 在知乎上的回答 

11.9.2 差分约束

Authors: Ir1d, Anguei, hsfzLZH1

定义

差分约束系统 是一种特殊的 n 元一次不等式组，它包含 n 个变量 $x_1, x_2, ..., x_n$ 以及 m 个约束条件，每个约束条件是由两个其中的变量做差构成的，形如 $x_i - x_j \le c_k$，其中 $1 \le i, j \le n, i \ne j, 1 \le k \le m$ 并且 c_k 是常数（可以是非负数，也可以是负数）．我们要解决的问题是：求一组解 $x_1 = a_1, x_2 = a_2, ..., x_n = a_n$，使得所有的约束条件得到满足，否则判断出无解．

差分约束系统中的每个约束条件 $x_i - x_j \le c_k$ 都可以变形成 $x_i \le x_j + c_k$，这与单源最短路中的三角形不等式 $dist[y] \le dist[x] + z$ 非常相似．因此，我们可以把每个变量 x_i 看做图中的一个结点，对于每个约束条件 $x_i - x_j \le c_k$，从结点 j 向结点 i 连一条长度为 c_k 的有向边．

注意到，如果 $\{a_1, a_2, ..., a_n\}$ 是该差分约束系统的一组解，那么对于任意的常数 d，$\{a_1 + d, a_2 + d, ..., a_n + d\}$ 显然也是该差分约束系统的一组解，因为这样做差后 d 刚好被消掉．

过程

设 $dist[0] = 0$ 并向每一个点连一条权重为 0 边，跑单源最短路，若图中存在负环，则给定的差分约束系统无解，否则，$x_i = dist[i]$ 为该差分约束系统的一组解．

性质

一般使用 Bellman–Ford 或队列优化的 Bellman–Ford（俗称 SPFA，在某些随机图跑得很快）判断图中是否存在负环，最坏时间复杂度为 $O(nm)$.

常用变形技巧

例题 luogu P1993 小 K 的农场[1]

题目大意：求解差分约束系统，有 m 条约束条件，每条都为形如 $x_a - x_b \geq c_k$，$x_a - x_b \leq c_k$ 或 $x_a = x_b$ 的形式，判断该差分约束系统有没有解.

题意	转化	连边
$x_a - x_b \geq c$	$x_b - x_a \leq -c$	`add(a, b, -c);`
$x_a - x_b \leq c$	$x_a - x_b \leq c$	`add(b, a, c);`
$x_a = x_b$	$x_a - x_b \leq 0,\ x_b - x_a \leq 0$	`add(b, a, 0), add(a, b, 0);`

跑判断负环，如果不存在负环，输出 `Yes`，否则输出 `No`.

参考代码

```cpp
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;

struct edge {
  int v, w, next;
} e[40005];

int head[10005], vis[10005], tot[10005], cnt;
long long ans, dist[10005];
queue<int> q;

void addedge(int u, int v, int w) {  // 加边
  e[++cnt].v = v;
  e[cnt].w = w;
  e[cnt].next = head[u];
  head[u] = cnt;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n, m;
  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int op, x, y, z;
    cin >> op;
    if (op == 1) {
      cin >> x >> y >> z;
      addedge(y, x, z);
    } else if (op == 2) {
      cin >> x >> y >> z;
      addedge(x, y, -z);
    } else {
```

```
      cin >> x >> y;
      addedge(x, y, 0);
      addedge(y, x, 0);
    }
  }
  for (int i = 1; i <= n; i++) addedge(0, i, 0);
  memset(dist, -0x3f, sizeof(dist));
  dist[0] = 0;
  vis[0] = 1;
  q.push(0);
  while (!q.empty()) {  // 判负环，看上面的
    int cur = q.front();
    q.pop();
    vis[cur] = 0;
    for (int i = head[cur]; i; i = e[i].next)
      if (dist[cur] + e[i].w > dist[e[i].v]) {
        dist[e[i].v] = dist[cur] + e[i].w;
        if (!vis[e[i].v]) {
          vis[e[i].v] = 1;
          q.push(e[i].v);
          tot[e[i].v]++;
          if (tot[e[i].v] >= n) {
            cout << "No\n";
            return 0;
          }
        }
      }
  }
  cout << "Yes\n";
  return 0;
}
```

例题 P4926[1007] 倍杀测量者[2]

不考虑二分等其他的东西，这里只论述差分系统 $\frac{x_i}{x_j} \le c_k$ 的求解方法.

对每个 x_i, x_j 和 c_k 取一个 log 就可以把乘法变成加法运算，即 $\log x_i - \log x_j \le \log c_k$，这样就可以用差分约束解决了.

Bellman–Ford 判负环代码实现

下面是用 Bellman–Ford 算法判断图中是否存在负环的代码实现，请在调用前先保证图是连通的.

实现

- C++

```
bool Bellman_Ford() {
  for (int i = 0; i < n; i++) {
    bool jud = false;
    for (int j = 1; j <= n; j++)
      for (int k = h[j]; ~k; k = nxt[k])
        if (dist[j] > dist[p[k]] + w[k])
          dist[j] = dist[p[k]] + w[k], jud = true;
```

```
    if (!jud) break;
  }
  for (int i = 1; i <= n; i++)
    for (int j = h[i]; ~j; j = nxt[j])
      if (dist[i] > dist[p[j]] + w[j]) return false;
  return true;
}
```

- Python

```
def Bellman_Ford():
    for i in range(0, n):
        jud = False
        for j in range(1, n + 1):
            while ~k:
                k = h[j]
                if dist[j] > dist[p[k]] + w[k]:
                    dist[j] = dist[p[k]] + w[k]
                    jud = True
                k = nxt[k]
        if jud == False:
            break
    for i in range(1, n + 1):
        while ~j:
            j = h[i]
            if dist[i] > dist[p[j]] + w[j]:
                return False
            j = nxt[j]
    return True
```

习题

Usaco2006 Dec Wormholes 虫洞[3]

「SCOI2011」糖果[4]

POJ 1364 King[5]

POJ 2983 Is the Information Reliable?[6]

外部参考资料

[1] luogu P1993 小 K 的农场

[2] P4926[1007] 倍杀测量者

[3] Usaco2006 Dec Wormholes 虫洞

[4] 「SCOI2011」糖果

[5] POJ 1364 King

[6] POJ 2983 Is the Information Reliable?

11.9.3 k 短路

前置知识：Dijkstra 算法→11.9.1@p2735、A* 算法→6.6@p667、可持久化可并堆→10.17.6@p2439

问题描述

给定一个有 n 个结点，m 条边的有向图，求从 s 到 t 的所有不同路径中的第 k 短路径的长度.

「路径」

本文所指的「路径」允许经过同一条边或同一个结点多次，因此严格的名称应为「途径 →11.2@p2598」而非「路径」. 本文讨论的问题严格地说也是 **第 k 短途径**（k shortest walk）问题. 但是，本文依据习惯仍然采用「路径」这一名称，而对于不自交的路径则称为「简单路径」.

A* 算法

A* 算法是一个搜索算法. 它为每个当前状态 x 都设置了一个估价函数 $f(x)=g(x)+h(x)$，其中 $g(x)$ 为从初始状态到达当前状态的实际代价，$h(x)$ 为从当前状态到达目标状态的最佳路径的估计代价. 搜索时，每次取出 $f(x)$ 最优的状态 x，扩展其所有后继状态. 可以用 **优先队列** 来维护这个值.

在求解 k 短路问题时，令 $h(x)$ 为从当前结点到达目标结点 t 的最短路径长度. 可以通过在反向图上对结点 t 跑单源最短路预处理出每个结点的这个值. 对于每个状态需要记录两个值，即当前到达的结点 x 和已经走过的距离 $g(x)$，将这种状态记为 $(x,g(x))$. 开始时，将初始状态 $(s,0)$ 加入优先队列. 每次取出估价函数 $f(x)=g(x)+h(x)$ 最小的一个状态，枚举该状态所在结点 x 的所有出边，将对应的后继状态加入优先队列. 当某个结点第 k 次出队时，对应状态的 $g(x)$ 就是从起始结点 s 到该结点的第 k 短路的长度.

这一搜索过程可以优化. 由于只需要求出从起始结点到目标结点的第 k 短路，所以当某个结点已经出队 k 次后，该结点后续出队的状态都可以不扩展其后继状态. 这些状态不会影响到最后的答案. 这是因为之前 k 次取出该结点时，已经形成了到达该结点的 k 条合法路径，足以构造到达目标结点的前 k 条最短路.

由于每个结点至多扩展 k 次，每条边至多会加入优先队列 k 次，所以算法的时间复杂度是 $O(km\log km)$ 的，空间复杂度是 $O(km)$ 的. 相较于直接搜索，A* 算法针对目标结点 t 进行了剪枝，但这仅仅改良了常数，而非渐近复杂度. 本节所述算法虽然复杂度并不优秀，但是可以在相同的复杂度内求出从起始结点 s 到（以 t 为根的最短路树中）每个结点的前 k 短路.

实现

模板题 Library Checker - K-Shortest Walk[1] 参考实现

```cpp
// Submission: https://judge.yosupo.jp/submission/311622
#include <iostream>
#include <queue>
#include <vector>

constexpr long long inf = 0x3f3f3f3f3f3f3f3f;

struct Edge {
  int u, v, c;

  Edge(int u, int v, int c) : u(u), v(v), c(c) {}
};

int n, m, s, t, k;
std::vector<std::vector<int>> gr, ig;  // Graph and inverse graph.
std::vector<Edge> edges;               // Edges.
std::priority_queue<std::pair<long long, int>,
                    std::vector<std::pair<long long, int>>, std::greater<>>
    pq;

std::vector<long long> dist_t;
```

```
22
23 // Calculate distances to the destination node t. (Dijkstra)
24 void calc_distances_to_t() {
25   dist_t.assign(n, inf);
26   dist_t[t] = 0;
27   pq.emplace(dist_t[t], t);
28   while (!pq.empty()) {
29     long long di = pq.top().first;
30     int cur = pq.top().second;
31     pq.pop();
32     if (di > dist_t[cur]) continue;
33     for (auto e : ig[cur]) {
34       int nxt = edges[e].u;
35       auto di_nxt = di + edges[e].c;
36       if (di_nxt < dist_t[nxt]) {
37         dist_t[nxt] = di_nxt;
38         pq.emplace(di_nxt, nxt);
39       }
40     }
41   }
42 }
43
44 std::vector<long long> ans;
45
46 // Find the k shortest walk. (A* search)
47 // Complexity: O(k * m * log(k * m)).
48 void find_k_shortest_walks() {
49   ans.assign(k, -1);
50   std::vector<int> cnt(n);
51   pq.emplace(dist_t[s], s);
52   while (!pq.empty()) {
53     long long cost = pq.top().first;
54     int cur = pq.top().second;
55     pq.pop();
56     // Skip unreachable nodes.
57     if (cost >= inf) continue;
58     if (cur == t) ans[cnt[t]] = cost;
59     ++cnt[cur];
60     // Terminate when the destination has been visited k times.
61     if (cnt[t] >= k) break;
62     // Expand the same node at most k times.
63     if (cnt[cur] > k) continue;
64     for (auto e : gr[cur]) {
65       int nxt = edges[e].v;
66       auto cost_nxt = cost - dist_t[cur] + edges[e].c + dist_t[nxt];
67       pq.emplace(cost_nxt, nxt);
68     }
69   }
70 }
71
72 int main() {
73   // Input.
```

```
  std::cin >> n >> m >> s >> t >> k;
  gr.resize(n);
  ig.resize(n);
  edges.reserve(m);
  for (int i = 0; i < m; ++i) {
    int u, v, c;
    std::cin >> u >> v >> c;
    edges.emplace_back(u, v, c);
    gr[u].push_back(i);
    ig[v].push_back(i);
  }
  // Calculate.
  calc_distances_to_t();
  find_k_shortest_walks();
  // Output.
  for (auto x : ans) std::cout << x << '\n';
  return 0;
}

```

可持久化可并堆做法

前述算法实际上求出了到达所有结点的 k 短路．如果仅仅是想要求得到达给定目标结点 t 的 k 短路，实际上可以做得更快．本节提供了一种基于可持久化可并堆的 $O(m \log m + k \log k)$ 的做法．

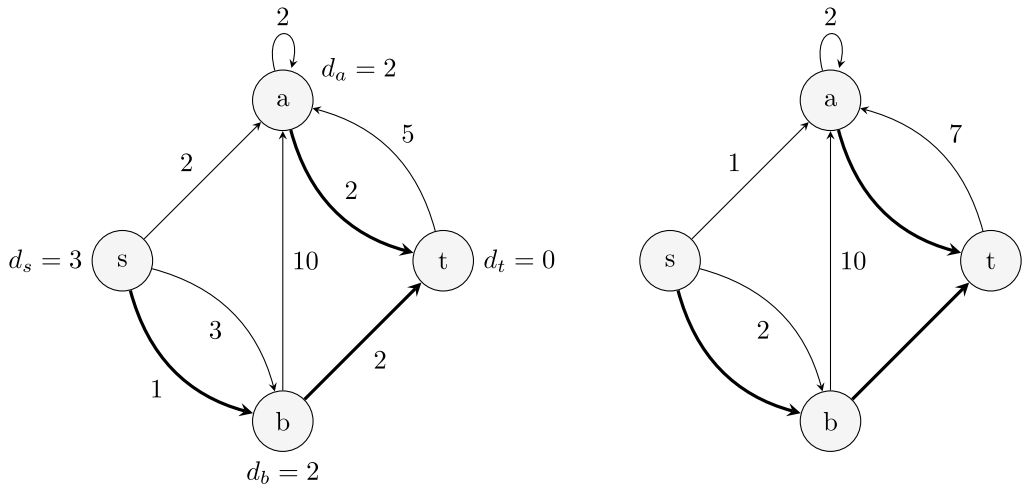
最短路树与偏离边

前述算法的瓶颈在于只有到达目标结点 t 时才会更新答案．但是，不同路径之间可能相差并不大．例如，次短路区别于最短路，可能仅仅是在一条边处多绕了一个结点，而路径的其他部分都是相同的；前述算法却可能需要重复搜索一遍这些相同的边才能找到次短路．由于只有绕路部分才是关键的，所以，要得到前 k 条最短的路径，只需要考虑代价最小的 k 种绕路方式即可．这就引出了最短路树的概念．

在反向图上从目标结点 t 开始跑单源最短路，记录每个结点 x 到 t 的最短路长度 $h(x)$，并记录从结点 x 开始的最短路经过的第一条边 f_x；如果有多个最优的选择，取 Dijkstra 算法松弛时记录的那一条即可．所有这些边 f_x 及其端点就构成一棵树，且从树上的每个结点 x 到根结点 t 的简单路径都是 x 到 t 的一条最短路径．这就是 **最短路树** T．

求得最短路树 T 后，就可以计算每条不在 T 上的边会多绕多少路．对于边 $e = (u, v) \notin T$，边权为 w，可以定义一条新的边，仍然从 u 指向 v，且代价为 $\Delta(e) = w + h(v) - h(u)$．本文形象地称这些权值为 $\Delta(e)$ 的边为 **偏离边**（sidetrack），权值 $\Delta(e)$ 则称为偏离成本．由于 $h(u) \le w + h(v)$，偏离成本总是非负的．如果一条边的端点并非全部在最短路树 T 里，它就不会影响到达结点 t 的 k 短路的计算，可以直接将它删掉．

下图左侧是有向图 G，右侧是它对应的最短路树 T（粗边）和相应的偏离边（细边）：



设一条从 s 到 t 的路径经过的边序列为 P，去掉 P 中与 T 的交集得到 P'．那么，将 P' 中的边顺次排列，它相邻的两条边 $e_1 = (u_1, v_1)$ 和 $e_2 = (u_2, v_2)$ 一定满足

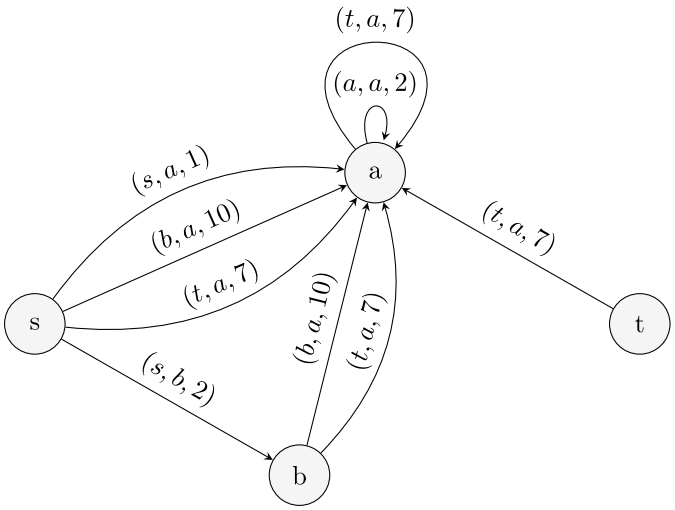
- 条件 (*)：后者的起点 u_2 是前者的终点 v_1 在最短路树 T 的祖先（包括其自身）．对于第一条边，约定「前者的终点」为 s．

这是因为对应的原始路径 P 中，v_1 和 u_2 之间连接了若干条 T 中的树边．反过来，对于一个满足条件 (*) 的边序列 P'，一定存在唯一一条图 G 中的路径 P 与之对应．这是因为 v_1 和 u_2 在最短路树 T 上的简单路径是唯一的．这样就说明，原图中的任意路径 P 与满足条件 (*) 的偏离边序列 P' 一一对应．而且，路径 P 的长度就等于最短路长度 $h(s)$ 与这些偏离成本的和：

$$h(s) + \sum_{e \in P'} \Delta(e).$$

这些讨论说明，寻找 k 短路的任务转化为寻找成本第 k 小且满足条件 (*) 的偏离边序列 P' 的任务．

为处理条件 (*)，与其每次查询时在最短路树上寻找祖先，不如直接将每个结点的偏离边集合下传到最短路树上的子孙结点．这相当于建了下面这样的图 G'：



在这个图上，条件 (*) 就转化为要求 P' 中的边首尾相接，也就是说，P' 是图 G' 中的一条路径．问题进一步转化为在这个图中寻找从 s 出发的长度第 k 小的 **到达任意结点的** 路径．相对于原始的 k 短路问题，此处不再要求路径一定要结束在目标结点 t．

转化后的问题很容易解决．直接从起始结点 s 处出发，用优先队列按路径长度从小到大扩展（同一结点可多次出队）．每次从优先队列中取出一个结点时，就相当于找到了一条图 G' 中的路径，也就对应着图 G 中一条到达目标结点 t 的路径．

可持久化可并堆优化

算法思路已经明晰．但是，朴素实现这一算法的复杂度过高．由于图 G' 中，单个结点处边的规模可能是 $\Theta(m)$ 的，所以每次扩展结点时，都可能需要将规模为 $\Theta(m)$ 的边集压入优先队列．实际上，没有必要将所有边都压入优先队列：很多情况下，压入队列的这些边，只有最短的那些可能会在后续计算中弹出队列．也就是说，完全可以将单个结点处的整个边集作为一个存储单元压入优先队列，每次只要能够快速访问边集中的最短边即可．

这启发我们使用小根堆来存储单个结点处的边集．在该优先队列中，只需要存储这些堆，它们的成本就是堆顶边对应的路径成本．每次弹出队首时，都需要一并从队首的堆中弹出堆顶边．然后，既要将弹出堆顶后的堆压回优先队列，也需要将堆顶边终点处的偏离边集合对应的堆压入优先队列．

使用堆来存储边集也解决了沿最短路树下传边集的问题．因为下传边集相当于需要将当前结点的边集合并到它的子结点，所以，堆还需要支持合并操作；合并到子结点的同时，还不能破坏当前结点处的边集，所以，堆还需要支持可持久化．这正是可持久化可并堆．

由此，就得到算法的完整过程：

1. 从目标结点 t 出发，跑单源最短路，求出最短路树．
2. 为最短路树上的每个结点都构建对应的偏离边集合，存储到可持久化可并堆里．
3. 沿着最短路树的边，从目标结点 t 开始自上而下（如按 BFS 序），将每个结点处的堆都合并到子结点的堆里．
4. 记录最短路长度 $h(s)$ 为第一个答案，再从起始结点 s 出发，将该处的堆压入优先队列．
5. 弹出队首的堆，记录答案，再将弹出堆顶后的堆压回优先队列，并将堆顶边的终点处的堆压入优先队列．

一般采用左偏树或随机堆实现可持久化可并堆．此时，最后一步还可以继续优化．这些堆的内部结构都是二叉树．弹出堆顶后，原本是要合并左右两个子结点，再将合并后的堆顶压入优先队列的；但是，本算法中，可以不执行合并操作，直接将两个子结点对应的堆分别压入优先队列．这样就省去了单次合并的 $O(\log m)$ 的复杂度．由于每次弹出队首堆后，至多会将三个新的堆压入优先队列，所以，优先队列的大小是 $O(k)$ 的．这样，单次查询的时间复杂度就降低到 $O(\log k)$．总查询复杂度就是 $O(k \log k)$ 的．

由于构建最短路树和构建可持久化可并堆的复杂度都是 $O(m \log m)$ 的，所以，算法的总时间复杂度为 $O(m \log m + k \log k)$ 的．

实现

模板题 Library Checker - K-Shortest Walk[2] 参考实现

```cpp
// Submission: https://judge.yosupo.jp/submission/311623
#include <algorithm>
#include <ctime>
#include <iostream>
#include <queue>
#include <random>
#include <vector>

std::mt19937_64
rng(static_cast<std::mt19937_64::result_type>(time(nullptr)));

```

```cpp
constexpr long long inf = 0x3f3f3f3f3f3f3f3f;

// Persistent Randomized Heap.
struct PersistentRandomizedHeap {
  static constexpr int N = 1e7;
  int id;
  std::vector<int> rt, lc, rc, to;
  std::vector<long long> va;

  int new_node(long long cost, int _to) {
    ++id;
    va[id] = cost;
    to[id] = _to;
    return id;
  }

  int copy_node(int x) {
    ++id;
    lc[id] = lc[x];
    rc[id] = rc[x];
    va[id] = va[x];
    to[id] = to[x];
    return id;
  }

  int meld(int x, int y, std::mt19937_64::result_type rand) {
    if (!x || !y) return x | y;
    if (va[x] > va[y]) std::swap(x, y);
    x = copy_node(x);
    if (rand & 1) std::swap(lc[x], rc[x]);
    rc[x] = meld(rc[x], y, rand >> 1);
    return x;
  }

  PersistentRandomizedHeap() : id(0), rt(N), lc(N), rc(N), to(N), va(N) {}

  void insert(int i, long long cost, int _to) {
    rt[i] = meld(rt[i], new_node(cost, _to), rng());
  }

  void merge(int i, int j) { rt[i] = meld(rt[i], rt[j], rng()); }

} heaps;

struct Edge {
  int u, v, c;

  Edge(int u, int v, int c) : u(u), v(v), c(c) {}
};

int n, m, s, t, k;
std::vector<std::vector<int>> gr, ig;
```

```
std::vector<Edge> edges;
std::priority_queue<std::pair<long long, int>,
                    std::vector<std::pair<long long, int>>, std::greater<>>
    pq;

std::vector<long long> dist_t;
std::vector<int> out;

// Calculate distances to the destination node t. (Dijkstra)
// Record the optimal outgoing edges which form the shortest path tree T.
void calc_distances_to_t() {
  dist_t.assign(n, inf);
  dist_t[t] = 0;
  out.assign(n, -1);
  pq.emplace(dist_t[t], t);
  while (!pq.empty()) {
    long long di = pq.top().first;
    int cur = pq.top().second;
    pq.pop();
    if (di > dist_t[cur]) continue;
    for (auto e : ig[cur]) {
      int nxt = edges[e].u;
      auto di_nxt = di + edges[e].c;
      if (di_nxt < dist_t[nxt]) {
        dist_t[nxt] = di_nxt;
        out[nxt] = e;
        pq.emplace(di_nxt, nxt);
      }
    }
  }
}

// Construct sidetracks and propagate them through tree T.
void build_sidetracks() {
  // Insert all valid sidetracks into heaps.
  for (int i = 0; i < m; ++i) {
    auto edge = edges[i];
    if (out[edge.u] != i && dist_t[edge.u] < inf && dist_t[edge.v] < inf) {
      heaps.insert(edge.u, edge.c + dist_t[edge.v] - dist_t[edge.u],
edge.v);
    }
  }
  // Propagate sidetracks down the shortest path tree.
  std::queue<int> q;
  q.push(t);
  while (!q.empty()) {
    auto cur = q.front();
    q.pop();
    for (auto e : ig[cur]) {
      auto nxt = edges[e].u;
      if (out[nxt] == e) {
        heaps.merge(nxt, cur);
```

```
        q.push(nxt);
      }
    }
  }
}

std::vector<long long> ans;

// Insert a non-empty heap into the priority queue.
// Total cost is the heap top value adjusted by accumulated cost.
void insert(int x, long long cost) {
  if (x) pq.emplace(cost + heaps.va[x], x);
}

// Find the k shortest paths in the sidetrack graph. (Dijkstra)
// These correspond to the k shortest walks in the original graph.
void find_k_shortest_walks() {
  int cnt = 0;
  ans.assign(k, -1);
  if (dist_t[s] >= inf) return;
  ans[cnt++] = dist_t[s];
  insert(heaps.rt[s], dist_t[s]);
  while (!pq.empty() && cnt < k) {
    auto cost = pq.top().first;
    int cur = pq.top().second;
    pq.pop();
    ans[cnt++] = cost;
    insert(heaps.lc[cur], cost - heaps.va[cur]);
    insert(heaps.rc[cur], cost - heaps.va[cur]);
    insert(heaps.rt[heaps.to[cur]], cost);
  }
}

int main() {
  // Input.
  std::cin >> n >> m >> s >> t >> k;
  gr.resize(n);
  ig.resize(n);
  edges.reserve(m);
  for (int i = 0; i < m; ++i) {
    int u, v, c;
    std::cin >> u >> v >> c;
    edges.emplace_back(u, v, c);
    gr[u].push_back(i);
    ig[v].push_back(i);
  }
  // Calculate.
  calc_distances_to_t();
  build_sidetracks();
  find_k_shortest_walks();
  // Output.
  for (auto x : ans) std::cout << x << '\n';
```

```
166   return 0;
167 }
168
```

习题

- 「SDOI2010」魔法猪学院[3]

参考资料与注释

- [Tutorial] k shortest paths and Eppstein's algorithm by meooow - Codeforces[4]

外部参考资料

[1] Library Checker - K-Shortest Walk 

[2] Library Checker - K-Shortest Walk 

[3] 「SDOI2010」魔法猪学院 

[4] [Tutorial] k shortest paths and Eppstein's algorithm by meooow - Codeforces 

11.9.4 同余最短路

当出现形如「给定 n 个整数，求这 n 个整数能拼凑出多少的其他整数（n 个整数可以重复取）」，以及「给定 n 个整数，求这 n 个整数不能拼凑出的最小（最大）的整数」，或者「至少要拼几次才能拼出模 K 余 p 的数」的问题时可以使用同余最短路的方法.

同余最短路利用同余来构造一些状态，可以达到优化空间复杂度的目的.

类比 差分约束→11.9.2@p2747 方法，利用同余构造的这些状态可以看作单源最短路中的点. 同余最短路的状态转移通常是这样的 $f(i+y)=f(i)+y$，类似单源最短路中 $f(v)=f(u)+edge(u,v)$.

例题

例题一

> **P3403 跳楼机**[1]
>
> 题目大意：给定 x,y,z,h，对于 $k\in[1,h]$，有多少个 k 能够满足 $ax+by+cz=k$.（$0\le a,b,c,1\le x,y,z\le 10^5$，$h\le 2^{63}-1$）

不妨假设 $x<y<z$.

令 d_i 为只通过 **操作 2** 和 **操作 3**，需满足 $p \bmod x=i$ 能够达到的最低楼层 p，即 **操作 2** 和 **操作 3** 操作后能得到的模 x 下与 i 同余的最小数，用来计算该同余类满足条件的数个数.

可以得到两个状态：

- $i \xrightarrow{y} (i+y) \bmod x$
- $i \xrightarrow{z} (i+z) \bmod x$

注意通常选取一组 a_i 中最小的那个数对它取模，也就是此处的 x，这样可以尽量减小空间复杂度（剩余系最小）.

那么实际上相当于执行了最短路中的建边操作：

```
add(i, (i+y) % x, y)
```

```
add(i, (i+z) % x, z)
```

接下来只需要求出 $d_0, d_1, d_2, ..., d_{x-1}$，只需要跑一次最短路就可求出相应的 d_i.

基于最短路的实现

```cpp
#include <iostream>
#include <queue>

using namespace std;
using ll = long long;
constexpr int MAXN = 100010;
constexpr ll linf = (1ull << 63) - 1;

ll h, x, y, z;
ll head[MAXN << 1], tot;
ll dis[MAXN], vis[MAXN];
queue<int> q;

struct edge {
  ll to, next, w;
} e[MAXN << 1];

void add(ll u, ll v, ll w) {
  e[++tot] = edge{v, head[u], w};
  head[u] = tot;
}

void spfa() {  // spfa算法，可看最短路部分
  dis[0] = 0;
  vis[0] = 1;
  q.push(0);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    vis[u] = 0;
    for (int i = head[u]; i; i = e[i].next) {
      int v = e[i].to, w = e[i].w;
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        if (!vis[v]) {
          q.push(v);
          vis[v] = 1;
        }
      }
    }
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> h;
  cin >> x >> y >> z;
  if (x == 1 || y == 1 || z == 1) {
    cout << h << '\n';
    return 0;
```

```
  }
  --h;
  for (int i = 0; i < x; i++) {
    add(i, (i + z) % x, z);
    add(i, (i + y) % x, y);
    dis[i] = linf;
  }
  spfa();
  ll ans = 0;
  for (int i = 0; i < x; i++) {
    if (h >= dis[i]) ans += (h - dis[i]) / x + 1;
  }
  cout << ans << '\n';
  return 0;
}

```

但是事实上也不需要进行正常的最短路求解，注意到有两个特殊的性质：

首先，只有两种边权，且对于每一条路径，由于加法交换律，走两种边权的顺序是无影响的．因此可以考虑做两次最短路，每次只建出一类边权的边；

其次，对于只有一类边权的图，每个点 u 都有一个入度（来自 $(u-y) \bmod x$）和一个出度（来自 $(u+y) \bmod x$），因此整个图必然由若干个环构成．并且可以证明共有 $\gcd(x,y)$ 个等长的环．

证明

设 $d=\gcd(x,y)$，设 $x=da, y=db$，有 $\gcd(a,b)=1$．

考虑从 u 出发走 k 步，到达 $(u+ky) \bmod x$．若成环，则 $ky \equiv 0 \pmod{x}$，即有 $kb \equiv 0 \pmod{a}$．

由于 $\gcd(a,b)=1$，最小的 $k=a$，即环长为 $a=\dfrac{x}{d}$．由于是从任意点开始，故每个可能的环长相等，环的数量为 d．

并且，边权为正，绕环两圈后，一定不能继续松弛．直接循环更新一遍就行了．这样处理就不会受限于最短路的复杂度，可以做到 $O(x)$．

与差分约束问题相同，当存在一组解 $\{a_1,a_2,\cdots,a_n\}$ 时，$\{a_1+d,a_2+d,\cdots,a_n+d\}$ 同样为一组解，因此在该题让 $i=1$ 作为源点，此时源点处的 $dis_1=1$ 在已知范围内最小，因此得到的也是一组最小的解．

答案即为：

$$\sum_{i=0}^{x-1}\left(\frac{h-d_i}{x}+1\right)$$

加 1 是由于 d_i 所在楼层也算一次．

代码实现上注意到 h 的范围是 $h \le 2^{63}-1$，所以在求解最短路之前 d_i 的初始值应至少设为 2^{63}，这超过了 C++ 中 `long long` 的最大值．所以可以使用 `unsigned long long` 或者先把 $h \leftarrow h-1$，然后把最低楼层设为 0 层，其他代码无异．

基于环优化的实现

```
#include <algorithm>
#include <climits>
```

```cpp
#include <cstring>
#include <iostream>
#include <vector>

using ll = long long;

constexpr int maxn = 1e5 + 100;

ll d[maxn];
bool vis[maxn];
ll ans;

int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }

void upd(int step, int M) {
  int D = gcd(step, M);
  int len = M / D;

  for (int st = 0; st < D; st++) {
    if (vis[st]) continue;
    std::vector<int> v;
    int u = st;
    for (int i = 0; i < len; i++) {
      v.push_back(u);
      vis[u] = true;
      u = (u + step) % M;
    }
    for (int r = 0; r < 2; r++) {
      for (int i = 0; i < len; i++) {
        int las = v[i];
        int now = v[(i + 1) % len];
        if (d[las] != LLONG_MAX) {
          // 如果d[las]=LLONG_MAX，d[las]+step会越界，需要特判
          d[now] = std::min(d[now], d[las] + step);
        }
      }
    }
  }
}

int main() {
  ll h;
  std::cin >> h;
  int x[3];
  for (int i = 0; i < 3; i++) {
    std::cin >> x[i];
  }
  std::sort(x, x + 3);
  int M = x[0];

  for (int i = 0; i < M; i++) {
    d[i] = LLONG_MAX;
```

```
    // 本题的h达到了ll的上界，如果使用ll的话必需把初值置为LLONG_MAX
  }
  d[0] = 0;

  upd(x[1], M);
  memset(vis, 0, sizeof(vis));
  upd(x[2], M);

  ll H = h - 1;
  ans = 0;
  for (int i = 0; i < M; i++) {
    if (d[i] <= H && d[i] != LLONG_MAX) {
      ans += (H - d[i]) / M + 1;
    }
  }
  printf("%lld\n", ans);
  return 0;
}

```

例题二

ARC084B Small Multiple[2]

题目大意：给定 n，求 n 的倍数中，数位和最小的那一个的数位和.（$1 \leq n \leq 10^5$）

本题可以使用循环卷积优化完全背包在 $O(n \log^2 n)$ 的时间内解决，但我们希望得到线性的算法.

观察到任意一个正整数都可以从 1 开始，按照某种顺序执行乘 10、加 1 的操作，最终得到，而其中加 1 操作的次数就是这个数的数位和. 这提示我们使用最短路.

对于所有 $0 \leq k \leq n-1$，从 k 向 $10k$ 连边权为 0 的边；从 k 向 $k+1$ 连边权为 1 的边.（点的编号均在模 n 意义下）

每个 n 的倍数在这个图中都对应了 1 号点到 0 号点的一条路径，求出 1 到 0 的最短路即可. 某些路径不合法（如连续走了 10 条边权为 1 的边），但这些路径产生的答案一定不优，不影响答案.

时间复杂度 $O(n)$.

习题

洛谷 P3403 跳楼机[3]

洛谷 P2662 牛场围栏[4]

[国家集训队] 墨墨的等式[5]

「NOIP2018」货币系统[6]

AGC057D - Sum Avoidance[7]

「THUPC 2023 初赛」背包[8]

外部参考资料

[1] P3403 跳楼机 [2] ARC084B Small Multiple [3] 洛谷 P3403 跳楼机

[4] 洛谷 P2662 牛场围栏 
[5] [国家集训队] 墨墨的等式 
[6] 「NOIP2018」货币系统 
[7] AGC057D - Sum Avoidance 
[8] 「THUPC 2023 初赛」背包 

11.10 生成树问题

11.10.1 最小生成树

Authors: Chrogeek, Enter-tainer, HeRaNO, Ir1d, Marcythm, ShadowsEpic, StudyingFather, Xeonacid, bear-good, billchenchina, diauweb, diauweb, greyqz, kawa-yoiko, ouuan, partychicken, sshwy, stevebraveman, zhouyuyang2002, renbaoshuo, Hszzzx, y-kx-b, toprise

定义

在阅读下列内容之前，请务必阅读 图论相关概念→11.2@p2598 与 树基础→11.6.1@p2621 部分，并了解以下定义：

1. 生成子图
2. 生成树

我们定义无向连通图的 **最小生成树**（Minimum Spanning Tree，MST）为边权和最小的生成树.

注意：只有连通图才有生成树，而对于非连通图，只存在生成森林.

Kruskal 算法

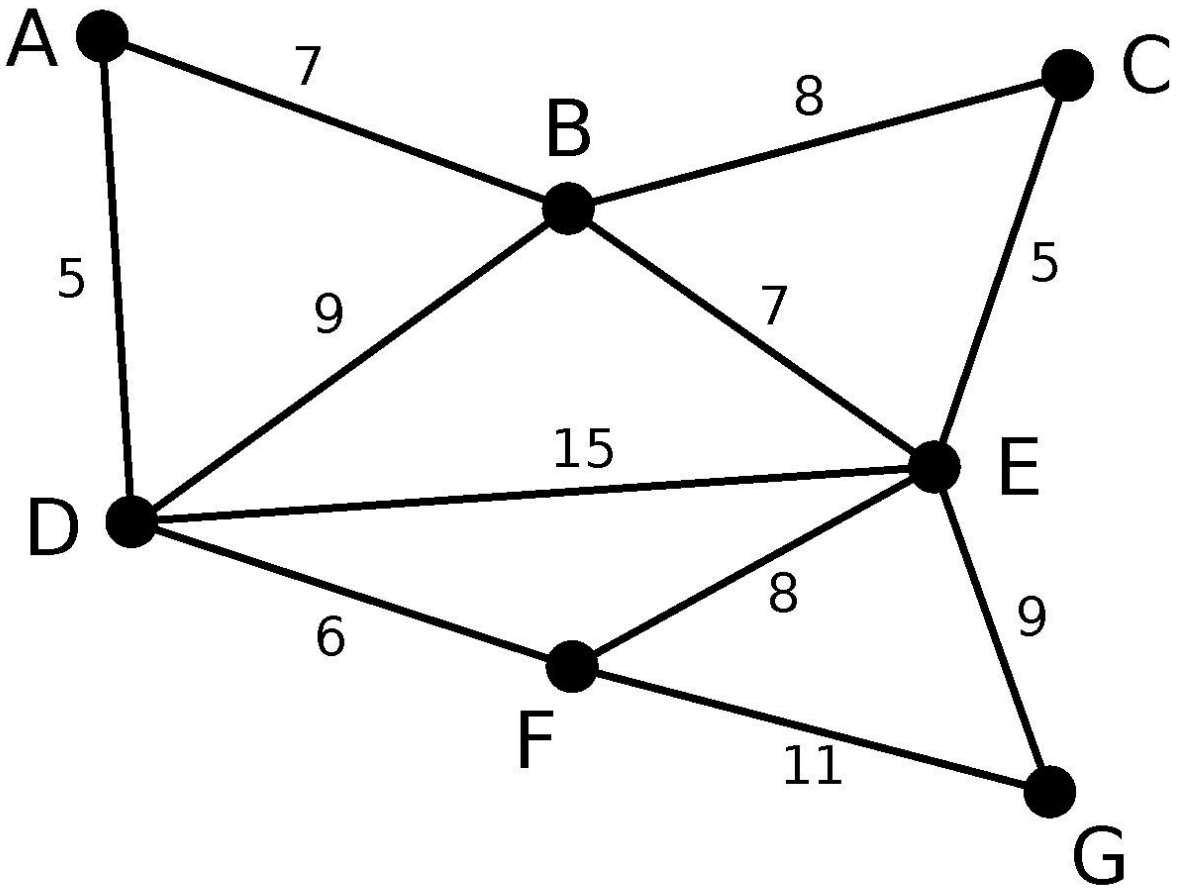
Kruskal 算法是一种常见并且好写的最小生成树算法，由 Kruskal 发明. 该算法的基本思想是从小到大加入边，是个贪心算法.

前置知识

并查集→10.6.1@p1999、贪心→5.6@p563、图的存储→11.3@p2605.

实现

图示：



伪代码：

```
1  Input. The edges of the graph e, where each element in e is (u, v, w)
   denoting that there is an edge between u and v weighted w.
2  Output. The edges of the MST of the input graph.
3  Method.
4  result ← ∅
5  sort e into nondecreasing order by weight w
6  for each (u, v, w) in the sorted e
7        if u and v are not connected in the union-find set
8              connect u and v in the union-find set
9              result ← result ⋃ {(u, v, w)}
10 return result
```

算法虽简单，但需要相应的数据结构来支持……具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树.

抽象一点地说，维护一堆 **集合**，查询两个元素是否属于同一集合，合并两个集合.

其中，查询两点是否连通和连接两点可以使用并查集维护.

如果使用 $O(m \log m)$ 的排序算法，并且使用 $O(m\alpha(m,n))$ 或 $O(m \log n)$ 的并查集，就可以得到时间复杂度为 $O(m \log m)$ 的 Kruskal 算法.

证明

思路很简单，为了造出一棵最小生成树，我们从最小边权的边开始，按边权从小到大依次加入，如果某次加边产生了环，就扔掉这条边，直到加入了 $n-1$ 条边，即形成了一棵树.

证明：使用归纳法，证明任何时候 K 算法选择的边集都被某棵 MST 所包含.

基础：对于算法刚开始时，显然成立（最小生成树存在）.

归纳：假设某时刻成立，当前边集为 F，令 T 为这棵 MST，考虑下一条加入的边 e.

如果 e 属于 T，那么成立.

否则，$T+e$ 一定存在一个环，考虑这个环上不属于 F 的另一条边 f（至少存在一条）.

首先，f 的权值一定不会比 e 小，不然 f 会在 e 之前被选取.

然后，f 的权值一定不会比 e 大，不然 $T+e-f$ 就是一棵比 T 还优的生成树了.

所以，$T+e-f$ 包含了 F，并且也是一棵最小生成树，归纳成立.

例题

洛谷 P1195 口袋的天空[1]

有 n 朵云，你要将它们连成 k 个棉花糖，将 X_i 云朵和 Y_i 连接起来需要 L_i 的代价，求最小代价.

例题代码

- C++

```cpp
#include <algorithm>
#include <iostream>
using namespace std;

int fa[1010];  // 定义父亲
int n, m, k;

struct edge {
  int u, v, w;
};

int l;
edge g[10010];

void add(int u, int v, int w) {
  l++;
  g[l].u = u;
  g[l].v = v;
  g[l].w = w;
}

// 标准并查集
int findroot(int x) { return fa[x] == x ? x : fa[x] =
findroot(fa[x]); }

void Merge(int x, int y) {
  x = findroot(x);
  y = findroot(y);
  fa[x] = y;
}

bool cmp(edge A, edge B) { return A.w < B.w; }
```

```cpp

// Kruskal 算法
void kruskal() {
  int tot = 0;  // 存已选了的边数
  int ans = 0;  // 存总的代价
  for (int i = 1; i <= m; i++) {
    int xr = findroot(g[i].u), yr = findroot(g[i].v);
    if (xr != yr) {         // 如果父亲不一样
      Merge(xr, yr);        // 合并
      tot++;                // 边数增加
      ans += g[i].w;        // 代价增加
      if (tot == n - k) {  // 检查选的边数是否满足 k 个棉花糖
        cout << ans << '\n';
        return;
      }
    }
  }
  cout << "No Answer\n";  // 无法连成
}

int main() {
  cin >> n >> m >> k;
  if (n == k) {  // 特判边界情况
    cout << "0\n";
    return 0;
  }
  for (int i = 1; i <= n; i++) {  // 初始化
    fa[i] = i;
  }
  for (int i = 1; i <= m; i++) {
    int u, v, w;
    cin >> u >> v >> w;
    add(u, v, w);  // 添加边
  }
  sort(g + 1, g + m + 1, cmp);  // 先按边权排序
  kruskal();
  return 0;
}

```

- Python

```python
class Edge:
    def __init__(self, u, v, w):
        self.u = u
        self.v = v
        self.w = w


fa = [0] * 1010  # 定义父亲
g = []


```

```python
def add(u, v, w):
    g.append(Edge(u, v, w))


# 标准并查集
def findroot(x):
    if fa[x] == x:
        return x
    fa[x] = findroot(fa[x])
    return fa[x]


def Merge(x, y):
    x = findroot(x)
    y = findroot(y)
    fa[x] = y


# Kruskal 算法
def kruskal():
    tot = 0  # 存已选了的边数
    ans = 0  # 存总的代价
    for e in g:
        x = findroot(e.u)
        y = findroot(e.v)
        if x != y:  # 如果父亲不一样
            fa[x] = y  # 合并
            tot += 1  # 边数增加
            ans += e.w  # 代价增加
            if tot == n - k:  # 检查选的边数是否满足 k 个棉花糖
                print(ans)
                return
    print("No Answer")  # 无法连成


if __name__ == "__main__":
    n, m, k = map(int, input().split())
    if n == k:  # 特判边界情况
        print("0")
        exit()
    for i in range(1, n + 1):  # 初始化
        fa[i] = i
    for i in range(1, m + 1):
        u, v, w = map(int, input().split())
        add(u, v, w)  # 添加边
    g.sort(key=lambda edge: edge.w)  # 先按边权排序
    kruskal()

```

- Java

```java
import java.util.Arrays;
import java.util.Scanner;

```

```
3
4  class Edge {
5      int u;
6      int v;
7      int w;
8
9      Edge(int u, int v, int w) {
10         this.u = u;
11         this.v = v;
12         this.w = w;
13     }
14 }
15
16 public class Main {
17     static int[] parent = new int[1010];  // 定义父亲
18     static int m, n, k;  // n 表示点的数量， m 表示边的数量，k 表示需要的棉花糖个数
19
20     static Edge[] edges = new Edge[10010];
21     static int l;
22
23     static void addEdge(int u, int v, int w) {
24         edges[++l] = new Edge(u, v, w);
25     }
26
27     // 标准并查集
28     static int findroot(int x) {
29         if (parent[x] != x) {
30             parent[x] = findroot(parent[x]);
31         }
32         return parent[x];
33     }
34
35     static void Merge(int x, int y) {
36         x = findroot(x);
37         y = findroot(y);
38         parent[x] = y;
39     }
40
41     static boolean cmp(Edge A, Edge B) {
42         return A.w < B.w;
43     }
44
45     // Kruskal 算法
46     static void kruskal() {
47         int tot = 0;  // 存已选了的边数
48         int ans = 0;  // 存总的代价
49
50         for (int i = 1; i <= m; i++) {
51             int xr = findroot(edges[i].u);
52             int yr = findroot(edges[i].v);
53             if (xr != yr) {   // 如果父亲不一样
```

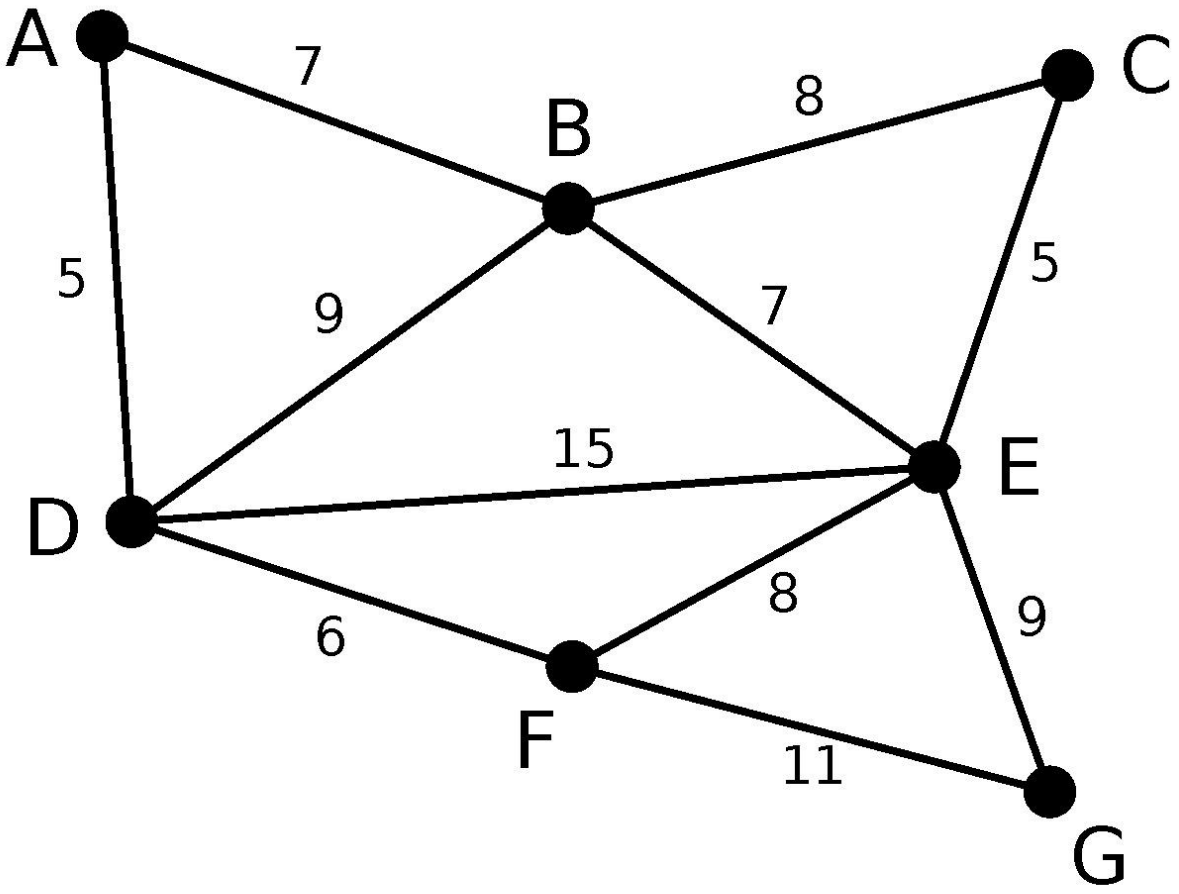
```
54                  Merge(xr, yr); // 合并
55                  tot++; // 边数增加
56                  ans += edges[i].w; // 代价增加
57                  if (tot == n - k) {  // 检查选的边数是否满足 k 个棉花糖
58                      System.out.println(ans);
59                      return;
60                  }
61              }
62          }
63          System.out.println("No Answer");  // 无法连成
64      }
65
66      public static void main(String[] args) {
67          Scanner scanner = new Scanner(System.in);
68          n = scanner.nextInt();
69          m = scanner.nextInt();
70          k = scanner.nextInt();
71
72          if (n == k) { // 特判边界情况
73              System.out.println("0");
74              return;
75          }
76
77          // 初始化
78          for (int i = 1; i <= n; i++) {
79              parent[i] = i;
80          }
81          for (int i = 1; i <= m; i++) {
82              int u = scanner.nextInt();
83              int v = scanner.nextInt();
84              int w = scanner.nextInt();
85              addEdge(u, v, w);  // 添加边
86          }
87          Arrays.sort(edges, 1, m + 1, (a, b) -> Integer.compare(a.w,
    b.w));  // 先按边权排序
88          kruskal();
89          scanner.close();
90      }
91  }
92
```

Prim 算法

Prim 算法是另一种常见并且好写的最小生成树算法. 该算法的基本思想是从一个结点开始, 不断加点(而不是 Kruskal 算法的加边).

实现

图示:



具体来说，每次要选择距离最小的一个结点，以及用新的边更新其他结点的距离.

其实跟 Dijkstra 算法一样，每次找到距离最小的一个点，可以暴力找也可以用堆维护.

堆优化的方式类似 Dijkstra 的堆优化，但如果使用二叉堆等不支持 $O(1)$ decrease-key 的堆，复杂度就不优于 Kruskal，常数也比 Kruskal 大. 所以，一般情况下都使用 Kruskal 算法，在稠密图尤其是完全图上，暴力 Prim 的复杂度比 Kruskal 优，但 **不一定** 实际跑得更快.

暴力：$O(n^2 + m)$.

二叉堆：$O((n + m) \log n)$.

Fib 堆：$O(n \log n + m)$.

伪代码：

Input. The nodes of the graph V; the function $g(u, v)$ which means the weight of the edge (u, v); the function $adj(v)$ which means the nodes adjacent to v.
Output. The sum of weights of the MST of the input graph.
Method.
$result \leftarrow 0$
choose an arbitrary node in V to be the $root$
$dis(root) \leftarrow 0$
for each node $v \in (V - \{root\})$
    $dis(v) \leftarrow \infty$
$rest \leftarrow V$
while $rest \neq \emptyset$
    $cur \leftarrow$ the node with the minimum dis in $rest$
    $result \leftarrow result + dis(cur)$
    $rest \leftarrow rest - \{cur\}$
    for each node $v \in adj(cur)$
        $dis(v) \leftarrow \min(dis(v), g(cur, v))$
return $result$

注意：上述代码只是求出了最小生成树的权值，如果要输出方案还需要记录每个点的 dis 代表的是哪条边.

代码实现

```cpp
// 使用二叉堆优化的 Prim 算法.
#include <cstring>
#include <iostream>
#include <queue>
using namespace std;
constexpr int N = 5050, M = 2e5 + 10;

struct E {
  int v, w, x;
} e[M * 2];

int n, m, h[N], cnte;

void adde(int u, int v, int w) { e[++cnte] = E{v, w, h[u]}, h[u] = cnte; }

struct S {
  int u, d;
};

bool operator<(const S &x, const S &y) { return x.d > y.d; }

priority_queue<S> q;
int dis[N];
bool vis[N];

int res = 0, cnt = 0;

void Prim() {
  memset(dis, 0x3f, sizeof(dis));
  dis[1] = 0;
  q.push({1, 0});
```

```cpp
  while (!q.empty()) {
    if (cnt >= n) break;
    int u = q.top().u, d = q.top().d;
    q.pop();
    if (vis[u]) continue;
    vis[u] = true;
    ++cnt;
    res += d;
    for (int i = h[u]; i; i = e[i].x) {
      int v = e[i].v, w = e[i].w;
      if (w < dis[v]) {
        dis[v] = w, q.push({v, w});
      }
    }
  }
}

int main() {
  cin >> n >> m;
  for (int i = 1, u, v, w; i <= m; ++i) {
    cin >> u >> v >> w, adde(u, v, w), adde(v, u, w);
  }
  Prim();
  if (cnt == n)
    cout << res;
  else
    cout << "No MST.";
  return 0;
}
```

证明

从任意一个结点开始，将结点分成两类：已加入的，未加入的.

每次从未加入的结点中，找一个与已加入的结点之间边权最小值最小的结点.

然后将这个结点加入，并连上那条边权最小的边.

重复 $n-1$ 次即可.

证明：还是说明在每一步，都存在一棵最小生成树包含已选边集.

基础：只有一个结点的时候，显然成立.

归纳：如果某一步成立，当前边集为 F，属于 T 这棵 MST，接下来要加入边 e.

如果 e 属于 T，那么成立.

否则考虑 $T+e$ 中环上另一条可以加入当前边集的边 f.

首先，f 的权值一定不小于 e 的权值，否则就会选择 f 而不是 e 了.

然后，f 的权值一定不大于 e 的权值，否则 $T+e-f$ 就是一棵更小的生成树了.

因此，e 和 f 的权值相等，$T+e-f$ 也是一棵最小生成树，且包含了 F.

Boruvka 算法

接下来介绍另一种求解最小生成树的算法——Boruvka 算法．该算法的思想是前两种算法的结合．它可以用于求解无向图的最小生成森林．（无向连通图就是最小生成树．）

在边具有较多特殊性质的问题中，Boruvka 算法具有优势．例如 CF888G[2] 的完全图问题．

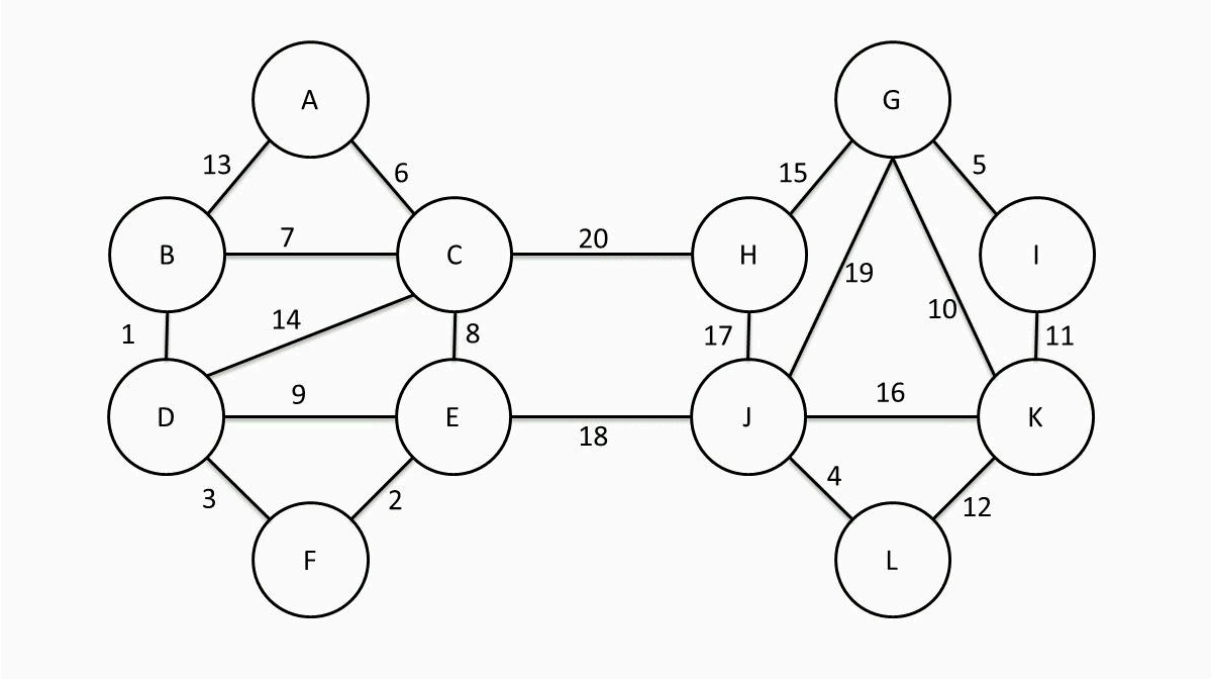
为了描述该算法，我们需要引入一些定义：

1. 定义 E' 为我们当前找到的最小生成森林的边．在算法执行过程中，我们逐步向 E' 加边，定义 **连通块** 表示一个点集 $V' \subseteq V$，且这个点集中的任意两个点 u，v 在 E' 中的边构成的子图上是连通的（互相可达）．
2. 定义一个连通块的 **最小边** 为它连向其它连通块的边中权值最小的那一条．

初始时，$E' = \emptyset$，每个点各自是一个连通块：

1. 计算每个点分别属于哪个连通块．将每个连通块都设为「没有最小边」．
2. 遍历每条边 (u, v)，如果 u 和 v 不在同一个连通块，就用这条边的边权分别更新 u 和 v 所在连通块的最小边．
3. 如果所有连通块都没有最小边，退出程序，此时的 E' 就是原图最小生成森林的边集．否则，将每个有最小边的连通块的最小边加入 E'，返回第一步．

下面通过一张动态图来举一个例子（图源自 维基百科[3]）：



当原图连通时，每次迭代连通块数量至少减半，算法只会迭代不超过 $O(\log V)$ 次，而原图不连通时相当于多个子问题，因此算法复杂度是 $O(E \log V)$ 的．给出算法的伪代码：（修改自 维基百科[4]）

```
1  Input. A graph G whose edges have distinct weights.
2  Output. The minimum spanning forest of G.
3  Method.
4  Initialize a forest F to be a set of one-vertex trees
5  while True
6      Find the components of F and label each vertex of G by its component
7      Initialize the cheapest edge for each component to "None"
8      for each edge (u, v) of G
9          if u and v have different component labels
10             if (u, v) is cheaper than the cheapest edge for the component of u
11                 Set (u, v) as the cheapest edge for the component of u
12             if (u, v) is cheaper than the cheapest edge for the component of v
13                 Set (u, v) as the cheapest edge for the component of v
14     if all components'cheapest edges are "None"
15         return F
16     for each component whose cheapest edge is not "None"
17         Add its cheapest edge to F
```

需要注意边与边的比较通常需要第二关键字（例如按编号排序），以便当边权相同时分出边的大小.

习题

- 「HAOI2006」聪明的猴子[5]
- 「SCOI2005」繁忙的都市[6]

最小生成树的唯一性

考虑最小生成树的唯一性. 如果一条边 **不在最小生成树的边集中**，并且可以替换与其 **权值相同、并且在最小生成树边集** 的另一条边. 那么，这个最小生成树就是不唯一的.

对于 Kruskal 算法，只要计算为当前权值的边可以放几条，实际放了几条，如果这两个值不一样，那么就说明这几条边与之前的边产生了一个环（这个环中至少有两条当前权值的边，否则根据并查集，这条边是不能放的），即最小生成树不唯一.

寻找权值与当前边相同的边，我们只需要记录头尾指针，用单调队列即可在 $O(\alpha(m))$（m 为边数）的时间复杂度里优秀解决这个问题（基本与原算法时间相同）.

例题：POJ 1679[7]

```cpp
#include <algorithm>
#include <iostream>

struct Edge {
  int x, y, z;
};

int f[100001];
Edge a[100001];

int cmp(const Edge& a, const Edge& b) { return a.z < b.z; }

int find(int x) { return f[x] == x ? x : f[x] = find(f[x]); }

using std::cin;
using std::cout;

int main() {
```

```
  cin.tie(nullptr)->sync_with_stdio(false);
  int t;
  cin >> t;
  while (t--) {
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) f[i] = i;
    for (int i = 1; i <= m; i++) cin >> a[i].x >> a[i].y >> a[i].z;
    std::sort(a + 1, a + m + 1, cmp);  // 先排序
    int num = 0, ans = 0, tail = 0, sum1 = 0, sum2 = 0;
    bool flag = true;
    for (int i = 1; i <= m + 1; i++) {  // 再并查集加边
      if (i > tail) {
        if (sum1 != sum2) {
          flag = false;
          break;
        }
        sum1 = 0;
        for (int j = i; j <= m + 1; j++) {
          if (j > m || a[j].z != a[i].z) {
            tail = j - 1;
            break;
          }
          if (find(a[j].x) != find(a[j].y)) ++sum1;
        }
        sum2 = 0;
      }
      if (i > m) break;
      int x = find(a[i].x);
      int y = find(a[i].y);
      if (x != y && num != n - 1) {
        sum2++;
        num++;
        f[x] = f[y];
        ans += a[i].z;
      }
    }
    if (flag)
      cout << ans << '\n';
    else
      cout << "Not Unique!\n";
  }
  return 0;
}

```

次小生成树

非严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **大于等于** 最小生成树边权和的生成树

求解方法

- 求出无向图的最小生成树 T，设其权值和为 M
- 遍历每条未被选中的边 $e=(u,v,w)$，找到 T 中 u 到 v 路径上边权最大的一条边 $e'=(s,t,w')$，则在 T 中以 e 替换 e'，可得一棵权值和为 $M'=M+w-w'$ 的生成树 T'.
- 对所有替换得到的答案 M' 取最小值即可

如何求 u,v 路径上的边权最大值呢?

我们可以使用倍增来维护，预处理出每个节点的 2^i 级祖先及到达其 2^i 级祖先路径上最大的边权，这样在倍增求 LCA 的过程中可以直接求得.

严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **严格大于** 最小生成树边权和的生成树

求解方法

考虑刚才的非严格次小生成树求解过程，为什么求得的解是非严格的?

因为最小生成树保证生成树中 u 到 v 路径上的边权最大值一定 **不大于** 其他从 u 到 v 路径的边权最大值. 换言之，当我们用于替换的边的权值与原生成树中被替换边的权值相等时，得到的次小生成树是非严格的.

解决的办法很自然：我们维护到 2^i 级祖先路径上的最大边权的同时维护 **严格次大边权**，当用于替换的边的权值与原生成树中路径最大边权相等时，我们用严格次大值来替换即可.

这个过程可以用倍增求解，复杂度 $O(m\log m)$.

代码实现

```cpp
#include <algorithm>
#include <iostream>

constexpr int INF = 0x3fffffff;
constexpr long long INF64 = 0x3fffffffffffffffLL;

struct Edge {
  int u, v, val;

  bool operator<(const Edge &other) const { return val < other.val; }
};

Edge e[300010];
bool used[300010];

int n, m;
long long sum;

class Tr {
 private:
  struct Edge {
    int to, nxt, val;
  } e[600010];

  int cnt, head[100010];
```

```
26
27   int pnt[100010][22];
28   int dpth[100010];
29   // 到祖先的路径上边权最大的边
30   int maxx[100010][22];
31   // 到祖先的路径上边权次大的边，若不存在则为 -INF
32   int minn[100010][22];
33
34 public:
35   void addedge(int u, int v, int val) {
36     e[++cnt] = Edge{v, head[u], val};
37     head[u] = cnt;
38   }
39
40   void insedge(int u, int v, int val) {
41     addedge(u, v, val);
42     addedge(v, u, val);
43   }
44
45   void dfs(int now, int fa) {
46     dpth[now] = dpth[fa] + 1;
47     pnt[now][0] = fa;
48     minn[now][0] = -INF;
49     for (int i = 1; (1 << i) <= dpth[now]; i++) {
50       pnt[now][i] = pnt[pnt[now][i - 1]][i - 1];
51       int kk[4] = {maxx[now][i - 1], maxx[pnt[now][i - 1]][i - 1],
52                    minn[now][i - 1], minn[pnt[now][i - 1]][i - 1]};
53       // 从四个值中取得最大值
54       std::sort(kk, kk + 4);
55       maxx[now][i] = kk[3];
56       // 取得严格次大值
57       int ptr = 2;
58       while (ptr >= 0 && kk[ptr] == kk[3]) ptr--;
59       minn[now][i] = (ptr == -1 ? -INF : kk[ptr]);
60     }
61
62     for (int i = head[now]; i; i = e[i].nxt) {
63       if (e[i].to != fa) {
64         maxx[e[i].to][0] = e[i].val;
65         dfs(e[i].to, now);
66       }
67     }
68   }
69
70   int lca(int a, int b) {
71     if (dpth[a] < dpth[b]) std::swap(a, b);
72
73     for (int i = 21; i >= 0; i--)
74       if (dpth[pnt[a][i]] >= dpth[b]) a = pnt[a][i];
75
76     if (a == b) return a;
77
```

```
    for (int i = 21; i >= 0; i--) {
      if (pnt[a][i] != pnt[b][i]) {
        a = pnt[a][i];
        b = pnt[b][i];
      }
    }
    return pnt[a][0];
  }

  int query(int a, int b, int val) {
    int res = -INF;
    for (int i = 21; i >= 0; i--) {
      if (dpth[pnt[a][i]] >= dpth[b]) {
        if (val != maxx[a][i])
          res = std::max(res, maxx[a][i]);
        else
          res = std::max(res, minn[a][i]);
        a = pnt[a][i];
      }
    }
    return res;
  }
} tr;

int fa[100010];

int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

void Kruskal() {
  int tot = 0;
  std::sort(e + 1, e + m + 1);
  for (int i = 1; i <= n; i++) fa[i] = i;

  for (int i = 1; i <= m; i++) {
    int a = find(e[i].u);
    int b = find(e[i].v);
    if (a != b) {
      fa[a] = b;
      tot++;
      tr.insedge(e[i].u, e[i].v, e[i].val);
      sum += e[i].val;
      used[i] = true;
    }
    if (tot == n - 1) break;
  }
}

int main() {
  std::ios::sync_with_stdio(false);
  std::cin.tie(nullptr);

  std::cin >> n >> m;
```

```cpp
  for (int i = 1; i <= m; i++) {
    int u, v, val;
    std::cin >> u >> v >> val;
    e[i] = Edge{u, v, val};
  }

  Kruskal();
  long long ans = INF64;
  tr.dfs(1, 0);

  for (int i = 1; i <= m; i++) {
    if (!used[i]) {
      int _lca = tr.lca(e[i].u, e[i].v);
      // 找到路径上不等于 e[i].val 的最大边权
      long long tmpa = tr.query(e[i].u, _lca, e[i].val);
      long long tmpb = tr.query(e[i].v, _lca, e[i].val);
      // 这样的边可能不存在，只在这样的边存在时更新答案
      if (std::max(tmpa, tmpb) > -INF)
        ans = std::min(ans, sum - std::max(tmpa, tmpb) + e[i].val);
    }
  }
  // 次小生成树不存在时输出 -1
  std::cout << (ans == INF64 ? -1 : ans) << '\n';
  return 0;
}
```

瓶颈生成树

定义

无向图 G 的瓶颈生成树是这样的一个生成树，它的最大的边权值在 G 的所有生成树中最小.

性质

最小生成树是瓶颈生成树的充分不必要条件. 即最小生成树一定是瓶颈生成树, 而瓶颈生成树不一定是最小生成树.

关于最小生成树一定是瓶颈生成树这一命题, 可以运用反证法证明: 我们设最小生成树中的最大边权为 w, 如果最小生成树不是瓶颈生成树的话, 则瓶颈生成树的所有边权都小于 w, 我们只需删去原最小生成树中的最长边, 用瓶颈生成树中的一条边来连接删去边后形成的两棵树, 得到的新生成树一定比原最小生成树的权值和还要小，这样就产生了矛盾.

例题

POJ 2395 Out of Hay

给出 n 个农场和 m 条边，农场按 1 到 n 编号，现在有一人要从编号为 1 的农场出发到其他的农场去，求在这途中他最多需要携带的水的重量，注意他每到达一个农场，可以对水进行补给，且要使总共的路径长度最小. 题目要求的就是瓶颈树的最大边，可以通过求最小生成树来解决.

最小瓶颈路

定义

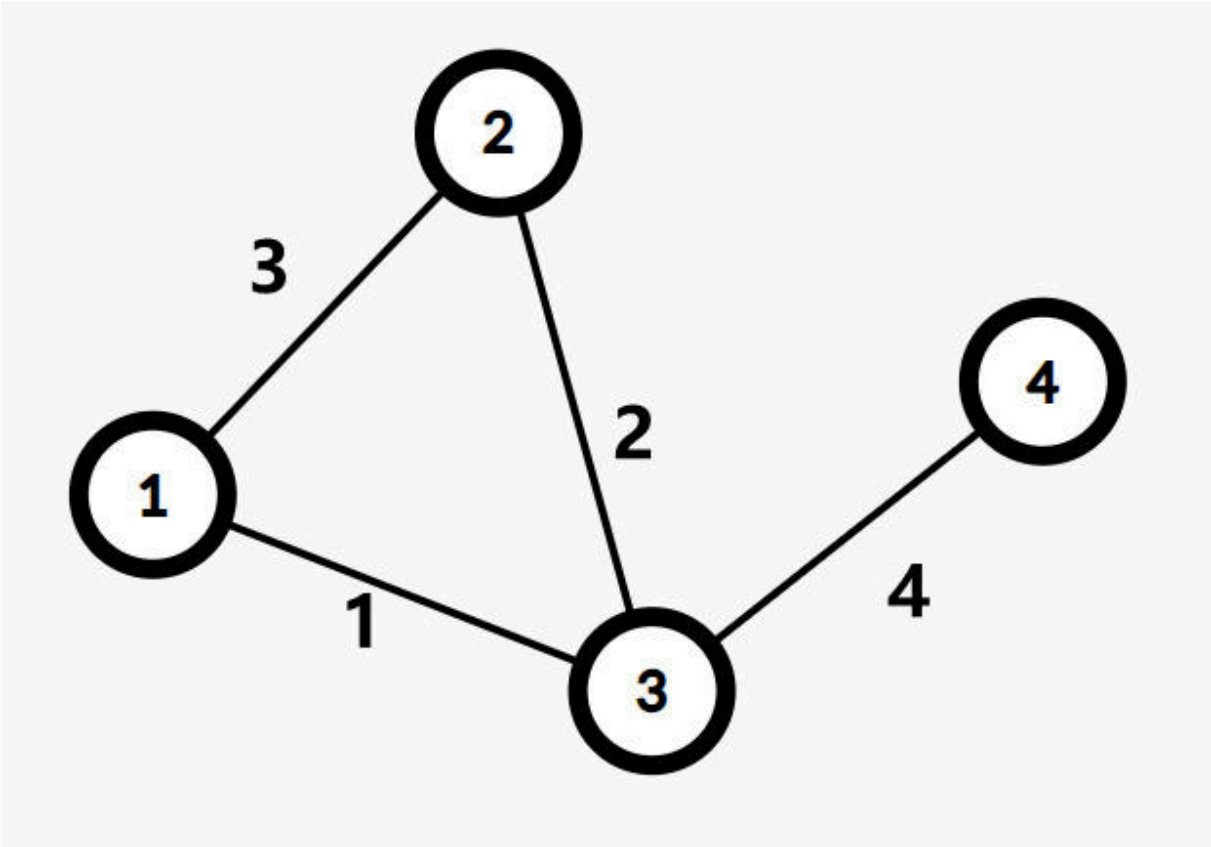
无向图 G 中 x 到 y 的最小瓶颈路是这样的一类简单路径，满足这条路径上的最大的边权在所有 x 到 y 的简单路径中是最小的.

性质

根据最小生成树定义，x 到 y 的最小瓶颈路上的最大边权等于最小生成树上 x 到 y 路径上的最大边权. 虽然最小生成树不唯一，但是每种最小生成树 x 到 y 路径的最大边权相同且为最小值. 也就是说，每种最小生成树上的 x 到 y 的路径均为最小瓶颈路.

但是，并不是所有最小瓶颈路都存在一棵最小生成树满足其为树上 x 到 y 的简单路径.

例如下图:



1 到 4 的最小瓶颈路显然有以下两条：1-2-3-4. 1-3-4.

但是，1-2 不会出现在任意一种最小生成树上.

应用

由于最小瓶颈路不唯一，一般情况下会询问最小瓶颈路上的最大边权.

也就是说，我们需要求最小生成树链上的 max.

倍增、树剖都可以解决，这里不再展开.

Kruskal 重构树

定义

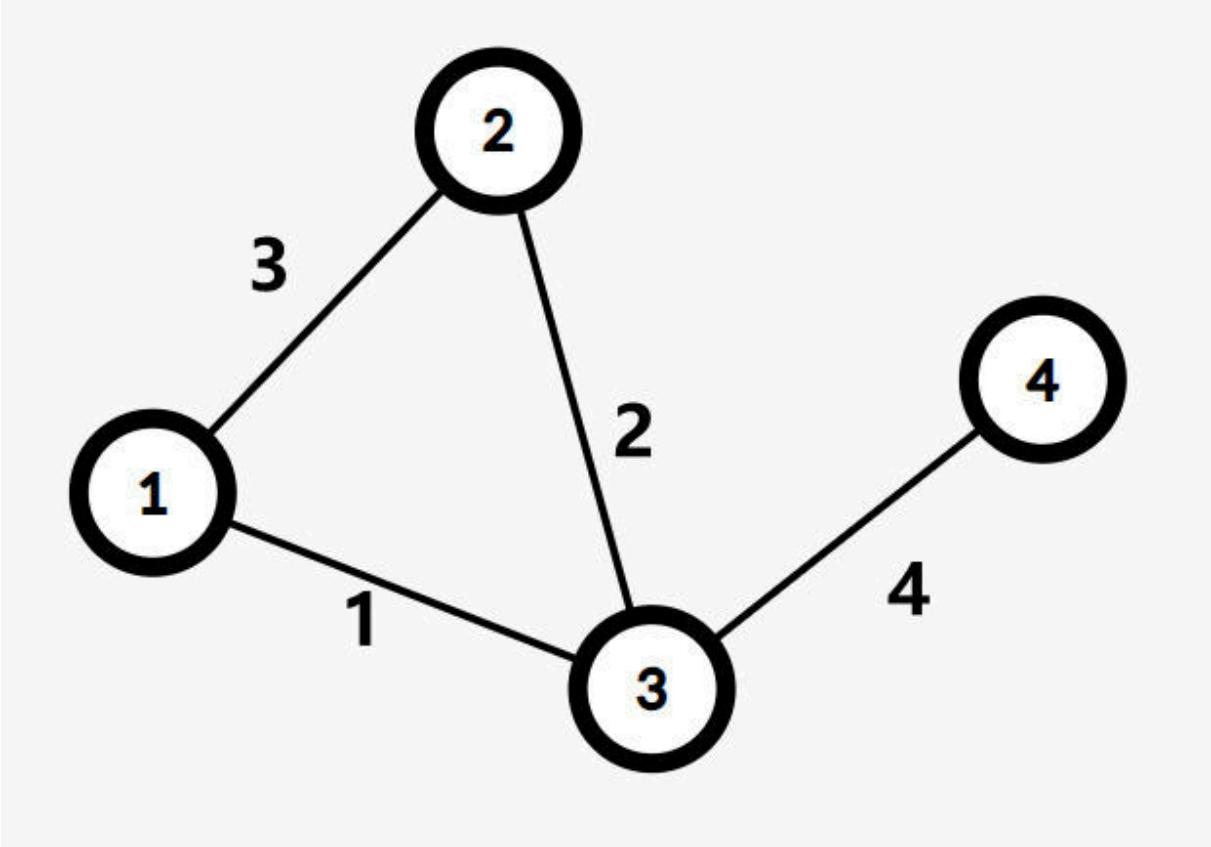
在跑 Kruskal 的过程中我们会从小到大加入若干条边. 现在我们仍然按照这个顺序.

首先新建 n 个集合，每个集合恰有一个节点，点权为 0.

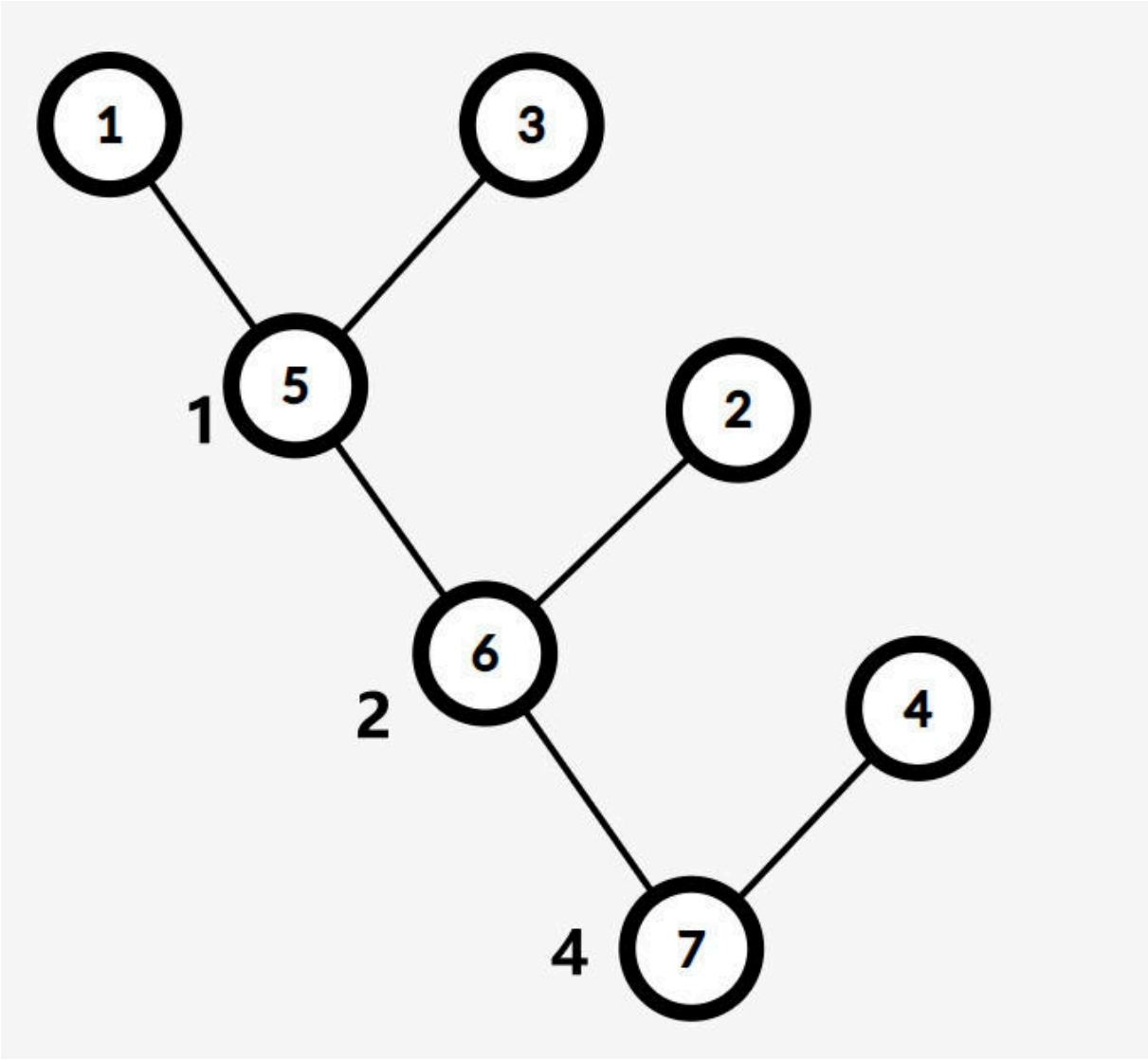
每一次加边会合并两个集合，我们可以新建一个点，点权为加入边的边权，同时将两个集合的根节点分别设为新建点的左儿子和右儿子．然后我们将两个集合和新建点合并成一个集合．将新建点设为根.

不难发现，在进行 $n-1$ 轮之后我们得到了一棵恰有 n 个叶子的二叉树，同时每个非叶子节点恰好有两个儿子．这棵树就叫 Kruskal 重构树.

举个例子：



这张图的 Kruskal 重构树如下：



性质

不难发现，原图中两个点之间的所有简单路径上最大边权的最小值 = 最小生成树上两个点之间的简单路径上的最大值 = Kruskal 重构树上两点之间的 LCA 的权值.

也就是说，到点 x 的简单路径上最大边权的最小值 $\leq val$ 的所有点 y 均在 Kruskal 重构树上的某一棵子树内，且恰好为该子树的所有叶子节点.

我们在 Kruskal 重构树上找到 x 到根的路径上权值 $\leq val$ 的最浅的节点. 显然这就是所有满足条件的节点所在的子树的根节点.

如果需要求原图中两个点之间的所有简单路径上最小边权的最大值，则在跑 Kruskal 的过程中按边权大到小的顺序加边.

「LOJ 137」最小瓶颈路 加强版[8]

```cpp
#include <algorithm>
#include <iostream>

using namespace std;

```

```cpp
constexpr int MAX_VAL_RANGE = 280010;

int n, m, log2Values[MAX_VAL_RANGE + 1];

namespace TR {
struct Edge {
  int to, nxt, val;
} e[400010];

int cnt, head[140010];

void addedge(int u, int v, int val = 0) {
  e[++cnt] = Edge{v, head[u], val};
  head[u] = cnt;
}

int val[140010];

namespace LCA {
int sec[280010], cnt;
int pos[140010];
int dpth[140010];

void dfs(int now, int fa) {
  dpth[now] = dpth[fa] + 1;
  sec[++cnt] = now;
  pos[now] = cnt;

  for (int i = head[now]; i; i = e[i].nxt) {
    if (fa != e[i].to) {
      dfs(e[i].to, now);
      sec[++cnt] = now;
    }
  }
}

int dp[280010][20];

void init() {
  dfs(2 * n - 1, 0);
  for (int i = 1; i <= 4 * n; i++) {
    dp[i][0] = sec[i];
  }
  for (int j = 1; j <= 19; j++) {
    for (int i = 1; i + (1 << j) - 1 <= 4 * n; i++) {
      dp[i][j] = dpth[dp[i][j - 1]] < dpth[dp[i + (1 << (j - 1))][j - 1]]
                     ? dp[i][j - 1]
                     : dp[i + (1 << (j - 1))][j - 1];
    }
  }
}

```

```
int lca(int x, int y) {
  int l = pos[x], r = pos[y];
  if (l > r) {
    swap(l, r);
  }
  int k = log2Values[r - l + 1];
  return dpth[dp[l][k]] < dpth[dp[r - (1 << k) + 1][k]]
             ? dp[l][k]
             : dp[r - (1 << k) + 1][k];
}
}  // namespace LCA
}  // namespace TR

using TR::addedge;

namespace GR {
struct Edge {
  int u, v, val;

  bool operator<(const Edge &other) const { return val < other.val; }
} e[100010];

int fa[140010];

int find(int x) { return fa[x] == 0 ? x : fa[x] = find(fa[x]); }

void kruskal() {  // 最小生成树
  int tot = 0, cnt = n;
  sort(e + 1, e + m + 1);
  for (int i = 1; i <= m; i++) {
    int fau = find(e[i].u), fav = find(e[i].v);
    if (fau != fav) {
      cnt++;
      fa[fau] = fa[fav] = cnt;
      addedge(fau, cnt);
      addedge(cnt, fau);
      addedge(fav, cnt);
      addedge(cnt, fav);
      TR::val[cnt] = e[i].val;
      tot++;
    }
    if (tot == n - 1) {
      break;
    }
  }
}
}  // namespace GR

int ans;
int A, B, C, P;

int rnd() { return A = (A * B + C) % P; }
```

```
110
111 void initLog2() {
112   for (int i = 2; i <= MAX_VAL_RANGE; i++) {
113     log2Values[i] = log2Values[i >> 1] + 1;
114   }
115 }
116
117 int main() {
118   initLog2();  // 预处理
119   cin >> n >> m;
120   for (int i = 1; i <= m; i++) {
121     int u, v, val;
122     cin >> u >> v >> val;
123     GR::e[i] = GR::Edge{u, v, val};
124   }
125   GR::kruskal();
126   TR::LCA::init();
127   int Q;
128   cin >> Q;
129   cin >> A >> B >> C >> P;
130
131   while (Q--) {
132     int u = rnd() % n + 1, v = rnd() % n + 1;
133     ans += TR::val[TR::LCA::lca(u, v)];
134     ans %= 1000000007;
135   }
136   cout << ans;
137   return 0;
138 }
```

NOI 2018 归程[9]

首先预处理出来每一个点到根节点的最短路.

我们构造出来根据海拔的最大生成树. 显然每次询问可以到达的节点是在最大生成树中和询问点的路径上最小边权 $> p$ 的节点.

根据 Kruskal 重构树的性质，这些节点满足均在一棵子树内同时为其所有叶子节点.

也就是说，我们只需要求出 Kruskal 重构树上每一棵子树叶子的权值 min 就可以支持子树询问.

询问的根节点可以使用 Kruskal 重构树上倍增的方式求出.

时间复杂度 $O((n+m+Q)\log n)$.

外部参考资料

[1] 洛谷 P1195 口袋的天空

[2] CF888G

[3] 维基百科

[4] 维基百科

[5] 「HAOI2006」聪明的猴子

[6] 「SCOI2005」繁忙的都市

[7] POJ 1679

[8] 「LOJ 137」最小瓶颈路 加强版

[9] NOI 2018 归程

11.10.2 最小树形图

定义

有向图上的最小生成树（Directed Minimum Spanning Tree）称为最小树形图.

常用的算法是朱刘算法（也称 Edmonds 算法），可以在 $O(nm)$ 时间内解决最小树形图问题.

过程

1. 对于每个点，选择指向它的边权最小的那条边.
2. 如果没有环，算法终止；否则进行缩环并更新其他点到环的距离.

实现

```
bool solve() {
  ans = 0;
  int u, v, root = 0;
  for (;;) {
    f(i, 0, n) in[i] = 1e100;
    f(i, 0, m) {
      u = e[i].s;
      v = e[i].t;
      if (u != v && e[i].w < in[v]) {
        in[v] = e[i].w;
        pre[v] = u;
      }
    }
    f(i, 0, m) if (i != root && in[i] > 1e50) return 0;
    int tn = 0;
    memset(id, -1, sizeof id);
    memset(vis, -1, sizeof vis);
    in[root] = 0;
    f(i, 0, n) {
      ans += in[i];
      v = i;
      while (vis[v] != i && id[v] == -1 && v != root) {
        vis[v] = i;
        v = pre[v];
      }
      if (v != root && id[v] == -1) {
        for (int u = pre[v]; u != v; u = pre[u]) id[u] = tn;
        id[v] = tn++;
      }
    }
    if (tn == 0) break;
    f(i, 0, n) if (id[i] == -1) id[i] = tn++;
    f(i, 0, m) {
      u = e[i].s;
      v = e[i].t;
      e[i].s = id[u];
      e[i].t = id[v];
      if (e[i].s != e[i].t) e[i].w -= in[v];
    }
    n = tn;
    root = id[root];
```

```
42    }
43    return ans;
44 }
```

Tarjan 的 DMST 算法

Tarjan 提出了一种能够在 $O(m + n \log n)$ 时间内解决最小树形图问题的算法.

这里的算法描述以及参考代码基于 Uri Zwick 教授的课堂讲义，更多的细节可以参考原文.

过程

Tarjan 的算法分为 **收缩** 与 **伸展** 两个过程. 接下来先介绍 **收缩** 的过程.

我们需要假设输入的图是满足强连通的，如果不满足那么就加入 $O(n)$ 条边使其满足，并且这些边的边权是无穷大的.

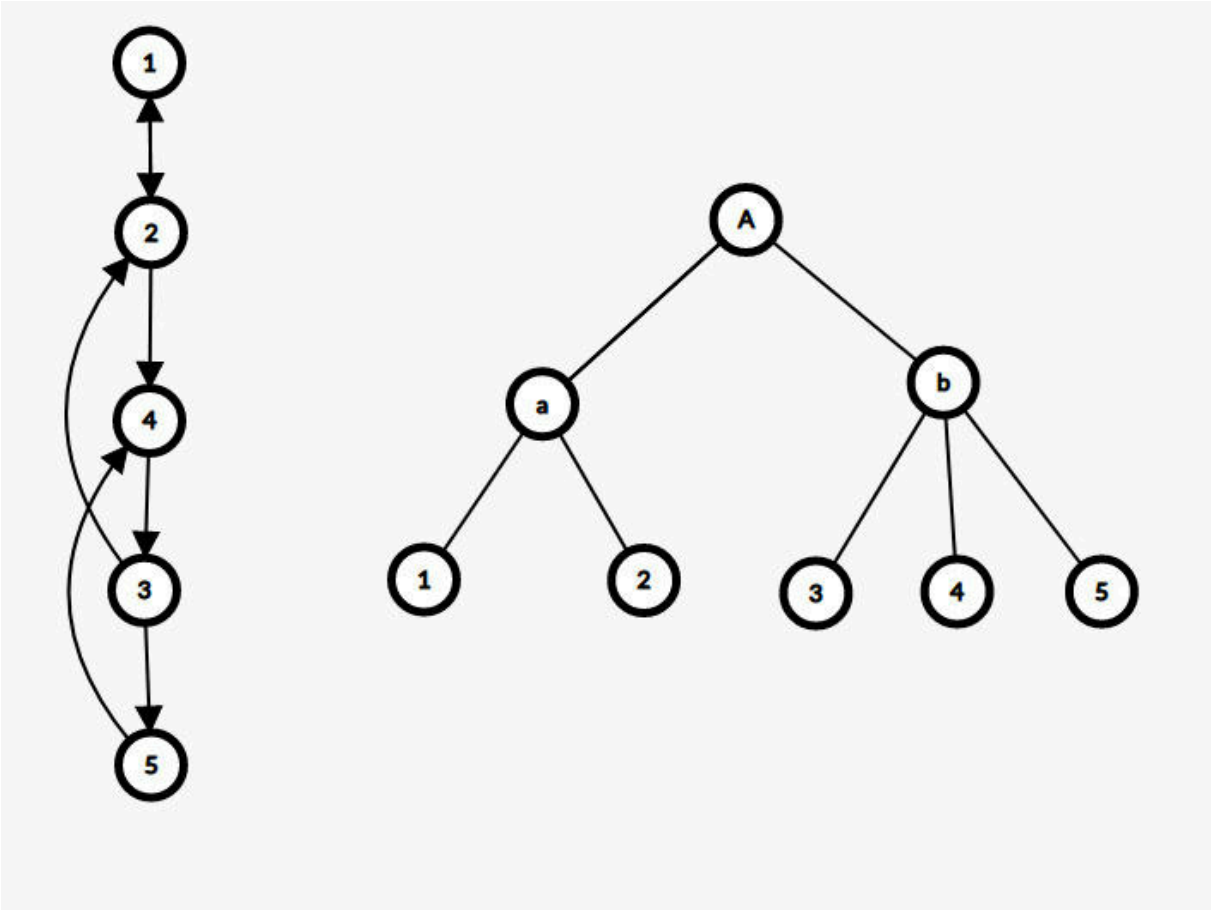
我们需要一个堆存储结点的入边编号，入边权值，结点总代价等相关信息，由于后续过程中会有堆的合并操作，这里采用 左偏树→10.7.4@p2035 与 并查集→10.6.1@p1999 实现. 算法的每一步都选择一个任意结点 v，需要保证 v 不是根节点，并且在堆中没有它的入边. 再将 v 的最小入边加入到堆中，如果新加入的这条边使堆中的边形成了环, 那么将构成环的那些结点收缩, 我们不妨将这些已经收缩的结点命名为 **超级结点**，再继续这个过程，如果所有的顶点都缩成了一个超级结点，那么收缩过程就结束了. 整个收缩过程结束后会得到一棵收缩树，之后将对它进行伸展操作.

堆中的边总是会形成一条路径 $v_0 \leftarrow v_1 \leftarrow \ldots \leftarrow v_k$，由于图是强连通的，这个路径必然存在，并且其中的 v_i 可能是最初的单一结点，也可能是压缩后的超级结点.

最初有 $v_o = a$，其中 a 是图中任意的一个结点，每一次选择一条最小入边 $v_k \leftarrow u$，如果 u 不是 $v_0, v_1, \ldots, v_k$ 中的一个结点，那么就将结点扩展到 $v_{k+1} = u$. 如果 u 是他们其中的一个结点 v_i，那么就找到了一个关于 $v_i \leftarrow \ldots \leftarrow v_k \leftarrow v_i$ 的环，再将他们收缩为一个超级结点 c.

向队列 P 中放入所有的结点或超级结点，并初始选择任意一节点 a，只要队列不为空，就进行以下步骤:

1. 选择 a 的最小入边，保证不存在自环，并找到另一头的结点 b. 如果结点 b 没有被记录过说明未形成环，令 $a \leftarrow b$，继续当前操作寻找环.
2. 如果 b 被记录过了，就说明出现了环. 总结点数加一，并将环上的所有结点重新编号，对堆进行合并, 以及结点/超级结点的总权值的更新. 更新权值操作就是将环上所有结点的入边都收集起来, 并减去环上入边的边权.



以图片为例，左边的强连通图在收缩后就形成了右边的一棵收缩树，其中 a 是结点 1 与结点 2 收缩后的超级结点，b 是结点 3，结点 4，结点 5 收缩后的超级结点，A 是两个超级结点 a 与 b 收缩后形成的.

伸展过程是相对简单的, 以原先要求的根节点 r 为起始点, 对 r 到收缩树的根上的每一个环进行伸展. 再以 r 的祖先结点 f_r 为起始点，将其到根的环展开，直到遍历完所有的结点.

实现

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
#include <vector>
using namespace std;

using ll = long long;
constexpr int MAXN = 102;
constexpr int INF = 0x3f3f3f3f;

struct UnionFind {
  int fa[MAXN << 1];

  UnionFind() { memset(fa, 0, sizeof(fa)); }

  void clear(int n) { memset(fa + 1, 0, sizeof(int) * n); }

  int find(int x) { return fa[x] ? fa[x] = find(fa[x]) : x; }

  int operator[](int x) { return find(x); }
```

```
};

struct Edge {
  int u, v, w, w0;
};

struct Heap {
  Edge *e;
  int rk, constant;
  Heap *lch, *rch;

  Heap(Edge *_e) : e(_e), rk(1), constant(0), lch(NULL), rch(NULL) {}

  void push() {
    if (lch) lch->constant += constant;
    if (rch) rch->constant += constant;
    e->w += constant;
    constant = 0;
  }
};

Heap *merge(Heap *x, Heap *y) {
  if (!x) return y;
  if (!y) return x;
  if (x->e->w + x->constant > y->e->w + y->constant) swap(x, y);
  x->push();
  x->rch = merge(x->rch, y);
  if (!x->lch || x->lch->rk < x->rch->rk) swap(x->lch, x->rch);
  if (x->rch)
    x->rk = x->rch->rk + 1;
  else
    x->rk = 1;
  return x;
}

Edge *extract(Heap *&x) {
  Edge *r = x->e;
  x->push();
  x = merge(x->lch, x->rch);
  return r;
}

vector<Edge> in[MAXN];
int n, m, fa[MAXN << 1], nxt[MAXN << 1];
Edge *ed[MAXN << 1];
Heap *Q[MAXN << 1];
UnionFind id;

void contract() {
  bool mark[MAXN << 1];
  // 将图上的每一个结点与其相连的那些结点进行记录.
  for (int i = 1; i <= n; i++) {
    queue<Heap *> q;
    for (int j = 0; j < in[i].size(); j++) q.push(new Heap(&in[i][j]));
```

```
    while (q.size() > 1) {
      Heap *u = q.front();
      q.pop();
      Heap *v = q.front();
      q.pop();
      q.push(merge(u, v));
    }
    Q[i] = q.front();
  }
  mark[1] = true;
  for (int a = 1, b = 1, p; Q[a]; b = a, mark[b] = true) {
    // 寻找最小入边以及其端点，保证无环.
    do {
      ed[a] = extract(Q[a]);
      a = id[ed[a]->u];
    } while (a == b && Q[a]);
    if (a == b) break;
    if (!mark[a]) continue;
    // 对发现的环进行收缩，以及环内的结点重新编号，总权值更新.
    for (a = b, n++; a != n; a = p) {
      id.fa[a] = fa[a] = n;
      if (Q[a]) Q[a]->constant -= ed[a]->w;
      Q[n] = merge(Q[n], Q[a]);
      p = id[ed[a]->u];
      nxt[p == n ? b : p] = a;
    }
  }
}

ll expand(int x, int r);

ll expand_iter(int x) {
  ll r = 0;
  for (int u = nxt[x]; u != x; u = nxt[u]) {
    if (ed[u]->w0 >= INF)
      return INF;
    else
      r += expand(ed[u]->v, u) + ed[u]->w0;
  }
  return r;
}

ll expand(int x, int t) {
  ll r = 0;
  for (; x != t; x = fa[x]) {
    r += expand_iter(x);
    if (r >= INF) return INF;
  }
  return r;
}

void link(int u, int v, int w) { in[v].push_back({u, v, w, w}); }

int main() {
```

```
129   int rt;
130   scanf("%d %d %d", &n, &m, &rt);
131   for (int i = 0; i < m; i++) {
132     int u, v, w;
133     scanf("%d %d %d", &u, &v, &w);
134     link(u, v, w);
135   }
136   // 保证强连通
137   for (int i = 1; i <= n; i++) link(i > 1 ? i - 1 : n, i, INF);
138   contract();
139   ll ans = expand(rt, n);
140   if (ans >= INF)
141     puts("-1");
142   else
143     printf("%lld\n", ans);
144   return 0;
145 }
```

参考文献

Uri Zwick. (2013),Directed Minimum Spanning Trees[1], Lecture notes on "Analysis of Algorithms"

https://riteme.site/blog/2018-6-18/mdst.html#_3[2]

外部参考资料

[1] Directed Minimum Spanning Trees 

[2] https://riteme.site/blog/2018-6-18/mdst.html#_3 

11.10.3 最小直径生成树

在学习最小直径生成树（Minimum Diameter Spanning Tree）前建议先阅读 树的直径→11.6.2@p2632 的内容.

定义

在无向图的所有生成树中，直径最小的那一棵生成树就是最小直径生成树.

图的绝对中心

求解直径最小生成树，首先需要找到 **图的绝对中心**，**图的绝对中心** 可以存在于一条边上或某个结点上，该中心到所有点的最短距离的最大值最小.

根据 **图的绝对中心** 的定义可以知道，到绝对中心距离最远的结点至少有两个.

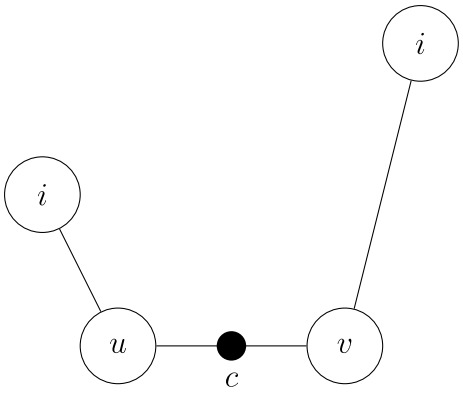
令 $d(i,j)$ 为顶点 i,j 间的最短路径长，通过多源最短路算法求出所有结点的最短路.

$rk(i,j)$ 记录点 i 到其他所有结点中第 j 小的那个结点.

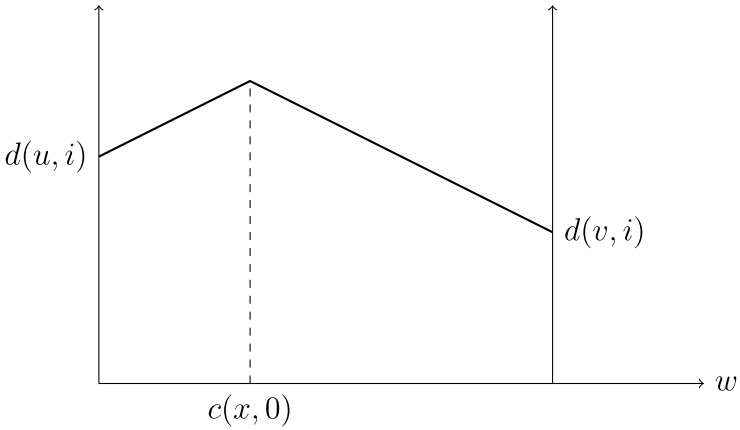
图的绝对中心可能在某条边上，枚举每一条边 $w=(u,v)$，并且假设图的绝对中心 c 就在这条边上. 那么距离 u 的长度为 x（$x\le w$），距离 v 的长度就是 $w-x$.

对于图中的任意一点 i，图的绝对中心 c 到 i 的距离为 $d(c,i)=\min(d(u,i)+x,d(v,i)+(w-x))$.

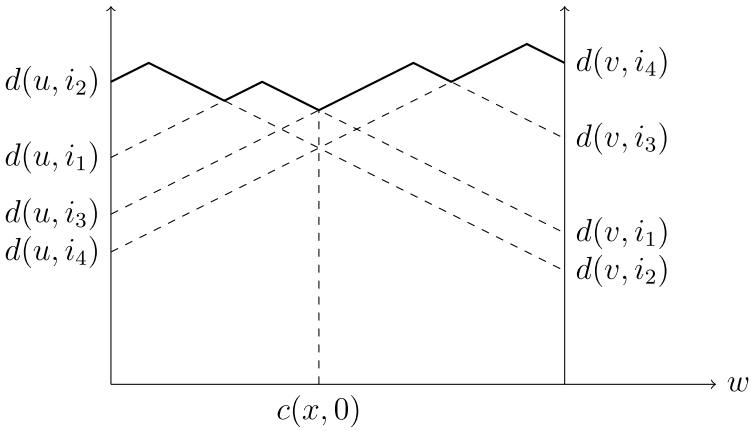
举例一个结点 i，该结点与图的绝对中心的位置关系如下图.



随着图的绝对中心 c 在边上的改变会生成一个距离与 c 位置的函数图像．显然的，当前的 $d(c,i)$ 的函数图像是一个两条斜率相同的线段构成的折线段．



对于图上的任意一结点，图的绝对中心到最远距离结点的函数就写作 $f=\max\{d(c,i)\},i\in[1,n]$，其函数图像如下．



并且这些折线交点中的最低点，横坐标就是图的绝对中心的位置．

图的绝对中心可能在某个结点上，用距离预选结点最远的那个结点来更新，即 $ans\leftarrow\min(ans,d(i,rk(i,n))\times 2)$．

过程

1. 使用多源最短路算法（Floyd→11.9.1@p2735，Johnson→11.9.1@p2735 等），求出 d 数组；
2. 求出 $rk(i,j)$，并将其升序排序；

3. 图的绝对中心可能在某个结点上，用距离预选结点最远的那个结点来更新，遍历所有结点并用 $ans \leftarrow \min(ans, d(i, rk(i, n)) \times 2)$ 更新最小值.
4. 图的绝对中心可能在某条边上，枚举所有的边. 对于一条边 $w(u, v)$ 从距离 u 最远的结点开始更新. 当出现 $d(v, rk(u, i)) > \max_{j=i+1}^{n} d(v, rk(u, j))$ 的情况时，用 $ans \leftarrow \min(ans, d(u, rk(u, i)) + \max_{j=i+1}^{n} d(v, rk(u, j)) + w(u, v))$ 来更新. 因为这种情况会使图的绝对中心改变.

实现

```
bool cmp(int a, int b) { return val[a] < val[b]; }

void Floyd() {
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

void solve() {
  Floyd();
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      rk[i][j] = j;
      val[j] = d[i][j];
    }
    sort(rk[i] + 1, rk[i] + 1 + n, cmp);
  }
  int ans = INF;
  // 图的绝对中心可能在结点上
  for (int i = 1; i <= n; i++) ans = min(ans, d[i][rk[i][n]] * 2);
  // 图的绝对中心可能在边上
  for (int i = 1; i <= m; i++) {
    int u = a[i].u, v = a[i].v, w = a[i].w;
    for (int p = n, i = n - 1; i >= 1; i--) {
      if (d[v][rk[u][i]] > d[v][rk[u][p]]) {
        ans = min(ans, d[u][rk[u][i]] + d[v][rk[u][p]] + w);
        p = i;
      }
    }
  }
}
```

例题

- CodeForce 266D BerDonalds[1]

最小直径生成树

根据图的绝对中心的定义，容易得知图的绝对中心是最小直径生成树的直径的中点.

求解最小直径生成树首先需要找到图的绝对中心. 以图的绝对中心为起点，生成一个最短路径树，那么就可以得到最小直径生成树了.

实现

```cpp
#include <algorithm>
#include <climits>
#include <iostream>
#include <vector>
using namespace std;
constexpr int MAXN = 502;
using ll = long long;
using pii = pair<int, int>;
ll d[MAXN][MAXN], dd[MAXN][MAXN], rk[MAXN][MAXN], val[MAXN];
constexpr ll INF = 1e17;
int n, m;

bool cmp(int a, int b) { return val[a] < val[b]; }

void floyd() {
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

struct node {
  ll u, v, w;
} a[MAXN * (MAXN - 1) / 2];

void solve() {
  // 求图的绝对中心
  floyd();
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      rk[i][j] = j;
      val[j] = d[i][j];
    }
    sort(rk[i] + 1, rk[i] + 1 + n, cmp);
  }
  ll P = 0, ansP = INF;
  // 在点上
  for (int i = 1; i <= n; i++) {
    if (d[i][rk[i][n]] * 2 < ansP) {
      ansP = d[i][rk[i][n]] * 2;
      P = i;
    }
  }
  // 在边上
  int f1 = 0, f2 = 0;
  ll disu = INT_MIN, disv = INT_MIN, ansL = INF;
  for (int i = 1; i <= m; i++) {
    ll u = a[i].u, v = a[i].v, w = a[i].w;
    for (int p = n, i = n - 1; i >= 1; i--) {
      if (d[v][rk[u][i]] > d[v][rk[u][p]]) {
        if (d[u][rk[u][i]] + d[v][rk[u][p]] + w < ansL) {
          ansL = d[u][rk[u][i]] + d[v][rk[u][p]] + w;
```

```
          f1 = u, f2 = v;
          disu = (d[u][rk[u][i]] + d[v][rk[u][p]] + w) / 2 - d[u][rk[u][i]];
          disv = w - disu;
        }
        p = i;
      }
    }
  }
  cout << min(ansP, ansL) / 2 << '\n';
  // 最小路径生成树
  vector<pii> pp;
  for (int i = 1; i <= 501; ++i)
    for (int j = 1; j <= 501; ++j) dd[i][j] = INF;
  for (int i = 1; i <= 501; ++i) dd[i][i] = 0;
  if (ansP <= ansL) {
    for (int j = 1; j <= n; j++) {
      for (int i = 1; i <= m; ++i) {
        ll u = a[i].u, v = a[i].v, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
        u = a[i].v, v = a[i].u, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
      }
    }
    for (auto [x, y] : pp) cout << x << ' ' << y << '\n';
  } else {
    d[n + 1][f1] = disu;
    d[f1][n + 1] = disu;
    d[n + 1][f2] = disv;
    d[f2][n + 1] = disv;
    a[m + 1].u = n + 1, a[m + 1].v = f1, a[m + 1].w = disu;
    a[m + 2].u = n + 1, a[m + 2].v = f2, a[m + 2].w = disv;
    n += 1;
    m += 2;
    floyd();
    P = n;
    for (int j = 1; j <= n; j++) {
      for (int i = 1; i <= m; ++i) {
        ll u = a[i].u, v = a[i].v, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
        }
        u = a[i].v, v = a[i].u, w = a[i].w;
        if (dd[P][u] + w == d[P][v] && dd[P][u] + w < dd[P][v]) {
          dd[P][v] = dd[P][u] + w;
          pp.push_back({u, v});
```

```
        }
      }
    }
    cout << f1 << ' ' << f2 << '\n';
    for (auto [x, y] : pp)
      if (x != n && y != n) cout << x << ' ' << y << '\n';
  }
}

void init() {
  for (int i = 1; i <= 501; ++i)
    for (int j = 1; j <= 501; ++j) d[i][j] = INF;
  for (int i = 1; i <= 501; ++i) d[i][i] = 0;
}

int main() {
  init();
  cin >> n >> m;
  for (int i = 1; i <= m; ++i) {
    ll u, v, w;
    cin >> u >> v >> w;
    w *= 2;
    d[u][v] = w, d[v][u] = w;
    a[i].u = u, a[i].v = v, a[i].w = w;
  }
  solve();
  return 0;
}
```

例题

SPOJ MDST[2]

timus 1569. Networking the "Iset"[3]

SPOJ PT07C - The GbAaY Kingdom[4]

参考文献

Play with Trees Solutions The GbAaY Kingdom[5]

外部参考资料

[1] CodeForce 266D BerDonalds 

[2] SPOJ MDST 

[3] timus 1569. Networking the "Iset" 

[4] SPOJ PT07C - The GbAaY Kingdom 

[5] Play with Trees Solutions The GbAaY Kingdom 

11.11 斯坦纳树

斯坦纳树问题是组合优化问题，与最小生成树相似，是最短网络的一种．最小生成树是在给定的点集和边中寻求最短网络使所有点连通．而最小斯坦纳树允许在给定点外增加额外的点，使生成的最短网络开销最小．

问题引入

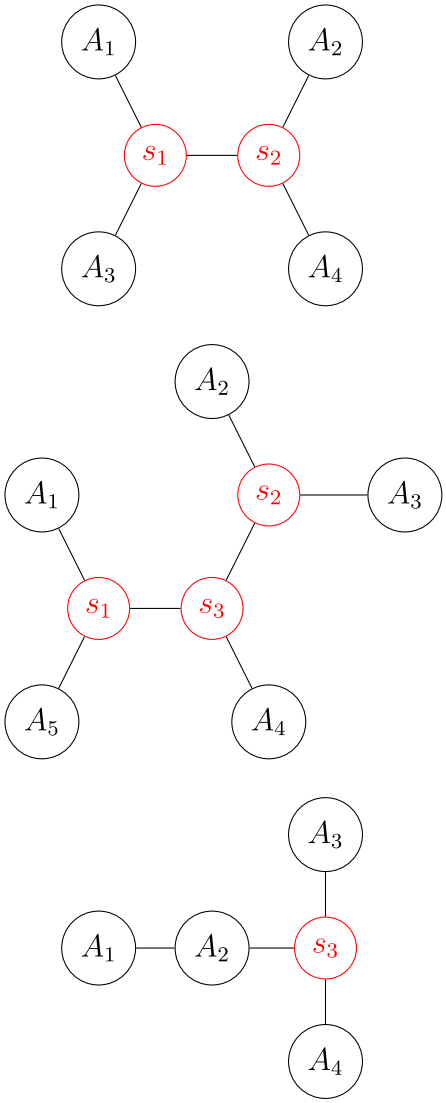
19 世纪初叶，柏林大学几何方面的著名学者斯坦纳，研究了一个非常简单却很有启示性的问题：将三个村庄用总长为极小的道路连接起来．从数学上说，就是在平面内给定三个点 A、B、C 找出平面内第四个点 P，使得和数 $a+b+c$ 为最短，这里 a、b、c 分别表示从 P 到 A、B、C 的距离．

问题的答案是：如果三角形 ABC 的每个内角都小于 120°，那么 P 就是使边 AB、BC、AC 对该点所张的角都是 120° 的点．如果三角形 ABC 的有一个角，例如 C 角，大于或等于 120°，那么点 P 与顶点 C 重合．

问题推广

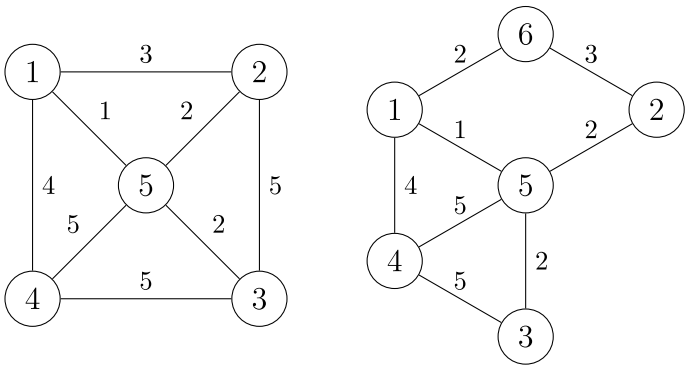
1. 在斯坦纳问题中，给定了三个固定点 A,B,C．很自然地可以把这个问题推广到给定 n 个点 $A_1,A_2,...,A_n$ 的情形；我们要求出平面内的点 P，使距离和 $a_1+a_2+...+a_n$ 为极小，其中 a_i 是距离 PA_i.
2. 考虑到点的其他相关因素，加入了权重的表示．n 个点的其他相关因素可以换算成一个权重表示，求出平面内的点 P，使距离与权重的乘积的总和 $a_1\cdot w_1+a_2\cdot w_2+...+a_n\cdot w_n$ 为极小，其中 w_i 是每个点的权重.
3. 库朗（R.Courant）和罗宾斯（H.Robbins）提出第一个定义的推广是肤浅的．为了求得斯坦纳问题真正有价值的推广，必须放弃寻找一个单独的点 P，而代之以具有最短总长的＂道路网＂．数学上表述成：给定 n 个点 $A_1,A_2,\cdots,A_n$，试求连接此 n 个点，总长最短的直线段连接系统，并且任意两点都可由系统中的直线段组成的折线连接起来．他们将此新问题称为 **斯坦纳树问题**．在给定 n 个点的情形，最多将有 $n-2$ 个复接点（斯坦纳点）．过每一斯坦纳点，至多有三条边通过．若为三条边，则它们两两交成 120° 角；若为两条边，则此斯坦纳点必为某一已给定的点，且此两条边交成的角必大于或等于 120°.

连接三个以上的点的最短网络



在第一种情形，解是由五条线段组成的，其中有两个斯坦纳点（红色 s_1, s_2），在那里有三条线段相交且相互间的交角为 120°. 第二种情形的解含有三个斯坦纳点. 第三种情形，一个或几个斯坦纳点可能退化，或被一个或几个给定的点所代替.

我们将斯坦纳树的问题模型以图论形式呈现.



对于形式一，如果令关键点为 $\{1, 2, 3, 4\}$，可以发现若直接将这四个关键点相连的最小边权和是 12，显然这不是最优的. 如果考虑使用 5 号节点那么最小边权和就会是 9，得到一个更优的答案.

对于形式二，如果令关键点为 $\{1,2,3,4\}$，可以发现这四个关键点中的一些点甚至没有直接相连的边，必须考虑使用复接点（斯坦纳点）．这时将 5 号考虑进去可以得到最小边权和 9．

并且我们可以发现在两张图中 1 号和 4 号的斯坦纳点是退化的，被 1 号或 4 号代替了．

例题

首先以一道模板题来带大家熟悉最小斯坦纳树问题．见 【模板】最小斯坦纳树[1]．

题意已经很明确了，给定连通图 G 中的 n 个点与 k 个关键点，连接 k 个关键点，使得生成树的所有边的权值和最小．

结合上面的知识我们可以知道直接连接这 k 个关键点生成的权值和不一定是最小的，或者这 k 个关键点不会直接（相邻）连接．所以应当使用剩下的 $n-k$ 个点．

我们使用状态压缩动态规划来求解．用 $f(i,S)$ 表示以 i 为根的一棵树，包含集合 S 中所有点的最小边权值和．

考虑状态转移：

- 首先对连通的子集进行转移，$f(i,S) \leftarrow \min(f(i,S), f(i,T)+f(i,S-T))$．
- 在当前的子集连通状态下进行边的松弛操作，$f(i,S) \leftarrow \min(f(i,S), f(j,S)+w(j,i))$．在下面的代码中用一个 `tree[tot]` 来记录两个相连节点 i,j 的相关信息．

参考实现

```cpp
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

constexpr int MAXN = 510;
constexpr int INF = 0x3f3f3f3f;
using P = pair<int, int>;
int n, m, k;

struct edge {
  int to, next, w;
} e[MAXN << 1];

int head[MAXN << 1], tree[MAXN << 1], tot;
int dp[MAXN][5000], vis[MAXN];
int key[MAXN];
priority_queue<P, vector<P>, greater<P>> q;

void add(int u, int v, int w) {
  e[++tot] = edge{v, head[u], w};
  head[u] = tot;
}

void dijkstra(int s) {  // 求解最短路
  memset(vis, 0, sizeof(vis));
  while (!q.empty()) {
    P item = q.top();
```

```
    q.pop();
    if (vis[item.second]) continue;
    vis[item.second] = 1;
    for (int i = head[item.second]; i; i = e[i].next) {
      if (dp[tree[i]][s] > dp[item.second][s] + e[i].w) {
        dp[tree[i]][s] = dp[item.second][s] + e[i].w;
        q.push(P(dp[tree[i]][s], tree[i]));
      }
    }
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  memset(dp, INF, sizeof(dp));
  cin >> n >> m >> k;
  int u, v, w;
  for (int i = 1; i <= m; i++) {
    cin >> u >> v >> w;
    add(u, v, w);
    tree[tot] = v;
    add(v, u, w);
    tree[tot] = u;
  }
  for (int i = 1; i <= k; i++) {
    cin >> key[i];
    dp[key[i]][1 << (i - 1)] = 0;
  }
  for (int s = 1; s < (1 << k); s++) {
    for (int i = 1; i <= n; i++) {
      for (int subs = s & (s - 1); subs;
           subs = s & (subs - 1))  // 状压 dp 可以看下题解里写的比较详细
        dp[i][s] = min(dp[i][s], dp[i][subs] + dp[i][s ^ subs]);
      if (dp[i][s] != INF) q.push(P(dp[i][s], i));
    }
    dijkstra(s);
  }
  cout << dp[key[1]][(1 << k) - 1] << '\n';
  return 0;
}
```

另外一道经典例题 [WC2008] 游览计划[2].

这道题是求点权和最小的斯坦纳树，用 $f(i,S)$ 表示以 i 为根的一棵树，包含集合 S 中所有点的最小点权值和．a_i 表示点权．

考虑状态转移：

- $f(i,S) \leftarrow \min(f(i,S), f(i,T) + f(i,S-T) - a_i)$．由于此处合并时同一个点 a_i，会被加两次，所以减去．
- $f(i,S) \leftarrow \min(f(i,S), f(j,S) + w(j,i))$．

可以发现状态转移与上面的模板题是类似的，麻烦的是对答案的输出，在 DP 的过程中还要记录路径．

用 `pre[i][s]` 记录转移到 i 为根，连通状态集合为 s 时的点与集合的信息. 在 DP 结束后从 `pre[root][S]` 出发，寻找与集合里的点相连的那些点并逐步分解集合 S，用 ans 数组来记录被使用的那些点，当集合分解完毕时搜索也就结束了.

参考实现

```cpp
#include <cstring>
#include <iostream>
#include <queue>

using namespace std;

#define mp make_pair
using P = pair<int, int>;
using PP = pair<P, int>;
constexpr int INF = 0x3f3f3f3f;
constexpr int dx[] = {0, 0, -1, 1};
constexpr int dy[] = {1, -1, 0, 0};
int n, m, K, root;
int f[101][1111], a[101], ans[11][11];
bool inq[101];
PP pre[101][1111];
queue<P> q;

bool legal(P u) {
  if (u.first >= 0 && u.second >= 0 && u.first < n && u.second < m) {
    return true;
  }
  return false;
}

int num(P u) { return u.first * m + u.second; }

void spfa(int s) {
  memset(inq, 0, sizeof(inq));
  while (!q.empty()) {
    P u = q.front();
    q.pop();
    inq[num(u)] = false;
    for (int d = 0; d < 4; d++) {
      P v = mp(u.first + dx[d], u.second + dy[d]);
      int du = num(u), dv = num(v);
      if (legal(v) && f[dv][s] > f[du][s] + a[dv]) {
        f[dv][s] = f[du][s] + a[dv];
        if (!inq[dv]) {
          inq[dv] = true;
          q.push(v);
        }
        pre[dv][s] = mp(u, s);
      }
    }
  }
}
```

```
48
49 void dfs(P u, int s) {
50   if (!pre[num(u)][s].second) return;
51   ans[u.first][u.second] = 1;
52   int nu = num(u);
53   if (pre[nu][s].first == u)
54     dfs(u, s ^ pre[nu][s].second);  // 通过 dfs 来找到答案
55   dfs(pre[nu][s].first, pre[nu][s].second);
56 }
57
58 int main() {
59   cin.tie(nullptr)->sync_with_stdio(false);
60   memset(f, INF, sizeof(f));
61   cin >> n >> m;
62   int tot = 0;
63   for (int i = 0; i < n; i++) {
64     for (int j = 0; j < m; j++) {
65       cin >> a[tot];
66       if (!a[tot]) {
67         f[tot][1 << (K++)] = 0;
68         root = tot;
69       }
70       tot++;
71     }
72   }
73   for (int s = 1; s < (1 << K); s++) {
74     for (int i = 0; i < n * m; i++) {
75       for (int subs = s & (s - 1); subs; subs = s & (subs - 1)) {
76         if (f[i][s] > f[i][subs] + f[i][s ^ subs] - a[i]) {
77           f[i][s] = f[i][subs] + f[i][s ^ subs] - a[i];  // 状态转移
78           pre[i][s] = mp(mp(i / m, i % m), subs);
79         }
80       }
81       if (f[i][s] < INF) q.push(mp(i / m, i % m));
82     }
83     spfa(s);
84   }
85   cout << f[root][(1 << K) - 1] << '\n';
86   dfs(mp(root / m, root % m), (1 << K) - 1);
87   for (int i = 0, tot = 0; i < n; i++) {
88     for (int j = 0; j < m; j++) {
89       if (!a[tot++])
90         cout << 'x';
91       else
92         cout << (ans[i][j] ? 'o' : '_');
93     }
94     if (i != n - 1) cout << '\n';
95   }
96   return 0;
97 }
```

习题

- 【模板】最小斯坦纳树[3]
- [WC2008] 游览计划[4]
- [JLOI2015] 管道连接[5]
- [APIO2013] 机器人[6]

外部参考资料

[1] 【模板】最小斯坦纳树 [2] [WC2008] 游览计划 [3] 【模板】最小斯坦纳树

[4] [WC2008] 游览计划 [5] [JLOI2015] 管道连接 [6] [APIO2013] 机器人

11.12 拆点

Authors: Anguei, sshwy, Xeonacid, Ir1d, MonkeyOliver, hsfzLZH1

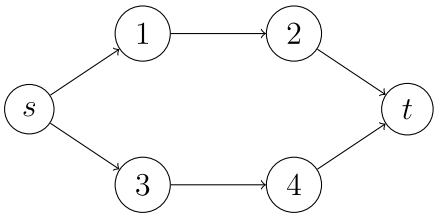
拆点是一种图论建模思想，常用于 网络流→11.23.1@p2906，用来处理 **点权或者点的流量限制** 的问题，也常用于 **分层图**.

结点有流量限制的最大流

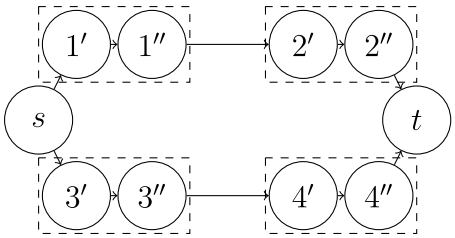
如果把结点转化成边，那么这个问题就可以套板子解决了.

我们考虑把有流量限制的结点转化成这样一种形式：由两个结点 u, v 和一条边 $\langle u, v\rangle$ 组成的部分. 其中，结点 u 承接所有从原图上其他点的出发到原图上该点的边，结点 v 引出所有从原图上该点出发到达原图上其他点的边. 边 $\langle u, v\rangle$ 的流量限制为原图该点的流量限制，再套板子就可以解决本题. 这就是拆点的基本思想.

如果原图是这样：



拆点之后的图是这个样子：



分层图最短路

分层图最短路，如：有 k 次零代价通过一条路径，求总的最小花费. 对于这种题目，我们可以采用 DP 相关的思想，设 $\mathrm{dis}_{i,j}$ 表示当前从起点 i 号结点，使用了 j 次免费通行权限后的最短路径. 显然，dis 数组可以这么转移：

$\text{dis}_{i,j} = \min\{\min\{\text{dis}_{from,j-1}\}, \min\{\text{dis}_{from,j} + w\}\}$

其中，$from$ 表示 i 的父亲节点，w 表示当前所走的边的边权．当 $j-1 \geq k$ 时，$\text{dis}_{from,j}=\infty$.

事实上，这个 DP 就相当于把每个结点拆分成了 $k+1$ 个结点，每个新结点代表使用不同多次免费通行后到达的原图结点．换句话说，就是每个结点 u_i 表示使用 i 次免费通行权限后到达 u 结点.

「JLOI2011」飞行路线[1]

题意：有一个 n 个点 m 条边的无向图，你可以选择 k 条道路以零代价通行，求 s 到 t 的最小花费.

参考核心代码：

```cpp
struct State {    // 优先队列的结点结构体
  int v, w, cnt;  // cnt 表示已经使用多少次免费通行权限

  State() {}

  State(int v, int w, int cnt) : v(v), w(w), cnt(cnt) {}

  bool operator<(const State &rhs) const { return w > rhs.w; }
};

void dijkstra() {
  memset(dis, 0x3f, sizeof dis);
  dis[s][0] = 0;
  pq.push(State(s, 0, 0));  // 到起点不需要使用免费通行权，距离为零
  while (!pq.empty()) {
    const State top = pq.top();
    pq.pop();
    int u = top.v, nowCnt = top.cnt;
    if (done[u][nowCnt]) continue;
    done[u][nowCnt] = true;
    for (int i = head[u]; i; i = edge[i].next) {
      int v = edge[i].v, w = edge[i].w;
      if (nowCnt < k && dis[v][nowCnt + 1] > dis[u][nowCnt]) {  // 可以免费通行
        dis[v][nowCnt + 1] = dis[u][nowCnt];
        pq.push(State(v, dis[v][nowCnt + 1], nowCnt + 1));
      }
      if (dis[v][nowCnt] > dis[u][nowCnt] + w) {  // 不可以免费通行
        dis[v][nowCnt] = dis[u][nowCnt] + w;
        pq.push(State(v, dis[v][nowCnt], nowCnt));
      }
    }
  }
}

int main() {
  n = read(), m = read(), k = read();
  // 笔者习惯从 1 到 n 编号，而这道题是从 0 到 n - 1，所以要处理一下
  s = read() + 1, t = read() + 1;
  while (m--) {
    int u = read() + 1, v = read() + 1, w = read();
    add(u, v, w), add(v, u, w);  // 这道题是双向边
  }
```

```
43   dijkstra();
44   int ans = std::numeric_limits<int>::max();  // ans 取 int 最大值为初值
45   for (int i = 0; i <= k; ++i)
46     ans = std::min(ans, dis[t][i]);  // 对到达终点的所有情况取最优值
47   println(ans);
48 }
```

外部参考资料

[1] 「JLOI2011」飞行路线 

11.13 连通性相关

11.13.1 强连通分量

简介

在阅读下列内容之前，请务必了解 图论相关概念→11.2@p2598 中的基础部分.

强连通的定义是：有向图 G 强连通是指，G 中任意两个结点连通.

强连通分量（Strongly Connected Components，SCC）的定义是：极大的强连通子图.

这里要介绍的是如何来求强连通分量.

Tarjan 算法

引入

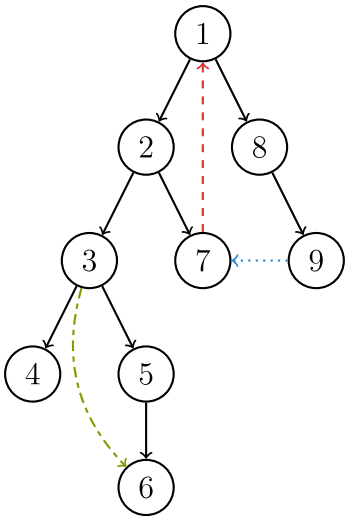
Robert E. Tarjan（罗伯特·塔扬，1948~），生于美国加州波莫纳，计算机科学家.

Tarjan 发明了很多算法和数据结构. 不少他发明的算法都以他的名字命名，以至于有时会让人混淆几种不同的算法. 比如求各种连通分量的 Tarjan 算法，求 LCA（Lowest Common Ancestor，最近公共祖先）的 Tarjan 算法. 并查集、Splay、Toptree 也是 Tarjan 发明的.

我们这里要介绍的是在有向图中求强连通分量的 Tarjan 算法.

DFS 生成树

在介绍该算法之前，先来了解 DFS **生成树**，我们以下面的有向图为例：



在有向图 G 上运行 DFS 算法时，由于边具有方向性，从单个结点出发可能无法访问到图中的全部结点．因此，我们需要遍历整个顶点集：对每个尚未被访问的结点，都重新发起一次 DFS．在每一次从某个起始结点出发并完成的 DFS 过程中，其所经过的树边（见下文）会构成一棵树，称为 **DFS 生成树**．当所有结点都被访问后，得到的 DFS 生成树的全体构成了该有向图的 **DFS 生成森林**．

需要注意的是，生成树（以及生成森林）的具体结构，以及下文中的边分类，都依赖于 DFS 的起始结点选择和邻接点的访问顺序．

有向图 G 的边可分为四类：

1. 树边（tree edge）：示意图中以黑色边表示，每次搜索找到一个还没有访问过的结点的时候就形成了一条树边．所有相邻的树边组成 DFS 生成树．
2. 返祖边（back edge）：也称回边，示意图中以红色边表示（即 $7 \to 1$），指在搜索过程中，从某个结点指向其祖先结点的非树边．
3. 前向边（forward edge）：示意图中以绿色边表示（即 $3 \to 6$），指在搜索过程中，从某个结点指向其子树中后代结点的非树边．
4. 横叉边（cross edge）：示意图中以蓝色边表示（即 $9 \to 7$），指在搜索过程中，从某个结点指向非祖先、非后代且已访问的结点的边，即不属于上述三类的边．

我们考虑 DFS 生成树与强连通分量之间的关系．

如果结点 u 是某个强连通分量在搜索树中遇到的第一个结点，那么这个强连通分量的其余结点肯定是在搜索树中以 u 为根的子树中．结点 u 被称为这个强连通分量的根．

反证法：假设有个结点 v 在该强连通分量中但是不在以 u 为根的子树中，那么 u 到 v 的路径中肯定有一条离开子树的边．但是这样的边只可能是横叉边或者返祖边，然而这两条边都要求指向的结点已经被访问过了，这就和 v 不在以 u 为根的子树中矛盾了．得证．

Tarjan 算法求强连通分量

Tarjan 算法基于对图进行 深度优先搜索→11.4@p2612．我们视每个连通分量为搜索树中的一棵子树，在搜索过程中，维护一个栈，每次把搜索树中尚未处理的节点加入栈中，并将确定下来答案的点从栈中弹出．

在 Tarjan 算法中为每个结点 u 维护了以下几个变量：

1. dfn_u：深度优先搜索遍历时结点 u 被搜索的次序．
2. low_u：在 u 的子树中能够回溯到的最早的已经在栈中的结点．设在搜索树中以 u 为根的子树为 $Subtree_u$．low_u 定义为以下结点的 dfn 的最小值：从 $Subtree_u$ 通过一条不在搜索树上的边能到达的在栈中的结点．

一个结点的子树内结点的 dfn 都大于该结点的 dfn．

从根开始的一条路径上的 dfn 严格递增，low 严格非降．

按照深度优先搜索算法搜索的次序对图中所有的结点进行搜索，维护每个结点的 `dfn` 与 `low` 变量，且让搜索到的结点入栈．每当找到一个强连通元素，就按照该元素包含结点数目让栈中元素出栈．在搜索过程中，对于结点 u 和与其相邻的结点 v（v 不是 u 的父节点）考虑 3 种情况：

1. v 未被访问：继续对 v 进行深度搜索．在回溯过程中，用 low_v 更新 low_u．因为存在从 u 到 v 的直接路径，所以 v 能够回溯到的已经在栈中的结点，u 也一定能够回溯到．
2. v 被访问过，已经在栈中：根据 low 值的定义，用 dfn_v 更新 low_u．
3. v 被访问过，已不在栈中：说明 v 已搜索完毕，其所在连通分量已被处理，所以不用对其做操作．

对于一个连通分量图，我们很容易想到，在该连通图中有且仅有一个 u 使得 $dfn_u = low_u$．该结点一定是在深度遍历的过程中，该连通分量中第一个被访问过的结点，因为它的 dfn 和 low 值最小，不会被该连通分量中的其他结点所影响．

因此，在回溯的过程中，判定 $dfn_u = low_u$ 是否成立，如果成立，则栈中 u 及其上方的结点构成一个 SCC．

将上述算法写成伪代码：

实现

```
TARJAN_SEARCH(int u)
    vis[u]=true
    low[u]=dfn[u]=++dfncnt
    push u to the stack
    for each (u,v) then do
        if v hasn't been searched then
            TARJAN_SEARCH(v) // 搜索
            low[u]=min(low[u],low[v]) // 回溯
        else if v has been in the stack then
            low[u]=min(low[u],dfn[v])
    if dfn[u] equal to low[u] then
        ++scccnt
        while top of stack not equal to u then
            scc[top of stack] = scccnt
            pop stack
        scc[u] = scccnt
        pop stack // 处理并删除残余的 u
```

实现

- C++

```cpp
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (in_stack[v]) {
      low[u] = min(low[u], dfn[v]);
```

```
    }
  }
  if (dfn[u] == low[u]) {
    ++sc;
    do {
      scc[s[tp]] = sc;
      sz[sc]++;
      in_stack[s[tp]] = 0;
    } while (s[tp--] != u);
  }
}
```

- Python

```
dfn = [0] * N
low = [0] * N
dfncnt = 0
s = [0] * N
in_stack = [0] * N
tp = 0
scc = [0] * N
sc = 0  # 结点 i 所在 SCC 的编号
sz = [0] * N  # 强连通 i 的大小


def tarjan(u):
    low[u] = dfn[u] = dfncnt
    s[tp] = u
    in_stack[u] = 1
    dfncnt = dfncnt + 1
    tp = tp + 1
    i = h[u]
    while i:
        v = e[i].t
        if dfn[v] == False:
            tarjan(v)
            low[u] = min(low[u], low[v])
        elif in_stack[v]:
            low[u] = min(low[u], dfn[v])
        i = e[i].nex
    if dfn[u] == low[u]:
        sc = sc + 1
        while s[tp] != u:
            scc[s[tp]] = sc
            sz[sc] = sz[sc] + 1
            in_stack[s[tp]] = 0
            tp = tp - 1
        scc[s[tp]] = sc
        sz[sc] = sz[sc] + 1
        in_stack[s[tp]] = 0
        tp = tp - 1
```

时间复杂度 $O(n+m)$.

分量标号和拓扑序的关系

Tarjan 算法在处理过程中，实际上是按照某种 **逆拓扑序** 来发现强连通分量的，这是因为算法在深度优先搜索的过程中会先访问完那些没有出边的节点，而这与拓扑排序的过程是相反的.

如果我们将图中的所有强连通分量缩成单个节点，那么在这些缩点后的节点形成的 DAG 中进行拓扑排序，得到的顺序将与 Tarjan 算法给出的强连通分量的标号顺序相反.

因此，可以说，在缩点后的 DAG 中，**强连通分量（缩点后）的标号顺序是其拓扑序的逆序**. 但要注意的是, 这种说法仅在考虑了强连通分量之间的依赖关系（即从一个强连通分量到另一个强连通分量的有向边）时才成立. 单个强连通分量内部的节点由于存在环，所以内部并不满足拓扑序的定义.

Kosaraju 算法

引入

Kosaraju 算法最早在 1978 年由 S. Rao Kosaraju 在一篇未发表的论文上提出，但 Micha Sharir 最早发表了它.

过程

该算法依靠两次简单的 DFS 实现:

第一次 DFS，选取任意顶点作为起点，遍历所有未访问过的顶点，并在回溯之前给顶点编号，也就是后序遍历.

第二次 DFS，对于反向后的图，以标号最大的顶点作为起点开始 DFS. 这样遍历到的顶点集合就是一个强连通分量. 对于所有未访问过的结点，选取标号最大的，重复上述过程.

两次 DFS 结束后，强连通分量就找出来了，Kosaraju 算法的时间复杂度为 $O(n+m)$.

实现

- C++

```cpp
// g 是原图，g2 是反图

void dfs1(int u) {
  vis[u] = true;
  for (int v : g[u])
    if (!vis[v]) dfs1(v);
  s.push_back(u);
}

void dfs2(int u) {
  color[u] = sccCnt;
  for (int v : g2[u])
    if (!color[v]) dfs2(v);
}

void kosaraju() {
  sccCnt = 0;
  for (int i = 1; i <= n; ++i)
    if (!vis[i]) dfs1(i);
  for (int i = n - 1; i >= 0; --i)
    if (!color[s[i]]) {
      ++sccCnt;
      dfs2(s[i]);
    }
```

```
}
```

- Python

```python
def dfs1(u):
    vis[u] = True
    for v in g[u]:
        if vis[v] == False:
            dfs1(v)
    s.append(u)


def dfs2(u):
    color[u] = sccCnt
    for v in g2[u]:
        if color[v] == False:
            dfs2(v)


def kosaraju(u):
    sccCnt = 0
    for i in range(1, n + 1):
        if vis[i] == False:
            dfs1(i)
    for i in range(n - 1, -1, -1):
        if color[s[i]] == False:
            sccCnt = sccCnt + 1
            dfs2(s[i])
```

Garbow 算法

过程

Garbow 算法是 Tarjan 算法的另一种实现，Tarjan 算法是用 dfn 和 low 来计算强连通分量的根，Garbow 维护一个节点栈，并用第二个栈来确定何时从第一个栈中弹出属于同一个强连通分量的节点. 从节点 w 开始的 DFS 过程中，当一条路径显示这组节点都属于同一个强连通分量时，只要栈顶节点的访问时间大于根节点 w 的访问时间，就从第二个栈中弹出这个节点，那么最后只留下根节点 w. 在这个过程中每一个被弹出的节点都属于同一个强连通分量.

当回溯到某一个节点 w 时，如果这个节点在第二个栈的顶部，就说明这个节点是强连通分量的起始节点，在这个节点之后搜索到的那些节点都属于同一个强连通分量，于是从第一个栈中弹出那些节点，构成强连通分量.

实现

- C++

```cpp
int garbow(int u) {
  stack1[++p1] = u;
  stack2[++p2] = u;
  low[u] = ++dfs_clock;
  for (int i = head[u]; i; i = e[i].next) {
    int v = e[i].to;
    if (!low[v])
      garbow(v);
    else if (!sccno[v])
```

```
      while (low[stack2[p2]] > low[v]) p2--;
  }
  if (stack2[p2] == u) {
    p2--;
    scc_cnt++;
    do {
      sccno[stack1[p1]] = scc_cnt;
      // all_scc[scc_cnt] ++;
    } while (stack1[p1--] != u);
  }
  return 0;
}

void find_scc(int n) {
  dfs_clock = scc_cnt = 0;
  p1 = p2 = 0;
  memset(sccno, 0, sizeof(sccno));
  memset(low, 0, sizeof(low));
  for (int i = 1; i <= n; i++)
    if (!low[i]) garbow(i);
}
```

- Python

```
def garbow(u):
    stack1[p1] = u
    stack2[p2] = u
    p1 = p1 + 1
    p2 = p2 + 1
    low[u] = dfs_clock
    dfs_clock = dfs_clock + 1
    i = head[u]
    while i:
        v = e[i].to
        if low[v] == False:
            garbow(v)
        elif sccno[v] == False:
            while low[stack2[p2]] > low[v]:
                p2 = p2 - 1
    if stack2[p2] == u:
        p2 = p2 - 1
        scc_cnt = scc_cnt + 1
        while stack1[p1] != u:
            p1 = p1 - 1
            sccno[stack1[p1]] = scc_cnt


def find_scc(n):
    dfs_clock = scc_cnt = 0
    p1 = p2 = 0
    sccno = []
    low = []
    for i in range(1, n + 1):
        if low[i] == False:
            garbow(i)
```

应用

我们可以将一张图的每个强连通分量都缩成一个点.

然后这张图会变成一个 DAG，可以进行拓扑排序以及更多其他操作.

举个简单的例子，求一条路径，可以经过重复结点，要求经过的不同结点数量最多.

习题

USACO Fall/HAOI 2006 受欢迎的牛[1]

POJ1236 Network of Schools[2]

外部参考资料

[1] USACO Fall/HAOI 2006 受欢迎的牛

[2] POJ1236 Network of Schools

11.13.2 双连通分量

简介

在阅读下列内容之前，请务必了解 图论相关概念→11.2@p2598 部分.

相关阅读：割点和桥→11.13.3@p2823

定义

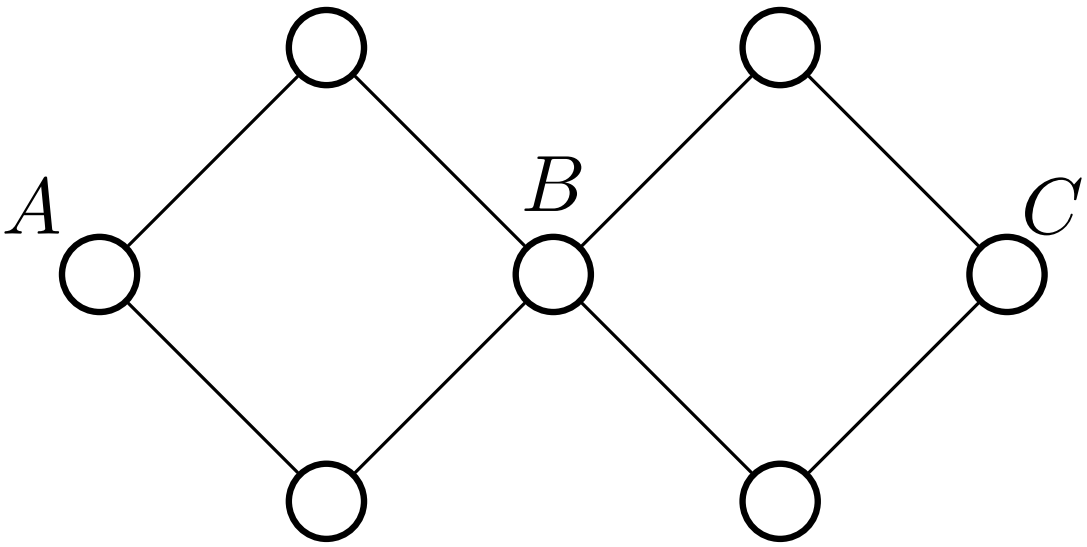
割点和桥更严谨的定义参见 图论相关概念→11.2@p2598.

在一张连通的无向图中，对于两个点 u 和 v，如果无论删去哪条边（只能删去一条）都不能使它们不连通，我们就说 u 和 v **边双连通**.

在一张连通的无向图中，对于两个点 u 和 v，如果无论删去哪个点（只能删去一个，且不能删 u 和 v 自己）都不能使它们不连通，我们就说 u 和 v **点双连通**.

边双连通具有传递性，即，若 x, y 边双连通，y, z 边双连通，则 x, z 边双连通.

点双连通 **不** 具有传递性，反例如下图，A, B 点双连通，B, C 点双连通，而 A, C **不** 点双连通.



对于一个无向图中的 **极大** 边双连通的子图，我们称这个子图为一个 **边双连通分量**.

对于一个无向图中的 **极大** 点双连通的子图，我们称这个子图为一个 **点双连通分量**.

DFS 生成树

对于一张连通的无向图，我们可以从任意一点开始 DFS，得到原图的一棵 DFS 生成树（以开始 DFS 的那个点为根），这棵生成树上的边称作 **树边**，不在生成树上的边称作 **非树边**.

由于 DFS 的性质，我们可以保证所有非树边连接的两个点在生成树上都满足其中一个是另一个的祖先.

DFS 的代码如下：

实现

- C++

```cpp
void DFS(int p) {
  visited[p] = true;
  for (int to : edge[p])
    if (!visited[to]) DFS(to);
}
```

- Python

```python
def DFS(p):
    visited[p] = True
    for to in edge[p]:
        if visited[to] == False:
            DFS(to)
```

边双连通分量

例题：洛谷 P8436【模版】边双连通分量[1]

对于一个 n 个节点 m 条无向边的图，请输出其边双连通分量的个数，并且输出每个边双连通分量.

Tarjan 算法 1

用 Tarjan 求双连通分量过程与求强连通分量类似，可以先阅读 强连通分量→11.13.1@p2806 的 Tarjan 算法.

我们考虑先求出所有的桥，再 DFS 求出边双连通分量.

求桥可参见 割点和桥→11.13.3@p2823 的桥部分.

时间复杂度 $O(n+m)$.

示例代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;
constexpr int N = 5e5 + 5, M = 2e6 + 5;
int n, m, ans;
int tot = 1, hd[N];
```

```cpp

struct edge {
  int to, nt;
} e[M << 1];

void add(int u, int v) { e[++tot].to = v, e[tot].nt = hd[u], hd[u] = tot; }

void uadd(int u, int v) { add(u, v), add(v, u); }

bool bz[M << 1];
int bcc_cnt, dfn[N], low[N], vis_bcc[N];
vector<vector<int>> bcc;

void tarjan(int x, int in) {
  dfn[x] = low[x] = ++bcc_cnt;
  for (int i = hd[x]; i; i = e[i].nt) {
    int v = e[i].to;
    if (dfn[v] == 0) {
      tarjan(v, i);
      if (dfn[x] < low[v]) bz[i] = bz[i ^ 1] = true;
      low[x] = min(low[x], low[v]);
    } else if (i != (in ^ 1))
      low[x] = min(low[x], dfn[v]);
  }
}

void dfs(int x, int id) {
  vis_bcc[x] = id, bcc[id - 1].push_back(x);
  for (int i = hd[x]; i; i = e[i].nt) {
    int v = e[i].to;
    if (vis_bcc[v] || bz[i]) continue;
    dfs(v, id);
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  int u, v;
  for (int i = 1; i <= m; i++) {
    cin >> u >> v;
    if (u == v) continue;
    uadd(u, v);
  }
  for (int i = 1; i <= n; i++)
    if (dfn[i] == 0) tarjan(i, 0);
  for (int i = 1; i <= n; i++)
    if (vis_bcc[i] == 0) {
      bcc.push_back(vector<int>());
      dfs(i, ++ans);
    }
  cout << ans << '\n';
```

```
  for (int i = 0; i < ans; i++) {
    cout << bcc[i].size();
    for (int j = 0; j < bcc[i].size(); j++) cout << ' ' << bcc[i][j];
    cout << '\n';
  }
  return 0;
}
```

Tarjan 算法 2

我们先总结出一个重要的性质，在无向图中，DFS 生成树上的边不是树边就只有非树边.

我们联系一下求强连通分量的方法，在无向图中只要一个分量没有桥，那么在 DFS 生成树上，它的所有点都在同一个强连通分量中.

反过来，在 DFS 生成树上的一个强连通分量，在原无向图中是边双连通分量.

可以发现，求边双连通分量的过程实际上就是求强连通分量的过程.

时间复杂度 $O(n+m)$.

示例代码

```
#include <algorithm>
#include <iostream>
#include <stack>
#include <vector>

using namespace std;
constexpr int N = 5e5 + 5, M = 2e6 + 5;
int n, m;

struct edge {
  int to, nt;
} e[M << 2];

int hd[N << 1], tot = 1;

void add(int u, int v) { e[++tot] = edge{v, hd[u]}, hd[u] = tot; }

void uadd(int u, int v) { add(u, v), add(v, u); }

int bcc_cnt, sum;
int dfn[N], low[N];
bool vis[N];
vector<vector<int>> ans;
stack<int> st;

void tarjan(int u, int in) {
  low[u] = dfn[u] = ++bcc_cnt;
  st.push(u), vis[u] = true;
  for (int i = hd[u]; i; i = e[i].nt) {
    int v = e[i].to;
    if (i == (in ^ 1)) continue;
```

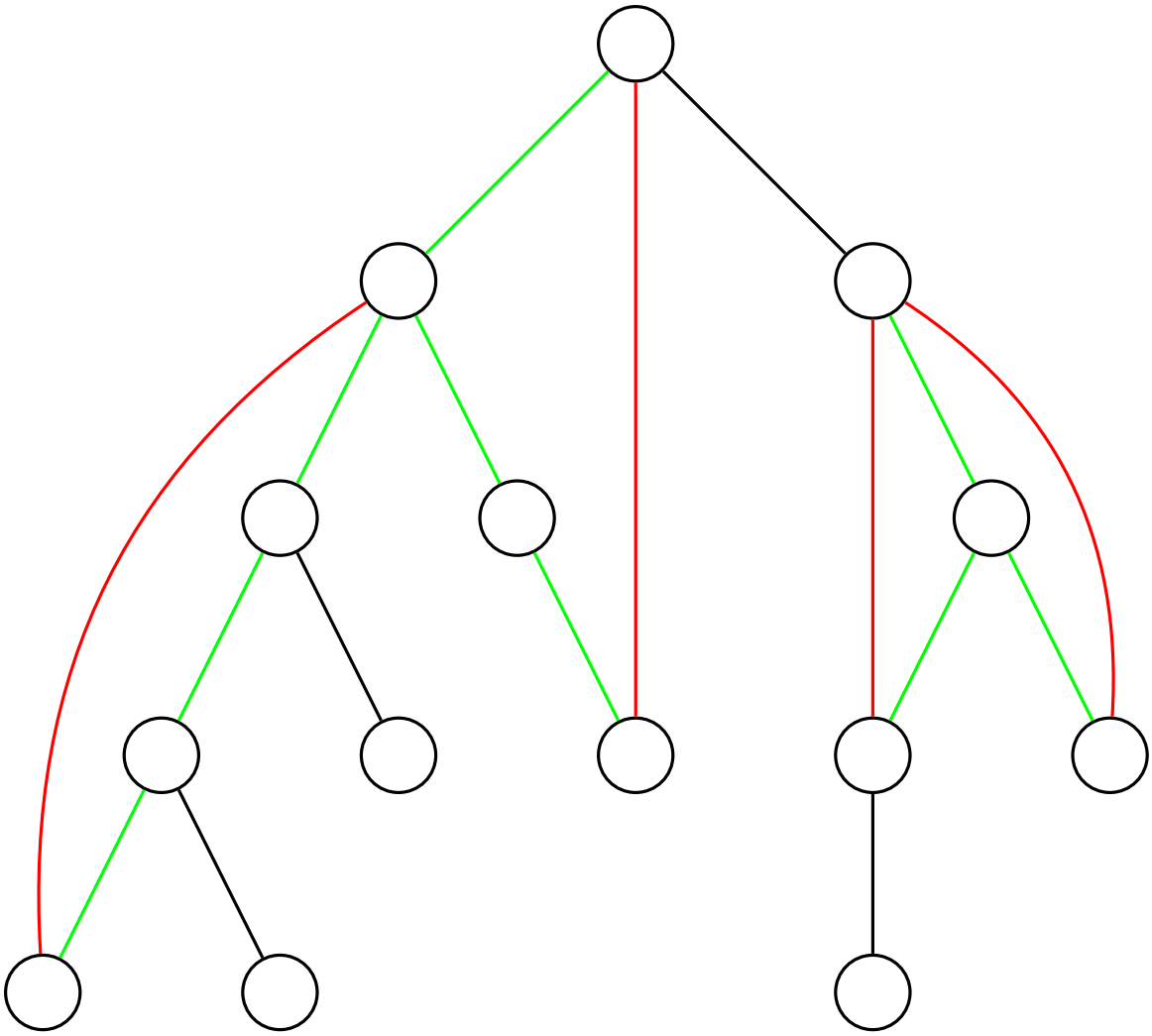
```cpp
    if (!dfn[v])
      tarjan(v, i), low[u] = min(low[u], low[v]);
    else if (vis[v])
      low[u] = min(low[u], dfn[v]);
  }
  if (dfn[u] == low[u]) {
    vector<int> t;
    t.push_back(u);
    while (st.top() != u)
      t.push_back(st.top()), vis[st.top()] = false, st.pop();
    st.pop(), ans.push_back(t);
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  int u, v;
  for (int i = 1; i <= m; i++) {
    cin >> u >> v;
    if (u != v) uadd(u, v);
  }
  for (int i = 1; i <= n; i++)
    if (!dfn[i]) tarjan(i, 0);
  cout << ans.size() << '\n';
  for (int i = 0; i < ans.size(); i++) {
    cout << ans[i].size() << ' ';
    for (int j = 0; j < ans[i].size(); j++) cout << ans[i][j] << ' ';
    cout << '\n';
  }
  return 0;
}
```

差分算法

和 Tarjan 算法 1 类似，我们先求出所有的桥，再差分求出边双连通分量.

首先，对原图进行 DFS.



如上图所示，黑色与绿色边为树边，红色边为非树边．每一条非树边的两个端点都唯一对应了树上的一条由树边构成的简单路径，我们说这条非树边 **覆盖** 了这条简单路径上所有的边．

在图中，绿色的树边 **至少** 被一条非树边覆盖，黑色的树边不被 **任何** 非树边覆盖．

显然，**非树边** 和 **绿色的树边** 一定不是桥，**黑色的树边** 一定是桥．

首先考虑一个暴力的做法，对于每一条非树边，都逐个地将它覆盖的每一条树边置成绿色，时间复杂度为 $O(nm)$．

考虑用差分优化．对于每一条非树边，在其树上深度较小的端点处打上 `-1` 标记，在其树上深度较大的端点处打上 `+1` 标记，然后 $O(n)$ 求出每个点的子树内部的标记和．

对于一个点 u，其子树内部的标记之和等于覆盖了 u 和 fa_u 之间的树边的非树边数量．若这个值等于 0，则 u 和 fa_u 之间的树边是 **桥**．

再用 DFS 求出边双连通分量．

时间复杂度 $O(n+m)$．

示例代码

```cpp
#include <cstring>
#include <iostream>
#include <vector>
```

```cpp
using namespace std;
#define min(x, y) (x < y ? x : y)
#define max(x, y) (x > y ? x : y)
constexpr int N = 5e5 + 5, M = 2e6 + 5;
int n, m;

struct edge {
  int to, nt;
} e[M << 1];

int hd[N], tot;

void add(int u, int v) { e[++tot] = {v, hd[u]}, hd[u] = tot; }

void uadd(int u, int v) { add(u, v), add(v, u); }  // 链式前向星

using ll = long long;
#define P(x, y) ((ll)min(x, y) * N + (ll)max(x, y))
constexpr int hmod = 1e5 + 7;

struct hash {
  vector<ll> v1[hmod];
  vector<short> v2[hmod];

  short &operator[](ll x) {
    int y = x % hmod;
    for (int i = 0; i < v1[y].size(); i++)
      if (v1[y][i] == x) return v2[y][i];
    v1[y].push_back(x), v2[y].push_back(0);
    return v2[y].back();
  }
} re, be;  // 用 vector 实现 hash 表，因为本题时空限制均比较紧

// re 判断是否有重边，be 记录边是不是桥

// #define P(x, y) {min(x, y), max(x, y)}
// using pii = pair<int, int>;
// map<pii, int> re, be; // 不紧时可以用 map 实现 hash 表

int dep[N], bz[N], sum[N];  // 记录深度、差分值、子树查分和
int vis[N], fa[N];          // 记录是否访问过，分组编号

void dfs(int x, int pre) {  // 计算每个点的深度和单点查分
  if (dep[x] < dep[pre]) bz[x]++, bz[pre]--;  // 回到祖先，更改差分值
  if (dep[x]) return;
  dep[x] = dep[pre] + 1;
  for (int i = hd[x]; i; i = e[i].nt) dfs(e[i].to, x);
}

int dfs2(int x, int pre) {  // 处理子树查分
  if (vis[x] == 1) return sum[x];
  vis[x] = 1, sum[x] = bz[x];
```

```
  for (int i = hd[x]; i; i = e[i].nt) {
    int v = e[i].to;
    if (dep[v] > dep[x] && !vis[v]) sum[x] += dfs2(v, x);
  }
  if (sum[x] == 0 && re[P(x, pre)] == 1) be[P(x, pre)] = 1;  // 记录桥
  return sum[x];
}

int cnt;
vector<int> ans[N];

void dfs3(int x) {  // 计算双连通分量
  if (fa[x]) return;
  ans[cnt].push_back(x), fa[x] = cnt;
  for (int i = hd[x]; i; i = e[i].nt) {
    int v = e[i].to;
    if (be[P(x, v)] != 1) dfs3(v);  // 不是桥，递归处理子树
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  int u, v;
  for (int i = 1; i <= m; i++) cin >> u >> v, uadd(u, v), re[P(u, v)]++;
  for (int i = 1; i <= n; i++)
    if (!dep[i]) dfs(i, 0);
  for (int i = 1; i <= n; i++)
    if (!vis[i]) dfs2(i, 0);
  for (int i = 1; i <= n; i++)
    if (!fa[i]) cnt++, dfs3(i);
  cout << cnt << '\n';
  for (int i = 1; i <= cnt; i++) {
    cout << ans[i].size() << ' ';
    for (int j = 0; j < ans[i].size(); j++) cout << ans[i][j] << ' ';
    cout << '\n';
  }
  return 0;
}
```

#2788.「CEOI2015 Day1」管道[2]

给出一个 N 点 M 边的无向图，不保证连通. 将每个连通块视为子图，请求出每一个子图中的桥. **你只有 16 MB 的内存空间.**

题解

此题最大的特征在于，你存不下所有的边.

考虑优化存边，若一条非树边被另一条非树边完全覆盖，则这条边无用.

用并查集维护即可.

点双连通分量

例题：洛谷 P8435【模板】点双连通分量[3]

对于一个 n 个节点 m 条无向边的图，请输出其点双连通分量的个数，并且输出每个点双连通分量.

Tarjan 算法

需要先学习割点，可以先参见 割点和桥→11.13.3@p2823 的割点部分.

先给出两个性质：

1. 两个点双最多只有一个公共点，且一定是割点.
2. 对于一个点双，它在 DFS 搜索树中 dfn 值最小的点一定是割点或者树根.

我们根据第二个性质，分类讨论：

1. 当这个点为割点时，它一定是点双连通分量的根，因为一旦包含它的父节点，他仍然是割点.
2. 当这个点为树根时：
    1. 有两个及以上子树，它是一个割点.
    2. 只有一个子树，它是一个点双连通分量的根.
    3. 它没有子树，视作一个点双.

示例代码

```cpp
#include <iostream>
#include <vector>
using namespace std;
constexpr int N = 5e5 + 5, M = 2e6 + 5;
int n, m;

struct edge {
  int to, nt;
} e[M << 1];

int hd[N], tot = 1;

void add(int u, int v) { e[++tot] = edge{v, hd[u]}, hd[u] = tot; }

void uadd(int u, int v) { add(u, v), add(v, u); }

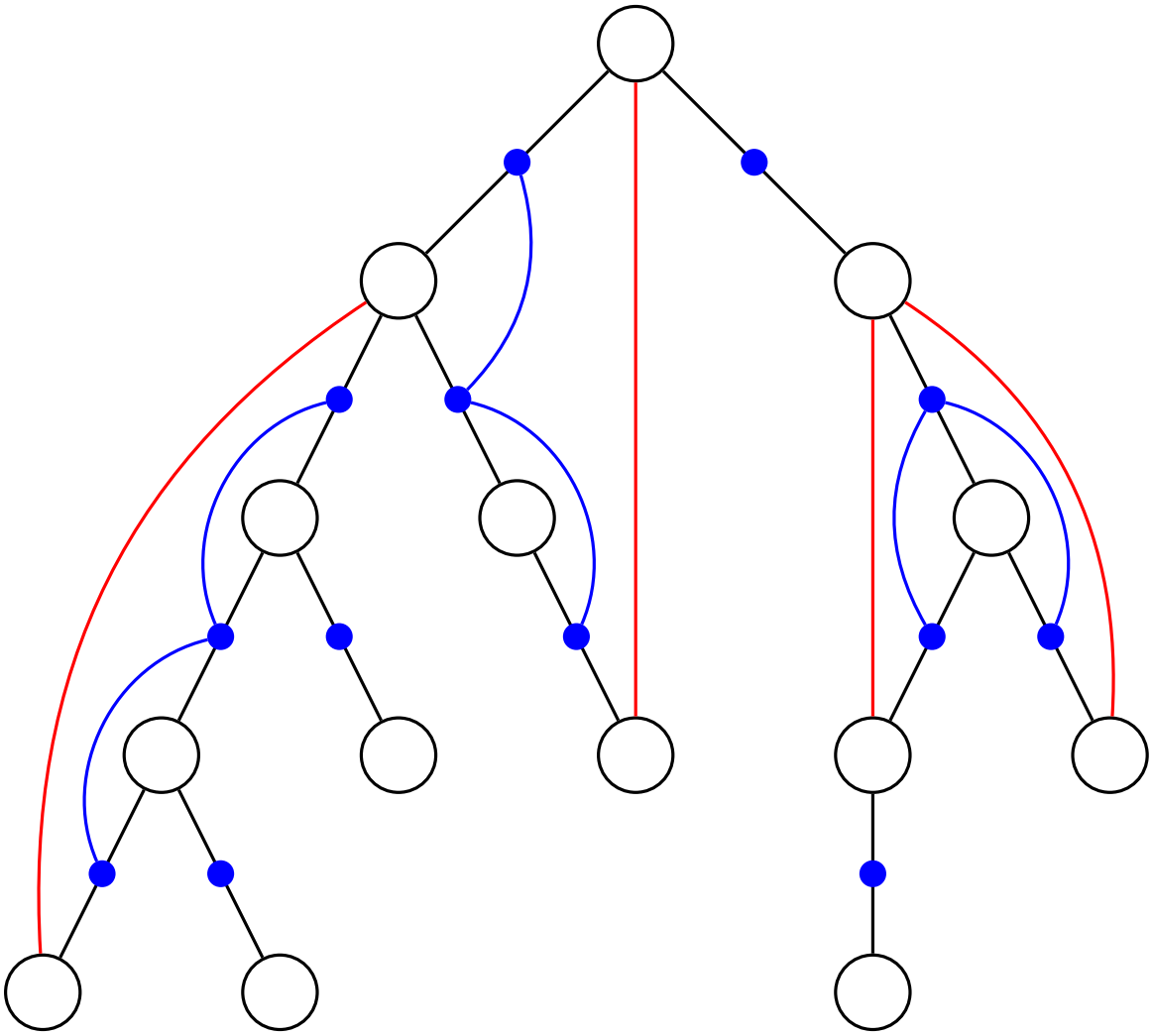
int ans;
int dfn[N], low[N], bcc_cnt;
int sta[N], top, cnt;
bool cut[N];
vector<int> dcc[N];
int root;

void tarjan(int u) {
  dfn[u] = low[u] = ++bcc_cnt, sta[++top] = u;
  if (u == root && hd[u] == 0) {
    dcc[++cnt].push_back(u);
    return;
  }
  int f = 0;
```

```
  for (int i = hd[u]; i; i = e[i].nt) {
    int v = e[i].to;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
      if (low[v] >= dfn[u]) {
        if (++f > 1 || u != root) cut[u] = true;
        cnt++;
        do dcc[cnt].push_back(sta[top--]);
        while (sta[top + 1] != v);
        dcc[cnt].push_back(u);
      }
    } else
      low[u] = min(low[u], dfn[v]);
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  int u, v;
  for (int i = 1; i <= m; i++) {
    cin >> u >> v;
    if (u != v) uadd(u, v);
  }
  for (int i = 1; i <= n; i++)
    if (!dfn[i]) root = i, tarjan(i);
  cout << cnt << '\n';
  for (int i = 1; i <= cnt; i++) {
    cout << dcc[i].size() << ' ';
    for (int j = 0; j < dcc[i].size(); j++) cout << dcc[i][j] << ' ';
    cout << '\n';
  }
  return 0;
}
```

差分算法



如上图所示，黑色边为树边，红色边为非树边，每一条非树边的两个端点都唯一对应了树上由树边构成的一条简单路径.

考虑一张新图，新图中的每一个点对应原图中的每一条树边（在图中用蓝色点表示）. 对于原图中的每一条非树边，将这条非树边对应的树上简单路径中的所有边在新图中对应的蓝点连成一个连通块（在图中用蓝色的边体现出来）.

这样，一个点若 **不是** 割点，当且仅当与其相连的所有边在新图中对应的蓝点都 **属于** 同一个连通块.

两个点 **是** 点双连通，当且仅当它们在原图的树上路径中的所有边在新图中对应的蓝点都 **属于** 同一个连通块，即图中的每个蓝点构成的连通块都是一个点双连通分量.

蓝点间的连通关系可以用与求边双连通时用到的差分类似的方法维护，时间复杂度 $O(n+m)$.

外部参考资料

[1] 例题：洛谷 P8436【模版】边双连通分量

[2] #2788.「CEOI2015 Day1」管道

[3] 例题：洛谷 P8435【模板】点双连通分量

11.13.3 割点和桥

Authors: Ir1d, sshwy, GavinZhengOI, Planet6174, ouuan, Marcythm, ylxmf2005, 0xis-cn

相关阅读：双连通分量→11.13.2@p2813

割点和桥更严谨的定义参见 图论相关概念→11.2@p2598.

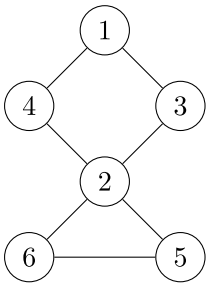
割点

> 对于一个无向图，如果把一个点删除后这个图的极大连通分量数增加了，那么这个点就是这个图的割点（又称割顶）.

过程

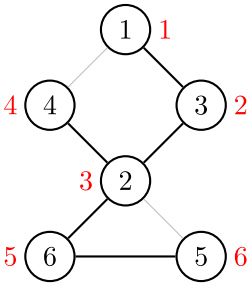
如果我们尝试删除每个点，并且判断这个图的连通性，那么复杂度会特别的高. 所以要介绍一个常用的算法：Tarjan.

首先，我们上一个图：



很容易的看出割点是 2，而且这个图仅有这一个割点.

首先，我们按照 DFS 序给他打上时间戳（访问的顺序）.



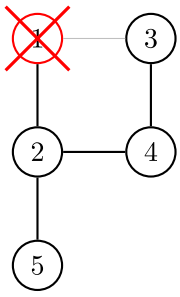
这些信息被我们保存在一个叫做 `dfn` 的数组中.

还需要另外一个数组 `low`，用它来存储不经过其父亲能到达的最小的时间戳.

例如 `low[2]` 是 1，`low[5]` 和 `low[6]` 是 3.

然后我们开始 DFS，我们判断某个点是否是割点的根据是：对于某个顶点 u，如果存在至少一个顶点 v（u 的儿子），使得 $low_v \geq dfn_u$，即不能回到祖先，那么 u 点为割点.

此根据惟独不适用于搜索的起始点，其需要特殊考虑：若该点不是割点，则其他路径亦能到达全部结点，因此从起始点只「向下搜了一次」，即在搜索树内仅有一个子结点. 如果在搜索树内有两个及以上的儿子，那么他一定是割点了（设想上图从 2 开始搜索，搜索树内应有两个子结点：3 或 4 及 5 或 6）. 如果只有一个儿子，那么把它删掉，不会有任何的影响. 比如下面这个图，此处形成了一个环.



我们在访问 1 的儿子时候，假设先 DFS 到了 2，然后标记用过，然后递归往下，来到了 4，4 又来到了 3，当递归回溯的时候，会发现 3 已经被访问过了，所以不是割点.

更新 `low` 的伪代码如下：

$$
\begin{aligned}
&\textbf{if } v \text{ is a son of } u \\
&\qquad \text{low}_u = \min(\text{low}_u, \text{low}_v) \\
&\textbf{else} \\
&\qquad \text{low}_u = \min(\text{low}_u, \text{dfn}_v)
\end{aligned}
$$

例题

洛谷 P3388【模板】割点（割顶）[1]

例题代码

```cpp
/*
洛谷 P3388 【模板】割点（割顶）
*/
#include <iostream>
#include <vector>
using namespace std;
int n, m;  // n: 点数 m: 边数
int dfn[100001], low[100001], idx, res;
// dfn: 记录每个点的时间戳
// low: 能不经过父亲到达最小的编号，idx: 时间戳，res: 答案数量
bool vis[100001], flag[100001];  // flag: 答案 vis: 标记是否重复
vector<int> edge[100001];        // 存图用的

void Tarjan(int u, int fa) {  // u 当前点的编号，fa 自己爸爸的编号
  vis[u] = true;              // 标记
  low[u] = dfn[u] = ++idx;    // 打上时间戳
  int child = 0;              // 每一个点儿子数量
  for (const auto &v : edge[u]) {  // 访问这个点的所有邻居 （C++11）
    if (!vis[v]) {
      child++;                        // 多了一个儿子
      Tarjan(v, u);                   // 继续
      low[u] = min(low[u], low[v]);  // 更新能到的最小节点编号
      if (fa != u && low[v] >= dfn[u] && !flag[u]) {  // 主要代码
        // 如果不是自己，且不通过父亲返回的最小点符合割点的要求，并且没有被标记过
        // 要求即为：删了父亲连不上去了，即为最多连到父亲
        flag[u] = true;
        res++;  // 记录答案
      }
    } else if (v != fa) {
      // 如果这个点不是自己的父亲，更新能到的最小节点编号
      low[u] = min(low[u], dfn[v]);
```

```
    }
  }
  // 主要代码，自己的话需要 2 个儿子才可以
  if (fa == u && child >= 2 && !flag[u]) {
    flag[u] = true;
    res++;  // 记录答案
  }
}

int main() {
  cin >> n >> m;                    // 读入数据
  for (int i = 1; i <= m; i++) {  // 注意点是从 1 开始的
    int x, y;
    cin >> x >> y;
    edge[x].push_back(y);
    edge[y].push_back(x);
  }  // 使用 vector 存图
  for (int i = 1; i <= n; i++)  // 因为 Tarjan 图不一定连通
    if (!vis[i]) {
      idx = 0;        // 时间戳初始为 0
      Tarjan(i, i);  // 从第 i 个点开始，父亲为自己
    }
  cout << res << endl;
  for (int i = 1; i <= n; i++)
    if (flag[i]) cout << i << " ";  // 输出结果
  return 0;
}

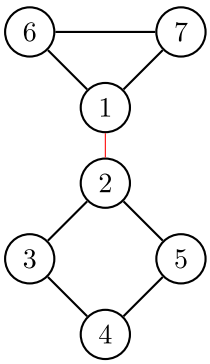
```

割边（无重边时）

和割点差不多，叫做桥.

> 对于一个无向图，如果删掉一条边后图中的连通分量数增加了，则称这条边为桥或者割边. 严谨来说，就是：假设有连通图 $G=\{V,E\}$，e 是其中一条边（即 $e\in E$），如果 $G-e$ 是不连通的，则边 e 是图 G 的一条割边（桥）.

比如说，下图中，



红色的边就是割边.

过程

和割点差不多，只要改一处：$low_v > dfn_u$ 就可以了，而且不需要考虑根节点的问题.

割边是和是不是根节点没关系的, 原来我们求割点的时候是指点 v 是不可能不经过父节点 u 为回到祖先节点（包括父节点），所以顶点 u 是割点. 如果 $low_v = dfn_u$ 表示还可以回到父节点，如果顶点 v 不能回到祖先也没有另外一条回到父亲的路，那么 $u - v$ 这条边就是割边.

实现

下面代码实现了对 **无重边** 的无向图求割边，其中，当 `isbridge[x]` 为真时，`(father[x],x)` 为一条割边.

- C++

```cpp
int low[MAXN], dfn[MAXN], idx;
bool isbridge[MAXN];
vector<int> G[MAXN];
int cnt_bridge;
int father[MAXN];

void tarjan(int u, int fa) {
  father[u] = fa;
  low[u] = dfn[u] = ++idx;
  for (const auto &v : G[u]) {
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u]) {
        isbridge[v] = true;
        ++cnt_bridge;
      }
    } else if (v != fa) {
      low[u] = min(low[u], dfn[v]);
    }
  }
}
```

- Python

```python
low = [0] * MAXN
dfn = [0] * MAXN
idx = 0
isbridge = [False] * MAXN
G = [[0 for i in range(MAXN)] for j in range(MAXN)]
cnt_bridge = 0
father = [0] * MAXN


def tarjan(u, fa):
    father[u] = fa
    idx = idx + 1
    low[u] = dfn[u] = idx
    for i in range(0, len(G[u])):
        v = G[u][i]
        if dfn[v] == False:
            tarjan(v, u)
            low[u] = min(low[u], low[v])
```

```
            if low[v] > dfn[u]:
                isbridge[v] = True
                cnt_bridge = cnt_bridge + 1
        elif v != fa:
            low[u] = min(low[u], dfn[v])
```

割边（有重边时）

然而，上述无重边时的做法在有重边的无向图上是有问题的.

因为两节点间可能不止有一条边，此时它们都不会是桥.

过程

一种思路是将参数 `fa` 改为刚刚走过的边的编号（每条边的编号一致）即可，即将「不用父节点更新」改为「不用来时的边更新」.

另一种更简单的思路是设立一个标记判断是否已有一条边抵达父节点，标记后再访问到父节点时正常更新.

下面代码实现了对可能 **有重边** 的无向图求割边.

- C++

```cpp
int low[MAXN], dfn[MAXN], idx;
bool isbridge[MAXN];
vector<int> G[MAXN];
int cnt_bridge;
int father[MAXN];

void tarjan(int u, int fa) {
  bool flag = false;
  father[u] = fa;
  low[u] = dfn[u] = ++idx;
  for (const auto &v : G[u]) {
    if (!dfn[v]) {
      tarjan(v, u);
      low[u] = min(low[u], low[v]);
      if (low[v] > dfn[u]) {
        isbridge[v] = true;
        ++cnt_bridge;
      }
    } else {
      if (v != fa || flag)
        low[u] = min(low[u], dfn[v]);
      else
        flag = true;
    }
  }
}
```

练习

- P3388【模板】割点（割顶）[2]
- POJ2117 Electricity[3]
- HDU4738 Caocao's Bridges[4]
- HDU2460 Network[5]

- POJ1523 SPF[6]

Tarjan 算法还有许多用途，常用的例如求强连通分量，缩点，还有求 2-SAT 的用途等.

外部参考资料

[1] 洛谷 P3388【模板】割点（割顶）

[2] P3388【模板】割点（割顶）

[3] POJ2117 Electricity

[4] HDU4738 Caocao's Bridges

[5] HDU2460 Network

[6] POJ1523 SPF

11.13.4 圆方树

> **Authors:** GitPinkRabbit, Early0v0, Backl1ght, mcendu, ksyx, iamtwz, Xeonacid, kenlig, Menci, Enter-tainer, CCXXXI, hcx2012Git

在阅读下列内容之前，请务必了解 图论相关概念→11.2@p2598 部分.

相关阅读：割点和桥→11.13.3@p2823.

引入

众所周知，树（或森林）有很好的性质，并且容易通过很多常见数据结构维护.

而一般图则没有那么好的性质，所幸有时我们可以把一般图上的某些问题转化到树上考虑.

而圆方树（Block forest 或 Round-square tree）[1]就是一种将图变成树的方法. 本文将介绍圆方树的构建，性质和一些应用.

限于篇幅，本文中有一些结论未经证明，读者可以自行理解或证明.

定义

圆方树最初是处理「仙人掌图」（每条边在不超过一个简单环中的无向图）的一种工具，不过发掘它的更多性质，有时我们可以在一般无向图上使用它.

要介绍圆方树，首先要介绍 **点双连通分量**.

一个 **点双连通图** 的一个定义是：图中任意两不同点之间都有至少两条点不重复的路径.

点不重复既指路径上点不重复（简单路径），也指两条路径的交集为空（当然，路径必然都经过出发点和到达点，这不在考虑范围内）.

可以发现对于只有一个点的图比较难定义它是不是一个点双，这里先不考虑节点数为 1 的图.

一个近乎等价的定义是：不存在割点的图.

这个定义只在图中只有两个点，一条连接它们的边时失效. 它没有割点，但是并不能找到两条不相交的路径，因为只有一条路径.

（也可以理解为那一条路径可以算两次，的确没有交，因为不经过其他点）

虽然原始的定义的确是前者，但是为了方便，我们规定点双图的定义采用后者.

而一个图的 **点双连通分量** 则是一个 **极大点双连通子图**.

与强连通分量等不同，一个点可能属于多个点双，但是一条边属于恰好一个点双（如果定义采用前者则有可能不属于任何点双）.

在圆方树中，原来的每个点对应一个 **圆点**，每一个点双对应一个 **方点**.

[1] 2017 年陈俊锟同学在他的 IOI2017 中国国家集训队论文《〈神奇的子图〉命题报告及其拓展》中定义并命名了圆方树这一结构.

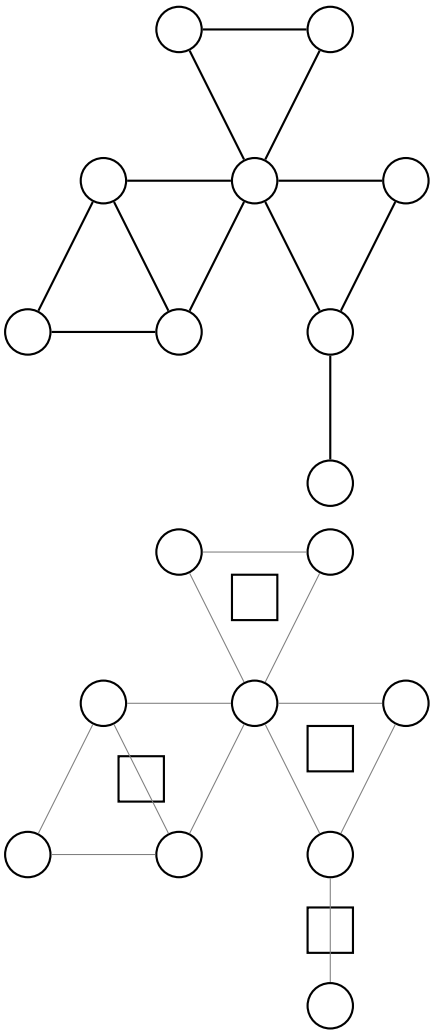
所以共有 $n+c$ 个点，其中 n 是原图点数，c 是原图点双连通分量的个数.

而对于每一个点双连通分量，它对应的方点向这个点双连通分量中的每个点连边.

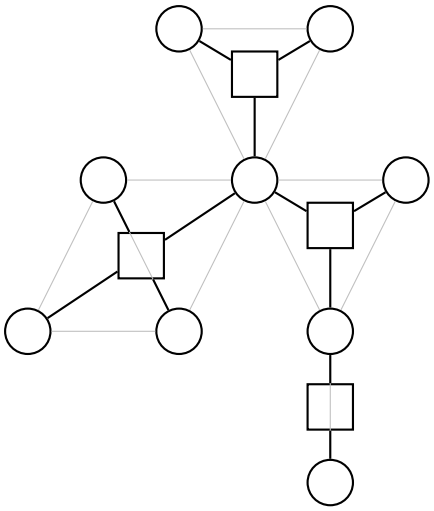
每个点双形成一个「菊花图」，多个「菊花图」通过原图中的割点连接在一起（因为点双的分隔点是割点）.

显然，圆方树中每条边连接一个圆点和一个方点.

下面的图显示了一张图对应的点双和圆方树形态. [2]



[2]陈俊锟，《平凡的圆方树和神奇的（动态）动态规划》，NOI2018 冬令营，第 4 页.



圆方树的点数小于 $2n$，这是因为割点的数量小于 n，所以请注意各种数组大小要开两倍.

其实，如果原图连通，则「圆方树」才是一棵树，如果原图有 k 个连通分量，则它的圆方树也会形成 k 棵树形成的森林.

如果原图中某个连通分量只有一个点，则需要具体情况具体分析，我们在后续讨论中不考虑孤立点.

过程

对于一个图, 如何构造出它的圆方树呢? 首先可以发现如果图不连通, 可以拆分成每个连通子图考虑, 所以我们只考虑连通图.

因为圆方树是基于点双连通分量的, 而点双连通分量又基于割点, 所以只需要用类似求割点的方法即可.

求割点的常用算法是 Tarjan 算法，如果你会了理解下面的内容就很简单了，如果你不会也没关系.

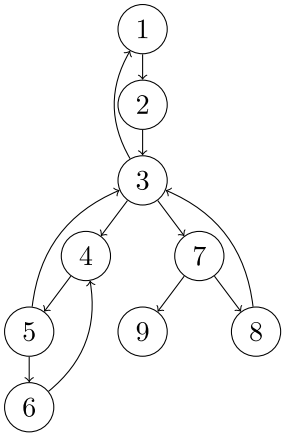
我们跳过 Tarjan 求割点，直接介绍圆方树使用的算法（其实是 Tarjan 的变体）:

对图进行 DFS，并且中间用到了两个关键数组 `dfn` 和 `low`（类似于 Tarjan）.

`dfn[u]` 存储的是节点 u 的 DFS 序，即第一次访问到 u 时它是第几个被访问的节点.

`low[u]` 存储的是节点 u 的 DFS 树中的子树中的某个点 v 通过 **最多一次返祖边或向父亲的树边** 能访问到的点的 **最小** DFS 序.

如果没有听说过 Tarjan 算法可能会有点难理解，让我们举个例子吧:



（可以发现这张图其实和上面图片中的图等价）

这里树边从上至下用直线画出，返祖边从下至上用曲线画出. 节点的编号便是它的 DFS 序.

则有 `low` 数组如下：

i	1	2	3	4	5	6	7	8	9
low[i]	1	1	1	3	3	4	3	3	7

并不是很难理解吧，注意这里 9 的 `low` 是 7，与一些求割点的做法有差异，因为为了方便，我们规定了可以通过父边向上，但主要思想是相同的.

我们可以很容易地写出计算 `dfn` 和 `low` 的 DFS 函数（初始时 `dfn` 数组清零）：

实现

- C++

```cpp
void Tarjan(int u) {
  low[u] = dfn[u] = ++dfc;                // low 初始化为当前节点 dfn
  for (int v : G[u]) {                    // 遍历 u 的相邻节点
    if (!dfn[v]) {                        // 如果未访问过
      Tarjan(v);                          // 递归
      low[u] = std::min(low[u], low[v]);  // 未访问的和 low 取 min
    } else
      low[u] = std::min(low[u], dfn[v]);  // 已访问的和 dfn 取 min
  }
}
```

- Python

```python
def Tarjan(u):
    low[u] = dfn[u] = dfc  # low 初始化为当前节点 dfn
    dfc = dfc + 1
    for v in G[u]:  # 遍历 u 的相邻节点
        if dfn[v] == False:  # 如果未访问过
            Tarjan(v)  # 递归
            low[u] = min(low[u], low[v])  # 未访问的和 low 取 min
        else:
            low[u] = min(low[u], dfn[v])  # 已访问的和 dfn 取 min
```

接下来，我们考虑点双和 DFS 树以及这两个数组之间的关联.

可以发现，每个点双在 DFS 树上是一棵连通子树，并至少包含两个点；特别地，最顶端节点仅往下接一个点.

同时还可以发现每条树边恰好在一个点双内.

我们考虑一个点双在 DFS 树中的最顶端节点 u，在 u 处确定这个点双，因为 u 的子树包含了整个点双的信息.

因为至少有两个点，考虑这个点双的下一个点 v，则有 u，v 之间存在一条树边.

不难发现，此时一定有 $\text{low}[v] = \text{dfn}[u]$.

更准确地说，对于一条树边 $u \to v$，u, v 在同一个点双中，且 u 是这个点双中深度最浅的节点 **当且仅当** $\text{low}[v] = \text{dfn}[u]$.

那么我们可以在 DFS 的过程中确定哪些地方存在点双，但是还不能准确确定一个点双所包含的点集.

这并不难处理，我们可以在 DFS 过程中维护一个栈，存储还未确定所属点双（可能有多个）的节点.

在找到点双时，点双中除了 u 以外的其他的点都集中在栈顶端，只需要不断弹栈直到弹出 v 为止即可.

当然，我们可以同时处理被弹出的节点，只要将其和新建的方点连边即可. 最后还要让 u 和方点连边.

这样就很自然地完成了圆方树的构建，我们可以给方点标号为 $n+1$ 开始的整数，这样可以有效区分圆点和方点.

这部分可能讲述得不够清晰，下面贴出一份代码，附有详尽注释以及帮助理解的输出语句和一份样例，建议读者复制代码并自行实践理解，毕竟代码才是最能帮助理解的（不要忘记开 c++11）.

实现

```cpp
#include <algorithm>
#include <cstdio>
#include <vector>

constexpr int MN = 100005;

int N, M, cnt;
std::vector<int> G[MN], T[MN * 2];

int dfn[MN], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {
  printf("  Enter : #%d\n", u);
  low[u] = dfn[u] = ++dfc;                // low 初始化为当前节点 dfn
  stk[++tp] = u;                          // 加入栈中
  for (int v : G[u]) {                    // 遍历 u 的相邻节点
    if (!dfn[v]) {                        // 如果未访问过
      Tarjan(v);                          // 递归
      low[u] = std::min(low[u], low[v]);  // 未访问的和 low 取 min
      if (low[v] == dfn[u]) {  // 标志着找到一个以 u 为根的点双连通分量
        ++cnt;                 // 增加方点个数
        printf("  Found a New BCC #%d.\n", cnt - N);
        // 将点双中除了 u 的点退栈，并在圆方树中连边
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
          printf("    BCC #%d has vertex #%d\n", cnt - N, x);
        }
        // 注意 u 自身也要连边（但不退栈）
        T[cnt].push_back(u);
        T[u].push_back(cnt);
        printf("    BCC #%d has vertex #%d\n", cnt - N, u);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);  // 已访问的和 dfn 取 min
  }
  printf("  Exit : #%d : low = %d\n", u, low[u]);
  printf("  Stack:\n    ");
  for (int i = 1; i <= tp; ++i) printf("%d, ", stk[i]);
  puts("");
}

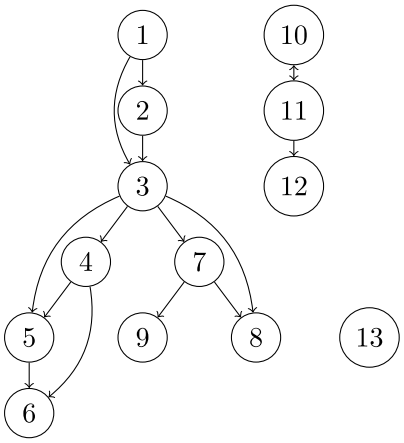
int main() {
  scanf("%d%d", &N, &M);
```

```
  cnt = N;  // 点双 / 方点标号从 N 开始
  for (int i = 1; i <= M; ++i) {
    int u, v;
    scanf("%d%d", &u, &v);
    G[u].push_back(v);  // 加双向边
    G[v].push_back(u);
  }
  // 处理非连通图
  for (int u = 1; u <= N; ++u)
    if (!dfn[u]) Tarjan(u), --tp;
  // 注意到退出 Tarjan 时栈中还有一个元素即根，将其退栈
  return 0;
}
```

提供一个测试用例：

```
13 15
1 2
2 3
1 3
3 4
3 5
4 5
5 6
4 6
3 7
3 8
7 8
7 9
10 11
11 10
11 12
```

这个例子对应的图（包含了重边和孤立点的情况）：



例题

我们讲一些可以使用圆方树求解的例题.

「APIO2018」铁人两项[1]

题意简述

给定一张简单无向图，问有多少对三元组 $\langle s,c,f\rangle$（s,c,f 互不相同）使得存在一条简单路径从 s 出发，经过 c 到达 f.

题解

说到简单路径，就必须提一个关于点双很好的性质：对于一个点双中的两点，它们之间简单路径的并集，恰好完全等于这个点双.

即同一个点双中的两不同点 u,v 之间一定存在一条简单路径经过给定的在同一个点双内的另一点 w.

这个性质的证明：

- 显然如果简单路径出了点双，就不可能再回到这个点双中，否则会和点双的定义冲突.
- 所以我们只需考虑证明一个点双连通图中任意三不同点 u,v,c，必存在一条从 u 到 v 的简单路径经过 c.
- 首先排除点数为 2 的情况，它满足这个性质，但是无法取出 3 个不同点.
- 对于余下的情况，考虑建立网络流模型，源点向 c 连容量为 2 的边，u 和 v 向汇点连容量为 1 的边.
- 原图中的双向边 $\langle x,y\rangle$，变成 x 向 y 连一条容量为 1 的边，y 也向 x 连一条容量为 1 的边.
- 最后，给除了源点，汇点和 c 之外的每个点赋上 1 的容量，这可以通过拆点实现.
- 因为源点到 c 的边的容量为 2，那么如果这个网络最大流为 2，则证明一定有路径经过 c.
- 考虑最大流最小割定理，显然最小割小于等于 2，接下来只要证最小割大于 1.
- 这等价于证明割掉任意一条容量为 1 的边，是无法使源点和汇点不连通的.
- 考虑割掉 u 或 v 与汇点连接的点，根据点双的第一种定义，必然存在简单路径从 c 到另一个没割掉的点.
- 考虑割掉一个节点拆点形成的边，这等价于删除一个点，根据点双的第二种定义，余下的图仍然连通.
- 考虑割掉一条由原先的边建出的边，这等价于删除一条边，这比删除一个点更弱，显然存在路径.
- 所以我们证明了最小割大于 1，即最大流等于 2. 证毕.

这个结论能告诉我们什么呢？它告诉了我们：考虑两圆点在圆方树上的路径，与路径上经过的方点相邻的圆点的集合，就等于原图中两点简单路径上的点集.

回到题目，考虑固定 s 和 f，求合法的 c 的数量，显然有合法 c 的数量等于 s,f 之间简单路径的并集的点数减 2（去掉 s,f 本身）.

那么，对原图建出圆方树后，两点之间简单路径的点数，就和它们在圆方树上路径经过的方点（点双）和圆点的个数有关.

接下来是圆方树的一个常用技巧：路径统计时，点赋上合适的权值.

本题中，每个方点的权值为对应点双的大小，而每个圆点权值为 -1.

这样赋权后则有两圆点间圆方树上路径点权和，恰好等于原图中简单路径并集大小减 2.

问题转化为统计圆方树上 $\sum$ 两圆点路径权值和.

换个角度考虑，改为统计每一个点对答案的贡献，即权值乘以经过它的路径条数，这可以通过简单的树形 DP 求出.

最后，不要忘记处理图不连通的情况. 下面是对应代码：

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

constexpr int MN = 100005;

int N, M, cnt;
std::vector<int> G[MN], T[MN * 2];
long long Ans;

int dfn[MN], low[MN], dfc, num;
int stk[MN], tp;

int wgh[MN * 2];

void Tarjan(int u) {  // 求点双
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  ++num;
  for (int v : G[u]) {
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        wgh[++cnt] = 0;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
          ++wgh[cnt];
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
        ++wgh[cnt];
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

int vis[MN * 2], siz[MN * 2];

void DFS(int u, int fz) {  // dfs求值
  vis[u] = 1;
```

```
45    siz[u] = (u <= N);
46    for (int v : T[u])
47      if (v != fz) {
48        DFS(v, u);
49        Ans += 2ll * wgh[u] * siz[u] * siz[v];
50        siz[u] += siz[v];
51      }
52    Ans += 2ll * wgh[u] * siz[u] * (num - siz[u]);
53  }
54
55  using std::cin;
56  using std::cout;
57
58  int main() {
59    cin.tie(nullptr)->sync_with_stdio(false);
60    cin >> N >> M;
61    for (int u = 1; u <= N; ++u) wgh[u] = -1;
62    cnt = N;
63    for (int i = 1; i <= M; ++i) {
64      int u, v;
65      cin >> u >> v;
66      G[u].push_back(v);
67      G[v].push_back(u);
68    }
69    for (int u = 1; u <= N; ++u)
70      if (!dfn[u]) {
71        num = 0;
72        Tarjan(u), --tp;
73        DFS(u, 0);
74      }
75    cout << Ans << '\n';
76    return 0;
77  }
```

顺带一提，刚刚的测试用例在这题的答案是 212.

Codeforces #487 E. Tourists[2]

题意简述

给定一张简单无向连通图，要求支持两种操作：

1. 修改一个点的点权.
2. 询问两点之间所有简单路径上点权的最小值.

题解

同样地，我们建出原图的圆方树，令方点权值为相邻圆点权值的最小值，问题转化为求路径上最小值.

路径最小值可以使用树链剖分和线段树维护，但是修改呢?

一次修改一个圆点的点权，需要修改所有和它相邻的方点，这样很容易被卡到 $O(n)$ 个修改.

这时我们利用圆方树是棵树的性质，令方点权值为自己的儿子圆点的权值最小值，这样的话修改时只需要修改父亲方点.

对于方点的维护，只需要对每个方点开一个 `multiset` 维护权值集合即可.

需要注意的是查询时若 LCA 是方点，则还需要查 LCA 的父亲圆点的权值.

注意：圆方树点数要开原图的两倍，否则会数组越界.

参考代码

```cpp
#include <algorithm>
#include <iostream>
#include <set>
#include <vector>

constexpr int MN = 100005;
constexpr int MS = 524288;
constexpr int Inf = 0x7fffffff;

int N, M, Q, cnt;
int w[MN * 2];
std::vector<int> G[MN], T[MN * 2];
std::multiset<int> S[MN * 2];

int dfn[MN * 2], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  for (int v : G[u]) {
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        ++cnt;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

```

```cpp
int idf[MN * 2], faz[MN * 2], siz[MN * 2], dep[MN * 2], son[MN * 2],
    top[MN * 2];

void DFS0(int u, int fz) {
  faz[u] = fz, dep[u] = dep[fz] + 1, siz[u] = 1;
  for (int v : T[u])
    if (v != fz) {
      DFS0(v, u);
      siz[u] += siz[v];
      if (siz[son[u]] < siz[v]) son[u] = v;
    }
}

void DFS1(int u, int fz, int tp) {
  dfn[u] = ++dfc, idf[dfc] = u, top[u] = tp;
  if (son[u]) DFS1(son[u], u, tp);
  for (int v : T[u])
    if (v != fz && v != son[u]) DFS1(v, u, v);
}

#define li (i << 1)
#define ri (i << 1 | 1)
#define mid ((l + r) >> 1)
#define ls li, l, mid
#define rs ri, mid + 1, r

int dat[MS];

void Build(int i, int l, int r) {  // 建树
  if (l == r) {
    dat[i] = w[idf[l]];
    return;
  }
  Build(ls), Build(rs);
  dat[i] = std::min(dat[li], dat[ri]);
}

void Mdf(int i, int l, int r, int p, int x) {  // 获取最小值
  if (l == r) {
    dat[i] = x;
    return;
  }
  if (p <= mid)
    Mdf(ls, p, x);
  else
    Mdf(rs, p, x);
  dat[i] = std::min(dat[li], dat[ri]);
}

int Qur(int i, int l, int r, int a, int b) {  // 查询
  if (r < a || b < l) return Inf;
```

```
  if (a <= l && r <= b) return dat[i];
  return std::min(Qur(ls, a, b), Qur(rs, a, b));
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> N >> M >> Q;
  for (int i = 1; i <= N; ++i) cin >> w[i];
  cnt = N;
  for (int i = 1; i <= M; ++i) {
    int u, v;
    cin >> u >> v;
    G[u].push_back(v);
    G[v].push_back(u);
  }
  Tarjan(1), DFS0(1, 0), dfc = 0, DFS1(1, 0, 1);
  for (int i = 1; i <= N; ++i)
    if (faz[i]) S[faz[i]].insert(w[i]);
  for (int i = N + 1; i <= cnt; ++i) w[i] = *S[i].begin();
  Build(1, 1, cnt);
  for (int q = 1; q <= Q; ++q) {
    char opt[3];
    int x, y;
    cin >> opt >> x >> y;
    if (*opt == 'C') {
      Mdf(1, 1, cnt, dfn[x], y);
      if (faz[x]) {
        int u = faz[x];
        S[u].erase(S[u].lower_bound(w[x]));
        S[u].insert(y);
        if (w[u] != *S[u].begin()) {
          w[u] = *S[u].begin();
          Mdf(1, 1, cnt, dfn[u], w[u]);
        }
      }
      w[x] = y;
    } else {
      int Ans = Inf;
      while (top[x] != top[y]) {
        if (dep[top[x]] < dep[top[y]]) std::swap(x, y);
        Ans = std::min(Ans, Qur(1, 1, cnt, dfn[top[x]], dfn[x]));
        x = faz[top[x]];
      }
      if (dfn[x] > dfn[y]) std::swap(x, y);
      Ans = std::min(Ans, Qur(1, 1, cnt, dfn[x], dfn[y]));
      if (x > N) Ans = std::min(Ans, w[faz[x]]);
      cout << Ans << '\n';
    }
```

```
142   }
143   return 0;
144 }
```

「SDOI2018」战略游戏[3]

题意简述

给出一个简单无向连通图．有 q 次询问：

每次给出一个点集 S（$2 \leq |S| \leq n$），问有多少个点 u 满足 $u \notin S$ 且删掉 u 之后 S 中的点不全在一个连通分量中．

每个测试点有多组数据．

题解

先建出圆方树，则变为询问 S 在圆方树上对应的连通子图中的圆点个数减去 $|S|$．

如何计算连通子图中的圆点个数？有一个方法：

把圆点的权值放到它和它的父亲方点的边上，问题转化为求边权和，这个问题可以参考「SDOI2015」寻宝游戏[4] 的一种解法．

即把 S 中的点按照 DFS 序排序，计算排序后相邻两点的距离和（还包括首尾两点之间的距离），答案就是距离和的一半，因为每条边只被经过两次．

最后，如果子图中的深度最浅的节点是圆点，答案还要加上 1，因为我们没有统计到它．

因为有多组数据，要注意初始化数组．

参考代码

```
#include <algorithm>
#include <iostream>
#include <vector>

constexpr int MN = 100005;

int N, M, Q, cnt;
std::vector<int> G[MN], T[MN * 2];

int dfn[MN * 2], low[MN], dfc;
int stk[MN], tp;

void Tarjan(int u) {  // 求点双，准备建树
  low[u] = dfn[u] = ++dfc;
  stk[++tp] = u;
  for (int v : G[u]) {
    if (!dfn[v]) {
```

```
      Tarjan(v);
      low[u] = std::min(low[u], low[v]);
      if (low[v] == dfn[u]) {
        ++cnt;
        for (int x = 0; x != v; --tp) {
          x = stk[tp];
          T[cnt].push_back(x);
          T[x].push_back(cnt);
        }
        T[cnt].push_back(u);
        T[u].push_back(cnt);
      }
    } else
      low[u] = std::min(low[u], dfn[v]);
  }
}

int dep[MN * 2], faz[MN * 2][18], dis[MN * 2];

void DFS(int u, int fz) {
  dfn[u] = ++dfc;
  dep[u] = dep[faz[u][0] = fz] + 1;
  dis[u] = dis[fz] + (u <= N);
  for (int j = 0; j < 17; ++j) faz[u][j + 1] = faz[faz[u][j]][j];
  for (int v : T[u])
    if (v != fz) DFS(v, u);
}

int LCA(int x, int y) {  // 最近公共祖先
  if (dep[x] < dep[y]) std::swap(x, y);
  for (int j = 0, d = dep[x] - dep[y]; d; ++j, d >>= 1)
    if (d & 1) x = faz[x][j];
  if (x == y) return x;
  for (int j = 17; ~j; --j)
    if (faz[x][j] != faz[y][j]) x = faz[x][j], y = faz[y][j];
  return faz[x][0];
}

using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int Ti;
  cin >> Ti;
  while (Ti--) {
    cin >> N >> M;
    for (int i = 1; i <= N; ++i) {
      G[i].clear();
      dfn[i] = low[i] = 0;
    }
```

```
    for (int i = 1; i <= N * 2; ++i) T[i].clear();
    for (int i = 1, x, y; i <= M; ++i) {
      cin >> x >> y;
      G[x].push_back(y);
      G[y].push_back(x);
    }
    cnt = N;
    dfc = 0, Tarjan(1), --tp;
    dfc = 0, DFS(1, 0);
    cin >> Q;
    while (Q--) {
      static int S, A[MN];
      cin >> S;
      int Ans = -2 * S;
      for (int i = 1; i <= S; ++i) cin >> A[i];
      std::sort(A + 1, A + S + 1, [](int i, int j) { return dfn[i]
< dfn[j]; });
      for (int i = 1; i <= S; ++i) {
        int u = A[i], v = A[i % S + 1];
        Ans += dis[u] + dis[v] - 2 * dis[LCA(u, v)];
      }
      if (LCA(A[1], A[S]) <= N) Ans += 2;
      cout << Ans / 2 << '\n';
    }
  }
  return 0;
}
```

习题

- UVa 1464 Traffic Real Time Query[5]
- 洛谷 P4320 道路相遇[6]
- 洛谷 P10517 国土规划[7]

外部链接

immortalCO，圆方树——处理仙人掌的利器[8]，Universal OJ.

参考资料与注释

[1] 「APIO2018」铁人两项

[2] Codeforces #487 E. Tourists

[3] 「SDOI2018」战略游戏

[4] 「SDOI2015」寻宝游戏

[5] UVa 1464 Traffic Real Time Query

[6] 洛谷 P4320 道路相遇

[7] 洛谷 P10517 国土规划

[8] 圆方树——处理仙人掌的利器

11.13.5 点/边连通度

Authors: jifbt, Mayuri0v0

定义

以下内容的定义，请参见 图论相关概念→11.2@p2598：

- 边连通度、边割集；
- 点连通度、点割集；
- 团.

性质

Whitney 不等式

Whitney 不等式（1932）给出了点连通度 κ、边连通度 λ 和最小度 δ 之间的关系：

$$\kappa \leq \lambda \leq \delta$$

证明

直觉上，如果有一个大小为 λ 的边割集，其中每一条边任选一个端点，就可以得到一个大小为 λ 的点割集，所以第一个不等式成立.

与度最小的结点（如有多个，任选一个）相邻的所有边构成大小为 δ 的边割集，所以第二个不等式也成立.

这个不等式不能改进；换言之，对每个满足它的三元组，均可以找出满足这个三元组的图.

构造

把两个大小为 $\delta+1$ 的团用 λ 条边连起来，使两个团分别有 λ 和 κ 个不同的结点被连在这些边上.

Menger 定理

由 最大流最小割定理→11.23.3@p2932（又名 Ford–Fulkerson 定理）可推出，两点间的不相交（指两两没有公共边）路径的最大数量等于割集的最小大小（这个推论又叫 **Menger 定理**——译者注）.

计算

以下图的边权均为 1.

用最大流计算边连通度

枚举点对 (s,t)，以 s 为源点，t 为汇点跑边权为 1 的最大流. 需要 $O(n^2)$ 次最大流，如果使用 Edmonds–Karp 算法，复杂度为 $O(|V|^3|E|^2)$. 使用 Dinic 算法可以更优，复杂度为 $O(|V|^2|E|\min(|V|^{2/3},|E|^{1/2}))$.

全局最小割

使用 Stoer–Wagner 算法→11.23.6@p2945 只需跑一次无源汇最小割即可. 复杂度为 $O(|V||E|+|V|^2\log|V|)$，一般可近似看作 $O(|V|^3)$.

点连通度

仍然枚举点对，这次把每个非源汇的点 x 拆成两个点 x_1 和 x_2，并连边 (x_1,x_2). 把原图中所有边 (u,v) 换成两条边 (u_2,v_1) 和 (v_2,u_1). 此时最大流等于 s、t 之间的最小点割集大小（又称局部点连通度）. 复杂度与用最大流计算边连通度相同.

本页面译自博文 Рёберная связность. Свойства и нахождение[1]、Вершинная связность. Свойства и нахождение[2] 与其英文翻译版 Edge connectivity/Vertex connectivity[3]. 其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0.

延伸阅读

- 论文 *Connectivity Algorithms*[4] 介绍了近年来连通度计算算法的进展．感兴趣的读者可以自行浏览．

外部参考资料

[1] Рёберная связность. Свойства и нахождение

[2] Вершинная связность. Свойства и нахождение

[3] Edge connectivity/Vertex connectivity

[4] *Connectivity Algorithms*

11.14 环计数问题

普通环计数

例题 1：Codeforces Beta Round 11 D. A Simple Task[1]

给定一个简单图，求图中简单环的数目．简单环是指没有重复顶点或边的环．

结点数目 $1 \leq n \leq 19$.

解题思路

考虑状态压缩动态规划．记 $f(s,i)$ 表示满足当前经过结点集合为 s，且现在在结点 i 上，且第一个结点为结点集合 s 中 **编号最小的那个** 的路径条数．

对于状态 $f(s,i)$，枚举下一个结点 u．若 u 在集合 s 中且是编号最小的那个（即起点），就将答案 A 加上 $f(s,i)$．若 u 不在 s 中，就将 $f(s,i)$ 加到 $f(s \cup \{u\},u)$ 上．

这样会把二元环（即重边）也算上，并且每个非二元环会被计算两次（因为固定起点可以向两个方向走），所以答案为 $\frac{A-m}{2}$，其中 m 表示边数．时间复杂度 $O(2^n m)$.

示例代码

```cpp
#include <iostream>
using namespace std;

int n, m;

struct Edge {
  int to, nxt;
} edge[500];

int cntEdge, head[20];

void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

long long answer, dp[1 << 19][20];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
```

```
  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v;
    cin >> u >> v;
    addEdge(u, v);
    addEdge(v, u);
  }
  for (int i = 1; i <= n; i++) dp[1 << (i - 1)][i] = 1;
  for (int s = 1; s < (1 << n); s++)
    for (int i = 1; i <= n; i++) {
      if (!dp[s][i]) continue;
      for (int j = head[i]; j; j = edge[j].nxt) {
        int u = i, v = edge[j].to;
        if ((s & -s) > (1 << (v - 1))) continue;
        if (s & (1 << (v - 1))) {
          if ((s & -s) == (1 << (v - 1))) answer += dp[s][u];
        } else
          dp[s | (1 << (v - 1))][v] += dp[s][u];
      }
    }
  cout << (answer - m) / 2 << '\n';
  return 0;
}
```

三元环计数

三元环 指的是一个简单图 G 中的一个无序三元组 (u, v, w) 满足存在三条边分别连接 (u, v)，(v, w) 和 (w, u)．而 **三元环计数问题** 要求计算出图中所有三元环的数量．

首先给所有边定向．我们规定从度数小的点指向度数大的点，度数相同就从编号小的点指向编号大的点．那么此时此图是一张有向无环图（DAG）．

该图没有环的证明

反证法，假设存在环，那么环中的点度数一个比一个大，要形成环，所有点的度数必须相等，但是编号必定不同，矛盾．

所以定向后图肯定不存在环．

事实上，可以根据上述定向规则构造一个 偏序→9.20@p1951，所以按此规则构造的图（也即该偏序的 Hasse 图→9.20@p1951）一定是一个 DAG．

枚举 u 和 u 指向的点 v，再在 v 指向的点中枚举 w，检验 u 是否与 w 相连即可．

这个算法的时间复杂度为 $O(m\sqrt{m})$．

时间复杂度证明

对于定向部分，遍历了所有的边，时间复杂度 $O(n+m)$．

对于每一对 (v, w)，u 的数量都不超过 v 的入度 $d^-(v)$．

若 $d^-(v) \le \sqrt{m}$，由于 w 的个数至多为 n，所以这部分时间复杂度为 $O(n\sqrt{m})$．

若 $d^-(v) > \sqrt{m}$，由于 v 指向 w，所以 $d(v) \le d(w)$，得出 $d(w) > \sqrt{m}$，但是总边数只有 m，所以这样的 w 的个数至多为 $\sqrt{m}$，故时间复杂度为 $O(m\sqrt{m})$.

总时间复杂度为 $O(n + m + n\sqrt{m} + m\sqrt{m}) = O(m\sqrt{m})$.

事实上，如果定向时从度数大的点指向度数小的点，复杂度也正确，只需要交换 u, w 两个点，上述证明也成立.

示例代码（洛谷 P1989 无向图三元环计数[2]）

```cpp
#include <iostream>
using namespace std;

int n, m, total;
int deg[200001], u[200001], v[200001];
bool vis[200001];

struct Edge {
  int to, nxt;
} edge[200001];

int cntEdge, head[100001];

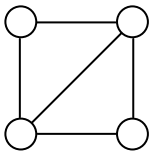
void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  for (int i = 1; i <= m; i++) cin >> u[i] >> v[i], deg[u[i]]++, deg[v[i]]++;
  for (int i = 1; i <= m; i++) {
    if ((deg[u[i]] == deg[v[i]] && u[i] > v[i]) || deg[u[i]] < deg[v[i]])
      swap(u[i], v[i]);
    addEdge(u[i], v[i]);
  }
  for (int u = 1; u <= n; u++) {
    for (int i = head[u]; i; i = edge[i].nxt) vis[edge[i].to] = true;
    for (int i = head[u]; i; i = edge[i].nxt) {
      int v = edge[i].to;
      for (int j = head[v]; j; j = edge[j].nxt) {
        int w = edge[j].to;
        if (vis[w]) total++;
      }
    }
    for (int i = head[u]; i; i = edge[i].nxt) vis[edge[i].to] = false;
  }
  cout << total << '\n';
  return 0;
}
```

例题 2

HDU 6184 Counting Stars[3]

给定一张有 n 个点和 m 条边的无向图，求下面图形的出现次数.



$2 \le n \le 10^5$，$1 \le m \le \min\left\{2 \times 10^5, \dfrac{n(n-1)}{2}\right\}$.

解题思路

这个图形是两个三元环共用了一条边形成的. 所以我们先跑一遍三元环计数, 统计出一条边上三元环的数量，然后枚举共用的那条边，设有 x 个三元环中有此边，那么对答案的贡献就是 $\binom{x}{2}$.

时间复杂度 $O(m\sqrt{m})$.

示例代码

```cpp
#include <cstring>
#include <iostream>
using namespace std;

int n, m, total;
int deg[200001], u[200001], v[200001];
int edgeId[200001], cnt[200001];

struct Edge {
  int to, nxt;
} edge[200001];

int cntEdge, head[100001];

void addEdge(int u, int v) {
  edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
}

int main() {
  while (cin >> n >> m) {
    cntEdge = total = 0;
    memset(deg, 0, sizeof deg);
    memset(head, 0, sizeof head);
    for (int i = 1; i <= m; i++) cin >> u[i] >> v[i], deg[u[i]]++,
deg[v[i]]++;
    for (int i = 1; i <= m; i++) {
      if ((deg[u[i]] == deg[v[i]] && u[i] > v[i]) || deg[u[i]] < deg[v[i]])
        swap(u[i], v[i]);
      addEdge(u[i], v[i]);
    }
```

```
30
31      for (int u = 1; u <= n; u++) {
32        for (int i = head[u]; i; i = edge[i].nxt) edgeId[edge[i].to] = i;
33        for (int i = head[u]; i; i = edge[i].nxt) {
34          int v = edge[i].to;
35          for (int j = head[v]; j; j = edge[j].nxt) {
36            int w = edge[j].to;
37            if (edgeId[w]) cnt[i]++, cnt[j]++, cnt[edgeId[w]]++;
38          }
39        }
40        for (int i = head[u]; i; i = edge[i].nxt) edgeId[edge[i].to] = 0;
41      }
42      for (int i = 1; i <= m; i++) total += cnt[i] * (cnt[i] - 1) / 2, cnt[i]
    = 0;
43      cout << total << '\n';
44    }
45    return 0;
46  }
```

四元环计数

类似地，**四元环** 就是指四个点 a, b, c, d 满足 (a, b)，(b, c)，(c, d) 和 (d, a) 均有边连接.

考虑先对点进行排序. 度数小的排在前面，度数大的排在后面.

考虑枚举排在最后面的点 a，此时只需要对于每个比 a 排名更前的点 c，都求出有多少个排名比 a 前的点 b 满足 (a, b)，(b, c) 有边. 然后只需要从这些 b 中任取两个都能成为一个四元环. 求 b 的数量只需要遍历一遍 b 和 c 即可.

注意到我们枚举的复杂度本质上与枚举三元环等价，所以时间复杂度也是 $O(m\sqrt{m})$（假设 n, m 同阶）.

值得注意的是，(a, b, c, d) 和 (a, c, b, d) 可以是两个不同的四元环.

另外，度数相同的结点的排名将不相同，并且需要注意判断 $a \neq c$.

示例代码（LibreOJ P191 无向图四元环计数[4]）

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int n, m, deg[100001], cnt[100001];
6  vector<int> E[100001], E1[100001];
7
8  long long total;
9
10 int main() {
11   cin.tie(nullptr)->sync_with_stdio(false);
12   cin >> n >> m;
13   for (int i = 1; i <= m; i++) {
14     int u, v;
15     cin >> u >> v;
```

```
    E[u].push_back(v);
    E[v].push_back(u);
    deg[u]++, deg[v]++;
  }
  for (int u = 1; u <= n; u++)
    for (int v : E[u])
      if (deg[u] > deg[v] || (deg[u] == deg[v] && u > v))
E1[u].push_back(v);
  for (int a = 1; a <= n; a++) {
    for (int b : E1[a])
      for (int c : E[b]) {
        if (deg[a] < deg[c] || (deg[a] == deg[c] && a <= c)) continue;
        total += cnt[c]++;
      }
    for (int b : E1[a])
      for (int c : E[b]) cnt[c] = 0;
  }
  cout << total << '\n';
  return 0;
}
```

例题 3

Gym 102028L Connected Subgraphs[5]

给定一张有 n 个点和 m 条边的无向图，求四条边的导出子图连通的情况数.

$4 \le n \le 10^5$，$4 \le m \le 2 \times 10^5$.

解题思路

容易把情况分为五种：菊花图、四元环、三元环上一个点连出一条边、四个点构成的链中间一个点连出一条边以及五个点构成的链.

菊花图直接枚举点的度数，用组合数解决即可. 四元环可以直接按照上述算法求得. 三元环部分只需枚举三元环 (u, v, w)，那么对答案的贡献就是 $[d(u)-2]+[d(v)-2]+[d(w)-2]$.

下面考虑第四种情况. 考虑枚举度数为 2 的点 x，再枚举与它相邻的一个结点 y 作为度数为 3 的那个点. 此时对答案的贡献为 $[d(x)-1]\cdot\binom{d(y)-1}{2}$. 但是注意到 y 的相邻节点可能会和 x 的相邻结点重合，此时的图形等价于第三种情况. 但是每种多算的第三种情况都会被多算两次（因为有两个度数为 3 的点），所以应该减去第三种情况数目的两倍.

对于最后一种情况，先枚举中间的点 x，那么容易发现对答案的贡献是

$$\sum_{y\in son_x}\sum_{z\in son_x}[d(y)-1]\cdot[d(z)-1].$$

同样地，这其中有多算的部分. 设 y 的相邻结点为 s，z 的相邻结点为 t，那么思考后发现多算的有如下几种情况：

1. y 与 t 重合，但是 s 与 z 不重合时，等价于第三种情况；
2. s 与 z 重合，但是 y 与 t 不重合时，同样等价于第三种情况；
3. y 与 t，s 与 z 都重合时，等价于一个三元环；
4. s 与 t 重合时，等价于一个四元环（第二种情况）.

考虑到第三种情况中两个度数 2 的点作为 x 时正好分别对应上述多算情况的 1 和 2，所以要额外减去第三种情况数目的两倍．对于一个三元环，三个结点都可以作为 x，多算了 3 次．同样的，四元环的情况被多算了 4 次．

于是我们就得出了所有情况的算法，时间复杂度为 $O(n+m\sqrt{m})$．

示例代码

```cpp
#include <iostream>
#include <vector>
using namespace std;
constexpr int mod = 1000000007;

constexpr long long power(long long a, long long n = mod - 2) {
  long long res = 1;
  while (n) {
    if (n & 1) res = res * a % mod;
    a = a * a % mod;
    n >>= 1;
  }
  return res;
}

constexpr long long power24 = power(24), power2 = power(2);

int n, m;
vector<int> E[100001], E1[100001], E2[100001];
bool vis[100001];
int cnt1[100001], cnt2[100001];

long long solve1();
long long solve2();
long long solve3();
long long solve4();
long long solve5();
long long ans[6];

long long solve1() {
  if (ans[1] != -1) return ans[1];
  ans[1] = 0;
  for (int i = 1; i <= n; i++) {
    int x = E[i].size();
    ans[1] +=
        1ll * x * (x - 1) % mod * (x - 2) % mod * (x - 3) % mod * power24
% mod;
  }
  return ans[1] %= mod;
}

long long solve2() {
  if (ans[2] != -1) return ans[2];
  ans[2] = 0;
```

```
  for (int i = 1; i <= n; i++) {
    for (int j : E1[i])
      for (int k : E1[j]) cnt1[k]++;
    for (int j : E2[i])
      for (int k : E1[j])
        if (k != i) cnt2[k]++;
    for (int j : E1[i])
      for (int k : E1[j])
        ans[2] +=
            (1ll * cnt1[k] * (cnt1[k] - 1) / 2 + 1ll * cnt1[k] * cnt2[k])
% mod,
            cnt1[k] = 0;
    for (int j : E2[i])
      for (int k : E1[j])
        if (k != i)
          ans[2] += 1ll * cnt2[k] * (cnt2[k] - 1) / 2 % mod * power2 % mod,
              cnt2[k] = 0;
  }
  return ans[2];
}

long long solve3() {
  if (ans[3] != -1) return ans[3];
  ans[0] = ans[3] = 0;
  for (int i = 1; i <= n; i++) {
    for (int j : E1[i]) vis[j] = true;
    for (int j : E1[i])
      for (int k : E1[j])
        if (vis[k])
          ans[3] =
              (1ll * ans[3] + E[i].size() + E[j].size() + E[k].size() - 6) %
              mod,
          ans[0]++;
    for (int j : E1[i]) vis[j] = false;
  }
  return ans[3];
}

long long solve4() {
  if (ans[4] != -1) return ans[4];
  ans[4] = 0;
  for (int i = 1; i <= n; i++)
    for (int j : E[i])
      (ans[4] += 1ll * (E[j].size() - 1) * (E[j].size() - 2) / 2 *
                 (E[i].size() - 1) % mod) %= mod;
  return ans[4] = (ans[4] - 2 * solve3()) % mod;
}

long long solve5() {
  if (ans[5] != -1) return ans[5];
  ans[5] = 0;
  for (int i = 1; i <= n; i++) {
```

```
    long long sum = 0;
    for (int j : E[i]) {
      ans[5] += sum * (E[j].size() - 1) % mod;
      sum += E[j].size() - 1;
    }
  }
  solve3();
  return ans[5] =
             (ans[5] % mod - 2 * solve3() - 4 * solve2() - 3 * ans[0])
% mod;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) {
    ans[5] = ans[1] = ans[2] = ans[4] = ans[3] = -1;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) E[i].clear(), E1[i].clear(), E2[i].clear();
    while (m--) {
      int x, y;
      cin >> x >> y;
      E[x].push_back(y), E[y].push_back(x);
    }
    for (int i = 1; i <= n; i++)
      for (int j : E[i]) {
        if (make_pair(E[i].size(), i) < make_pair(E[j].size(), j))
          E1[i].push_back(j);
        else
          E2[i].push_back(j);
      }
    cout << ((solve5() + solve1() + solve2() + solve4() + solve3()) % mod +
             mod) %
                mod
         << '\n';
  }
  return 0;
}
```

习题

洛谷 P3547 [POI2013] CEN-Price List[6]

CodeForces 985G Team Players[7]（容斥原理）

外部参考资料

[1] 例题 1：Codeforces Beta Round 11 D. A Simple Task

[2] 洛谷 P1989 无向图三元环计数

[3] HDU 6184 Counting Stars

[4] LibreOJ P191 无向图四元环计数

[5] Gym 102028L Connected Subgraphs

[6] 洛谷 P3547 [POI2013] CEN-Price List

[7] CodeForces 985G Team Players 

11.15 最小环

引入

问题

给出一个图，问其中的由 n 个节点构成的边权和最小的环 ($n \geq 3$) 是多大.

图的最小环也称围长.

过程

暴力解法

设 u 和 v 之间有一条边长为 w 的边，$dis(u,v)$ 表示删除 u 和 v 之间的连边之后，u 和 v 之间的最短路.

那么无向图中的最小环是 $dis(u,v)+w$.

注意若是在有向图中求最小环，相对应的公式要修改，最小环是 $dis(v,u)+w$.

总时间复杂度 $O(n^2m)$.

Dijkstra

相关链接：最短路/Dijkstra→11.9.1@p2735

过程

枚举所有边，每一次求删除一条边之后对这条边的起点跑一次 Dijkstra，道理同上.

性质

时间复杂度 $O(m(n+m)\log n)$.

Floyd

相关链接：最短路/Floyd→11.9.1@p2735

过程

记原图中 u,v 之间边的边权为 $val(u,v)$.

我们注意到 Floyd 算法有一个性质：在最外层循环到点 k 时（尚未开始第 k 次循环），最短路数组 dis 中，$dis_{u,v}$ 表示的是从 u 到 v 且仅经过编号在 $[1,k)$ 区间中的点的最短路.

由最小环的定义可知其至少有三个顶点，设其中编号最大的顶点为 w，环上与 w 相邻两侧的两个点为 u,v，则在最外层循环枚举到 $k=w$ 时，该环的长度即为 $dis_{u,v}+val(v,w)+val(w,u)$.

故在循环时对于每个 k 枚举满足 $i<k,j<k$ 的 (i,j)，更新答案即可.

记录路径

现在已经知道了环的形式为 $u\to k\to v$，然后再从 v 回到 u（经过的点编号均 $<k$）.

问题转化为求 $v\rightsquigarrow u$ 的路径. 由三角不等式 $dis_{u,v}\leq dis_{u,i}+dis_{i,v}$，考虑记录 $pos_{u,v}=j$ 表示使得 $dis_{u,v}=dis_{u,j}+dis_{j,v}$ 的点. 显然 j 就在 $v\rightsquigarrow u$ 的路径上.

于是可以将路径转化为 $v\rightsquigarrow j$ 和 $j\rightsquigarrow u$ 两段，分别递归处理即可.

证明递归不会陷入死循环

使用反证法.

假设环会重复经过一个点 u，那么环上必然会有一条 u 出发经过若干条边回到 u 的边. 这就构成了一个新的环.

由于图中不会出现负环（如果出现负环就不会有最小环了），所以新环的边权和必然是小于等于原环的.

所以只需要截取这一个环，那么就不会重复经过一个点 u，假设不成立，故环不会重复经过一个点.

所以当递归到 u, v 两点时，其 $pos_{u,v}$ 必然不等于两点编号，也就是新加入了一个点.

特别地，当 u 和 v 相邻时，直接返回即可.

由于总点数为 n，那么新加入的次数（即递归次数）不会超过 n，因此递归不会陷入死循环.

性质

时间复杂度：$O(n^3)$.

实现

下面给出 C++ 和 Python 的参考实现（记录路径）：

- C++

```cpp
// 图的点数为 n
int val[MAXN + 1][MAXN + 1];  // 原图的邻接矩阵
int cnt, path[MAXN + 5];      // 记录最小环的路径和长度

void get_path(int u, int v) {  // 获得 u 到 v 之间的路径
  if (pos[u][v] == 0) return;

  int k = pos[u][v];
  get_path(u, k);
  path[++cnt] = k;
  get_path(k, v);
}

void Floyd(const int &n) {
  static int dis[MAXN + 1][MAXN + 1];  // 最短路矩阵
  static int pos[MAXN + 1][MAXN + 1];
  memcpy(dis, val, sizeof(val));
  memset(pos, 0, sizeof(pos));
  for (int k = 1; k <= n; ++k) {
    for (int i = 1; i < k; ++i)
      for (int j = 1; j < i; ++j)
        if (ans >
            (long long)val[i][k] + val[k][j] + dis[i][j]) {  // 发现了更短的环
          // 由于这里保证了 j<i<k，所以三个点不同，不会出现零环.
          ans = val[i][k] + val[k][j] + dis[i][j], cnt = 0;
          path[++cnt] = i, path[++cnt] = k,
          path[++cnt] = j;  // 依次加入 i,k,j 三点
          get_path(j, i);   // 加入 j 到 i 的路径
        }
```

```
30
31      for (int i = 1; i <= n; ++i)  // 正常 Floyd 更新最短路
32        for (int j = 1; j <= n; ++j) {
33          if (dis[i][j] > dis[i][k] + dis[k][j]) {
34            dis[i][j] = dis[i][k] + dis[k][j];
35            pos[i][j] = k;  // 当前路径可以由 k 更新得到
36          }
37        }
38    }
39 }
```

- Python

```
1  # 定义一个足够大的值表示无穷大
2  INF = sys.maxsize
3
4
5  def get_path(i, j, pos, path, cnt):
6      """
7      递归地获取从节点 i 到节点 j 的最短路径上的中间节点.
8
9      Args:
10         i (int): 起始节点 (0-based index).
11         j (int): 结束节点 (0-based index).
12         pos (list[list[int]]): 记录最短路径中间节点的矩阵. pos[i][j] = k 表示从 i 到 j 的最短路径经过 k.
13         path (list[int]): 存储路径节点的列表 (使用 0-based index).
14         cnt (int): 当前路径节点的数量.
15
16     Returns:
17         int: 更新后的路径节点数量.
18     """
19     # 如果 pos[i][j] 为 -1, 表示 i 到 j 没有中间节点
20     if pos[i][j] == -1:
21         return cnt
22
23     # 获取中间节点 k
24     k = pos[i][j]
25     # 递归获取 i 到 k 的路径
26     cnt = get_path(i, k, pos, path, cnt)
27     # 将中间节点 k 加入路径
28     path[cnt] = k
29     cnt += 1
30     # 递归获取 k 到 j 的路径
31     cnt = get_path(k, j, pos, path, cnt)
32     return cnt
33
34
35 def find_minimum_cycle_undirected(n, edges):
36     """
37     使用 Floyd-Warshall 算法查找无向图中的最小环.
38
39     Args:
40         n (int): 图的节点数 (1 到 n).
```

```
        edges (list[tuple]): 边的列表，每个元素是 (u, v, w)，表示节点 u 和 v 之间有一条权重为 w 的边.
                             节点索引是 1 到 n.

    Returns:
        tuple: 包含最小环的长度和路径.
               如果不存在环，返回 (INF, []).
               路径是一个节点索引列表 (1-based index).
    """
    # 内部使用 0-based indexing
    N = n
    # 初始化邻接矩阵 g，表示原始边的权重
    g = [[INF for _ in range(N)] for _ in range(N)]
    # 初始化最短路矩阵 dis，开始时与 g 相同
    dis = [[INF for _ in range(N)] for _ in range(N)]
    # 初始化 pos 矩阵，记录最短路径的中间节点
    pos = [[-1 for _ in range(N)] for _ in range(N)]

    # 初始化对角线为 0（节点到自身的距离）
    for i in range(N):
        g[i][i] = 0
        dis[i][i] = 0

    # 根据输入的边构建邻接矩阵（无向图）
    for u, v, w in edges:
        # 将 1-based 索引转换为 0-based
        u -= 1
        v -= 1
        # 对于无向图，边是双向的
        g[u][v] = min(g[u][v], w)
        g[v][u] = min(g[v][u], w)
        dis[u][v] = min(dis[u][v], w)
        dis[v][u] = min(dis[v][u], w)

    # 初始化最小环长度为无穷大
    min_cycle_len = INF
    # 初始化最小环路径
    min_cycle_path = []

    # Floyd-Warshall 算法核心部分
    # k 作为中间节点 (0-based index)
    for k in range(N):
        # 在更新 dis[i][j] 之前，检查通过节点 k 是否能形成更小的环
        # 环由 i -> k -> j -> ... -> i 组成
        # 这里的 dis[i][j] 是在考虑节点 0 到 k-1 作为中间节点时的最短路径
        # C++ 代码中使用 i < k 和 j < i 的循环顺序，这里也遵循这个逻辑 (0-based)
        for i in range(k):  # 0 <= i < k
            for j in range(i):  # 0 <= j < i
                # 检查 i, k, j 是否构成一个环，并且通过 dis[i][j] 连接
                # 确保原始边 g[i][k] 和 g[k][j] 存在 (不为 INF)
                # 并且 i 到 j 的最短路径 dis[i][j] 存在 (不为 INF)
                if g[i][k] != INF and g[k][j] != INF and dis[i][j] != INF:
                    current_cycle_len = g[i][k] + g[k][j] + dis[i][j]
                    if current_cycle_len < min_cycle_len:
```

```
                            min_cycle_len = current_cycle_len
                            # 重构路径
                            path = [0] * (N + 5)  # 临时存储路径的数组，长度足够大
                            cnt = 0
                            # 按照 i, k, j 的顺序加入路径
                            path[cnt] = i
                            cnt += 1
                            path[cnt] = k
                            cnt += 1
                            path[cnt] = j
                            cnt += 1
                            # 获取 j 到 i 的最短路径上的中间节点（使用之前计算的 dis 和 pos）
                            cnt = get_path(j, i, pos, path, cnt)
                            # 提取实际路径节点（去除未使用的部分）
                            # 将 0-based 索引转换为 1-based
                            min_cycle_path = [node + 1 for node in path[:cnt]]

        # 标准 Floyd-Warshall 更新最短路径
        for i in range(N):
            for j in range(N):
                if (
                    dis[i][k] != INF
                    and dis[k][j] != INF
                    and dis[i][j] > dis[i][k] + dis[k][j]
                ):
                    dis[i][j] = dis[i][k] + dis[k][j]
                    # 记录从 i 到 j 的最短路径经过 k
                    pos[i][j] = k

    return min_cycle_len, min_cycle_path
```

模板题

AcWing 344 观光之旅[1]

给定一张 n 个点无向图，求图中一个至少包含 3 个点的环，环上的节点不重复，并且环上的边的长度之和最小.

该问题称为无向图的最小环问题.

你需要输出最小环的方案，若最小环不唯一，输出任意一个均可.

$n \leq 100$

时间复杂度接受 $O(n^3)$ 的做法，套用 Floyd 求最小环的做法即可.

- C++

```
#include <bits/stdc++.h>
using lint = long long;
// 定义一个足够大的常量，用于表示图的最大节点数
const int MAXN = 110;

// 定义一个足够大的值，用于表示无穷大，初始化最小环长度
lint ans = 1e9;  // lint 是 long long 的别名
```

```

// 图的节点数 n，边数 m
// cnt 记录最小环路径中的节点数量
// path 存储最小环的路径节点
int n, m, cnt, path[MAXN];

// g 存储原始图的邻接矩阵
// dis 存储最短路径矩阵 (Floyd-Warshall 算法计算过程中更新)
// pos 记录最短路径的中间节点，pos[i][j] = k 表示从 i 到 j 的最短路径经过 k
int g[MAXN][MAXN], dis[MAXN][MAXN], pos[MAXN][MAXN];

// 递归函数：获取从节点 u 到节点 v 的最短路径上的中间节点
// 根据 pos 矩阵重构路径
void get_path(int u, int v) {
  // 如果 pos[u][v] 为 0，表示 u 到 v 没有中间节点，直接返回
  if (pos[u][v] == 0) return;

  // 获取中间节点 k
  int k = pos[u][v];
  // 递归获取 u 到 k 的路径
  get_path(u, k);
  // 将中间节点 k 加入路径
  path[++cnt] = k;
  // 递归获取 k 到 v 的路径
  get_path(k, v);
}

// Floyd-Warshall 算法函数：查找图中的最小环
void Floyd() {
  // 外层循环：k 作为中间节点 (1 到 n)
  for (int k = 1; k <= n; ++k) {
    // 内层循环：i 和 j，用于检查通过节点 k 是否能形成更小的环
    // 这里循环顺序是 i 从 1 到 k-1，j 从 1 到 i-1
    // 这样可以检查由 i -> k -> j -> ... -> i 构成的环
    for (int i = 1; i < k; ++i)
      for (int j = 1; j < i; ++j)
        // 检查通过节点 k 连接 i 和 j 是否形成更小的环
        // 环的长度是 i 到 k 的原始边权重 g[i][k] + k 到 j 的原始边权重 g[k][j]
        // + i 到 j 的当前最短路径 dis[i][j]
        if (ans > (long long)g[i][k] + g[k][j] + dis[i][j]) {
          // 发现了更小的环
          ans = g[i][k] + g[k][j] + dis[i][j];  // 更新最小环长度
          cnt = 0;                              // 重置路径计数
          // 将 i, k, j 依次加入路径
          path[++cnt] = i, path[++cnt] = k, path[++cnt] = j;
          // 获取 j 到 i 的最短路径上的中间节点，加入路径
          get_path(j, i);
        }

    // 标准 Floyd-Warshall 更新最短路径
    // i 从 1 到 n，j 从 1 到 n
    for (int i = 1; i <= n; ++i)
      for (int j = 1; j <= n; ++j) {
        // 如果通过中间节点 k 可以获得更短的从 i 到 j 的路径
```

```cpp
        if (dis[i][j] > dis[i][k] + dis[k][j]) {
          // 更新最短路径
          dis[i][j] = dis[i][k] + dis[k][j];
          // 记录从 i 到 j 的最短路径经过 k
          pos[i][j] = k;
        }
      }
  }
}

// 主函数
int main() {
  // 读取节点数 n 和边数 m
  std::cin >> n >> m;
  // 初始化原始邻接矩阵 g，所有边的权重设为无穷大（0x3f 通常表示一个很大的值）
  memset(g, 0x3f, sizeof(g));
  // 将节点到自身的距离设为 0
  for (int i = 1; i <= n; ++i) g[i][i] = 0;
  // 读取 m 条边，构建原始邻接矩阵 g
  // 对于无向图，边是双向的，取较小的权重
  for (int i = 0, u, v, w; i < m; ++i) {
    std::cin >> u >> v >> w;
    g[u][v] = g[v][u] = std::min(g[u][v], w);
  }
  // 将原始邻接矩阵 g 复制到最短路径矩阵 dis
  memcpy(dis, g, sizeof(g));
  // 调用 Floyd 算法查找最小环
  Floyd();
  // 根据最小环长度判断是否存在环
  if (ans == 1e9) {  // 如果最小环长度仍为无穷大，表示不存在环
    puts("No solution.");
  } else {
    // 如果存在环，打印路径节点
    // std::cout << "ans = " << ans << std::endl; // 打印最小环长度（注释掉）
    // 打印路径节点，用空格分隔
    for (int i = 1; i <= cnt; ++i)
      std::cout << path[i]
                << (i == cnt ? "" : " ");  // 最后一个节点后面没有空格
    std::cout << std::endl;                 // 路径打印完后换行
  }
  return 0;
}
```

- Python

```python
import copy
import sys

# 定义一个足够大的值表示无穷大
INF = sys.maxsize


def get_path(i, j, pos, path, cnt):
    """
    递归地获取从节点 i 到节点 j 的最短路径上的中间节点.
```

```
11
12      Args:
13          i (int): 起始节点 (0-based index).
14          j (int): 结束节点 (0-based index).
15          pos (list[list[int]]): 记录最短路径中间节点的矩阵. pos[i][j] = k 表示从 i 到 j 的最短路径经过 k.
16          path (list[int]): 存储路径节点的列表 (使用 0-based index).
17          cnt (int): 当前路径节点的数量.
18
19      Returns:
20          int: 更新后的路径节点数量.
21      """
22      # 如果 pos[i][j] 为 -1，表示 i 到 j 没有中间节点
23      if pos[i][j] == -1:
24          return cnt
25
26      # 获取中间节点 k
27      k = pos[i][j]
28      # 递归获取 i 到 k 的路径
29      cnt = get_path(i, k, pos, path, cnt)
30      # 将中间节点 k 加入路径
31      path[cnt] = k
32      cnt += 1
33      # 递归获取 k 到 j 的路径
34      cnt = get_path(k, j, pos, path, cnt)
35      return cnt
36
37
38  def find_minimum_cycle_undirected(n, edges):
39      """
40      使用 Floyd-Warshall 算法查找无向图中的最小环.
41
42      Args:
43          n (int): 图的节点数 (1 到 n).
44          edges (list[tuple]): 边的列表，每个元素是 (u, v, w)，表示节点 u 和 v 之间有一条权重为 w 的边.
45                               节点索引是 1 到 n.
46
47      Returns:
48          tuple: 包含最小环的长度和路径.
49                 如果不存在环，返回 (INF, []).
50                 路径是一个节点索引列表 (1-based index).
51      """
52      # 内部使用 0-based indexing
53      N = n
54      # 初始化邻接矩阵 g，表示原始边的权重
55      g = [[INF for _ in range(N)] for _ in range(N)]
56      # 初始化最短路矩阵 dis，开始时与 g 相同
57      dis = [[INF for _ in range(N)] for _ in range(N)]
58      # 初始化 pos 矩阵，记录最短路径的中间节点
59      pos = [[-1 for _ in range(N)] for _ in range(N)]
60
61      # 初始化对角线为 0 (节点到自身的距离)
62      for i in range(N):
```

```
        g[i][i] = 0
        dis[i][i] = 0

    # 根据输入的边构建邻接矩阵（无向图）
    for u, v, w in edges:
        # 将 1-based 索引转换为 0-based
        u -= 1
        v -= 1
        # 对于无向图，边是双向的
        g[u][v] = min(g[u][v], w)
        g[v][u] = min(g[v][u], w)
        dis[u][v] = min(dis[u][v], w)
        dis[v][u] = min(dis[v][u], w)

    # 初始化最小环长度为无穷大
    min_cycle_len = INF
    # 初始化最小环路径
    min_cycle_path = []

    # Floyd-Warshall 算法核心部分
    # k 作为中间节点 (0-based index)
    for k in range(N):
        # 在更新 dis[i][j] 之前，检查通过节点 k 是否能形成更小的环
        # 环由 i -> k -> j -> ... -> i 组成
        # 这里的 dis[i][j] 是在考虑节点 0 到 k-1 作为中间节点时的最短路径
        # C++ 代码中使用 i < k 和 j < i 的循环顺序，这里也遵循这个逻辑 (0-based)
        for i in range(k):  # 0 <= i < k
            for j in range(i):  # 0 <= j < i
                # 检查 i, k, j 是否构成一个环，并且通过 dis[i][j] 连接
                # 确保原始边 g[i][k] 和 g[k][j] 存在 (不为 INF)
                # 并且 i 到 j 的最短路径 dis[i][j] 存在 (不为 INF)
                if g[i][k] != INF and g[k][j] != INF and dis[i][j] != INF:
                    current_cycle_len = g[i][k] + g[k][j] + dis[i][j]
                    if current_cycle_len < min_cycle_len:
                        min_cycle_len = current_cycle_len
                        # 重构路径
                        path = [0] * (N + 5)  # 临时存储路径的数组，长度足够大
                        cnt = 0
                        # 按照 i, k, j 的顺序加入路径
                        path[cnt] = i
                        cnt += 1
                        path[cnt] = k
                        cnt += 1
                        path[cnt] = j
                        cnt += 1
                        # 获取 j 到 i 的最短路径上的中间节点（使用之前计算的 dis
和 pos)
                        cnt = get_path(j, i, pos, path, cnt)
                        # 提取实际路径节点（去除未使用的部分）
                        # 将 0-based 索引转换为 1-based
                        min_cycle_path = [node + 1 for node in path[:cnt]]

        # 标准 Floyd-Warshall 更新最短路径
        for i in range(N):
```

```
            for j in range(N):
                if (
                    dis[i][k] != INF
                    and dis[k][j] != INF
                    and dis[i][j] > dis[i][k] + dis[k][j]
                ):
                    dis[i][j] = dis[i][k] + dis[k][j]
                    # 记录从 i 到 j 的最短路径经过 k
                    pos[i][j] = k

    return min_cycle_len, min_cycle_path


# --- 主程序入口 ---
if __name__ == "__main__":
    # 读取节点数 n 和边数 m
    n, m = map(int, sys.stdin.readline().split())

    # 读取边信息
    edges = []
    for _ in range(m):
        u, v, w = map(int, sys.stdin.readline().split())
        edges.append((u, v, w))

    # 查找最小环
    min_len, path = find_minimum_cycle_undirected(n, edges)

    # 输出结果
    if min_len == INF:
        print("No solution.")
    else:
        # 打印路径节点 (1-based index)，用空格分隔
        print(" ".join(map(str, path)))
```

例题 2

GDOI2018 Day2 巡逻

给出一张 n 个点的无负权边无向图，要求执行 q 个操作，三种操作

1. 删除一个图中的点以及与它有关的边
2. 恢复一个被删除点以及与它有关的边
3. 询问点 x 所在的最小环大小

对于 50% 的数据，有 $n, q \leq 100$

对于每一个点 x 所在的简单环，都存在两条与 x 相邻的边，删去其中的任意一条，简单环将变为简单路径.

那么枚举所有与 x 相邻的边，每次删去其中一条，然后跑一次 Dijkstra.

或者直接对每次询问跑一遍 Floyd 求最小环，$O(qn^3)$

对于 100% 的数据，有 $n, q \leq 400$.

还是利用 Floyd 求最小环的算法.

若没有删除，删去询问点将简单环裂开成为一条简单路.

然而第二步的求解改用 Floyd 来得出.

那么答案就是要求出不经过询问点 x 的情况下任意两点之间的距离.

怎么在线?

强行离线，利用离线的方法来避免删除操作.

将询问按照时间顺序排列，对这些询问建立一个线段树.

每个点的出现时间覆盖所有除去询问该点的时刻外的所有询问，假设一个点被询问 x 次，则它的出现时间可以视为 $x+1$ 段区间，插入到线段树上.

完成之后遍历一遍整棵线段树，在经过一个点时存储一个 Floyd 数组的备份，然后加入被插入在这个区间上的所有点，在离开时利用备份数组退回去即可.

这个做法的时间复杂度为 $O(qn^2 \log q)$.

还有一个时间复杂度更优秀的在线做法.

对于一个对点 x 的询问，我们以 x 为起点跑一次最短路，然后把最短路树建出来，顺便处理出每个点是在 x 的哪棵子树内.

那么一定能找出一条非树边，满足这条非树边的两个端点在根的不同子树中，使得这条非树边 + 两个端点到根的路径就是最小环.

证明:

显然最小环包含至少两个端点在根的不同子树中一条非树边.

假设这条边为 (u,v)，那么最短路树上 x 到 u 的路径是所有 x 到 u 的路径中最短的那条，x 到 v 的路径也是最短的那条，那么 $x \to u \to v \to x$ 这个环肯定不会比最小环要长.

那么就可以枚举所有非树边，更新答案.

每次询问的复杂度为跑一次单源最短路的复杂度，为 $O(n^2)$.

总时间复杂度为 $O(qn^2)$.

外部参考资料

[1] AcWing 344 观光之旅 

11.16 2-SAT

Authors: chu-yuehan

SAT 是适定性（Satisfiability）问题的简称. 一般形式为 k - 适定性问题，简称 k-SAT. 而当 $k>2$ 时该问题为 NP 完全的. 所以我们只研究 $k=2$ 的情况.

定义

2-SAT，简单的说就是给出 n 个布尔方程，每个方程和两个变量相关，如 $a \vee b$，表示变量 a,b 至少满足一个. 然后判断是否存在可行方案，显然可能有多种选择方案，一般题中只需要求出一种即可. 另外，$\neg a$ 表示 a 取反.

解决思路

洛谷 P4782「模板」2-SAT[1]

有 n 个布尔变量 $x_1 \sim x_n$，另有 m 个需要满足的条件，每个条件的形式都是「x_i 为 `true/false` 或 x_j 为 `true/false`」. 比如「x_1 为真或 x_3 为假」、「x_7 为假或 x_2 为假」.

2-SAT 问题的目标是给每个变量赋值使得所有条件得到满足.

使用布尔方程表示上述问题. 设 a 表示 x_a 为真（$\neg a$ 就表示 x_a 为假）. 如果有个人提出的要求分别是 a 和 b，即 $(a \vee b)$（变量 a, b 至少满足一个）. 对这些变量关系建有向图，则把 a 成立或不成立用图中的点表示，$\neg a \to b$ $\neg b \to a$，表示 a **不成立** 则 b **一定成立**；同理，b **不成立** 则 a **一定成立**. 建图之后，我们就可以使用缩点算法来求解 2-SAT 问题了.

原式	建图
$\neg a \vee b$	$a \to b$ 和 $\neg b \to \neg a$
$a \vee b$	$\neg a \to b$ 和 $\neg b \to a$
$\neg a \vee \neg b$	$a \to \neg b$ 和 $b \to \neg a$

许多 2-SAT 问题都需要找出如 a **不成立**，则 b **成立** 的关系.

求解

思考如果两点在同一强连通分量里有什么含义. 根据前文边的逻辑意义可知: 若两点在同一强连通分量内，则这两点代表的条件 **要么都满足，要么都不满足**.

建图后我们使用 Tarjan 算法找 SCC→11.13.1@p2806，判断对于任意布尔变量 a，表示 a 成立的点和表示 a 不成立的点是否在同一个 SCC 中（同一条件不可能既满足又不满足，或既不满足又并非不满足），若有则输出无解，否则有解.

输出方案时可以通过变量在图中的拓扑序确定该变量的取值. 如果变量 x 的拓扑序在 $\neg x$ 之后，那么取 x 值为真. 应用到 Tarjan 算法的缩点，即 x 所在 SCC 编号在 $\neg x$ 之前时，取 x 为真. 因为 Tarjan 算法求强连通分量时使用了栈，如果跑完 Tarjan 缩点之后呈现出的拓扑序更大，在 Tarjan 会更晚被遍历到，就会更早地被弹出栈而缩点，分量编号会更小，所以 Tarjan 求得的 SCC 编号相当于 **反拓扑序**.

算法会把整张图遍历一遍，由于这张图 n 和 m 同阶，计算答案时复杂度为 $O(n)$，因此总复杂度为 $O(n)$.

代码实现

```cpp
#include <algorithm>
#include <cstdio>
#include <stack>
using namespace std;
const int N = 2e6 + 2;
int n, m, dfn[N], low[N], t, tot, head[N], a[N];
bool vis[N];
stack<int> s;

struct node {
  int to, Next;
} e[N];

void adde(int u, int v) {
```

```
  e[++tot].to = v;
  e[tot].Next = head[u];
  head[u] = tot;
}

void Tarjan(int u) {
  dfn[u] = low[u] = ++t;
  s.push(u);
  vis[u] = 1;
  for (int i = head[u]; i; i = e[i].Next) {
    int v = e[i].to;
    if (!dfn[v]) {
      Tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (vis[v])
      low[u] = min(low[u], dfn[v]);
  }
  if (dfn[u] == low[u]) {
    int cur;
    ++tot;
    do {
      cur = s.top();
      s.pop();
      vis[cur] = 0;
      a[cur] = tot;
    } while (cur != u);
  }
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1, I, J, A, B; i <= m; i++) {
    scanf("%d%d%d%d", &I, &A, &J, &B);
    adde(A ? I + n : I, B ? J : J + n);
    adde(B ? J + n : J, A ? I : I + n);
  }
  tot = 0;
  for (int i = 1; i <= (n << 1); i++)
    if (!dfn[i]) Tarjan(i);
  for (int i = 1; i <= n; i++) {
    if (a[i] == a[i + n]) {
      printf("IMPOSSIBLE");
      return 0;
    }
  }
  puts("POSSIBLE");
  for (int i = 1; i <= n; i++)
    printf("%c%c", a[i] < a[i + n] ? '1' : '0', " \n"[i == n]);
  return 0;
}

```

例题

例题 1

HDU3062 Party[2]

有 n 对夫妻被邀请参加一个聚会，因为场地的问题，每对夫妻中只有一人可以列席．在 $2n$ 个人中，某些人之间有着很大的矛盾（当然夫妻之间是没有矛盾的），有矛盾的两个人是不会同时出现在聚会上的．有没有可能会有 n 个人同时列席？

按照上面的分析，如果 a_1 中的丈夫和 a_2 中的妻子不合，我们就把 a_1 中的丈夫和 a_2 中的丈夫连边，把 a_2 中的妻子和 a_1 中的妻子连边，然后缩点染色判断即可．

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
constexpr int MAXN = 2018;
constexpr int MAXM = 4000400;
using namespace std;
int Index, instack[MAXN], DFN[MAXN], LOW[MAXN];
int tot, color[MAXN];
int numedge, head[MAXN];

struct Edge {
  int nxt, to;
} edge[MAXM];

int sta[MAXN], top;
int n, m;

void add(int x, int y) {
  edge[++numedge].to = y;
  edge[numedge].nxt = head[x];
  head[x] = numedge;
}

void tarjan(int x) {  // 缩点看不懂请移步强连通分量上面有一个链接可以点。
  sta[++top] = x;
  instack[x] = 1;
  DFN[x] = LOW[x] = ++Index;
  for (int i = head[x]; i; i = edge[i].nxt) {
    int v = edge[i].to;
    if (!DFN[v]) {
      tarjan(v);
      LOW[x] = min(LOW[x], LOW[v]);
    } else if (instack[v])
      LOW[x] = min(LOW[x], DFN[v]);
  }
  if (DFN[x] == LOW[x]) {
    tot++;
    do {
```

```
39        color[sta[top]] = tot;  // 染色
40        instack[sta[top]] = 0;
41      } while (sta[top--] != x);
42    }
43  }
44
45  bool solve() {
46    for (int i = 0; i < 2 * n; i++)
47      if (!DFN[i]) tarjan(i);
48    for (int i = 0; i < 2 * n; i += 2)
49      if (color[i] == color[i + 1]) return false;
50    return true;
51  }
52
53  void init() {
54    top = 0;
55    tot = 0;
56    Index = 0;
57    numedge = 0;
58    memset(sta, 0, sizeof(sta));
59    memset(DFN, 0, sizeof(DFN));
60    memset(instack, 0, sizeof(instack));
61    memset(LOW, 0, sizeof(LOW));
62    memset(color, 0, sizeof(color));
63    memset(head, 0, sizeof(head));
64  }
65
66  int main() {
67    cin.tie(nullptr)->sync_with_stdio(false);
68    while (cin >> n >> m) {
69      init();
70      for (int i = 1; i <= m; i++) {
71        int a1, a2, c1, c2;
72        cin >> a1 >> a2 >> c1 >> c2;
73        add(2 * a1 + c1, 2 * a2 + 1 - c2);
74        // 对于第 i 对夫妇，我们用 2i+1 表示丈夫，2i 表示妻子。
75        add(2 * a2 + c2, 2 * a1 + 1 - c1);
76      }
77      if (solve())
78        cout << "YES\n";
79      else
80        cout << "NO\n";
81    }
82    return 0;
83  }
```

例题 2

2018-2019 ACM-ICPC Asia Seoul Regional K TV Show Game[3]

有 k 盏灯，每盏灯是红色或者蓝色，但是初始的时候不知道灯的颜色．有 n 个人，每个人选择三盏灯并猜灯的颜色．一个人猜对两盏灯或以上的颜色就可以获得奖品．判断是否存在一个灯的着色方案使得每个人都能领奖，若有则输出一种灯的着色方案．

根据 伍昱 -《由对称性解 2-sat 问题》[4]，我们可以得出：如果要输出 2-SAT 问题的一个可行解，只需要在 tarjan 缩点后所得的 DAG 上自底向上地进行选择和删除．

具体实现的时候，可以通过构造 DAG 的反图后在反图上进行拓扑排序实现；也可以根据 tarjan 缩点后，所属连通块编号越小，节点越靠近叶子节点这一性质，优先对所属连通块编号小的节点进行选择．

下面给出第二种实现方法的代码．

参考代码

```cpp
#include <algorithm>
#include <iostream>
using namespace std;
constexpr int MAXN = 1e4 + 5;
constexpr int MAXK = 5005;

int n, k;
int id[MAXN][5];
char s[MAXN][5][5], ans[MAXK];
bool vis[MAXN];

struct Edge {
  int v, nxt;
} e[MAXN * 100];

int head[MAXN], tot = 1;

void addedge(int u, int v) {
  e[tot].v = v;
  e[tot].nxt = head[u];
  head[u] = tot++;
}

int dfn[MAXN], low[MAXN], color[MAXN], stk[MAXN], ins[MAXN], top, dfs_clock,
c;

void tarjan(int x) {  // tarjan算法求强联通
  stk[++top] = x;
  ins[x] = 1;
  dfn[x] = low[x] = ++dfs_clock;
  for (int i = head[x]; i; i = e[i].nxt) {
    int v = e[i].v;
    if (!dfn[v]) {
      tarjan(v);
      low[x] = min(low[x], low[v]);
    } else if (ins[v])
```

```
      low[x] = min(low[x], dfn[v]);
    }
    if (dfn[x] == low[x]) {
      c++;
      do {
        color[stk[top]] = c;
        ins[stk[top]] = 0;
      } while (stk[top--] != x);
    }
}

int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    cin >> k >> n;
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= 3; j++) cin >> id[i][j] >> s[i][j];

      for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 3; k++) {
          if (j == k) continue;
          int u = 2 * id[i][j] - (s[i][j][0] == 'B');
          int v = 2 * id[i][k] - (s[i][k][0] == 'R');
          addedge(u, v);
        }
      }
    }

    for (int i = 1; i <= 2 * k; i++)
      if (!dfn[i]) tarjan(i);

    for (int i = 1; i <= 2 * k; i += 2)
      if (color[i] == color[i + 1]) {
        cout << "-1\n";
        return 0;
      }

    for (int i = 1; i <= 2 * k; i += 2) {
      int f1 = color[i], f2 = color[i + 1];
      if (vis[f1]) {
        ans[(i + 1) >> 1] = 'R';
        continue;
      }
      if (vis[f2]) {
        ans[(i + 1) >> 1] = 'B';
        continue;
      }
      if (f1 < f2) {
        vis[f1] = true;
        ans[(i + 1) >> 1] = 'R';
      } else {
        vis[f2] = true;
        ans[(i + 1) >> 1] = 'B';
```

```
    }
  }
  ans[k + 1] = 0;
  cout << (ans + 1) << '\n';
  return 0;
}
```

习题

- 洛谷 P5782 和平委员会[5]
- POJ3683 Priest John's Busiest Day[6]

外部参考资料

[1] 洛谷 P4782「模板」2-SAT

[2] HDU3062 Party

[3] 2018-2019 ACM-ICPC Asia Seoul Regional K TV Show Game

[4] 伍昱 -《由对称性解 2-sat 问题》

[5] 洛谷 P5782 和平委员会

[6] POJ3683 Priest John's Busiest Day

11.17 欧拉图

本页面将简要介绍欧拉图的概念、实现和应用.

定义

本文中仅讨论有限图.

在图论中, **欧拉路径 (Eulerian path) 是经过图中每条边恰好一次的路径，欧拉回路 (Eulerian circuit) 是经过图中每条边恰好一次的回路. 如果一个图中存在欧拉回路，则这个图被称为欧拉图 (Eulerian graph)；如果一个图中不存在欧拉回路但是存在欧拉路径，则这个图被称为半欧拉图 (semi-Eulerian graph)** .

> **Warning**
>
> 此处定义中虽然使用「路径」一词，但严格说来此处使用的概念应该是「迹 (trail)」. 欧拉路径与欧拉回路仅能使用每条边恰好一次，但并没有对经过顶点的情况进行限制.

性质

以下我们假设所讨论的图 G 中不存在孤立顶点. 该假设不失一般性，因为对于存在孤立顶点的图 G，以下性质对从 G 中删除孤立顶点后得到的图 G' 仍然成立.

对于连通图 G，以下三个性质是互相等价的:

1. G 是欧拉图;
2. G 中所有顶点的度数都是偶数 (对于有向图，每个顶点的入度等于出度);
3. G 可被分解为若干条不共边回路的并.

以下我们对等价性进行证明.

若一个图 G 是欧拉图，那么 G 中所有顶点的度数都是偶数: 考虑从任意顶点开始沿着欧拉回路走一圈，则每个点 v 的度数等于离开点 v 的次数加到达点 v 的次数. 又由于行动的轨迹是一个回路，则对于每个点 v，离开该点的次数等于到达该点的次数. 这也就是说，每个点的度数都形如 $2k$，即偶数. 特别地，对于有向图，根据相同的证明过程，每个顶点的入度等于出度.

若一个图 G 中所有顶点的度数都是偶数（或入度与出度相等），则它可被分解为若干条不共边回路的不交并：考虑从任意顶点 u 开始，选择任意出边 (u,v)，走向对应的相邻顶点 v 并删除 (u,v)，直到返回最初开始的顶点 u．可以证明该过程必定会最终回到 u：每当到达一个新的顶点 $v\neq u$ 时，根据上一条性质，该顶点剩余的度数为奇数，也就是说必定存在一条出边，该过程不会在点 v 终止．（换句话说，该过程会且仅会在回到点 u 时停止．）又因图 G 中的边数是有限的，该过程必定会在有限步内停止，则最终必然可以返回 u 并得到一条回路．注意到在前述证明中我们仅使用了点度数均为偶数的性质，且在找到并删除一条回路后剩下部分的图仍然满足该性质，我们可以不断重复该过程直到剩下的图为空图，从而将 G 拆分为若干条不共边的回路．更进一步地，每条回路都可以被从其多次经过的顶点处分解成若干简单环的不交并，所以上述性质中的简单回路亦可被替换为简单环．

若一个连通图 G 可被分解为若干条不共边回路的不交并，则 G 是欧拉图：对于一组不共边回路，每次从中选出两条有共同顶点的回路并将其合并为一条，重复该过程直到不存在有共同顶点的两条回路．可以证明该过程结束时剩下的回路唯一．对于任意两条不共边回路 P_1,P_2，若 P_1 与 P_2 共点，则可以在共点处直接进行合并；否则，任取 P_1 上的点 v_1 与 P_2 上的点 v_2，根据 G 的连通性，存在连接 v_1 和 v_2 的路径 $e_1,e_2,...,e_k$，其中的每条边 e_i 都被一个回路 C_i 包含，且 P_1 与 C_1，C_i 与 C_{i+1}，C_k 与 P_2 均存在共点（或者 $C_i=C_{i+1}$，此情况不影响证明）．此情况下，P_1 与 P_2 可以通过 $C_1,...,C_k$ 进行合并．也就是说，任意两条回路都可以进行合并，最后剩下的回路必定唯一，且组成该回路的边集是所有不共边回路的并即 $E(G)$，该回路为 G 上的欧拉回路，G 为欧拉图．

以上的性质同时也构成了欧拉图的判断条件．具体地说，一个图是欧拉图当且仅当非零度顶点互相（强）连通，且顶点的度数都是偶数（或入度与出度相等）．

对于半欧拉图，其性质与欧拉图相似：一个半欧拉图具有恰好两个奇度数的顶点，且这两个顶点就是欧拉路径的两个端点．通过将这两个点连接起来，可以将半欧拉图转化为欧拉图．通过删除欧拉图中的任意一条边，可以得到一个半欧拉图．由此可以导出半欧拉图的判别法：一个图是半欧拉图当且仅当非零度顶点互相（强）连通，且奇度数顶点恰好有两个．对于有向图，第二个条件为恰存在两个顶点 u,v，其中 $\deg^+(u)-\deg^-(u)=1,\deg^+(v)-\deg^-(v)=-1$，且其余顶点的入度等于出度．

欧拉回路/欧拉路径的构造

此处我们介绍最常用的 Hierholzer 算法，该算法的核心思想为利用上述欧拉图性质中的第三点，即欧拉图可以被拆解为若干条不共边回路的并．可以注意到，在上述证明中其实已经提到了完整可行的将不共边回路合并为欧拉回路的操作，且在使用合适的数据结构储存时（如使用类链表的结构储存环）实现并不困难．

算法的具体流程为先从图中找到一条回路作为当前回路，每次从当前回路中选取剩余度数不为零的点，从该点出发找到一条新的简单回路，并将该简单回路与当前回路合并，重复该过程直到当前回路中的所有点均无剩余度数，此时的当前回路即为欧拉回路．

该算法同样适用于有向图．对于半欧拉图，可以从图中找到一条连接两个奇度数点的路径作为当前路径，每次选取度数非零的点寻找简单回路并将其与当前路径合并，最后得到欧拉路径．

实现

Hierholzer 算法的伪代码如下：

```
1  Input. The edges of the graph e, where each element in e is (u, v)
2  Output. The vertex of the Euler Road of the input graph.
3  Method.
4  Function Hierholzer (v)
5      circle ← Find a Circle in e Begin with v
6      if circle = ∅
7          return v
8      e ← e − circle
9      for each v ∈ circle
10         v ← Hierholzer(v)
11     return circle
12 Endfunction
13 return Hierholzer(any vertex)
```

时间复杂度分析

Hierholzer 算法的时间复杂度为 $O(|E|+|V|)$.

注意到在前述正确性分析中，在欧拉图或半欧拉图上寻找简单回路（或半欧拉图的初始路径）的过程是 **无需回溯** 的，只要沿着剩下的边一直走就必定可以发现所求的回路或路径，且 **每条边仅会被访问一次**. 为了利用这一性质，在实现上应采取类链表的方式储存图中的边，如邻接表或链式前向星，以便每条边在被访问过后即刻删除之. 如果采用朴素的邻接矩阵进行储存，则每次寻边耗时 $O(|V|)$，总复杂度为 $O(|V||E|)$.

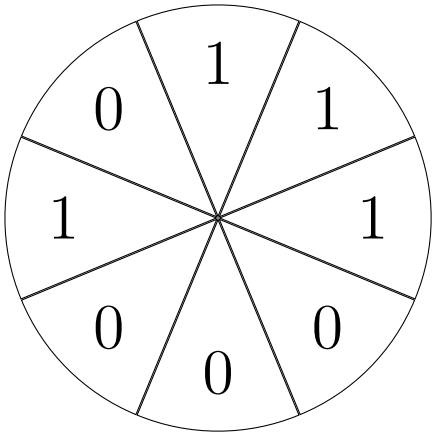
> **Note**
>
> 事实上，该算法的准确复杂度应为 $O(|E|)$ 而非 $O(|V|+|E|)$，这是因为该算法的实现方式可以采取依赖于边而不依赖于点的方法，通过维护剩余边的总链表来进行下一步回路的寻找.

如果需要输出字典序最小的欧拉路或欧拉回路，则需要将边排序，时间复杂度为 $\Theta(|E|\log|E|)$ 或 $\Theta(|E|)$（使用计数排序或者基数排序）.

应用

有向欧拉图可用于计算机译码.

设有 m 个字母，希望构造一个有 m^n 个扇形的圆盘，每个圆盘上放一个字母，使得圆盘上每连续 n 位对应长为 n 的符号串. 转动一周（m^n 次）后得到由 m 个字母产生的长度为 n 的 m^n 个各不相同的符号串.



构造如下有向欧拉图：

设 $S=\{a_1,a_2,\cdots,a_m\}$，构造 $D=\langle V,E\rangle$，如下：

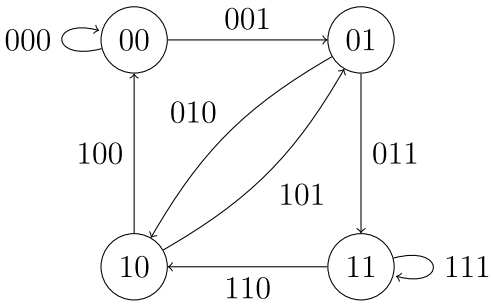
$V=\{a_{i_1}a_{i_2}\cdots a_{i_{n-1}}\mid a_i\in S,1\le i\le n-1\}$

$E=\{a_{j_1}a_{j_2}\cdots a_{j_{n-1}}\mid a_j\in S,1\le j\le n\}$

规定 D 中顶点与边的关联关系如下：

顶点 $a_{i_1}a_{i_2}\cdots a_{i_{n-1}}$ 引出 m 条边：$a_{i_1}a_{i_2}\cdots a_{i_{n-1}}a_r,r=1,2,\cdots,m.$

边 $a_{j_1}a_{j_2}\cdots a_{j_{n-1}}$ 引入顶点 $a_{j_2}a_{j_3}\cdots a_{j_n}$.



这样的 D 是连通的，且每个顶点入度等于出度（均等于 m），所以 D 是有向欧拉图.

任求 D 中一条欧拉回路 C，取 C 中各边的最后一个字母，按各边在 C 中的顺序排成圆形放在圆盘上即可.

例题

洛谷 P2731 骑马修栅栏[1]

给定一张有 500 个顶点的无向图，求这张图的一条欧拉路或欧拉回路. 如果有多组解，输出最小的那一组.

在本题中，欧拉路或欧拉回路不需要经过所有顶点.

边的数量 m 满足 $1\le m\le 1024$.

解题思路

本题为 Hierholzer 算法的直接应用.

保存答案可以使用 `std::stack<int>`，因为如果找的不是回路的话必须将那一部分放在最后.

注意，不能使用邻接矩阵存图，否则时间复杂度会退化为 $\Theta(nm)$. 由于需要将边排序，建议使用前向星或者 `std::vector` 存图. 示例代码使用 `std::vector`.

示例代码

```cpp
#include <algorithm>
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

struct edge {
  int to;
  bool exists;
  int revref;

```

```cpp
  bool operator<(const edge& b) const { return to < b.to; }
};

vector<edge> beg[505];
int cnt[505];

constexpr int dn = 500;
stack<int> ans;

void Hierholzer(int x) {  // 关键函数
  for (int& i = cnt[x]; i < (int)beg[x].size();) {
    if (beg[x][i].exists) {
      edge e = beg[x][i];
      beg[x][i].exists = beg[e.to][e.revref].exists = false;
      ++i;
      Hierholzer(e.to);
    } else {
      ++i;
    }
  }
  ans.push(x);
}

int deg[505];
int reftop[505];

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  for (int i = 1; i <= dn; ++i) {
    beg[i].reserve(1050);  // vector 用 reserve 避免动态分配空间，加快速度
  }

  int m;
  cin >> m;
  for (int i = 1; i <= m; ++i) {
    int a, b;
    cin >> a >> b;
    beg[a].push_back(edge{b, true, 0});
    beg[b].push_back(edge{a, true, 0});
    ++deg[a];
    ++deg[b];
  }

  for (int i = 1; i <= dn; ++i) {
    if (!beg[i].empty()) {
      sort(beg[i].begin(), beg[i].end());  // 为了要按字典序贪心，必须排序
    }
  }

  for (int i = 1; i <= dn; ++i) {
    for (int j = 0; j < (int)beg[i].size(); ++j) {
      beg[i][j].revref = reftop[beg[i][j].to]++;
```

```
64      }
65    }
66
67    int bv = 0;
68    for (int i = 1; i <= dn; ++i) {
69      if (!deg[bv] && deg[i]) {
70        bv = i;
71      } else if (!(deg[bv] & 1) && (deg[i] & 1)) {
72        bv = i;
73      }
74    }
75
76    Hierholzer(bv);
77
78    while (!ans.empty()) {
79      cout << ans.top() << '\n';
80      ans.pop();
81    }
82  }
```

习题

- SGU 101 Domino[2]
- POJ 1780 Code[3]
- 洛谷 P1127 词链[4]
- 洛谷 P1333 瑞瑞的木棍[5]
- 洛谷 P1341 无序字母对[6]
- 洛谷 P6066 [USACO05JAN]Watchcow S[7]
- 洛谷 P6628 [省选联考 2020 B 卷] 丁香之路[8]
- 洛谷 P3520 [POI 2011] SMI-Garbage[9]

外部参考资料

[1] 洛谷 P2731 骑马修栅栏 

[2] SGU 101 Domino 

[3] POJ 1780 Code 

[4] 洛谷 P1127 词链 

[5] 洛谷 P1333 瑞瑞的木棍 

[6] 洛谷 P1341 无序字母对 

[7] 洛谷 P6066 [USACO05JAN]Watchcow S 

[8] 洛谷 P6628 [省选联考 2020 B 卷] 丁香之路 

[9] 洛谷 P3520 [POI 2011] SMI-Garbage 

11.18 哈密顿图

定义

通过图中所有顶点一次且仅一次的通路称为哈密顿通路.

通过图中所有顶点一次且仅一次的回路称为哈密顿回路.

具有哈密顿回路的图称为哈密顿图.

具有哈密顿通路而不具有哈密顿回路的图称为半哈密顿图.

性质

设 $G=\langle V,E\rangle$ 是哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G-V_1)\leq |\ V_1|$．其中 $p(x)$ 为 x 的连通分支数.

推论：设 $G=\langle V,E\rangle$ 是半哈密顿图，则对于 V 的任意非空真子集 V_1，均有 $p(G-V_1)\leq |\ V_1|+1$．其中 $p(x)$ 为 x 的连通分支数.

完全图 $K_{2k+1}(k\geq 1)$ 中含 k 条边不重的哈密顿回路，且这 k 条边不重的哈密顿回路含 K_{2k+1} 中的所有边.

完全图 $K_{2k}(k\geq 2)$ 中含 $k-1$ 条边不重的哈密顿回路，从 K_{2k} 中删除这 $k-1$ 条边不重的哈密顿回路后所得图含 k 条互不相邻的边.

充分条件

设 G 是 $n(n\geq 2)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i,v_j，均有 $d(v_i)+d(v_j)\geq n-1$，则 G 中存在哈密顿通路.

推论 1：设 G 是 $n(n\geq 3)$ 的无向简单图，若对于 G 中任意不相邻的顶点 v_i,v_j，均有 $d(v_i)+d(v_j)\geq n$，则 G 中存在哈密顿回路，从而 G 为哈密顿图.

推论 2：设 G 是 $n(n\geq 3)$ 的无向简单图，若对于 G 中任意顶点 v_i，均有 $d(v_i)\geq \frac{n}{2}$，则 G 中存在哈密顿回路，从而 G 为哈密顿图.

设 D 为 $n(n\geq 2)$ 阶竞赛图，则 D 具有哈密顿通路.

若 D 含 $n(n\geq 2)$ 阶竞赛图作为子图，则 D 具有哈密顿通路.

强连通的竞赛图为哈密顿图.

若 D 含 $n(n\geq 2)$ 阶强连通的竞赛图作为子图，则 D 具有哈密顿回路.

11.19 二分图

引入

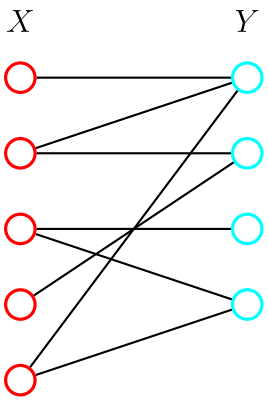
二分图，又称二部图，是一类结构特殊的图．它的顶点集可以划分为两个互不相交的子集，使得图中的每条边都连接这两个集合之间的一对点，而不会连接同一集合内部的点.

得益于这种简单的结构，二分图不仅展现出许多优雅的性质，也广泛应用于现实生活中的建模场景，例如任务分配、推荐系统、匹配市场等．许多在一般图上困难的优化问题，在二分图上都可以高效、准确地求解.

定义

如果图 $G=(V,E)$ 的顶点集 V 可以分为两个互不相交的子集 X 和 Y，使得每条边 $e\in E$ 的两个端点都分别属于 X 和 Y，就称图 G 是一个 **二分图**（bipartite graph）．集合 X 和 Y 常称作它的两个 **部分**（part），或者分别称为二分图的左部和右部．当二分图的两个部分 X 和 Y 已知时，也可以用三元组 (X,Y,E) 来表示二分图 G.

一个典型的二分图如下图所示.



树、偶环、网格图等都是常见的二分图的例子.

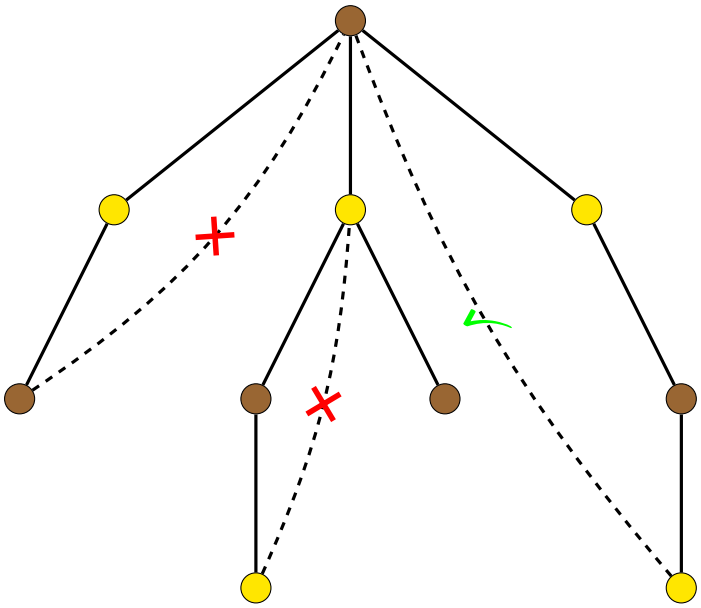
刻画

二分图也可以由下列性质等价地定义：

- 图 G 是可 2-着色的. 也就是说，可以用至多两种颜色给图的所有顶点染色，并且保证相邻顶点颜色不同.
- 图 G 中不存在奇数长度的环.

很显然，第一条性质与二分图的定义等价：只需要将二分图的两个部分各染一种颜色就好了.

第二条性质稍微复杂一些. 可以考虑用两种颜色尝试给图 G 染色. 因为不同连通分量之间染色互不干扰，只需要逐个考虑连通分量就好了. 任选连通分量中的一个顶点 s，进行 DFS，并记录连通分量中每个顶点 v 在 DFS 生成树上与 s 的距离（即深度）. 从 s 开始，在 DFS 生成树上进行归纳可知，如果存在一种可行的染色方法，一定是根据每个顶点 v 到起点 s 的距离的奇偶性分别染成两种颜色.



继而考虑那些不在生成树中的边. 如果这些非树边的两个端点的颜色都不一样，就说明当前的染色方案可行；否则，就不存在可行的方案. 进一步地，两个顶点颜色不同，当且仅当它们到树根 s 的距离一奇一偶，这又等价于加入该非树边形成的是一个偶环而非奇环. 因此，只要没有奇环，这些非树边必然连接颜色不同的点，进而整张图都可以用两种颜色染色，图就一定是二分图.

判定

要判定一个图是不是二分图，只需要利用上述等价刻画，尝试给这张图染色即可．为此，可以使用 DFS→11.4@p2612 或者 BFS→11.5@p2616 来遍历这张图．如果发现了奇环，也就是出现无法染色的情况，那么就不是二分图；否则，就是二分图．

具体流程如下：

- 遍历顶点，如果发现还没有染色的顶点，说明发现新的连通分量．
- 任选一种颜色给该顶点染色，并以它为起点做 DFS→11.4@p2612 或者 BFS→11.5@p2616，尝试给该连通分量染色．
- 遍历相邻的顶点时，如果发现已经染色的顶点，检查颜色是否与当前顶点相同．相同，则不是二分图，直接返回；否则，继续遍历．
- 如果发现尚未染色的顶点，就将它染上与当前顶点相反的颜色．

参考代码如下：

参考代码

```cpp
int n;
std::vector<std::vector<int>> gr;
std::vector<int> colors, vis;

// Depth-first search to color vertices.
bool dfs(int cr) {
  vis[cr] = true;
  for (int nt : gr[cr]) {
    if (vis[nt]) {
      if (colors[cr] == colors[nt]) return false;
    } else {
      colors[nt] = colors[cr] ^ 1;
      if (!dfs(nt)) return false;
    }
  }
  return true;
}

// Check whether the graph GR is bipartite.
// If so, the vector COLORS will store a feasible coloring.
bool check_bipartite() {
  for (int i = 1; i <= n; ++i) {
    // Check connected components one by one.
    if (!vis[i]) {
      colors[i] = 0;
      if (!dfs(i)) return false;
    }
  }
  return true;
}

```

时间复杂度为 $O(|V|+|E|)$.

应用

由于结构简单，很多图论优化问题都可以在二分图上高效解决．详情参考相关主条目．

- 极大团（平凡）
- 最小点着色（平凡）
- 最小边着色→11.22@p2896
- 最大匹配→11.24.2@p2959
- 最小边覆盖→11.24.1@p2949
- 最小点覆盖→11.24.2@p2959
- 最大独立集→11.24.2@p2959
- 最大权匹配→11.24.3@p2983
- 二分图博弈→9.18.2@p1903

11.20 平面图

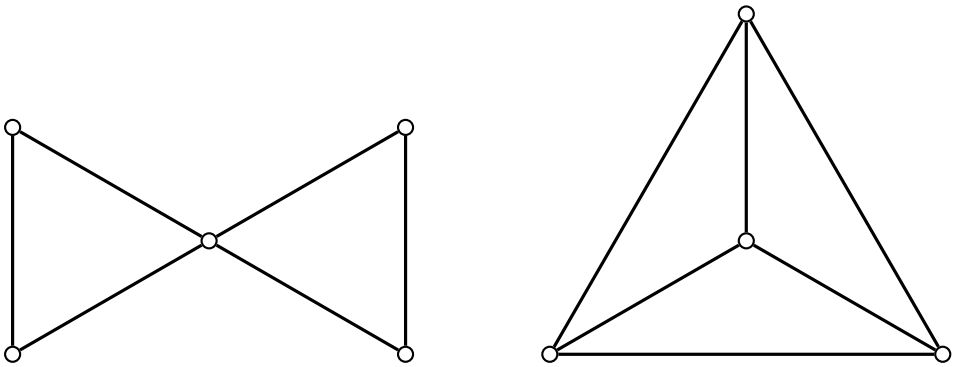
本文介绍（可）平面图及其相关概念.

平面图

如果图 G 能画在平面 S 上，即除顶点处外无边相交，则称 G 可嵌入平面 S，G 为 **可平面图**（planar graph）. 画出的没有边相交的图称为 G 的平面表示或 **平面嵌入**（planar embedding）. 可平面图的这个平面嵌入也称为 **平面图**（plane graph）.

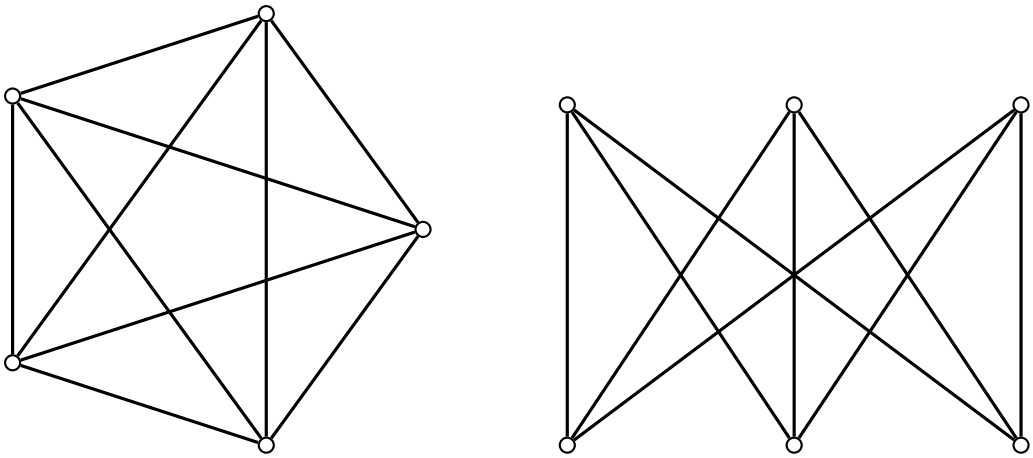
> **「平面图」**
>
> 不同中文文本中，「平面图」的含义可能不同. 在本文的定义中，可平面图是一个图论对象，它可能以不同的方式嵌入平面中；平面图则是一个几何对象，除了图论结构外，它还需要指定图的绘制方式. 同一个可平面图往往对应着多个平面图. 因此，本文叙述的结论如果只依赖于图论结构，将使用「可平面图」一词；如果还依赖于图的平面嵌入方式，将使用「平面图」一词.

以下是平面图的简单例子：



（左：蝴蝶图；右：4 阶完全图 K_4）

以下是不可平面图的简单例子：



（左：5 阶完全图 K_5；右：两部分各 3 个顶点的完全二分图 $K_{3,3}$）

性质

本节介绍平面图的性质.

面及其次数

设 G 是平面图，由 G 的边将 G 所在的平面划分成若干个区域，每个区域称为 G 的一个 **面**（face）. 其中，无界的面称为 **无限面**（unbounded face）或 **外部面**（external face），有界的称为有限面或内部面. 每一个平面图有且仅有一个外部面.

包围每个面的所有边组成的回路称为该面的 **边界**（boundary），并称边界中的边与该面 **关联**（incident）. 边界的长度称为该面的 **次数**（degree）. 计算面的次数时，每条割边都算作两次. 平面图中所有面的次数之和等于边数 $|E|$ 的 2 倍.

平面图中，1 次面的边界对应于图的自环，2 次面的边界通常对应于图的一对重边[1]. 顶点数 $|V| \geq 3$ 的简单连通平面图中，所有面次数都至少为 3.

欧拉公式

平面图的一个重要性质是 **欧拉公式**（Euler's formula）. 它给出了图的顶点数 $|V|$、边数 $|E|$ 和面数 $|F|$ 之间的关系.

欧拉公式

对于连通的平面图 G，有

$$|V|-|E|+|F|=2.$$

证明

对于面数 $|F|$ 应用数学归纳法. 归纳起点是 $|F|=1$. 此时，平面图有且只有一个外部面，全部边都是割边. 所以，图 G 是一棵树，必然有 $|E|=|V|-1$，代入欧拉公式就可以发现它成立. 假设欧拉公式对于面数 $|F|=k$ 的平面图成立. 对于面数 $|F|=k+1$ 的平面图 G，必然存在非割边 e，它是两个不同的面的公共边. 将边 e 从图中删除，得到图 $G-e$，它有 $|V|$ 个顶点、$|E|-1$ 条边和 $|F|-1$ 个面. 由归纳假设，对图 $G-e$ 成立欧拉公式，即 $|V|-(|E|-1)+(|F|-1)=2$，整理就得到关于图 G 的欧拉公式. 所以，根据数学归纳法，欧拉公式对于所有平面图都成立.

[1]但这并非唯一的可能. 两个嵌套的自环也会形成二次面. 另外，有二次面未必意味着图不是简单的，例如，一个只有一条边的图中，唯一的面（即外部面）也是二次的.

推论

对于有 k 个连通分支的平面图 G，有

$$|V|-|E|+|F|=k+1.$$

证明

图 G 的每个连通分支都是平面图，但是这些连通分支共用同一个外部面．所以，直接对这些连通分支应用欧拉公式，并累加到一起，总顶点数和总边数都是正确的，但是总面数多了 $(k-1)$，因为唯一的外部面总共计数了 k 次．将这一修正考虑在内，就得到 $|V|-|E|+|F|=2k-(k-1)=k+1$.

由此，可以推出平面图的边与顶点的数量关系．

定理

对于有 k 个连通分支的平面图 G，如果图 G 的每个面次数都至少为 $l\geq 3$，那么有

$$|E|\leq \frac{l}{l-2}(|V|-k-1).$$

证明

因为 G 的各面的次数至少为 l，所以所有面的次数和至少为 $l|F|$，亦即 $2|E|\geq l|F|$．代入欧拉公式的推论 $|V|-|E|+|F|=k+1$，就得到

$$2|E|\geq l(k+1-|V|+|E|).$$

利用 $l\geq 2$ 解出 $|E|$，就得到

$$|E|\leq \frac{l}{l-2}(|V|-k-1).$$

推论

设 G 是简单可平面图，且 $|V|\geq 3$，那么，有

$$|E|\leq 3|V|-6.$$

证明

当 G 连通时，所有面次数都至少是 3．在上述定理中，取 $k=1$ 且 $l=3$，就得到 $|E|\leq 3|V|-6$.

当 G 不连通时，分为两种情形：

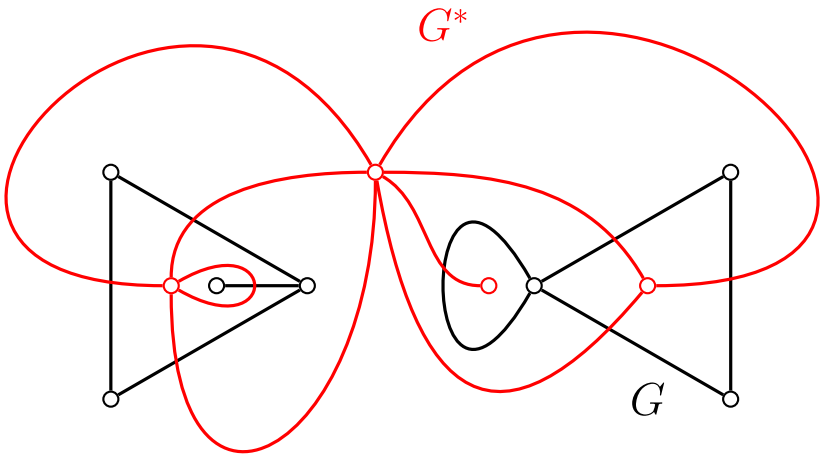
- 如果存在连通分支顶点数至少是 3，那么对这些顶点数至少为 3 的连通分支可以分别建立不等式 $|E_i|\leq 3|V_i|-6$．因为那些顶点数小于 3 的连通分支一定有 $|E_i|\leq |V_i|\leq 3|V_i|$．将所有连通分支对应的不等式相加，就得到 $|E|\leq 3|V|-6$.
- 如果所有连通分支顶点数都小于 3，那么整体一定有 $|E|\leq |V|$．又因为 $|V|\geq 3$ 时，$|V|\leq 3|V|-6$，所以 $|E|\leq 3|V|-6$ 仍然成立．

综上，命题得证．

这一推论说明，简单可平面图是稀疏图．

对偶图

平面图都有相应的（几何）对偶图．



设 G 是平面图，可以绘制图 G^* 如下：

1. 在 G 的每个面 f_i 内部都绘制一个点 v_i^*.
2. 对 G 的每条边 e，如果 e 在面 f_i 和 f_j 的公共边界上，就绘制一条连接 v_i^* 和 v_j^* 的边 e^*，使之与 e 恰相交一次，且不与其他图 G 或图 G^* 的边相交. 特别地，当 e 只出现在一个面 f_i 的边界上时，需要绘制一条与 v_i^* 关联的自环，使之与 e 相交.

这样得到的图 G^* 就称作图 G 的 **对偶图**（dual graph）.

定理

设图 G^* 是平面图 G 的对偶图. 那么，图 G^* 是连通的平面图. 而且，图 G^{**} 与 G 同构，当且仅当 G 是连通图.

证明

图 G^* 是平面图这一点可以由它的构造过程保证. 还需要证明图 G^* 是连通的. 对于图 G^* 中任意两个顶点 v_i^*, v_j^*，设平面中连接 v_i^* 和 v_j^* 的直线段经过图 G 中的面和边依次为 $f_i, e_{s_1}, f_{s_1}, \cdots, f_{s_{r-1}}, e_{s_r}, f_j$，它们分别对应对偶图中的顶点和边 $v_i^*, e_{s_1}^*, v_{s_1}^*, \cdots, v_{s_{r-1}}^*, e_{s_r}^*, v_j^*$. 由图 G^* 的构造可知，序列中相邻的顶点和边是相关联的，所以，这描述了图 G^* 中的一条途径. 所以，图 G^* 是连通的.

图 G^{**} 是图 G^* 的对偶图，必然是连通的. 所以，G 与 G^{**} 同构，必要条件是图 G 连通. 接下来，需要证明这一条件也是充分的. 为此，只需要证明当图 G 连通时，图 G 满足图 G^* 的对偶图的构造要求. 因为图 G^* 的边和图 G 的边天然是对应的，所以，只需要证明图 G^* 的每一个面都恰好包含图 G 的一个顶点. 对于图 G^* 的任一个面 f^*，设 e^* 是它边界上的一条边，那么图 G 中相对应的边 e 的端点之一必然在面 f^* 之内；因此，面 f^* 中至少存在图 G 的一个顶点. 由于图 G^* 和图 G 都是连通的，欧拉公式成立；而图 G 和图 G^* 边数相同，图 G 的面数等于图 G^* 的顶点数，所以图 G 的顶点数就等于图 G^* 的面数. 所以，图 G^* 的每个面都恰好只有图 G 的一个顶点. 命题得证.

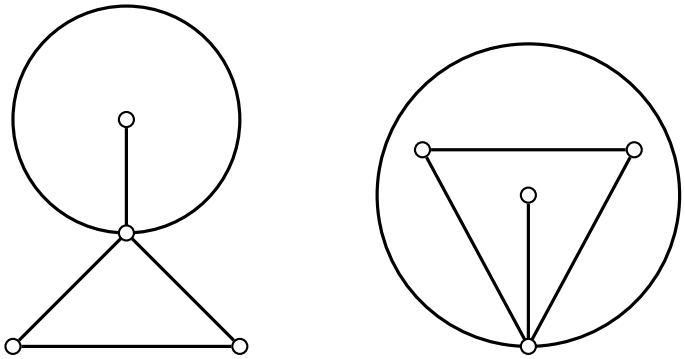
平面图与其对偶图的结构之间有很多对应关系：

- G 中的面对应 G^* 中的点，G 中的边对应 G^* 中的边，G 中的点对应 G^* 中的面.
- G 中的自环对应 G^* 中的割边，G^* 中的自环对应 G 中的割边.
- G 中的边割集对应 G^* 中的回路，G^* 中的回路对应 G 中的边割集.

需要注意的是，对偶图的概念仅对具体的平面图成立，而无法定义在任意可平面图上．事实上，两个同构的平面图的对偶图未必是同构的．也就是说，同一个图的不同平面嵌入的对偶图可能并不相同．

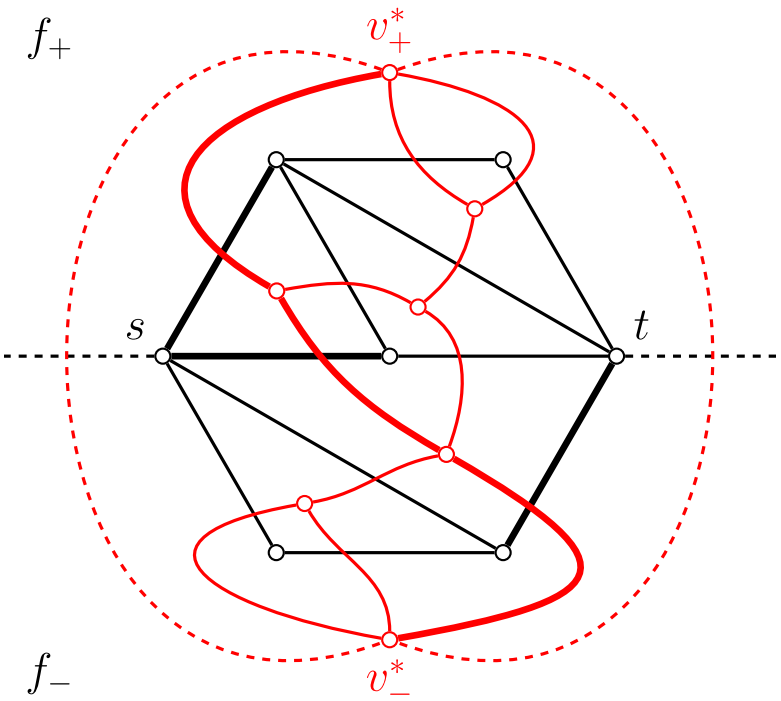
例子

下图画了两个同构的平面图，它们的对偶图并不同构．



对偶图不同构的原因是，右图有一次面，它的对偶图有一度顶点，而左图没有．

将可平面图的问题转化到对偶图上，有时更容易解决．一个典型的例子是，可平面图 最小割→11.23.3@p2932 问题可以转化为对偶图 最短路→11.9.1@p2735 问题．设 G 是带边权的可平面图，s,t 是它的两个顶点，需要求最小的 s-t 割．



如图所示，通过选取合适的平面嵌入，使得 s,t 出现在图 G 外部面边界上．另外，添加自 s 和 t 延伸出去的射线，将外部面分为两部分 f_+ 和 f_-．基于该图，建立对偶图，并将边权赋给对偶图中的对应边．那么，对偶图 G^* 中面 f_+ 和 f_- 对应顶点之间的路径（红色粗线）就和图 G 的 s-t 边割集（黑色粗线）之间一一对应，且二者权值相同．这样，求解对偶图中的最短路，就得到了图 G 中的最小 s-t 割．

一个常见的误区是根据上述转化方法宣称平面图最小割等于对偶图最短路．事实上，它只适用于存在 G 的平面嵌入使得 s,t 共面的情况，只是在考察此知识点的算法竞赛题目中，给出的图往往具有并且附带这样的平面嵌入．这里给出一个可以用于判定其存在性的定理．

定理

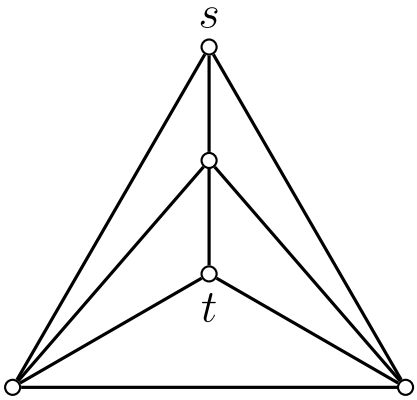
对于可平面图 $G=(V,E)$ 的两个顶点 s,t，存在 G 的平面嵌入使得 s,t 处于同一个面上，当且仅当 $(V,E\cup\{(s,t)\}$ 是可平面图.

证明

如果存在 G 的平面嵌入使得 s,t 处于同一个面上，那么就可以在这个面内添加一条边 (s,t)，保持图的平面性.

如果 $(V,E\cup\{(s,t)\})$ 是可平面图，那么任取它的一个平面嵌入，s,t 都同处边 (s,t) 所处的面. 删除 (s,t) 之后，s,t 仍然同处一面. 命题得证.

例如，下图中添加边 (s,t) 之后得到非平面图 K_5，故而不具有这样的平面嵌入，上述转化不适用.



更多结果

当然，平面图还有很多著名的结果. 本节简单列举它们，但并不做出讨论.

四色定理

（没有自环的）平面图都是可 4-着色的.

Fáry 定理

简单可平面图总是存在一种平面嵌入，使得图的所有边都是直线段.

定理（Wood）

可平面图至多只有 $8|V|-16$ 个极大团.

定理（Tutte）

4-点连通的可平面图都是哈密顿图.

判定

本节讨论给定一个图，判定它是不是可平面图的方法.

禁用图

可平面图最经典的刻画方式是利用 **禁用图**（forbidden graph）给出的.

首先，K_5 和 $K_{3,3}$ 不是可平面图.

定理

K_5 和 $K_{3,3}$ 不是可平面图.

证明

前文说明，$|V| \geq 3$ 的简单连通平面图都需要满足

$$|E| \leq \frac{l}{l-2}(|V|-2).$$

其中，l 是面次数的最小值. 对于 K_5，有 $l=3$，$|V|=5$，$|E|=10$，所以 K_5 不可能画成平面图. 对于 $K_{3,3}$，有 $l=4$，$|V|=6$，$|E|=9$，所以 $K_{3,3}$ 不可能画成平面图.

事实上，它们就是使得一个图不可平面的最小结构. 也就是说，只要图不（以某种方式）包含这两个图为子结构，该图就一定是可平面的.

第一个可平面性判定定理是 Kuratowski 定理. 它用到了图同胚的概念：若两个图 G_1 与 G_2 同构，或通过反复插入或消去 2 度顶点后是同构的，则称二者是 **同胚的**（homeomorphic）. 由此，可以叙述如下结果：

Kuratowski 定理

图 G 是可平面图，当且仅当 G 不含与 K_5 或 $K_{3,3}$ 同胚的子图.

另外一个与此相关的定理是 Wagner 定理. 它利用收缩操作来刻画可平面图. 收缩操作是指，重复多次将图的一条边收缩为一个点. 由此，可以叙述如下结果：

Wagner 定理

图 G 是可平面图，当且仅当 G 中没有可以收缩到 K_5 或 $K_{3,3}$ 的子图.

可平面图不包含这些类型的子图相对显然，所以这两个定理的关键部分都在于相应的禁用图条件的充分性. 由于与 K_5 或 $K_{3,3}$ 同胚的子图一定可以收缩到它们，反过来却未必成立，所以 Kuratowski 定理提供了一个更弱的也更容易检验的判定可平面图的条件.

平面性判定算法

尽管看起来并不容易，平面性判定问题实际上有很多线性算法. 但是，由于这些算法的实现通常都比较复杂，它们几乎从未出现在算法竞赛中.

最早的线性算法是 Hopcroft–Tarjan 算法[2]，但它的实现相当复杂. de Fraysseix–Ossona de Mendez–Rosenstiehl 算法（也称为 LR 平面性算法）[3][4][5] 进一步改进了 Hopcroft–Tarjan 算法的流程，是目前最优秀的平面性判定算法之一. Python 的 NetworkX 库中就 实现[1] 了这一算法.

另外一个同样优秀的算法是 Boyer–Myrvold 算法 [6][7]. 它可以在线性时间内判定给定图是否可平面. 而且，如果图是可平面的，算法将输出一个平面嵌入；否则，算法将输出一个 Kuratowski 子图（即与 K_5 或 $K_{3,3}$ 同胚的子图）. C++ 的 Boost 库就 实现[2] 了这一算法.

---

[2] Hopcroft, John, and Robert Tarjan. "Efficient planarity testing." Journal of the ACM (JACM) 21, no. 4 (1974): 549-568.

[3] De Fraysseix, Hubert, Patrice Ossona De Mendez, and Pierre Rosenstiehl. "Trémaux trees and planarity." International Journal of Foundations of Computer Science 17, no. 05 (2006): 1017-1029.

[4] De Fraysseix, Hubert. "Trémaux trees and planarity." Electronic Notes in Discrete Mathematics 31 (2008): 169-180.

[5] Brandes, Ulrik. "The left-right planarity test." Manuscript submitted for publication 3 (2009).

[6] Boyer, John M., and Wendy J. Myrvold. "Stop Minding Your p's and q's: A Simplified O (n) Planar Embedding Algorithm." In SODA, vol. 99, pp. 140-146. 1999.

[7] Boyer, John M., and Wendy J. Myrvold. "Simplified o (n) planarity by edge addition." Graph Algorithms and Applications 5 (2006): 241.

更多相关算法可以参考文末提供的文献.

特殊的平面图

本节介绍几类特殊的可平面图.

极大平面图

对于简单可平面图 G，如果在它的任意不相邻顶点间添加边，所得图都不再是可平面图，就称 G 为**极大可平面图**（maximal planar graph）. 极大可平面图的平面嵌入称为 **极大平面图**.

定理

极大可平面图 G 必然连通. 而且，当顶点数 $|V| \geq 3$ 时，图 G 没有割边.

证明

如果可平面图 G 不连通，那么任选它的一个平面嵌入，都可以选择属于不同连通分支的两个顶点，在外部面内连接起来，得到的图显然仍然是平面图，这说明图 G 不是极大可平面图. 所以，图 G 是极大可平面图，就一定连通.

如果可平面图 G 顶点数 $|V| \geq 3$，且 G 有割边 $e=(u,v)$，那么，删去边 e 后的图 $G-e$ 中恰有两个连通分支，且 u,v 属于不同的连通分支. 假设 v 所在连通分支至少有两个顶点. 那么，可以先将 u 所在连通分支 G_1 画在平面上，并选取图 G_1 中边界含有 u 的任意面 f，并将另一个连通分支 G_2 画在面 f 中. 由于 G_2 是简单图，它的外部面的边界一定不是一个自环，故而至少还存在另一个顶点 $w \neq u,v$. 将 v,w 分别连接到 u 上，就得到包含 G 为子图的平面图. 所以，图 G 不是极大可平面图. 因此，顶点数 $|V| \geq 3$ 的极大可平面图一定没有割边.

极大平面图的结构可以更准确地描述.

定理

对于顶点数 $|V| \geq 3$ 的平面图 G，它是极大平面图当且仅当它是简单图，且它的每个面次数均为 3.

证明

条件的充分性显然. 只需要说明必要性，即要证明：顶点数 $|V| \geq 3$ 的极大平面图 G 中，每个面次数都是 3. 由于图 G 是连通简单平面图且 $|V| \geq 3$，所以全部面的次数都至少是 3. 所以，假设命题不成立，就一定存在一个面 f 的边界长度至少是 4. 又因为图 G 不存在割边，该边界只能是一个环. 设这个环是 $v_1v_2v_3v_4\cdots v_1$. 那么，如果 v_1 与 v_3 不相邻，那么在面 f 内连接 v_1 和 v_3 不会破坏平面性，与 G 的极大性矛盾，所以 v_1 与 v_3 相邻；同理，v_2 与 v_4 相邻. 但是，边 (v_1,v_3) 和 (v_2,v_4) 都不会出现在面 f 中. 这意味着，两条边必然在面 f 的外部. 但这是不可能的：无论如何绘制，这两条边都必然相交. 所以，图 G 中不存在高于 3 次的面. 原命题得证.

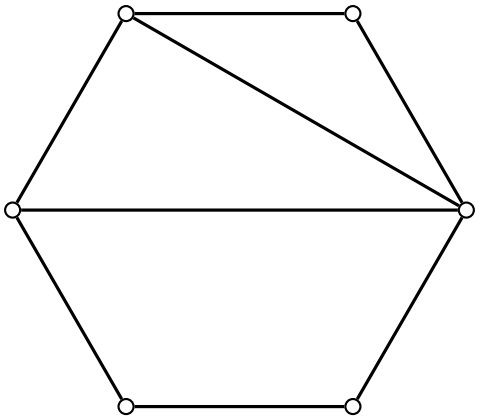
推论

对于顶点数 $|V| \geq 3$ 的图 G，总是有边数 $|E| = 3|V| - 6$ 且面数 $|F| = 2|V| - 4$.

由于极大平面图中，每个面都是由三条边围成，所以极大平面图也称为 **平面三角剖分**（plane triangulation）.

外平面图

设 G 为可平面图，若 G 存在平面嵌入 $\tilde{G}$，使得 G 中所有顶点都在 $\tilde{G}$ 的一个面的边界上，则称 G 为**外可平面图**（outerplanar graph）．这一嵌入也称为外平面嵌入或**外平面图**．通常将边界经过所有顶点的那个面绘制为外部面．



外可平面图都是可平面图，反之未必成立．外可平面图同样可以使用禁用图刻画．

定理

一个图 G 是外平面图有当且仅当 G 中不含与 K_4 或 $K_{2,3}$ 同胚的子图．

对于外可平面图，同样可以讨论极大外可平面图的概念．对于简单外可平面图 G，如果在它的任意不相邻顶点间添加边，所得图都不再是外可平面图，就称 G 为**极大外可平面图**（maximal outerplanar graph）．极大外可平面图的外平面嵌入称为**极大外平面图**．极大外平面图其实就是平面上多边形的三角剖分．

定理

对于顶点数 $|V|\geq 3$ 的极大外平面图 G，且所有顶点都在外部面的边界上，那么图 G 恰有 $|V|-2$ 个内部面．

证明

对 $|V|$ 应用数学归纳法．归纳起点是 $|V|=3$．此时，图 G 是三元环，只有 1 个内部面，命题成立．假设命题对于 $|V|=k$ 成立．现在要证明，当 $|V|=k+1$ 时，命题仍然成立．

首先，图 G 一定存在 2 度顶点．否则，除了外部面边界上相邻的顶点外，所有顶点都需要和第三个顶点相连接．不妨将外部面边界上的顶点顺次编号，并对每一个 $i=1,2,\cdots,k+1$，都定义 $f(i)$ 为与顶点 i 连接且编号不与之相邻的顶点的最小编号．考虑 $f(i)$ 的可能取值．首先，$1<f(1)$．由于点 1 已经和 $f(1)$ 连接，点 2 与 $f(2)$ 的连线不能越过边 $(1,f(1))$，就必然有 $1<2<f(2)<f(1)$．同理，$2<3<f(3)<f(2)$．由于顶点只有有限多个，这个逐渐缩小的过程必然在有限步后终止．令 i^* 为满足 $1<\cdots<i-1<i<f(i)<f(i-1)<\cdots<f(1)$ 的编号 i 最大值．那么，由于点 i^* 和点 $f(i^*)$ 不相邻，必然有 $i^*<i^*+1<f(i^*)$．而重复之前的论述，仍应该有 $i^*<i^*+1<f(i^*+1)<f(i^*)$，这与 i^* 的最大性矛盾．这一矛盾说明，图 G 必然存在 2 度顶点．

设 v 就是一个 2 度顶点．将这一顶点从图 G 中删除，就得到顶点数为 k 的外平面图 $G-v$．它必然是极大外平面图，否则在它上面合法添加边的方法，必然对图 G 也适用．由归纳假设，图 $G-v$ 恰有 $k-2$ 个内部面，而删去顶点 v 时，恰好减少了一个图 G 的内部面．所以，图 G 内部面数目为 $k-1$．命题得证．

定理

对于顶点数 $|V| \geq 3$ 的外平面图 G，且所有顶点都在外部面的边界上，那么图 G 是极大外平面图，当且仅当图 G 的外部面边界是长为 $|V|$ 的环，且所有内部面边界均是长为 3 的环.

证明

充分性显然. 事实上，考虑连接外部面边界上的两个不相邻顶点. 如果连接发生在外部面中，那么，所有顶点无法都出现在一个面的边界上；否则，它们的连线必然与内部面的边界相交.

接下来，证明必要性. 假设图 G 的外部面边界 $v_1v_2v_3\cdots v_nv_1$ $(n=|V|)$ 不是一个环. 那么，它会重复经过一个顶点多次，亦即存在 $i \neq j$ 且 $i-j \neq \pm 1 \pmod n$ 使得 $v_i = v_j$. 不妨设 $1 < i < j < n$. 此时，与 v_{i-1} 相关联的边只能出现在回路 $v_jv_{j+1}\cdots v_nv_1\cdots v_{i-1}v_i$ 围成的有界区域内部，与 v_{i+1} 相关联的边只能出现在回路 $v_iv_{i+1}\cdots v_{j-1}v_j$ 围成的有界区域内部，所以 v_{i-1} 和 v_{i+1} 无法相邻. 可以在外部面内添加一条连接 v_{i-1} 和 v_{i+1} 的边 e，得到图 $G+e$. 这显然也是平面图，且外部面边界上包含所有顶点. 这就与图 G 的极大外平面性矛盾. 所以，图 G 的外部面必然是长度为 $|V|$ 的环. 而图 G 内部面边界均为长为 3 的环的原因，和极大平面图一致，不再赘述.

推论

对于顶点数 $|V| \geq 3$ 的极大外平面图 G，有：

1. $|E| = 2|V| - 3$.
2. G 中至少有 3 个顶点度数小于等于 3，且至少有 2 个顶点度数为 2.
3. G 的点连通度为 2.

习题

- Luogu P3209 [HNOI2010] 平面图判定[3]
- Luogu P3249 [HNOI2016] 矿区[4]
- Luogu P4001 [ICPC-Beijing 2006] 狼抓兔子[5]
- Luogu P4073 [WC2013] 平面图[6]
- Luogu P7295 [USACO21JAN] Paint by Letters P[7]

参考资料与注释

- Planar graph - Wikipedia[8]
- Planarity testing - Wikipedia[9]

- Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. Graph theory with applications. Vol. 290. London: Macmillan, 1976.
- Diestel, Reinhard. Graph theory. Vol. 173. Springer Nature, 2025.
- Patrignani, Maurizio. "Planarity Testing and Embedding." (2013): 1-42.

[1] 实现 [2] 实现 [3] Luogu P3209 [HNOI2010] 平面图判定

[4] Luogu P3249 [HNOI2016] 矿区 [5] Luogu P4001 [ICPC-Beijing 2006] 狼抓兔子 [6] Luogu P4073 [WC2013] 平面图

[7] Luogu P7295 [USACO21JAN] Paint by Letters P [8] Planar graph - Wikipedia [9] Planarity testing - Wikipedia

11.21 弦图

弦图是一种特殊的图，很多在一般图上的 NP-Hard 问题在弦图上都有优秀的线性时间复杂度算法.

一些定义与性质

子图：点集和边集均为原图点集和边集子集的图.

导出子图（诱导子图）：点集为原图点集子集，边集为所有满足 **两个端点均在选定点集中** 的图.

团：完全子图.

极大团：不是其他团子图的图.

最大团：点数最大的团.

团数：最大团的点数，记为 $\omega(G)$.

最小染色：用最少的颜色给点染色使得所有边连接的两点颜色不同.

色数：最小染色的颜色数，记为 $\chi(G)$.

最大独立集：最大的点集使得点集中任意两点都没有边直接相连．该集合的大小记为 $\alpha(G)$.

最小团覆盖：用最少的团覆盖所有的点．使用团的数量记为 $\kappa(G)$.

弦：连接环中不相邻两点的边.

弦图：任意长度大于 3 的环都有一个弦的图称为弦图.

Lemma 1：团数 $\omega(G) \leq \chi(G)$ 色数

证明：考虑单独对最大团的导出子图进行染色，至少需要 $\omega(G)$ 种颜色.

Lemma 2：最大独立集数 $\alpha(G) \leq \kappa(G)$ 最小团覆盖数

证明：每个团中至多选择一个点.

Lemma 3：弦图的任意导出子图一定是弦图.

证明：如果弦图有导出子图不是弦图，说明在这个导出子图上存在大于 3 的无弦环，则无论原图如何（怎么加边）都不会使得原图是弦图，矛盾.

Lemma 4：弦图的任意导出子图一定不可能是一个点数大于 3 的环.

证明：一个点数大于 3 的环不是弦图，用以上定理即可.

弦图的判定

问题描述

给定一个无向图，判断其是否为弦图.

点割集

对于图 G 上的两点 u, v，定义这两点间的 **点割集** 为满足删除这一集合后，u, v 两点之间不连通．如果关于 u, v 两点间的一个点割集的任意子集都不是点割集，则称这个点割集为 **极小点割集**.

Lemma 5：图关于 u, v 的极小点割集将原图分成了若干个连通块，设包含 u 的连通块为 V_1，包含 v 的连通块为 V_2，则对于极小点割集上的任意一点 a，$N(a)$ 一定包含 V_1 和 V_2 中的点.

证明：若 $N(a)$ 只包含 V_1 或 V_2 中的至多一个连通块中的点，从点割集中删去 a 点，仍不连通，则原点割集不是最小点割集.

Lemma 6：弦图上任意两点间的极小点割集的导出子图一定为一个团.

证明：极小点割集大小 ≤ 1 时，导出子图一定为一个团.

否则，设极小点割集上有两点为 x, y，由 **Lemma 5** 得，$N(x)$ 中有 V_1, V_2 中的点，设为 x_1, x_2，同样的，设 y_1, y_2，注意，可能有 $x_1 = y_1, x_2 = y_2$.

由于 V_1, V_2 均为连通块，则在 x_1, y_1 和 x_2, y_2 两个点对之间存在最短路径．设 x, y 在 V_1, V_2 内部的最短路为 $x - x_1 \sim y_1 - y, x - x_2 \sim y_2 - y$，则图上存在一个环 $x - x_1 \sim y_1 - y - y_2 \sim x_2 - x$，该环的大小一定 ≥ 4，根据弦图的定义，此时该环上一定存在一条弦.

若这条弦连接了 V_1, V_2 两个连通块，则点集不是点割集．若这条弦连接了单个连通块内部的两个点或一个连通块内部的一个点和一个点割集上的点，都不满足最短路的性质．所以这条弦只能连接 x, y 两点.

由此，可证弦图中每个极小点割集中的两点都有边直接相连，故性质得证.

单纯点

设 $N(x)$ 表示与点 x 相邻的点集．若点集 $\{x\} + N(x)$ 的导出子图为一个团，则称点 x 为单纯点.

Lemma 7：任何一个弦图都至少有一个单纯点，不是完全图的弦图至少有两个不相邻的单纯点.

证明：数学归纳法．单独考虑每一连通块.

归纳基底：当图与完全图同构时，图上任意一点都是单纯点．当图的点数 ≤ 3 时，引理成立.

若图上的点数 ≥ 4 且图不为完全图，可知必然存在 u, v 使得 $(u, v) \notin E$．设 I 是图关于 u, v 的极小点割集．设 A, B 分别是删去 I 后的导出子图上 u, v 所在的连通块．由于问题的对称性，我们只考虑 A 一侧的情况，设 $L = A + I$．若 L 为完全图，则 u 为单纯点；若不是，因为 L 是原图的导出子图，一定也是弦图，所以有两个不相邻的单纯点，因为 I 是一个团，其上两点都相邻，所以 A 中一定有一个单纯点．该单纯点扩展到全图也为单纯点.

由于每次将整个图分成若干个连通块证明，大小一定减小，且都满足性质，故归纳成立.

完美消除序列

令 $n = |\ V\ |$，完美消除序列 $v_1, v_2, ..., v_n$ 为 $1, 2, ..., n$ 的一个排列，满足 v_i 在 $\{v_i, v_{i+1}, ..., v_n\}$ 的导出子图中为单纯点.

Lemma 8：一个无向图是弦图当且仅当其有一个完全消除序列.

充分性：点数为 1 的弦图有完全消除序列．由 **Lemma 3** 和 **Lemma 7**，点数为 n 的弦图的完美消除序列可以由点数为 $n - 1$ 的弦图的完美消除序列加上一个单纯点得到.

必要性：假设有无向图存在结点数 > 3 的环且拥有完美消除序列，设在完美消除序列中出现的第一个环上的点为 v，设 v 在环上与 v_1, v_2 相连，则有完美消除序列的性质即单纯点的定义可得 v_1, v_2 直接有边相连，矛盾.

朴素算法

每次找到一个 **单纯点** v，加入到完美消除序列中.

将点 v 与其相邻的边从图上删除.

重复以上过程, 若所有点都被删除, 则原图是弦图且求得了一个完美消除序列; 若图上不存在单纯点, 则原图不是弦图.

时间复杂度 $O(n^4)$.

MCS 算法

最大势算法（Maximum Cardinality Search）是一种可以在 $O(n + m)$ 的时间复杂度内求出无向图的完美消除序列的方法.

逆序给结点编号，即按从 n 到 1 的顺序给点标号.

设 $label_x$ 表示第 x 个点与多少个已经标号的点相邻，每次选择 $label$ 值最大的未标号结点进行标号.

用链表维护对于每个 i，满足 $label_x = i$ 的 x.

由于每条边对 $\sum_{i=1}^{n} label_i$ 的贡献最多是 2，时间复杂度 $O(n+m)$.

正确性证明:

设 $\alpha(x)$ 为 x 在这个序列中的位置. 我们需要证明对于任何一个弦图，算法求出的序列一定是一个完美消除序列，即在序列中位于某个点后面且与这个点相连的所有点两两相连.

Lemma 9：考虑三个点 u, v, w 满足 $\alpha(u) < \alpha(v) < \alpha(w)$，如果 uw 相连，vw 不相连，则 w 只给 u 的 $label$ 贡献，不给 v 贡献. 为了让 v 比 u 先加入序列，需要一个 x 满足 $\alpha(v) < \alpha(x)$ 且 vx 相连，ux 不相连，即 x 只给 v 贡献而不给 u 贡献.

Lemma 10：任意一个弦图一定不存在一个序列 $v_0, v_1, ..., v_k (k \geq 2)$ 满足下列性质：

1. $v_i v_j$ 相连当且仅当 $| i - j | = 1$.
2. $\alpha(v_0) > \alpha(v_i) (i \in [1, k])$.
3. 存在 $i \in [1, k-1]$，满足 $\alpha(v_i) < \alpha(v_{i+1}) < ... < \alpha(v_k)$ 且 $\alpha(v_i) < \alpha(v_{i-1}) < ... < \alpha(v_1) < \alpha(v_k) < \alpha(v_0)$.

证明：

由于 $\alpha(v_1) < \alpha(v_k) < \alpha(v_0)$，且 $v_1 v_0$ 相连，$v_k v_0$ 不相连，所以由性质一，存在 x 满足 $\alpha(v_k) < \alpha(x)$ 且 $v_k x$ 相连，$v_1 x$ 不相连.

考虑最小的 $j \in (1, k]$ 满足 $v_j x$ 相连，我们可以推出 $v_0 x$ 不相连，否则 $v_0 v_1 \cdots v_j x$ 构成了一个长度 ≥ 4 且无弦的环.

如果 $x < v_0$，则 $v_0, v_1, ..., v_j, x$ 也是一个满足性质的序列；如果 $v_0 < x$ 则 $x, v_j, ..., v_1, v_0$ 也是一个满足性质的序列.

在上面的推导中，我们扩大了 $\min(v_0, v_k)$，于是一直推下去，一定会产生矛盾.

Theorem 1：对于任何一个弦图，最大势算法求出的序列一定是一个完美消除序列.

证明：考虑任意三个点 u, v, w 满足 $\alpha(u) < \alpha(v) < \alpha(w)$，我们需要证明若 uv 相连，uw 相连，则 vw 一定相连.

考虑反证法，假设不相连，那么 w, u, v 就是一个满足 **Lemma 10** 中性质的序列，我们证明了这样序列不存在，所以矛盾，vw 相连.

参考实现

```cpp
using namespace std;

int n;
vector<vector<int>> G;
vector<int> p, rnk;

void mcs() {
  vector<int> head(n, 0), prev(n + 1), next(n + 1), label(n + 1, 0);
  p.assign(n + 1, 0);
  rnk.assign(n + 1, 0);
  auto insert = [&](int v) {
    prev[v] = 0;
    next[v] = head[label[v]];
    if (next[v]) prev[next[v]] = v;
    head[label[v]] = v;
  };
```

```
  auto erase = [&](int v) {
    if (prev[v])
      next[prev[v]] = next[v];
    else
      head[label[v]] = next[v];
    if (next[v]) prev[next[v]] = prev[v];
  };
  for (int v = 1; v <= n; v++) insert(v);
  int largest = 0;
  for (int i = n; i >= 1; i--) {
    while (!head[largest]) largest--;
    int v = head[largest];
    erase(v);
    p[i] = v;
    rnk[v] = i;
    for (int u : G[v]) {
      if (rnk[u]) continue;
      erase(u);
      label[u]++;
      insert(u);
      largest = max(largest, label[u]);
    }
  }
}

```

如果此时原图是弦图，此时求出的就是完美消除序列；但是由于原图可能不是弦图，此时求出的一定不是完美消除序列，所以问题转化为 **判断求出的序列是否是原图的完美消除序列**.

判断一个序列是否是完美消除序列

朴素算法

根据定义，依次判断完美消除序列 v 上 $\{v_i, v_{i+1}, ..., v_n\}$ 中与 v_i 相邻的点是否构成了一个团. 时间复杂度 $O(nm)$.

优化后的算法

设 $N^+(u)$ 为序列中位于 u 之后且与 u 相邻的点集，$f(u)$ 为其中在序列中最靠前的点. 对于每个 $N^+(u)$ 非空的点 u，只需检查 $N^+(u) \setminus \{f(u)\}$ 中的每个点是否都与 $f(u)$ 相邻.

这个条件显然是必要的. 充分性可以按序列从后往前归纳：若后面的点均满足完美消除序列的要求，则 $N^+(f(u))$ 是一个团. 检查通过时，$N^+(u) \setminus \{f(u)\} \subseteq N^+(f(u))$，且其中的点都与 $f(u)$ 相邻，所以 $N^+(u)$ 也是一个团. $N^+(u)$ 为空时无需检查.

先扫描邻接表求出每个点的 $f(u)$，并将 $f(u)$ 相同的点分为一组. 处理 $f(u) = v$ 的这一组时，先将 v 的所有邻点标记，再扫描组内每个点 u 的邻接表，检查序列中位于 v 之后的邻点是否都被标记. 用顶点编号 v 作为标记值，就不必在每组检查后清空标记数组. 每个点至多属于一个组，每个点的邻接表在求 $f(u)$、设置标记和检查时各至多扫描一次，因此总时间复杂度为 $O(n + m)$.

参考实现

```
using namespace std;
```

```cpp

int n;
vector<vector<int>> G;
vector<int> p, rnk;

int fail_u, fail_v, fail_w;

bool check() {
  vector<vector<int>> bucket(n + 1);
  vector<int> mark(n + 1, 0);
  for (int i = 1; i <= n; i++) {
    int u = p[i], first = 0;
    for (int v : G[u]) {
      if (rnk[v] > rnk[u] && (!first || rnk[v] < rnk[first])) first = v;
    }
    if (first) bucket[first].push_back(u);
  }

  fail_u = 0;
  for (int v = 1; v <= n; v++) {
    // 同一组共用一次标记，mark[w] == v 表示 w 与当前的 v 相邻
    for (int w : G[v]) mark[w] = v;
    for (int u : bucket[v]) {
      for (int w : G[u]) {
        if (rnk[w] > rnk[v] && mark[w] != v) {
          // 记录序列中最靠后的失败点及对应的两个邻点
          if (!fail_u || rnk[u] > rnk[fail_u]) {
            fail_u = u;
            fail_v = v;
            fail_w = w;
          }
          break;
        }
      }
    }
  }
  return !fail_u;
}

```

至此，**弦图判定问题** 可以在 $O(n+m)$ 的时间复杂度内解决.

弦图的极大团

令 $N(x)$ 为满足与 x 直接有边相连且在完美消除序列上的 x 之后的序列. 则弦图的极大团一定为 $\{x\}+N(x)$.

证明：考虑弦图的一个极大团 V，其中的点在完美消除序列中出现的第一个点 x，一定有 $V\subseteq\{x\}+N(x)$，又因为 V 是极大团，所以 $V=\{x\}+N(x)$.

弦图最多有 n 个极大团. 求出弦图的每个极大团，可以判断每个 $\{x\}+N(x)$ 是否为极大团.

设 $A=\{x\}+N(x), B=\{y\}+N(y)$，若 $A\subsetneq B$，则 A 不是极大团. 此时在完美消除序列上显然有 y 在 x 前.

设 nxt_x 表示 $N(x)$ 中在完美消除序列上最靠前的点，$y*$ 表示所有满足 $A \subseteq B$ 的 y 中的最靠后的点. 此时必然有 $nxt_{y*} = x$，否则 $y*$ 不是最靠后的，令 $y* = nxt_{y*}$ 仍然满足条件.

$A \subsetneq B$ 当且仅当 $|A| + 1 \leq |B|$.

问题转化为判断是否存在 y，满足 $nxt_y = x$ 且 $|N(x)| + 1 \leq |N(y)|$. 时间复杂度 $O(n + m)$.

```
for (int i = 1; i <= n; i++) {
  cur = 0;
  for (vector<int>::iterator it = G[p[i]].begin(); it != G[p[i]].end(); it++)
    if (rnk[p[i]] < rnk[*it]) {
      s[++cur] = *it;
      if (rnk[s[cur]] < rnk[s[1]]) swap(s[1], s[cur]);
    }
  fst[p[i]] = s[1];
  N[p[i]] = cur;
}
for (int i = 1; i <= n; i++) {
  if (!vis[p[i]]) ans++;
  if (N[p[i]] >= N[fst[p[i]]] + 1) vis[fst[p[i]]] = true;
}
```

弦图的色数/弦图的团数

一种构造方法：按完美消除序列从后往前依次给每个点染色，给每个点染上可以染的最小颜色. 时间复杂度 $O(m + n)$.

正确性证明：设以上方法使用了 t 种颜色，则 $t \geq \chi(G)$. 由于团上每个点都是不同的颜色，所以 $t = \omega(G)$，由 **Lemma 1**，$t = \omega(G) \leq \chi(G)$. 综上，可得 $t = \chi(G) = \omega(G)$.

无需染色方案，只需求出弦图的色数/团数时，可以取 $|\{x\} + N(x)|$ 的最大值得到.

```
for (int i = 1; i <= n; i++) ans = max(ans, deg[i] + 1);
```

弦图的最大独立集/最小团覆盖

最大独立集：完美消除序列从前往后，选择所有没有与已经选择的点有直接连边的点.

最小团覆盖：设最大独立集为 $\{v_1, v_2, ..., v_t\}$，则团的集合 $\{\{v_1 + N(v_1)\}, \{v_2 + N(v_2)\}, ..., \{v_t + N(v_t)\}\}$ 为图的最小团覆盖. 时间复杂度均为 $O(n + m)$.

正确性证明：设以上方案独立集数和团覆盖数为 t，由定义得 $t \leq \alpha(G), t \geq \kappa(G)$，由 **Lemma 2** 得，$\alpha(G) \leq \kappa(G)$，所以 $t = \alpha(G) = \kappa(G)$.

```
for (int i = 1; i <= n; i++)
  if (!vis[p[i]]) {
    ans++;
    for (vector<int>::iterator it = G[p[i]].begin(); it != G[p[i]].end(); it++)
      vis[*it] = true;
  }
```

习题

Library Checker - Chordal Graph Recognition[1]

SPOJ FISHNET - Fishing Net[2]

P3196[HNOI2008] 神奇的国度[3]

P3852[TJOI2007] 小朋友[4]

参考资料

弦图相关[5]

2009 WC 讲稿[6]

弦图总结 - 租酥雨[7]

R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs,test acyclicity of hypergraphs,and selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984), pp. 566–579.[8]

外部参考资料

[1] Library Checker - Chordal Graph Recognition

[2] SPOJ FISHNET - Fishing Net

[3] P3196[HNOI2008] 神奇的国度

[4] P3852[TJOI2007] 小朋友

[5] 弦图相关

[6] 2009 WC 讲稿

[7] 弦图总结 - 租酥雨

[8] R. E. Tarjan and M. Yannakakis, Simple linear-time algorithms to test chordality of graphs,test acyclicity of hypergraphs,and selectively reduce acyclic hypergraphs, SIAM J. Comput., 13 (1984), pp. 566–579.

11.22 图的着色

点着色

（讨论的是无自环无向图）

对无向图顶点着色，且相邻顶点不能同色．若 G 是 k- 可着色的，但不是 $(k-1)$- 可着色的，则称 k 是 G 的色数，记为 $\chi(G)$.

对任意图 G，有 $\chi(G) \leq \Delta(G)+1$，其中 $\Delta(G)$ 为最大度.

Brooks 定理

设连通图不是完全图也不是奇圈，则 $\chi(G) \leq \Delta(G)$.

证明

证明

设 $|V(G)|=n$，考虑数学归纳法.

首先，$n \leq 3$ 时，命题显然成立.

接下来，假设对于 $n-1$ 时的命题成立，下面我们要逐步强化命题.

不妨只考虑 $\Delta(G)$- 正则图，因为对于非正则图来说，可以看作在正则图里删去一些边构成的，而这一过程并不会影响结论.

对于任意不是完全图也不是奇圈的正则图 G，任取其中一点 v，考虑子图 $H:=G-v$，由归纳假设知 $\chi(H) \leq \Delta(H)=\Delta(G)$，接下来我们只需证明在 H 中插入 v 不会影响结论即可.

令 $\Delta:=\Delta(G)$，设 H 染的 Δ 种颜色分别为 $c_1, c_2, ..., c_\Delta$，v 的 Δ 个邻接点为 $v_1, v_1, ..., v_\Delta$．不妨假设 v 的这些邻接点颜色两两不同，否则命题得证.

接下来我们设所有在 H 中染成 c_i 或 c_j 的点以及它们之间的所有边构成子图 $H_{i,j}$．不妨假设任意 2 个不同的点 v_i，v_j 一定在 $H_{i,j}$ 的同一个连通分量中，否则若在两个连通分量中的话，可以交换其中一个连通分量所有点的颜色，从而 v_i，v_j 颜色相同．

> 这里的交换颜色指的是若图中只有两种颜色 a，b，那么把图中原来染成颜色 a 的点全部染成颜色 b，把图中原来染成颜色 b 的点全部染成颜色 a．

我们设上述连通分量为 $C_{i,j}$，那么 $C_{i,j}$ 一定只能是 v_i 到 v_j 的路．因为 v_i 在 H 中的度为 $\Delta-1$，所以 v_i 在 H 中的邻接点颜色一定两两不同，否则可以给 v_i 染别的颜色，从而和 v 的其他邻接点颜色重复，所以 v_i 在 $C_{i,j}$ 中邻接点数量为 1，v_j 同理．然后我们在 $C_{i,j}$ 中取一条 v_i 到 v_j 的路，令其为 P，若 $C_{i,j}\neq P$，那么我们沿着 P 顺次给路上的点染色，设遇到的第一个度数大于 2 的点为 u，注意到 u 的邻接点最多只用了 $\Delta-2$ 种颜色，所以 u 可以重新染色，从而使 v_i，v_j 不连通．

然后我们不难发现，对任意 3 个不同的点 v_i，v_j，v_k，$V(C_{i,j})\cap V(C_{j,k})=\{v_j\}$．

到这里我们对命题的强化工作就已经做完了．

接下来就很简单．首先，如果 v 的邻接点两两相邻，那么命题得证．不妨设 v_1，v_2 不相邻，在 $C_{1,2}$ 中取 v_1 的邻接点 w，交换 $C_{1,3}$ 中的颜色．得到的新图中，$w\in V(C_{1,2})\cap V(C_{2,3})$，矛盾．

至此命题证明完毕．

Welsh–Powell 算法

Welsh–Powell 算法是一种在 **不限制最大着色数** 时寻找着色方案的贪心算法．

对于无自环无向图 G，设 $V(G):=\{v_1,v_2,...,v_n\}$ 满足．

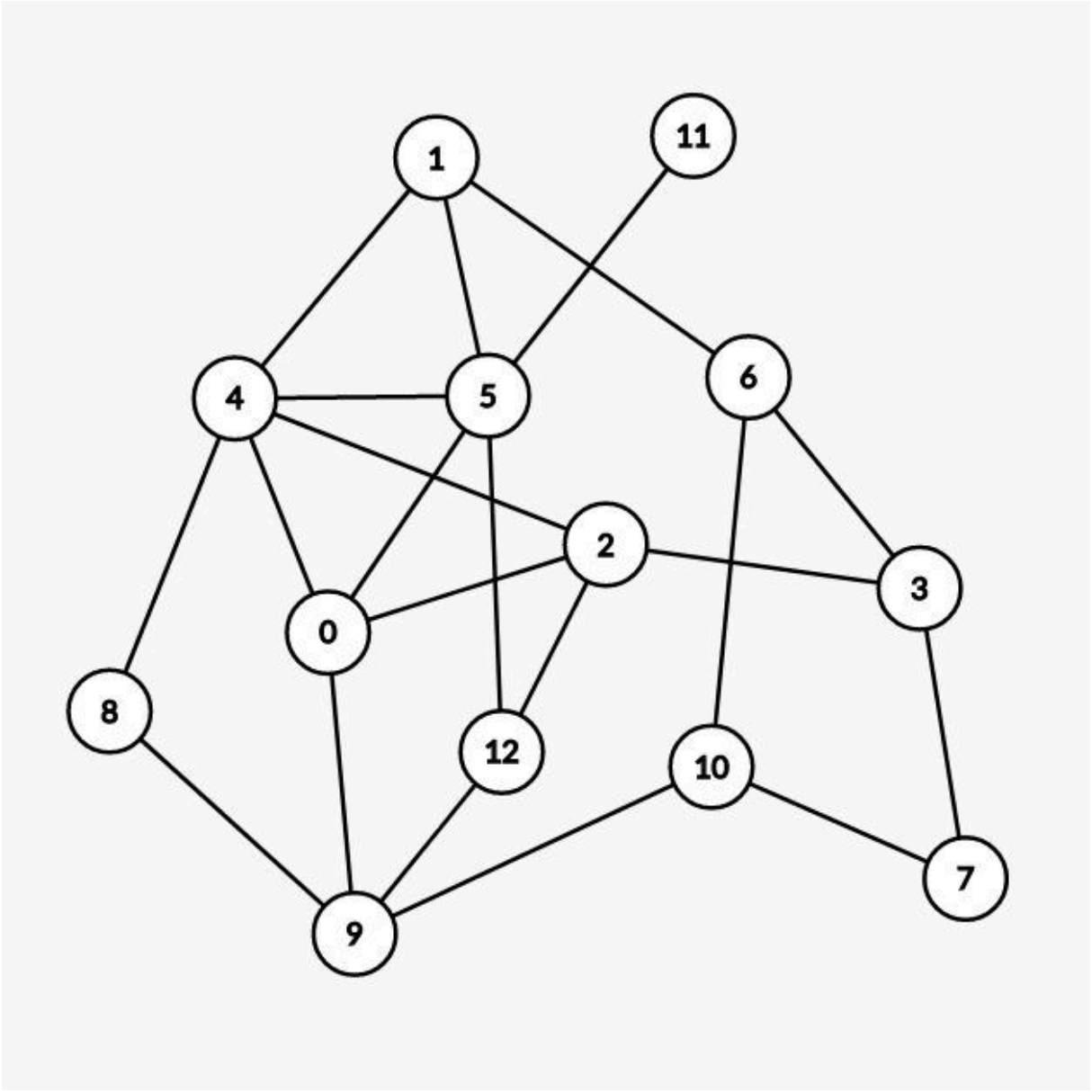
$\deg(v_i)\geq\deg(v_{i+1}),\ \forall 1\leq i\leq n-1$

按 Welsh–Powell 算法着色后的颜色数至多为 $\max_{i=1}^{n}\min\{\deg(v_i)+1,i\}$，该算法的时间复杂度为 $O(n\max_{i=1}^{n}\min\{\deg(v_i)+1,i\})=O(n^2)$．

过程

1. 将当前未着色的点按度数降序排列．
2. 将第一个点染成一个未被使用的颜色．
3. 顺次遍历接下来的点，若当前点和所有与第一个点颜色 **相同** 的点 **不相邻**，则将该点染成与第一个点相同的颜色．
4. 若仍有未着色的点，则回到步骤 1，否则结束．

示例如下：



（由 Graph Editor[1] 生成）

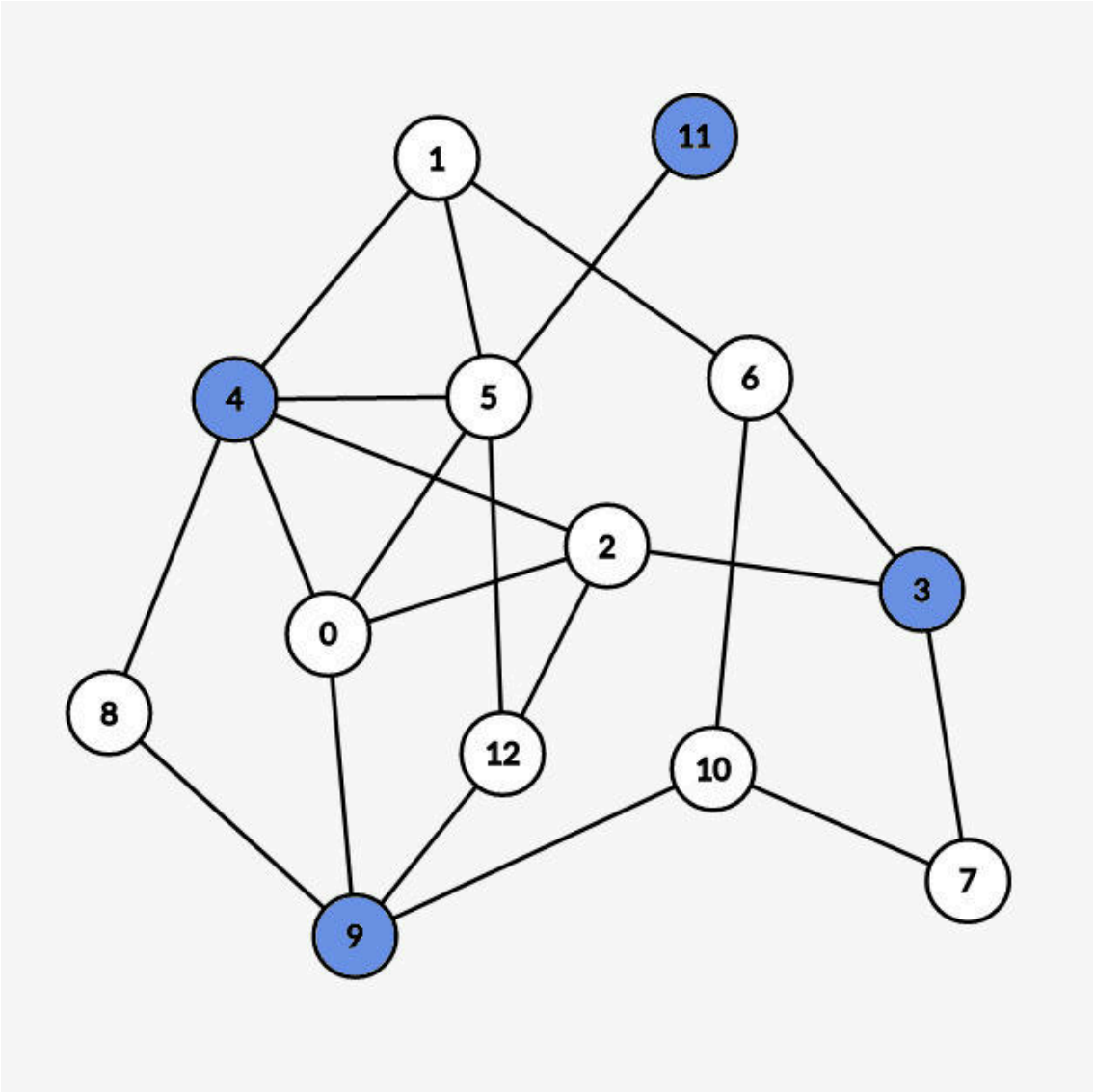
我们先对点按度数降序排序，得：

次序	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
点的编号	4	5	0	2	9	1	3	6	10	12	7	8	11
度数	5	5	4	4	4	3	3	3	3	3	2	2	1
$\min\{\deg(v_i)+1, i\}$	1	2	3	4	5	4	4	4	4	4	3	3	2

所以 Welsh–Powell 算法着色后的颜色数最多为 5.

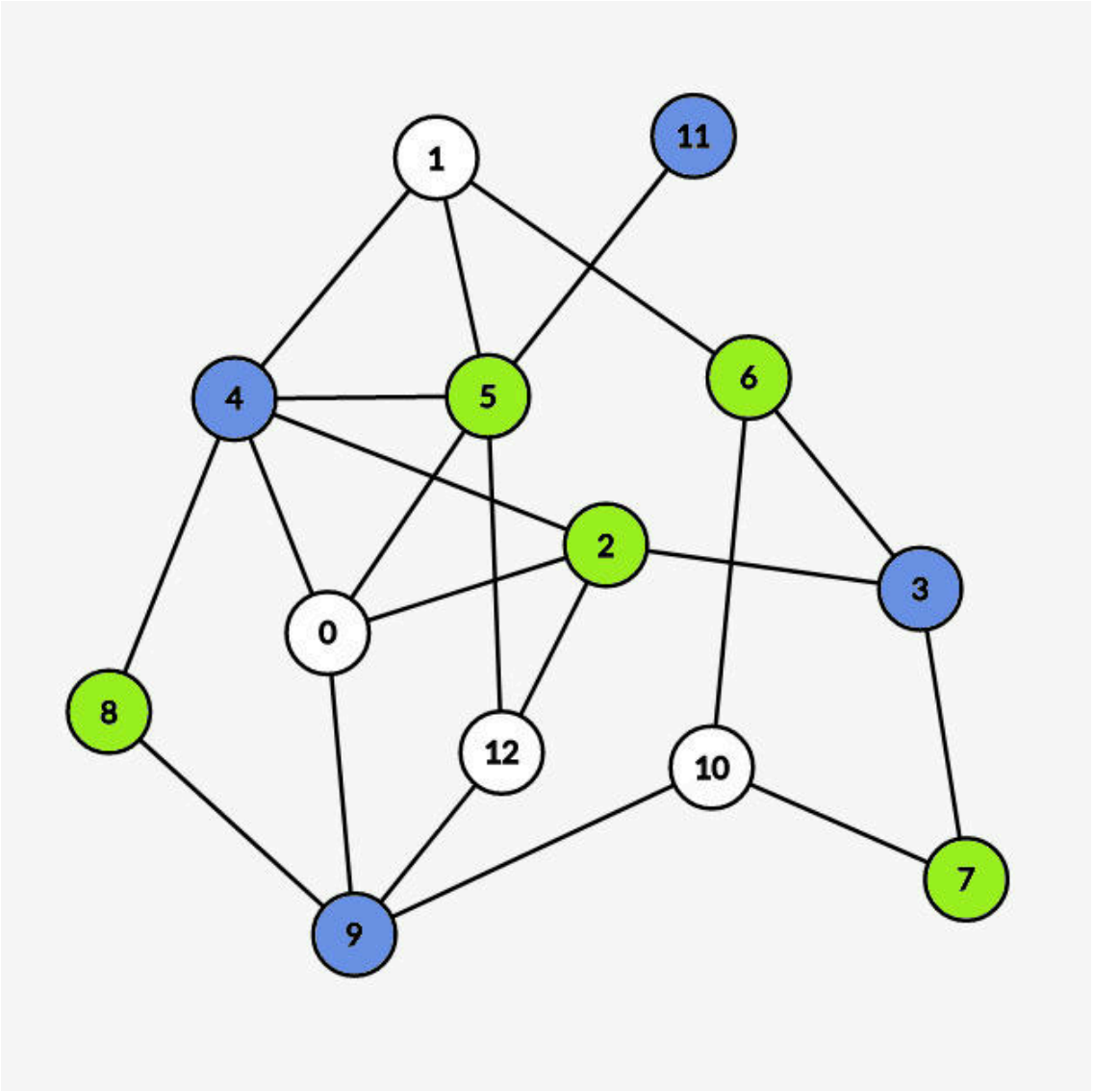
另外因为该图有子图 C_3, 所以色数一定大于等于 3.

- 第一次染色：



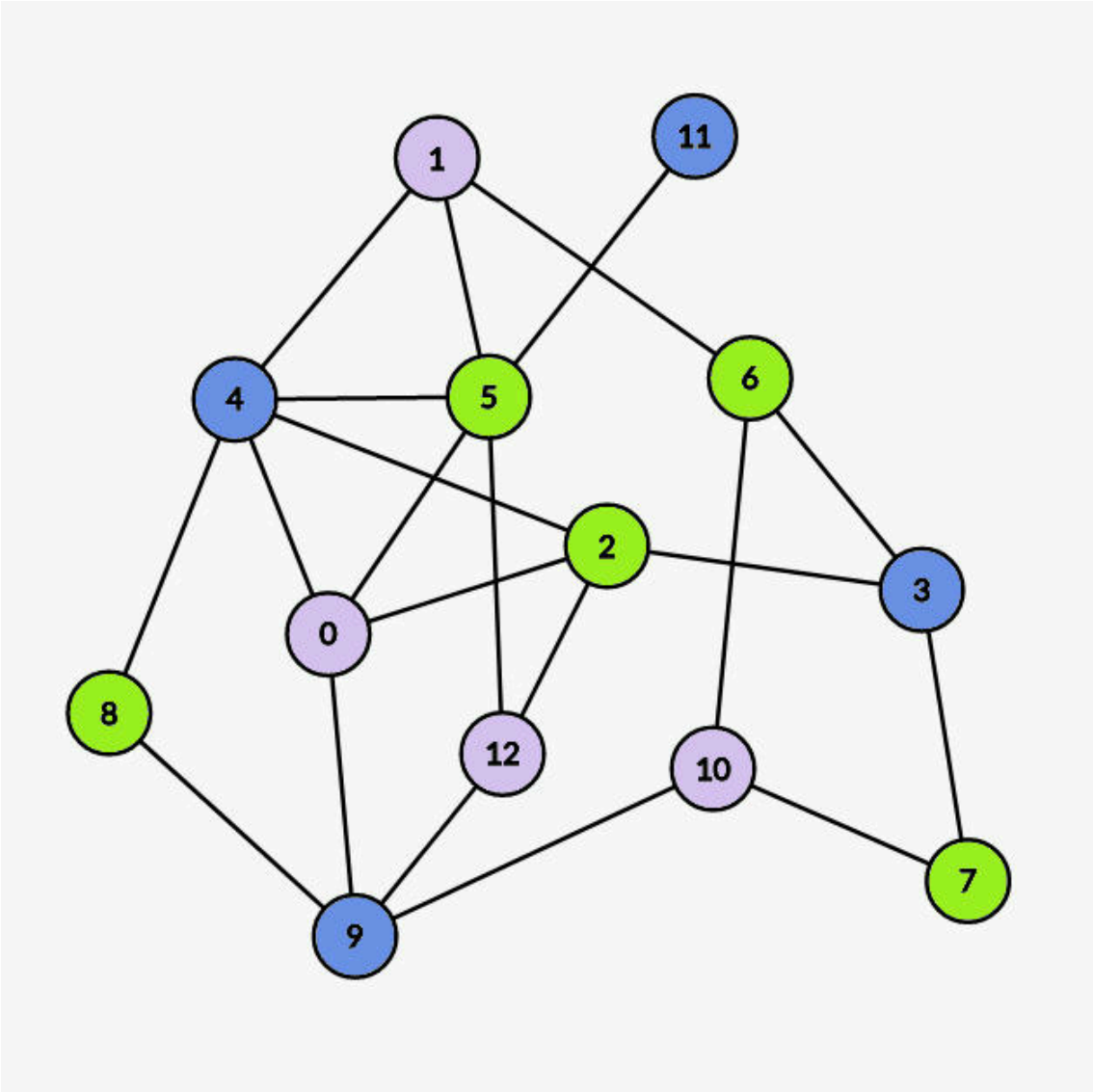
染 4 9 3 11 号点.

- 第二次染色:



染 5 2 6 7 8 号点.

- 第三次染色:



染 `0 1 10 12` 号点.

证明

证明

对于无自环无向图 G，设 $V(G) := \{v_1, v_2, ..., v_n\}$ 满足

$\deg(v_i) \geq \deg(v_{i+1}),\ \forall 1 \leq i \leq n-1$

令 $V_0 = \emptyset$, 我们取 $V(G) \setminus \bigcup_{i=0}^{m-1} V_i$ 中的子集 V_m, 其中的元素满足

1. $v_{k_m} \in V_m$, 其中 $k_m = \min\{k : v_k \notin \bigcup_{i=0}^{m-1} V_i\}$
2. 若

   $\{v_{i_{m,1}}, v_{i_{m,2}}, ..., v_{i_{m,l_m}}\} \subset V_m,\ i_{m,1} < i_{m,2} < ... < i_{m,l_m}$

   则 $v_j \in V_m$ 当且仅当
   1. $j > i_{m,l_m}$
   2. v_j 与 $v_{i_{m,1}}, v_{i_{m,2}}, ..., v_{i_{m,l_m}}$ 均不相邻

显然若将 V_i 中的点染成第 i 种颜色，则该染色方案即为 Welsh–Powell 算法给出的方案，显然有

- $V_1 \neq \emptyset$

- $V_i \cap V_j = \emptyset \iff i \neq j$
- $\exists \alpha(G) \in \mathbb{N}^*, \forall i > \alpha(G),\ s.t.\ V_i = \emptyset$

我们只需要证明：

$\bigcup_{i=1}^{\alpha(G)} V_i = V(G)$

其中

$\chi(G) \leq \alpha(G) \leq \max_{i=1}^{n} \min\{\deg(v_i) + 1, i\}$

上式左边的不等号显然成立，我们考虑右边.

首先我们不难得出：

若 $v \notin \bigcup_{i=1}^{m} V_i$，则 v 与 $V_1, V_2, ..., V_m$ 中分别至少有一个点相邻，从而有 $\deg(v) \geq m$

进而

$v_j \in \bigcup_{i=1}^{\deg(v_j)+1} V_i$

另一方面，基于序列 $\{V_i\}$ 的构造方法，我们不难发现

$v_j \in \bigcup_{i=1}^{j} V_i$

两式结合即得证.

边着色

对无向图的边着色，要求相邻的边涂不同种颜色．若 G 是 k- 边可着色的，但不是 $(k-1)$- 边可着色的，则称 k 是 G 的边色数，记为 $\chi'(G)$.

Vizing 定理

设 G 是简单图，则 $\Delta(G) \leq \chi'(G) \leq \Delta(G) + 1$

若 G 是二部图，则 $\chi'(G) = \Delta(G)$

当 n 为奇数（$n \neq 1$）时，$\chi'(K_n) = n$ 当 n 为偶数时，$\chi'(K_n) = n - 1$

二分图 Vizing 定理的构造性证明

证明

按照顺序在二分图中加边.

我们在尝试加入边 (x, y) 的时候，我们尝试寻找对于 x 和 y 的编号最小的尚未被使用过的颜色，假设分别为 l_x 和 l_y.

如果 $l_x = l_y$ 此时我们可以直接将这条边的颜色设置为 l_x.

否则假设 $l_x < l_y$, 我们可以尝试将节点 y 连出去的颜色为 l_x 的边的颜色修改为 l_y.

修改的过程可以被近似的看成是一条从 y 出发，依次经过颜色为 $l_x, l_y, \cdots$ 的边的有限唯一增广路.

因为增广路有限所以我们可以将增广路上所有的边反色，即原来颜色为 l_x 的修改为 l_y，原来颜色为 l_y 的修改为 l_x.

根据二分图的性质，节点 x 不可能为增广路节点，否则与最小未使用颜色为 l_x 矛盾.

所以我们可以在增广之后直接将连接 x 和 y 的边的颜色设为 l_x.

总构造时间复杂度为 $O(nm)$.

示例代码 UVa10615 Rooks[2]

```cpp
#include <iostream>

constexpr int MAXN = 100;

int t;
int n;
char a[MAXN + 10][MAXN + 10];
int c[MAXN + 10][MAXN + 10];
int ans;

namespace graph {
struct Vertex {
  int head;
  int deg;
  int vis[MAXN + 10];
} vertex[2 * MAXN + 10], e;

struct Edge {
  int head;
  int next;
  int col;
} edge[2 * MAXN * MAXN + 10];

int ecnt;

void init() {
  for (int i = 0; i < MAXN + 10; i++) std::fill(c[i], c[i] + MAXN + 10, 0);
  for (int i = 1; i <= 2 * MAXN; i++) vertex[i] = e;
  for (int i = 0; i <= 2 * MAXN * MAXN; i++) edge[i].col = 0;
  ecnt = 1;
  ans = 0;
  return;
}

void addEdge(int tail, int head) {
  ecnt++;
  edge[ecnt].head = head;
  edge[ecnt].next = vertex[tail].head;
  vertex[tail].head = ecnt;
  vertex[tail].deg++;
  return;
}

int get(int u) {
  for (int i = 1; i <= n; i++)
    if (!vertex[u].vis[i]) return i;
  exit(1);
}

void DFS(int u, int ori, int upd) {
  if (vertex[u].vis[ori] == 0) {
```

```
52      vertex[u].vis[upd] = 0;
53      return;
54    }
55    int e = vertex[u].vis[ori];
56    int v = edge[e].head;
57    DFS(v, upd, ori);
58    edge[e].col = upd;
59    edge[e ^ 1].col = upd;
60    vertex[u].vis[upd] = e;
61    vertex[v].vis[upd] = e ^ 1;
62    return;
63  }
64
65  void AddEdge(int u, int v) {
66    addEdge(u, v);
67    addEdge(v, u);
68    return;
69  }
70
71  void solve() {
72    for (int u = 1; u <= n; u++) {
73      for (int e = vertex[u].head; e; e = edge[e].next) {
74        int v = edge[e].head;
75        if (edge[e].col) continue;
76        int lu = get(u);
77        int lv = get(v);
78        if (lu < lv) DFS(v, lu, lv);
79        if (lu > lv) DFS(u, lv, lu);
80        int l = std::min(lu, lv);
81        vertex[u].vis[l] = e;
82        vertex[v].vis[l] = e ^ 1;
83        edge[e].col = l;
84        edge[e ^ 1].col = l;
85      }
86    }
87  }
88
89  void generate() {
90    for (int u = 1; u <= n; u++) {
91      for (int e = vertex[u].head; e; e = edge[e].next) {
92        int v = edge[e].head;
93        c[u][v - n] = edge[e].col;
94      }
95    }
96    return;
97  }
98  }  // namespace graph
99
100 void mian() {
101   graph::init();
102   std::cin >> n;
103   for (int i = 1; i <= n; i++)
```

```
104     for (int j = 1; j <= n; j++) std::cin >> a[i][j];
105   for (int i = 1; i <= n; i++)
106     for (int j = 1; j <= n; j++)
107       if (a[i][j] == '*') graph::AddEdge(i, j + n);
108   for (int i = 1; i <= n * 2; i++) ans = std::max(ans,
    graph::vertex[i].deg);
109   graph::solve();
110   graph::generate();
111   std::cout << ans << '\n';
112   for (int i = 1; i <= n; i++) {
113     for (int j = 1; j <= n; j++) std::cout << c[i][j] << ' ';
114     std::cout << '\n';
115   }
116   return;
117 }
118
119 int main() {
120   std::cin >> t;
121   while (t--) mian();
122   return 0;
123 }
```

一道很不简单的例题 uoj 444 二分图[3]

本题为笔者于 2018 年命制的集训队第一轮作业题.

首先我们可以发现答案下界为度数不为 k 倍数的点的个数.

下界的构造方法是对二分图进行拆点.

若 $degree \bmod k \neq 0$, 我们将其拆为 $degree/k$ 个度数为 k 的节点和一个度数为 $degree \bmod k$ 的节点.

若 $degree \bmod k = 0$, 我们将其拆为 $degree/k$ 个度数为 k 的节点.

拆出来的点在原图中的意义相同，也就是说，在满足度数限制的情况下，一条边端点可以连接任意一个拆出来的点.

根据 Vizing 定理，我们显然可以构造出该图的一种 k 染色方案.

删边部分由于和 Vizing 定理关系不大这里不再展开.

有兴趣的读者可以自行阅读笔者当时写的题解.

色多项式

$P(G,k)$ 表示 G 的不同 k 着色方式的总数.

$P(K_n,k) = k(k-1)\cdots(k-n+1)$

$P(N_n,k) = k^n$

在无向无环图 G 中,

1. $e=(v_i,v_j) \notin E(G)$，则 $P(G,k) = P(G \cup e,k) + P(G \setminus e,k)$
2. $e=(v_i,v_j) \in E(G)$，则 $P(G,k) = P(G-e,k) - P(G \setminus e,k)$

定理：设 V_1 是 G 的点割集，且 $G[V_1]$ 是 G 的 $|V_1|$ 阶完全子图，$G-V_1$ 有 $p(p \geq 2)$ 个连通分支，则:

$P(G,k)=\frac{\Pi_{i=1}^{p}(P(H_i,k))}{P(G[V_1],k)^{p-1}}$

其中 $H_i=G[V_1\cup V(G_i)]$

参考资料

1. Graph coloring - Wikipedia[4]
2. Welsh, D. J. A.; Powell, M. B. (1967), "An upper bound for the chromatic number of a graph and its application to timetabling problems[5]", The Computer Journal, 10 (1): 85–86

外部参考资料

[1] Graph Editor 

[2] UVa10615 Rooks 

[3] uoj 444 二分图 

[4] Graph coloring - Wikipedia 

[5] An upper bound for the chromatic number of a graph and its application to timetabling problems 

11.23 网络流

11.23.1 网络流简介

本页面主要介绍网络流相关的基本知识.

概述

网络（network）是指一个特殊的有向图 $G=(V,E)$，其与一般有向图的不同之处在于有容量和源汇点.

- E 中的每条边 (u,v) 都有一个被称为容量（capacity）的权值，记作 $c(u,v)$. 当 $(u,v)\notin E$ 时，可以假定 $c(u,v)=0$.
- V 中有两个特殊的点：源点（source）s 和汇点（sink）t（$s\neq t$）.

对于网络 $G=(V,E)$，流（flow）是一个从边集 E 到整数集或实数集的函数，其满足以下性质.

1. 容量限制：对于每条边，流经该边的流量不得超过该边的容量，即 $0\leq f(u,v)\leq c(u,v)$;
2. 流守恒性：除源汇点外，任意结点 u 的净流量为 0. 其中，我们定义 u 的净流量为 $f(u)=\sum_{x\in V}f(u,x)-\sum_{x\in V}f(x,u)$.

对于网络 $G=(V,E)$ 和其上的流 f，我们定义 f 的流量 $|f|$ 为 s 的净流量 $f(s)$. 作为流守恒性的推论，这也等于 t 的净流量的相反数 $-f(t)$.

对于网络 $G=(V,E)$, 如果 $\{S,T\}$ 是 V 的划分（即 $S\cup T=V$ 且 $S\cap T=\emptyset$），且满足 $s\in S,t\in T$, 则我们称 $\{S,T\}$ 是 G 的一个 s-t 割（cut）. 我们定义 s-t 割 $\{S,T\}$ 的容量为 $||S,T||=\sum_{u\in S}\sum_{v\in T}c(u,v)$.

常见问题

常见的网络流问题包括但不限于以下类型问题.

- 最大流问题：对于网络 $G=(V,E)$，给每条边指定流量，得到合适的流 f，使得 f 的流量尽可能大. 此时我们称 f 是 G 的最大流.
- 最小割问题：对于网络 $G=(V,E)$，找到合适的 s-t 割 $\{S,T\}$，使得 $\{S,T\}$ 的总容量尽可能小. 此时我们称 $\{S,T\}$ 的总容量是 G 的最小割.
- 最小费用最大流问题：在网络 $G=(V,E)$ 上，对每条边给定一个权值 $w(u,v)$，称为费用（cost），含义是单位流量通过 (u,v) 所花费的代价. 对于 G 所有可能的最大流，我们称其中总费用最小的一者为最小费用最大流.

我们将在稍后的章节中对它们进行详细介绍.

例题：网络流 24 题

网络流 24 题是中文互联网上广泛流传的一个题单（LibreOJ[1]/洛谷[2]），至少在 2010 年前后就已经存在．该题单引入了一些经典的将其他问题建模为网络流问题的技巧．由于时代的局限性，这些问题未必是最具代表性的网络流问题，但仍值得有志于算法竞赛的读者一阅．

外部参考资料

[1] LibreOJ  [2] 洛谷 

11.23.2 最大流

本页面主要介绍最大流问题相关的算法知识．

概述

网络流基本概念参见 网络流简介→11.23.1@p2906.

令 $G=(V,E)$ 是一个有源汇点的网络，我们希望在 G 上指定合适的流 f，以最大化整个网络的流量 $|f|$（即 $\sum_{x\in V} f(s,x)-\sum_{x\in V} f(x,s)$），这一问题被称作最大流问题（Maximum flow problem）．

Ford–Fulkerson 增广

Ford–Fulkerson 增广是计算最大流的一类算法的总称．该方法运用贪心的思想，通过寻找增广路来更新并求解最大流．

概述

给定网络 G 及 G 上的流 f，我们做如下定义．

对于边 (u,v)，我们将其容量与流量之差称为剩余容量 $c_f(u,v)$（Residual Capacity），即 $c_f(u,v)=c(u,v)-f(u,v)$．

我们将 G 中所有结点和剩余容量大于 0 的边构成的子图称为残量网络 G_f（Residual Network），即 $G_f=(V,E_f)$，其中 $E_f=\{(u,v)\mid c_f(u,v)>0\}$．

> **Warning**
>
> 正如我们马上要提到的，流量可能是负值，因此，E_f 的边有可能并不在 E 中．引入增广的概念后，下文将具体解释这一点．

我们将 G_f 上一条从源点 s 到汇点 t 的路径称为增广路（Augmenting Path）．对于一条增广路，我们给每一条边 (u,v) 都加上等量的流量，以令整个网络的流量增加，这一过程被称为增广（Augment）．由此，最大流的求解可以被视为若干次增广分别得到的流的叠加．

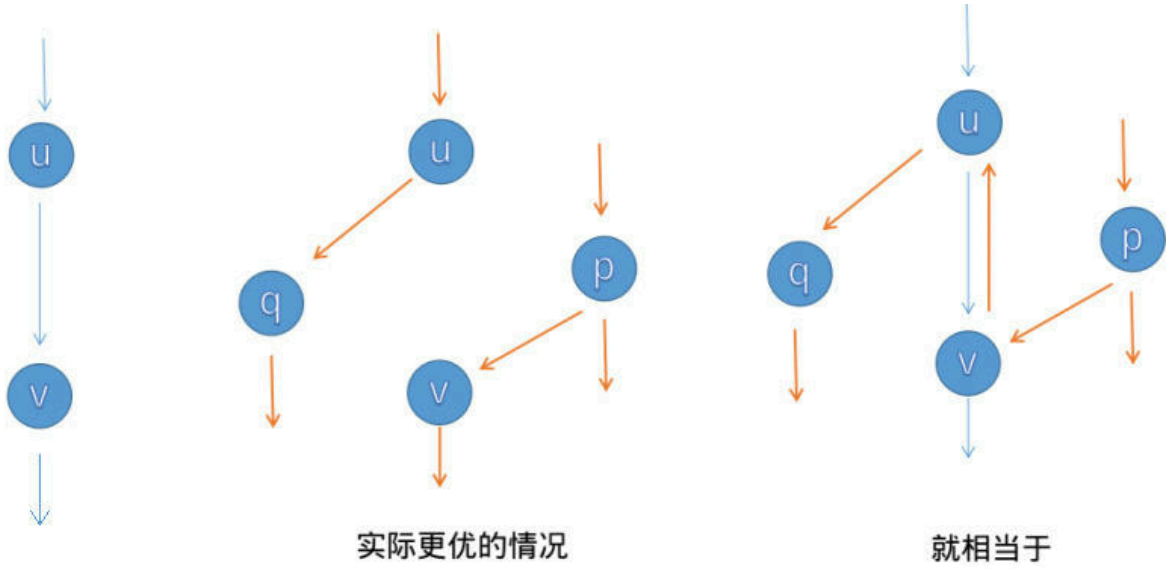
此外，在 Ford–Fulkerson 增广的过程中，对于每条边 (u,v)，我们都新建一条反向边 (v,u)．我们约定 $f(u,v)=-f(v,u)$，这一性质可以通过在每次增广时引入退流操作来保证，即 $f(u,v)$ 增加时 $f(v,u)$ 应当减少同等的量．

> **Tip**
>
> 在最大流算法的代码实现中，我们往往需要支持快速访问反向边的操作．在邻接矩阵中，这一操作是 trivial 的（$g_{u,v}\leftrightarrow g_{v,u}$）．但主流的实现是更加优秀的链式前向星．其中，一个常用的技巧是，我们令边从偶数（通常为 0）开始编号，并在加边时总是紧接着加入其反向边使得它们的编号相邻．由此，我们可以令编号为 i 的边和编号为 $i\oplus 1$ 的边始终保持互为反向边的关系．

初次接触这一方法的读者可能察觉到一个违反直觉的情形——反向边的流量 $f(v,u)$ 可能是一个负值．实际上我们可以注意到，在 Ford–Fulkerson 增广的过程中，真正有意义的是剩余容量 c_f，而 $f(v,u)$ 的绝对值是无关紧要的，我们可以将反向边流量的减少视为反向边剩余容量 $c_f(v,u)$ 的增加——这也与退流的意义相吻合——反向边剩余容量的增加意味着我们接下来可能通过走反向边来和原先正向的增广抵消，代表一种「反悔」的操作．

以下案例有可能帮助你理解这一过程．假设 G 是一个单位容量的网络，我们考虑以下过程：

- G 上有多条增广路，其中，我们选择进行一次先后经过 u,v 的增广（如左图所示），流量增加 1．
- 我们注意到，如果进行中图上的增广，这个局部的最大流量不是 1 而是 2．但由于指向 u 的边和从 v 出发的边在第一次增广中耗尽了容量，此时我们无法进行中图上的增广．这意味着我们当前的流是不够优的，但局部可能已经没有其他（只经过原图中的边而不经过反向边的）增广路了．
- 现在引入退流操作．第一次增广后，退流意味着 $c_f(v,u)$ 增加了 1 剩余容量，即相当于新增 (v,u) 这条边，因此我们可以再进行一次先后经过 p,v,u,q 的增广（如右图橙色路径所示）．无向边 (u,v) 上的流量在两次增广中抵消，我们惊奇地发现两次增广叠加得到的结果实际上和中图是等价的．



以上案例告诉我们，退流操作带来的「抵消」效果使得我们无需担心我们按照「错误」的顺序选择了增广路．

容易发现，只要 G_f 上存在增广路，那么对其增广就可以令总流量增加；否则说明总流量已经达到最大可能值，求解过程完成．这就是 Ford–Fulkerson 增广的过程．

最大流最小割定理

我们大致了解了 Ford–Fulkerson 增广的思想，可是如何证明这一方法的正确性呢？为什么增广结束后的流 f 是一个最大流？

实际上，Ford–Fulkerson 增广的正确性和最大流最小割定理（The Maxflow-Mincut Theorem）等价．这一定理指出，对于任意网络 $G=(V,E)$，其上的最大流 f 和最小割 $\{S,T\}$ 总是满足 $|f|=||S,T||$．

为了证明最大流最小割定理，我们先从一个引理出发：对于网络 $G=(V,E)$，任取一个流 f 和一个割 $\{S,T\}$，总是有 $|f|\leq||S,T||$，其中等号成立当且仅当 $\{(u,v)\mid u\in S,v\in T\}$ 的所有边均满流，且 $\{(u,v)\mid u\in T,v\in S\}$ 的所有边均空流．

证明

$$
\begin{aligned}
|f| &= f(s) \\
&= \sum_{u\in S} f(u) \\
&= \sum_{u\in S}\left(\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u)\right) \\
&= \sum_{u\in S}\left(\sum_{v\in T} f(u,v) + \sum_{v\in S} f(u,v) - \sum_{v\in T} f(v,u) - \sum_{v\in S} f(v,u)\right) \\
&= \sum_{u\in S}\left(\sum_{v\in T} f(u,v) - \sum_{v\in T} f(v,u)\right) + \sum_{u\in S}\sum_{v\in S} f(u,v) - \sum_{u\in S}\sum_{v\in S} f(v,u) \\
&= \sum_{u\in S}\left(\sum_{v\in T} f(u,v) - \sum_{v\in T} f(v,u)\right) \\
&\le \sum_{u\in S}\sum_{v\in T} f(u,v) \\
&\le \sum_{u\in S}\sum_{v\in T} c(u,v) \\
&= ||S,T||
\end{aligned}
$$

为了取等，第一个不等号需要 $\{(u,v) \mid u\in T, v\in S\}$ 的所有边均空流，第二个不等号需要 $\{(u,v) \mid u\in S, v\in T\}$ 的所有边均满流．原引理得证．

那么，对于任意网络，以上取等条件是否总是能被满足呢? 如果答案是肯定的，则最大流最小割定理得证．以下我们尝试证明．

证明

假设某一轮增广后，我们得到流 f 使得 G_f 上不存在增广路，即 G_f 上不存在 s 到 t 的路径．此时我们记从 s 出发可以到达的结点组成的点集为 S，并记 $T = V \setminus S$．

显然，$\{S,T\}$ 是 G_f 的一个割，且 $||S,T|| = \sum_{u\in S}\sum_{v\in T} c_f(u,v) = 0$．由于剩余容量是非负的，这也意味着对于任意 $u\in S, v\in T, (u,v)\in E_f$，均有 $c_f(u,v) = 0$．以下我们将这些边分为存在于原图中的边和反向边两种情况讨论：

- $(u,v)\in E$：此时，$c_f(u,v) = c(u,v) - f(u,v) = 0$，因此有 $c(u,v) = f(u,v)$，即 $\{(u,v) \mid u\in S, v\in T\}$ 的所有边均满流；
- $(v,u)\in E$：此时，$c_f(u,v) = c(u,v) - f(u,v) = 0 - f(u,v) = f(v,u) = 0$，即 $\{(v,u) \mid u\in S, v\in T\}$ 的所有边均空流．

因此，增广停止后，上述流 f 满足取等条件．根据引理指出的大小关系，自然地，f 是 G 的一个最大流，$\{S,T\}$ 是 G 的一个最小割．

容易看出，Kőnig 定理是最大流最小割定理的特殊情形．实际上，它们都和线性规划中的对偶有关．

时间复杂度分析

在整数流量的网络 $G = (V,E)$ 上，平凡地，我们假设每次增广的流量都是整数，则 Ford–Fulkerson 增广的时间复杂度的一个上界是 $O(|E|\,|f|)$，其中 f 是 G 上的最大流．这是因为单轮增广的时间复杂度是 $O(|E|)$，而增广会导致总流量增加，故增广轮数不可能超过 $|f|$．

对于 Ford–Fulkerson 增广的不同实现，时间复杂度也各不相同．其中较主流的实现有 Edmonds–Karp, Dinic, SAP, ISAP 等算法，我们将在下文中分别介绍．

Edmonds–Karp 算法

算法思想

如何在 G_f 中寻找增广路呢？当我们考虑 Ford–Fulkerson 增广的具体实现时，最自然的方案就是使用 BFS．此时，Ford–Fulkerson 增广表现为 Edmonds–Karp 算法．其具体流程如下：

- 如果在 G_f 上我们可以从 s 出发 BFS 到 t，则我们找到了新的增广路．
- 对于增广路 p，我们计算出 p 经过的边的剩余容量的最小值 $\Delta = \min_{(u,v)\in p} c_f(u,v)$．我们给 p 上的每条边都加上 Δ 流量，并给它们的反向边都退掉 Δ 流量，令最大流增加了 Δ．
- 因为我们修改了流量，所以我们得到新的 G_f，我们在新的 G_f 上重复上述过程，直至增广路不存在，则流量不再增加．

以上算法即 Edmonds–Karp 算法．

时间复杂度分析

接下来让我们尝试分析 Edmonds–Karp 算法的时间复杂度．

显然，单轮 BFS 增广的时间复杂度是 $O(|E|)$．

增广总轮数的上界是 $O(|V||E|)$．这一论断在网络资料中常被伪证（或被含糊其辞略过）．以下我们尝试给出一个较正式的证明[1]．

增广总轮数的上界的证明

首先，我们引入一个引理——最短路非递减引理．具体地，我们记 $d_f(u)$ 为 G_f 上结点 u 到源点 s 的距离（即最短路长度，下同）．对于某一轮增广，我们用 f 和 f' 分别表示增广前的流和增广后的流，我们断言，对于任意结点 u，增广总是使得 $d_{f'}(u) \geq d_f(u)$．我们将在稍后证明这一引理．

不妨称增广路上剩余容量最小的边是饱和边（存在多条边同时最小则取任一）．如果一条有向边 (u,v) 被选为饱和边，增广会清空其剩余容量导致饱和边的消失，并且退流导致反向边的新增（如果原先反向边不存在），即 $(u,v)\notin E_{f'}$ 且 $(v,u)\in E_{f'}$．以上分析使我们知道，对于无向边 (u,v)，其被增广的两种方向总是交替出现．

在 G_f 上沿 (u,v) 增广时，$d_f(u)+1=d_f(v)$，此后残量网络变为 $G_{f'}$．在 $G_{f'}$ 上沿 (v,u) 增广时，$d_{f'}(v)+1=d_{f'}(u)$．根据最短路非递减引理又有 $d_{f'}(v)\geq d_f(v)$，我们连接所有式子，得到 $d_{f'}(u)\geq d_f(u)+2$．换言之，如果有向边 (u,v) 被选为饱和边，那么与其上一次被选为饱和边时相比，u 到 s 的距离至少增加 2．

s 到任意结点的距离不可能超过 $|V|$，结合上述性质，我们发现每条边被选为饱和边的次数是 $O(|V|)$ 的，与边数相乘后得到增广总轮数的上界 $O(|V||E|)$．

接下来我们证明最短路非递减引理，即 $d_{f'}(u)\geq d_f(u)$．这一证明并不难，但可能稍显绕口，读者可以停下来认真思考片刻．

最短路非递减引理的证明

考虑反证．对于某一轮增广，我们假设存在若干结点，它们在该轮增广后到 s 的距离较增广前减小．我们记 v 为其中到 s 的距离最小的一者（即 $v=\arg\min_{x\in V, d_{f'}(x)<d_f(x)} d_{f'}(x)$）．注意，根据反证假设，此时 $d_{f'}(v)<d_f(v)$ 是已知条件．

在 $G_{f'}$ 中 s 到 v 的最短路上，我们记 u 是 v 的上一个结点，即 $d_{f'}(u)+1=d_{f'}(v)$．

为了不让 u 破坏 v 的「距离最小」这一性质，u 必须满足 $d_{f'}(u)\geq d_f(u)$．

[1] http://pisces.ck.tp.edu.tw/~peng/index.php?action=showfile&file=f6cdf7ef750d7dc79c7d599b942acbaaee86a2e3e[1]

对于上式，我们令不等号两侧同加，得 $d_{f'}(v) \geq d_f(u) + 1$．根据反证假设进行放缩，我们得到 $d_f(v) > d_f(u) + 1$．

以下我们尝试讨论 (u, v) 上的增广方向．

- 假设有向边 $(u, v) \in E_f$．根据 BFS「广度优先」的性质，我们有 $d_f(u) + 1 \geq d_f(v)$．该式与放缩结果冲突，导出矛盾．
- 假设有向边 $(u, v) \notin E_f$．根据 u 的定义我们已知 $(u, v) \in E_{f'}$，因此这条边的存在必须是当前轮次的增广经过了 (v, u) 并退流产生反向边的结果，也即 $d_f(v) + 1 = d_f(u)$．该式与放缩结果冲突，导出矛盾．

由于 (u, v) 沿任何方向增广都会导出矛盾，我们知道反证假设不成立，最短路非递减引理得证．

将单轮 BFS 增广的复杂度与增广轮数的上界相乘，我们得到 Edmonds–Karp 算法的时间复杂度是 $O(|V||E|^2)$．

代码实现

Edmonds–Karp 算法的可能实现如下．

参考代码

```cpp
constexpr int MAXN = 250;
constexpr int INF = 0x3f3f3f3f;

struct Edge {
  int from, to, cap, flow;

  Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
};

struct EK {
  int n, m;              // n: 点数，m: 边数
  vector<Edge> edges;    // edges: 所有边的集合
  vector<int> G[MAXN];   // G: 点 x -> x 的所有边在 edges 中的下标
  int a[MAXN], p[MAXN];  // a: 点 x -> BFS 过程中最近接近点 x 的边给它的最大流
                         // p: 点 x -> BFS 过程中最近接近点 x 的边

  void init(int n) {
    for (int i = 0; i < n; i++) G[i].clear();
    edges.clear();
  }

  void AddEdge(int from, int to, int cap) {
    edges.push_back(Edge(from, to, cap, 0));
    edges.push_back(Edge(to, from, 0, 0));
    m = edges.size();
    G[from].push_back(m - 2);
    G[to].push_back(m - 1);
  }

  int Maxflow(int s, int t) {
```

```
    int flow = 0;
    for (;;) {
      memset(a, 0, sizeof(a));
      queue<int> Q;
      Q.push(s);
      a[s] = INF;
      while (!Q.empty()) {
        int x = Q.front();
        Q.pop();
        for (int i = 0; i < G[x].size(); i++) {  // 遍历以 x 作为起点的边
          Edge& e = edges[G[x][i]];
          if (!a[e.to] && e.cap > e.flow) {
            p[e.to] = G[x][i];  // G[x][i] 是最近接近点 e.to 的边
            a[e.to] =
                min(a[x], e.cap - e.flow);  // 最近接近点 e.to 的边赋给它的流
            Q.push(e.to);
          }
        }
        if (a[t]) break;  // 如果汇点接受到了流，就退出 BFS
      }
      if (!a[t])
        break;  // 如果汇点没有接受到流，说明源点和汇点不在同一个连通分量上
      for (int u = t; u != s;
           u = edges[p[u]].from) {  // 通过 u 追寻 BFS 过程中 s -> t 的路径
        edges[p[u]].flow += a[t];       // 增加路径上边的 flow 值
        edges[p[u] ^ 1].flow -= a[t];  // 减小反向路径的 flow 值
      }
      flow += a[t];
    }
    return flow;
  }
};
```

Dinic 算法

算法思想

考虑在增广前先对 G_f 做 BFS 分层，即根据结点 u 到源点 s 的距离 $d(u)$ 把结点分成若干层．令经过 u 的流量只能流向下一层的结点 v，即删除 u 向层数标号相等或更小的结点的出边，我们称 G_f 剩下的部分为层次图（Level Graph）．形式化地，我们称 $G_L=(V,E_L)$ 是 $G_f=(V,E_f)$ 的层次图，其中 $E_L=\{(u,v)\mid (u,v)\in E_f, d(u)+1=d(v)\}$．

如果我们在层次图 G_L 上找到一个极大的增广流 f_b，使得仅在 G_L 上是不可能进一步扩大流 f_b 的，则我们称 f_b 是 G_L 的阻塞流（Blocking Flow）．

> ⚠ **Warning**
>
> 尽管在上文中我们仅在单条增广路上定义了增广/增广流，广义地，「增广」一词不仅可以用于单条路径上的增广流，也可以用于若干增广流的并——后者才是我们定义阻塞流时使用的意义．

定义层次图和阻塞流后，Dinic 算法的流程如下．

1. 在 G_f 上 BFS 出层次图 G_L．
2. 在 G_L 上 DFS 出阻塞流 f_b．

3. 将 f_b 并到原先的流 f 中，即 $f \leftarrow f + f_b$.
4. 重复以上过程直到不存在从 s 到 t 的路径.

此时的 f 即为最大流.

在分析这一算法的复杂度之前，我们需要特别说明「在 G_L 上 DFS 出阻塞流 f_b」的过程. 尽管 BFS 层次图对于本页面的读者应当是 trivial 的，但 DFS 阻塞流的过程则稍需技巧——我们需要引入当前弧优化.

注意到在 G_L 上 DFS 的过程中，如果结点 u 同时具有大量入边和出边，并且 u 每次接受来自入边的流量时都遍历出边表来决定将流量传递给哪条出边，则 u 这个局部的时间复杂度最坏可达 $O(|E|^2)$. 为避免这一缺陷，如果某一时刻我们已经知道边 (u, v) 已经增广到极限（边 (u, v) 已无剩余容量或 v 的后侧已增广至阻塞），则 u 的流量没有必要再尝试流向出边 (u, v). 据此，对于每个结点 u，我们维护 u 的出边表中第一条还有必要尝试的出边. 习惯上，我们称维护的这个指针为当前弧，称这个做法为当前弧优化.

多路增广

多路增广是 Dinic 算法的一个常数优化——如果我们在层次图上找到了一条从 s 到 t 的增广路 p，则接下来我们未必需要重新从 s 出发找下一条增广路，而可能从 p 上最后一个仍有剩余容量的位置出发寻找一条岔路进行增广. 考虑到其与回溯形式的一致性，这一优化在 DFS 的代码实现中也是自然的.

× 常见误区

可能是由于大量网络资料的错误表述引发以讹传讹的情形，相当数量的选手喜欢将当前弧优化和多路增广并列称为 Dinic 算法的两种优化. 实际上，当前弧优化是用于保证 Dinic 时间复杂度正确性的一部分，而多路增广只是一个不影响复杂度的常数优化.

时间复杂度分析

应用当前弧优化后，对 Dinic 算法的时间复杂度分析如下.

首先，我们尝试证明单轮增广中 DFS 求阻塞流的时间复杂度是 $O(|V||E|)$.

单轮增广的时间复杂度的证明

考虑阻塞流 f_b 中的每条增广路，它们都是在 G_L 上每次沿当前弧跳转而得到的结果，其中每条增广路经历的跳转次数不可能多于 $|V|$.

每找到一条增广路就有一条饱和边消失（剩余容量清零）. 考虑阻塞流 f_b 中的每条增广路，我们将被它们清零的饱和边形成的边集记作 E_1. 考虑到 G_L 分层的性质，饱和边消失后其反向边不可能在同一轮增广内被其他增广路经过，因此，E_1 是 E_L 的子集.

此外，对于沿当前弧跳转但由于某个位置阻塞所以没有成功得到增广路的情形，我们将这些不完整的路径上的最后一条边形成的边集记作 E_2. E_2 的成员不饱和，所以 E_1 与 E_2 不交，且 $E_1 \cup E_2$ 仍是 E_L 的子集.

由于 $E_1 \cup E_2$ 的每个成员都没有花费超过 $|V|$ 次跳转（且在使用多路增广优化后一些跳转将被重复计数），因此，综上所述，DFS 过程中的总跳转次数不可能多于 $|V||E_L|$.

× 常见伪证一则

对于每个结点，我们维护下一条可以增广的边，而当前弧最多变化 $|E|$ 次，从而单轮增广的最坏时间复杂度为 $O(|V||E|)$.

Bug

「当前弧最多变化 $|E|$ 次」并不能推得「每个结点最多访问其出边 $|E|$ 次」. 这是因为, 访问当前弧并不一定耗尽上面的剩余容量, 结点 u 可能多次访问同一条当前弧.

注意到层次图的层数显然不可能超过 $|V|$, 如果我们可以证明层次图的层数在增广过程中严格单增, 则 Dinic 算法的增广轮数是 $O(|V|)$ 的. 接下来我们尝试证明这一结论[2].

层次图层数单调性的证明

我们需要引入预流推进类算法 (另一类最大流算法) 中的一个概念——高度标号. 为了更方便地结合高度标号表述我们的证明, 在证明过程中, 我们令 $d_f(u)$ 为 G_f 上结点 u 到 **汇点** t 的距离, 从 **汇点** 而非源点出发进行分层 (这并没有本质上的区别). 对于某一轮增广, 我们用 f 和 f' 分别表示增广前的流和增广后的流. 在该轮增广中求解并加入阻塞流后, 记层次图由 $G_L=(V,E_L)$ 变为 $G'_L=(V,E'_L)$.

我们给高度标号一个不严格的临时定义——在网络 $G=(V,E)$ 上, 令 h 是点集 V 到整数集 N 上的函数, h 是 G 上合法的高度标号当且仅当 $h(u)\le h(v)+1$ 对于 $(u,v)\in E$ 恒成立.

考察所有 $E_{f'}$ 的成员 (u,v), 我们发现 $(u,v)\in E_{f'}$ 的原因是以下二者之一.

- $(u,v)\in E_f$, 且剩余容量在该轮增广过程中未耗尽——根据最短路的定义, 此时我们有 $d_f(u)\le d_f(v)+1$;
- $(u,v)\notin E_f$, 但在该轮增广过程中阻塞流经过 (v,u) 并退流产生反向边——根据层次图和阻塞流的定义, 此时我们有 $d_f(u)+1=d_f(v)$.

以上观察让我们得出一个结论——d_f 在 $G_{f'}$ 上是一个合法的高度标号. 当然, 在 $G_{f'}$ 的子图 G'_L 上也是.

现在, 对于一条 G'_L 上的增广路 $p=(s,\dots,u,v,\dots,t)$, 按照 p 上结点的反序 (从 t 到 s 的顺序) 考虑从空路径开始每次添加一个结点的过程. 假设结点 v 已加入, 结点 u 正在加入, 我们发现, 加入结点 u 后, 根据层次图的定义, $d_{f'}(u)$ 的值较 $d_{f'}(v)$ 增加 1; 与此同时, 由于 d_f 是 G'_L 上的高度标号, $d_f(u)$ 的值既可能较 $d_f(v)$ 增加 1, 也可能保持不变或减少. 因此, 在整条路径被添加完成后, 我们得到 $d_{f'}(s)\ge d_f(s)$, 其中取等的充要条件是 $d_f(u)=d_f(v)+1$ 对于 $(u,v)\in p$ 恒成立. 如果该不等式不能取等, 则有 $d_{f'}(s)>d_f(s)$——即我们想要的结论「层次图的层数在增广过程中严格单增」. 以下我们尝试证明该不等式不能取等.

考虑反证, 我们假设 $d_{f'}(s)=d_f(s)$ 成立, 并尝试导出矛盾. 现在我们断言, 在 G'_L 上, p 至少包含一条边 (u,v) 满足 (u,v) 在 G_L 上不存在. 如果没有这样的边, 考虑到 $d_f(s)=d_{f'}(s)$, 结合层次图和阻塞流的定义, G_L 上的增广应尚未完成. 为了不产生以上矛盾, 我们的断言只好是正确的.

令 (u,v) 是满足断言条件的那条边, 其满足断言的原因只能是以下二者之一.

- $(u,v)\in E_f$ 但 $d_f(u)\le d_f(v)+1$ 未取等, 故根据层次图的定义可知 $(u,v)\notin E_L$, 并在增广后新一轮重分层中被加入到 E'_L 中;
- $(u,v)\notin E_f$, 这意味着 (u,v) 这条边的产生是当前轮次增广中阻塞流经过 (v,u) 并退流产生反向边的结果, 也即 $d_f(u)=d_f(v)-1$.

由于我们无论以何种方式满足断言均得到 $d_f(u)\ne d_f(v)+1$, 也即 $d_{f'}(s)\ge d_f(s)$ 取等的充要条件无法被满足, 这与反证假设 $d_{f'}(s)=d_f(s)$ 冲突, 原命题得证.

[2] https://people.orie.cornell.edu/dpw/orie633/LectureNotes/lecture9.pdf

× 常见伪证另一则

考虑反证. 假设层次图的层数在一轮增广结束后较原先相等, 则层次图上应仍存在至少一条从 s 到 t 的增广路满足相邻两点间的层数差为 1. 这条增广路未被增广说明该轮增广尚未结束. 为了不产生上述矛盾，原命题成立.

Bug

「一轮增广结束后新的层次图上 s-t 最短路较原先相等」并不能推得「旧的层次图上该轮增广尚未结束」. 这是因为，没有理由表明两张层次图的边集相同，新的层次图上的 s-t 最短路有可能经过旧的层次图上不存在的边.

将单轮增广的时间复杂度 $O(|V||E|)$ 与增广轮数 $O(|V|)$ 相乘，Dinic 算法的时间复杂度是 $O(|V|^2|E|)$.

如果需要令 Dinic 算法的实际运行时间接近其理论上界，我们需要构造有特殊性质的网络作为输入. 由于在算法竞赛实践中，对于网络流知识相关的考察常侧重于将原问题建模为网络流问题的技巧. 此时，我们的建模通常不包含令 Dinic 算法执行缓慢的特殊性质；恰恰相反，Dinic 算法在大部分图上效率非常优秀. 因此，网络流问题的数据范围通常较大，「将 $|V|,|E|$ 的值代入 $|V|^2|E|$ 以估计运行时间」这一方式并不适用. 实际上，进行准确的估计需要选手对 Dinic 算法的实际效率有一定的经验，读者可以多加练习.

特殊情形下的时间复杂度分析

在一些性质良好的图上，Dinic 算法有更好的时间复杂度.

对于网络 $G=(V,E)$，如果其所有边容量均为 1，即 $c(u,v)\in\{0,1\}$ 对于 $(u,v)\in E$ 恒成立，则我们称 G 是单位容量（Unit Capacity）的.

在单位容量的网络中，Dinic 算法的单轮增广的时间复杂度为 $O(|E|)$.

证明

这是因为, 每次增广都会导致增广路上的所有边均饱和并消失, 故单轮增广中每条边只能被增广一次.

在单位容量的网络中，Dinic 算法的增广轮数是 $O(|E|^{\frac{1}{2}})$ 的.

证明

以源点 s 为中心分层，记 $d_f(u)$ 为 G_f 上结点 u 到源点 s 的距离. 另外，我们定义将点集 $\{u\mid u\in V,d_f(u)=k\}$ 定义为编号为 k 的层次 D_k，并记 $S_k=\cup_{i\le k}D_i$.

假设我们已经进行了 $|E|^{\frac{1}{2}}$ 轮增广. 根据鸽巢原理，至少存在一个 k 满足边集 $\{(u,v)\mid u\in D_k,v\in D_{k+1},(u,v)\in E_f\}$ 的大小不超过 $\frac{|E|}{|E|^{\frac{1}{2}}}\approx|E|^{\frac{1}{2}}$. 显然，$\{S_k,V-S_k\}$ 是 G_f 上的 s-t 割，且其割容量不超过 $|E|^{\frac{1}{2}}$. 根据最大流最小割定理，G_f 上的最大流不超过 $|E|^{\frac{1}{2}}$，也即 G_f 上最多还能执行 $|E|^{\frac{1}{2}}$ 轮增广. 因此，总增广轮数是 $O(|E|^{\frac{1}{2}})$ 的.

在单位容量的网络中，Dinic 算法的增广轮数是 $O(|V|^{\frac{2}{3}})$ 的.

证明

假设我们已经进行了 $2 \mid V \mid^{\frac{2}{3}}$ 轮增广. 由于至多有半数的 ($\mid V \mid^{\frac{2}{3}}$ 个) 层次包含多于 $\mid V \mid^{\frac{1}{3}}$ 个点，故无论我们如何分配所有层次的大小，至少存在一个 k 满足相邻两个层次同时包含不多于 $\mid V \mid^{\frac{1}{3}}$ 个点，即 $\mid D_k \mid \leq \mid V \mid^{\frac{1}{3}}$ 且 $\mid D_{k+1} \mid \leq \mid V \mid^{\frac{1}{3}}$.

为最大化 D_k 和 D_{k+1} 之间的边数，我们假定这是一个完全二分图，此时边集 $\{(u,v) \mid u \in D_k, v \in D_{k+1}, (u,v) \in E_f\}$ 的大小不超过 $\mid V \mid^{\frac{2}{3}}$. 显然，$\{S_k, V - S_k\}$ 是 G_f 上的 s-t 割，且其割容量不超过 $\mid V \mid^{\frac{2}{3}}$. 根据最大流最小割定理，$G_f$ 上的最大流不超过 $\mid V \mid^{\frac{2}{3}}$，也即 G_f 上最多还能执行 $\mid V \mid^{\frac{2}{3}}$ 轮增广. 因此，总增广轮数是 $O(\mid V \mid^{\frac{2}{3}})$ 的.

在单位容量的网络中，如果除源汇点外每个结点 u 都满足 $deg_{in}(u) = 1$ 或 $deg_{out}(u) = 1$，则 Dinic 算法的增广轮数是 $O(\mid V \mid^{\frac{1}{2}})$ 的. 其中，$deg_{in}(u)$ 和 $deg_{out}(u)$ 分别代表结点 u 的入度和出度.

证明

我们引入以下引理——对于这一形式的网络，其上的任意流总是可以分解成若干条单位流量的、**点不交** 的增广路.

假设我们已经进行了 $\mid V \mid^{\frac{1}{2}}$ 轮增广. 根据层次图的定义，此时任意新的增广路的长度至少为 $\mid V \mid^{\frac{1}{2}}$.

考虑 G_f 上的最大流的增广路分解，我们得到的增广路的数量不能多于 $\frac{\mid V \mid}{\mid V \mid^{\frac{1}{2}}} \approx \mid V \mid^{\frac{1}{2}}$，这意味着 G_f 上最多还能执行 $\mid V \mid^{\frac{1}{2}}$ 轮增广. 因此，总增广轮数是 $O(\mid V \mid^{\frac{1}{2}})$ 的.

综上，我们得出一些推论.

- 在单位容量的网络上，Dinic 算法的总时间复杂度是 $O(\mid E \mid \min(\mid E \mid^{\frac{1}{2}}, \mid V \mid^{\frac{2}{3}}))$.
- 在单位容量的网络上，如果除源汇点外每个结点 u 都满足 $deg_{in}(u) = 1$ 或 $deg_{out}(u) = 1$，Dinic 算法的总时间复杂度是 $O(\mid E \mid \mid V \mid^{\frac{1}{2}})$. 对于二分图最大匹配问题，我们常使用 Hopcroft–Karp 算法解决，而这一算法实际上是 Dinic 算法在满足上述度数限制的单位容量网络上的特例.

代码实现

参考代码

```
struct MF {
  struct edge {
    int v, nxt, cap, flow;
  } e[N];

  int fir[N], cnt = 0;

  int n, S, T;
  ll maxflow = 0;
  int dep[N], cur[N];

  void init() {
    memset(fir, -1, sizeof fir);
    cnt = 0;
  }

  void addedge(int u, int v, int w) {
    e[cnt] = {v, fir[u], w, 0};
```

```
    fir[u] = cnt++;
    e[cnt] = {u, fir[v], 0, 0};
    fir[v] = cnt++;
  }

  bool bfs() {
    queue<int> q;
    memset(dep, 0, sizeof(int) * (n + 1));

    dep[S] = 1;
    q.push(S);
    while (q.size()) {
      int u = q.front();
      q.pop();
      for (int i = fir[u]; ~i; i = e[i].nxt) {
        int v = e[i].v;
        if ((!dep[v]) && (e[i].cap > e[i].flow)) {
          dep[v] = dep[u] + 1;
          q.push(v);
        }
      }
    }
    return dep[T];
  }

  int dfs(int u, int flow) {
    if ((u == T) || (!flow)) return flow;

    int ret = 0;
    for (int& i = cur[u]; ~i; i = e[i].nxt) {
      int v = e[i].v, d;
      if ((dep[v] == dep[u] + 1) &&
          (d = dfs(v, min(flow - ret, e[i].cap - e[i].flow)))) {
        ret += d;
        e[i].flow += d;
        e[i ^ 1].flow -= d;
        if (ret == flow) return ret;
      }
    }
    return ret;
  }

  void dinic() {
    while (bfs()) {
      memcpy(cur, fir, sizeof(int) * (n + 1));
      maxflow += dfs(S, INF);
    }
  }
} mf;
```

MPM 算法

MPM(Malhotra, Pramodh-Kumar and Maheshwari) 算法得到最大流的方式有两种：使用基于堆的优先队列，时间复杂度为 $O(n^3 \log n)$；常用 BFS 解法，时间复杂度为 $O(n^3)$．注意，本章节只专注于分析更优也更简洁的 $O(n^3)$ 算法．

MPM 算法的整体结构和 Dinic 算法类似，也是分阶段运行的．在每个阶段，在 G 的残量网络的分层网络中找到增广路．它与 Dinic 算法的主要区别在于寻找增广路的方式不同：MPM 算法中寻找增广路的部分的只花了 $O(n^2)$，时间复杂度要优于 Dinic 算法．

MPM 算法需要考虑顶点而不是边的容量．在分层网络 L 中，如果定义点 v 的容量 $p(v)$ 为其传入残量和传出残量的最小值，则有：

$$p_{in}(v) = \sum_{(u,v)\in L} (c(u,v) - f(u,v))$$

$$p_{out}(v) = \sum_{(v,u)\in L} (c(v,u) - f(v,u))$$

$$p(v) = \min(p_{in}(v), p_{out}(v))$$

我们称节点 r 是参考节点当且仅当 $p(r) = \min p(v)$．对于一个参考节点 r，我们一定可以让经过 r 的流量增加 $p(r)$ 以使其容量变为 0．这是因为 L 是有向无环图且 L 中节点容量至少为 $p(r)$，所以我们一定能找到一条从 s 经过 r 到达 t 的有向路径．那么我们让这条路上的边流量都增加 $p(r)$ 即可．这条路即为这一阶段的增广路．寻找增广路可以用 BFS．增广完之后所有满流边都可以从 L 中删除，因为它们不会在此阶段后被使用．同样，所有与 s 和 t 不同且没有出边或入边的节点都可以删除．

时间复杂度分析

MPM 算法的每个阶段都需要 $O(V^2)$，因为最多有 V 次迭代（因为至少删除了所选的参考节点），并且在每次迭代中，我们删除除最多 V 之外经过的所有边．求和，我们得到 $O(V^2 + E) = O(V^2)$．由于阶段总数少于 V，因此 MPM 算法的总运行时间为 $O(V^3)$．

阶段总数小于 V 的证明

MPM 算法在少于 V 个阶段内结束．为了证明这一点，我们必须首先证明两个引理．

引理 1：每次迭代后，从 s 到每个点的距离不会减少，也就是说，$level_{i+1}[v] \geq level_i[v]$.

证明：固定一个阶段 i 和点 v. 考虑 G_i^R 中从 s 到 v 的任意最短路径 P. P 的长度等于 $level_i[v]$. 注意 G_i^R 只能包含 G_i^R 的后向边和前向边. 如果 P 没有 G_i^R 的后边，那么 $level_{i+1}[v] \geq level_i[v]$. 因为 P 也是 G_i^R 中的一条路径. 现在，假设 P 至少有一个后向边且第一个这样的边是 (u, w)，那么 $level_{i+1}[u] \geq level_i[u]$（因为第一种情况）. 边 (u, w) 不属于 G_i^R，因此 (u, w) 受到前一次迭代的增广路的影响. 这意味着 $level_i[u] = level_i[w] + 1$. 此外，$level_{i+1}[w] = level_{i+1}[u] + 1$. 从这两个方程和 $level_{i+1}[u] \geq level_i[u]$ 我们得到 $level_{i+1}[w] \geq level_i[w] + 2$. 路径的剩余部分也可以使用相同思想.

引理 2：$level_{i+1}[t] > level_i[t]$.

证明：从引理一我们得出，$level_{i+1}[t] \geq level_i[t]$. 假设 $level_{i+1}[t] = level_i[t]$，注意 G_i^R 只能包含 G_i^R 的后向边和前向边. 这意味着 G_i^R 中有一条最短路径未被增广路阻塞. 这就形成了矛盾.

实现

参考代码

```
struct MPM {
  struct FlowEdge {
    int v, u;
```

```
    long long cap, flow;

    FlowEdge() {}

    FlowEdge(int _v, int _u, long long _cap, long long _flow)
        : v(_v), u(_u), cap(_cap), flow(_flow) {}

    FlowEdge(int _v, int _u, long long _cap)
        : v(_v), u(_u), cap(_cap), flow(0ll) {}
};

constexpr static long long flow_inf = 1e18;
vector<FlowEdge> edges;
vector<char> alive;
vector<long long> pin, pout;
vector<list<int>> in, out;
vector<vector<int>> adj;
vector<long long> ex;
int n, m = 0;
int s, t;
vector<int> level;
vector<int> q;
int qh, qt;

void resize(int _n) {
  n = _n;
  ex.resize(n);
  q.resize(n);
  pin.resize(n);
  pout.resize(n);
  adj.resize(n);
  level.resize(n);
  in.resize(n);
  out.resize(n);
}

MPM() {}

MPM(int _n, int _s, int _t) {
  resize(_n);
  s = _s;
  t = _t;
}

void add_edge(int v, int u, long long cap) {
  edges.push_back(FlowEdge(v, u, cap));
  edges.push_back(FlowEdge(u, v, 0));
  adj[v].push_back(m);
  adj[u].push_back(m + 1);
  m += 2;
}

```

```
  bool bfs() {
    while (qh < qt) {
      int v = q[qh++];
      for (int id : adj[v]) {
        if (edges[id].cap - edges[id].flow < 1) continue;
        if (level[edges[id].u] != -1) continue;
        level[edges[id].u] = level[v] + 1;
        q[qt++] = edges[id].u;
      }
    }
    return level[t] != -1;
  }

  long long pot(int v) { return min(pin[v], pout[v]); }

  void remove_node(int v) {
    for (int i : in[v]) {
      int u = edges[i].v;
      auto it = find(out[u].begin(), out[u].end(), i);
      out[u].erase(it);
      pout[u] -= edges[i].cap - edges[i].flow;
    }
    for (int i : out[v]) {
      int u = edges[i].u;
      auto it = find(in[u].begin(), in[u].end(), i);
      in[u].erase(it);
      pin[u] -= edges[i].cap - edges[i].flow;
    }
  }

  void push(int from, int to, long long f, bool forw) {
    qh = qt = 0;
    ex.assign(n, 0);
    ex[from] = f;
    q[qt++] = from;
    while (qh < qt) {
      int v = q[qh++];
      if (v == to) break;
      long long must = ex[v];
      auto it = forw ? out[v].begin() : in[v].begin();
      while (true) {
        int u = forw ? edges[*it].u : edges[*it].v;
        long long pushed = min(must, edges[*it].cap - edges[*it].flow);
        if (pushed == 0) break;
        if (forw) {
          pout[v] -= pushed;
          pin[u] -= pushed;
        } else {
          pin[v] -= pushed;
          pout[u] -= pushed;
        }
        if (ex[u] == 0) q[qt++] = u;
```

```
108         ex[u] += pushed;
109         edges[*it].flow += pushed;
110         edges[(*it) ^ 1].flow -= pushed;
111         must -= pushed;
112         if (edges[*it].cap - edges[*it].flow == 0) {
113           auto jt = it;
114           ++jt;
115           if (forw) {
116             in[u].erase(find(in[u].begin(), in[u].end(), *it));
117             out[v].erase(it);
118           } else {
119             out[u].erase(find(out[u].begin(), out[u].end(), *it));
120             in[v].erase(it);
121           }
122           it = jt;
123         } else
124           break;
125         if (!must) break;
126       }
127     }
128   }
129
130   long long flow() {
131     long long ans = 0;
132     while (true) {
133       pin.assign(n, 0);
134       pout.assign(n, 0);
135       level.assign(n, -1);
136       alive.assign(n, true);
137       level[s] = 0;
138       qh = 0;
139       qt = 1;
140       q[0] = s;
141       if (!bfs()) break;
142       for (int i = 0; i < n; i++) {
143         out[i].clear();
144         in[i].clear();
145       }
146       for (int i = 0; i < m; i++) {
147         if (edges[i].cap - edges[i].flow == 0) continue;
148         int v = edges[i].v, u = edges[i].u;
149         if (level[v] + 1 == level[u] && (level[u] < level[t] || u == t)) {
150           in[u].push_back(i);
151           out[v].push_back(i);
152           pin[u] += edges[i].cap - edges[i].flow;
153           pout[v] += edges[i].cap - edges[i].flow;
154         }
155       }
156       pin[s] = pout[t] = flow_inf;
157       while (true) {
158         int v = -1;
159         for (int i = 0; i < n; i++) {
```

```
160           if (!alive[i]) continue;
161           if (v == -1 || pot(i) < pot(v)) v = i;
162         }
163         if (v == -1) break;
164         if (pot(v) == 0) {
165           alive[v] = false;
166           remove_node(v);
167           continue;
168         }
169         long long f = pot(v);
170         ans += f;
171         push(v, s, f, false);
172         push(v, t, f, true);
173         alive[v] = false;
174         remove_node(v);
175       }
176     }
177     return ans;
178   }
179 };
```

ISAP

在 Dinic 算法中，我们每次求完增广路后都要跑 BFS 来分层，有没有更高效的方法呢?

答案就是下面要介绍的 ISAP 算法.

过程

和 Dinic 算法一样，我们还是先跑 BFS 对图上的点进行分层，不过与 Dinic 略有不同的是，我们选择在反图上，从 t 点向 s 点进行 BFS.

执行完分层过程后，我们通过 DFS 来找增广路.

增广的过程和 Dinic 类似，我们只选择比当前点层数少 1 的点来增广.

与 Dinic 不同的是，我们并不会重跑 BFS 来对图上的点重新分层，而是在增广的过程中就完成重分层过程.

具体来说，设 i 号点的层为 d_i，当我们结束在 i 号点的增广过程后，我们遍历残量网络上 i 的所有出边，找到层最小的出点 j，随后令 $d_i \leftarrow d_j + 1$. 特别地，若残量网络上 i 无出边，则 $d_i \leftarrow n$.

容易发现，当 $d_s \geq n$ 时，图上不存在增广路，此时即可终止算法.

和 Dinic 类似，ISAP 中也存在 **当前弧优化**.

而 ISAP 还存在另外一个优化，我们记录层数为 i 的点的数量 num_i，每当将一个点的层数从 x 更新到 y 时，同时更新 num 数组的值，若在更新后 $num_x = 0$，则意味着图上出现了断层，无法再找到增广路，此时可以直接终止算法（实现时直接将 d_s 标为 n），该优化被称为 GAP **优化**.

实现

参考代码

```
1 struct Edge {
2   int from, to, cap, flow;
```

```
3
4   Edge(int u, int v, int c, int f) : from(u), to(v), cap(c), flow(f) {}
5 };
6
7 bool operator<(const Edge& a, const Edge& b) {
8   return a.from < b.from || (a.from == b.from && a.to < b.to);
9 }
10
11 struct ISAP {
12   int n, m, s, t;
13   vector<Edge> edges;
14   vector<int> G[MAXN];
15   bool vis[MAXN];
16   int d[MAXN];
17   int cur[MAXN];
18   int p[MAXN];
19   int num[MAXN];
20
21   void AddEdge(int from, int to, int cap) {
22     edges.push_back(Edge(from, to, cap, 0));
23     edges.push_back(Edge(to, from, 0, 0));
24     m = edges.size();
25     G[from].push_back(m - 2);
26     G[to].push_back(m - 1);
27   }
28
29   bool BFS() {
30     memset(vis, 0, sizeof(vis));
31     queue<int> Q;
32     Q.push(t);
33     vis[t] = true;
34     d[t] = 0;
35     while (!Q.empty()) {
36       int x = Q.front();
37       Q.pop();
38       for (int i = 0; i < G[x].size(); i++) {
39         Edge& e = edges[G[x][i] ^ 1];
40         if (!vis[e.from] && e.cap > e.flow) {
41           vis[e.from] = true;
42           d[e.from] = d[x] + 1;
43           Q.push(e.from);
44         }
45       }
46     }
47     return vis[s];
48   }
49
50   void init(int n) {
51     this->n = n;
52     for (int i = 0; i < n; i++) G[i].clear();
53     edges.clear();
54   }
```

```
55
56  int Augment() {
57    int x = t, a = INF;
58    while (x != s) {
59      Edge& e = edges[p[x]];
60      a = min(a, e.cap - e.flow);
61      x = edges[p[x]].from;
62    }
63    x = t;
64    while (x != s) {
65      edges[p[x]].flow += a;
66      edges[p[x] ^ 1].flow -= a;
67      x = edges[p[x]].from;
68    }
69    return a;
70  }
71
72  int Maxflow(int s, int t) {
73    this->s = s;
74    this->t = t;
75    int flow = 0;
76    BFS();
77    memset(num, 0, sizeof(num));
78    for (int i = 0; i < n; i++) num[d[i]]++;
79    int x = s;
80    memset(cur, 0, sizeof(cur));
81    while (d[s] < n) {
82      if (x == t) {
83        flow += Augment();
84        x = s;
85      }
86      int ok = 0;
87      for (int i = cur[x]; i < G[x].size(); i++) {
88        Edge& e = edges[G[x][i]];
89        if (e.cap > e.flow && d[x] == d[e.to] + 1) {
90          ok = 1;
91          p[e.to] = G[x][i];
92          cur[x] = i;
93          x = e.to;
94          break;
95        }
96      }
97      if (!ok) {
98        int m = n - 1;
99        for (int i = 0; i < G[x].size(); i++) {
100         Edge& e = edges[G[x][i]];
101         if (e.cap > e.flow) m = min(m, d[e.to]);
102       }
103       if (--num[d[x]] == 0) break;
104       num[d[x] = m + 1]++;
105       cur[x] = 0;
106       if (x != s) x = edges[p[x]].from;
```

```
      }
    }
    return flow;
  }
};
```

Push-Relabel 预流推进算法

该方法在求解过程中忽略流守恒性，并每次对一个结点更新信息，以求解最大流.

通用的预流推进算法

首先我们介绍预流推进算法的主要思想，以及一个可行的暴力实现算法.

预流推进算法通过对单个结点的更新操作，直到没有结点需要更新来求解最大流.

算法过程维护的流函数不一定保持流守恒性，对于一个结点，我们允许进入结点的流超过流出结点的流，超过的部分被称为结点 $u(u \in V-\{s,t\})$ 的 **超额流** $e(u)$：

$$e(u)=\sum_{(x,u)\in E} f(x,u)-\sum_{(u,y)\in E} f(u,y)$$

若 $e(u)>0$，称结点 u **溢出**[3]，注意当我们提到溢出结点时，并不包括 s 和 t.

预流推进算法维护每个结点的高度 $h(u)$，并且规定溢出的结点 u 如果要推送超额流，只能向高度小于 u 的结点推送；如果 u 没有相邻的高度小于 u 的结点，就修改 u 的高度（重贴标签）.

高度函数[4]

准确地说，预流推进维护以下的一个映射 $h:V\to \mathbf{N}$：

- $h(s)=\mid V\mid, h(t)=0$
- $\forall (u,v)\in E_f, h(u)\le h(v)+1$

称 h 是残量网络 $G_f=(V_f,E_f)$ 的高度函数.

引理 1：设 G_f 上的高度函数为 h，对于任意两个结点 $u,v\in V$，如果 $h(u)>h(v)+1$，则 (u,v) 不是 G_f 中的边.

算法只会在 $h(u)=h(v)+1$ 的边执行推送.

推送（Push）

适用条件：结点 u 溢出，且存在结点 $v((u,v)\in E_f, c(u,v)-f(u,v)>0, h(u)=h(v)+1)$，则 push 操作适用于 (u,v).

于是，我们尽可能将超额流从 u 推送到 v，推送过程中我们只关心超额流和 $c(u,v)-f(u,v)$ 的最小值，不关心 v 是否溢出.

如果 (u,v) 在推送完之后满流，将其从残量网络中删除.

重贴标签（Relabel）

适用条件：如果结点 u 溢出，且 $\forall (u,v)\in E_f, h(u)\le h(v)$，则 relabel 操作适用于 u.

则将 $h(u)$ 更新为 $\min_{(u,v)\in E_f} h(v)+1$ 即可.

---

[3] 英语文献中通常称为「active」.

[4] 在英语文献中，一个结点的高度通常被称为「distance label」. 此处使用的「高度」这个术语源自算法导论中的相关章节. 你可以在机械工业出版社算法导论（原书第 3 版）的 P432 脚注中找到这么做的理由.

初始化

$$\forall (u,v)\in E,\quad f(u,v)=\begin{cases}c(u,v), u=s\\0, \quad u\neq s\end{cases}$$

$$\forall u\in V,\quad h(u)=\begin{cases}|\ V\ |, u=s\\0, \quad u\neq s\end{cases}$$

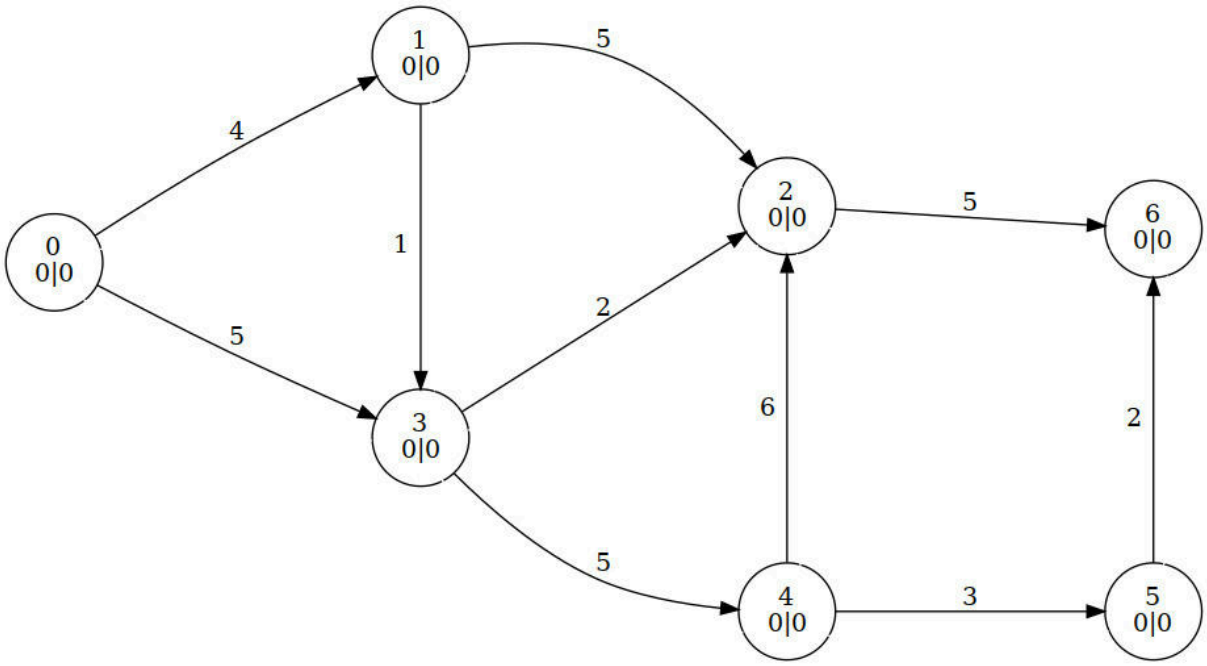
$$e(u)=\sum_{(x,u)\in E} f(x,u)-\sum_{(u,y)\in E} f(u,y)$$

上述将 $(s,v)\in E$ 充满流，并将 $h(s)$ 抬高，使得 $(s,v)\notin E_f$，因为 $h(s)>h(v)$，而且 (s,v) 毕竟满流，没必要留在残量网络中；上述还将 $e(s)$ 初始化为 $\sum_{(s,v)\in E} f(s,v)$ 的相反数.

过程

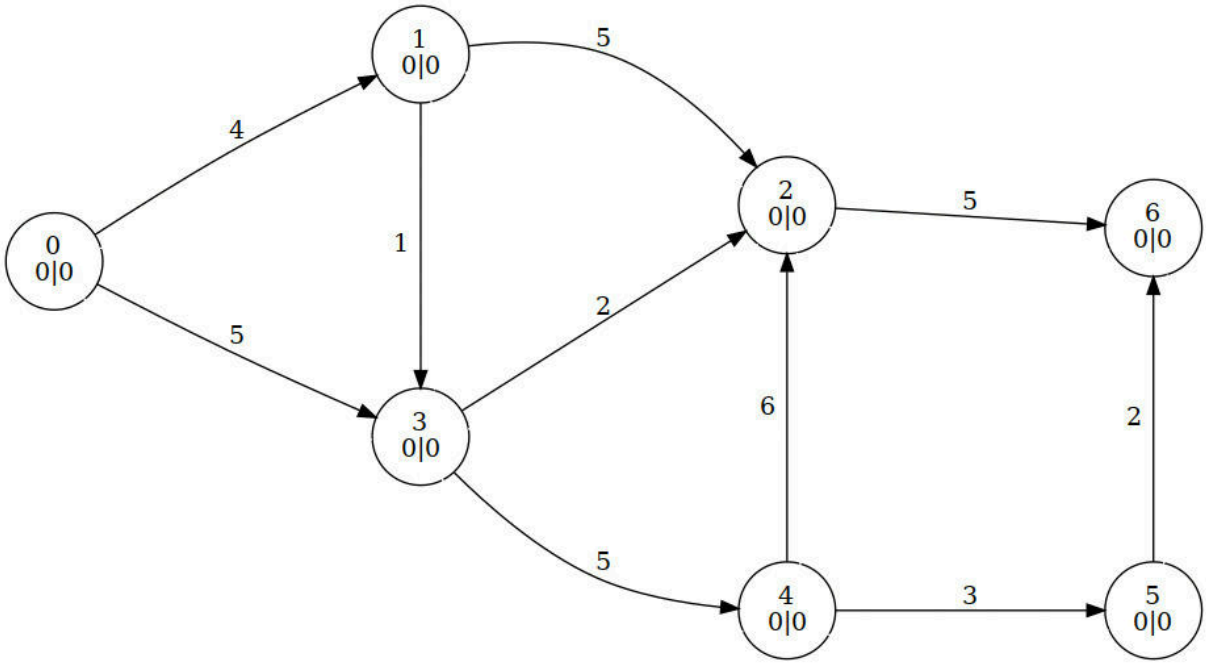
我们每次扫描整个图，只要存在结点 u 满足 push 或 relabel 操作的条件，就执行对应的操作.

如图，每个结点中间表示编号，左下表示高度值 $h(u)$，右下表示超额流 $e(u)$，结点颜色的深度也表示结点的高度；边权表示 $c(u,v)-f(u,v)$，绿色的边表示满足 $h(u)=h(v)+1$ 的边 (u,v)（即残量网络的边 E_f）：



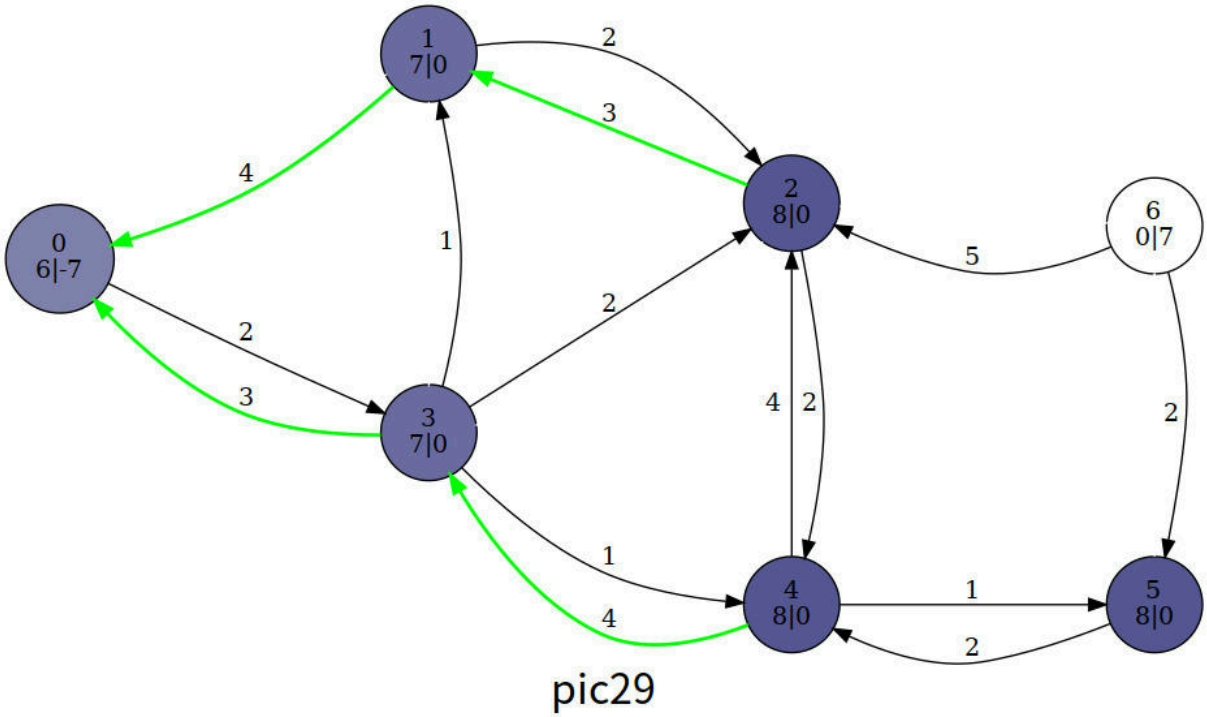
pic1

整个算法我们大致浏览一下过程，这里笔者使用的是一个暴力算法，即暴力扫描是否有溢出的结点，有就更新



pic1

最后的结果



pic29

可以发现，最后的超额流一部分回到了 s，且除了源点汇点，其他结点都没有溢出；这时的流函数 f 满足流守恒性，为最大流，流量即为 $e(t)$.

但是实际上论文[5]指出只处理高度小于 n 的溢出节点也能获得正确的最大流值，不过这样一来算法结束的时候预流还不满足流函数性质，不能知道每条边上真实的流量.

[5]Cherkassky B V, Goldberg A V. On implementing push-relabel method for the maximum flow problem[C]//International Conference on Integer Programming and Combinatorial Optimization. Springer, Berlin, Heidelberg, 1995: 157-171.

实现

核心代码

```
constexpr int N = 1e4 + 4, M = 1e5 + 5, INF = 0x3f3f3f3f;
int n, m, s, t, maxflow, tot;
int ht[N], ex[N];

void init() {  // 初始化
  for (int i = h[s]; i; i = e[i].nex) {
    const int &v = e[i].t;
    ex[v] = e[i].v, ex[s] -= ex[v], e[i ^ 1].v = e[i].v, e[i].v = 0;
  }
  ht[s] = n;
}

bool push(int ed) {
  const int &u = e[ed ^ 1].t, &v = e[ed].t;
  int flow = min(ex[u], e[ed].v);
  ex[u] -= flow, ex[v] += flow, e[ed].v -= flow, e[ed ^ 1].v += flow;
  return ex[u];  // 如果 u 仍溢出，返回 1
}

void relabel(int u) {
  ht[u] = INF;
  for (int i = h[u]; i; i = e[i].nex)
    if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
  ++ht[u];
}
```

HLPP 算法

最高标号预流推进算法（Highest Label Preflow Push）在上述通用的预流推送算法中，在每次选择结点时，都优先选择高度最高的溢出结点，其算法复杂度为 $O(n^2\sqrt{m})$.

过程

具体地说，HLPP 算法过程如下：

1. 初始化（基于预流推进算法）；
2. 选择溢出结点中高度最高的结点 u，并对它所有可以推送的边进行推送；
3. 如果 u 仍溢出，对它重贴标签，回到步骤 2；
4. 如果没有溢出的结点，算法结束.

一篇对最大流算法实际表现进行测试的论文[6]表明，实际上基于预流的算法，有相当一部分时间都花在了重贴标签这一步上. 以下介绍两种来自论文[7]的能显著减少重贴标签次数的优化.

BFS 优化

HLPP 的上界为 $O(n^2\sqrt{m})$，但在使用时卡得比较紧；我们可以在初始化高度的时候进行优化. 具体来说，我们初始化 $h(u)$ 为 u 到 t 的最短距离；特别地，$h(s) = n$.

在 BFS 的同时我们顺便检查图的连通性，排除无解的情况.

[6]Ahuja R K, Kodialam M, Mishra A K, et al. Computational investigations of maximum flow algorithms[J]. European Journal of Operational Research, 1997, 97(3): 509-542.

[7]Derigs U, Meier W. Implementing Goldberg's max-flow-algorithm—A computational investigation[J]. Zeitschrift für Operations Research, 1989, 33(6): 383-403.

GAP 优化

HLPP 推送的条件是 $h(u) = h(v) + 1$，而如果在算法的某一时刻，存在某个 k，使得 $h(u) = k$ 的结点个数为 0，那么对于 $h(u) > k$ 的结点就永远无法推送超额流到 t，因此只能送回 s，那么我们就在这时直接让他们的高度变成至少 $n + 1$，以尽快推送回 s，减少重贴标签的操作.

以下的实现采取论文[8]中的实现方法，使用 $N*2 - 1$ 个桶 `B`，其中 `B[i]` 中记录所有当前高度为 i 的溢出节点. 加入了以上提到的两种优化，并且只处理了高度小于 n 的溢出节点.

值得注意的是论文[9]中使用的桶是基于链表的栈，而 STL 中的 `stack` 默认的容器是 `deque`. 经过简单的测试发现 `vector`，`deque`，`list` 在本题的实际运行过程中效率区别不大.

实现

LuoguP4722【模板】最大流 加强版/预流推进

```cpp
#include <cstdio>
#include <cstring>
#include <queue>
#include <stack>
using namespace std;
constexpr int N = 1200, M = 120000, INF = 0x3f3f3f3f;
int n, m, s, t;

struct qxx {
  int nex, t;
  long long v;
};

qxx e[M * 2 + 1];
int h[N + 1], cnt = 1;

void add_path(int f, int t, long long v) {
  e[++cnt] = qxx{h[f], t, v}, h[f] = cnt;
}

void add_flow(int f, int t, long long v) {
  add_path(f, t, v);
  add_path(t, f, 0);
}

int ht[N + 1];          // 高度;
long long ex[N + 1];    // 超额流;
int gap[N];             // gap 优化. gap[i] 为高度为 i 的节点的数量
stack<int> B[N];        // 桶 B[i] 中记录所有 ht[v]==i 的v
int level = 0;          // 溢出节点的最高高度

int push(int u) {       // 尽可能通过能够推送的边推送超额流
  bool init = u == s;   // 是否在初始化
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
```

[8] Ahuja R K, Kodialam M, Mishra A K, et al. Computational investigations of maximum flow algorithms[J]. European Journal of Operational Research, 1997, 97(3): 509-542.

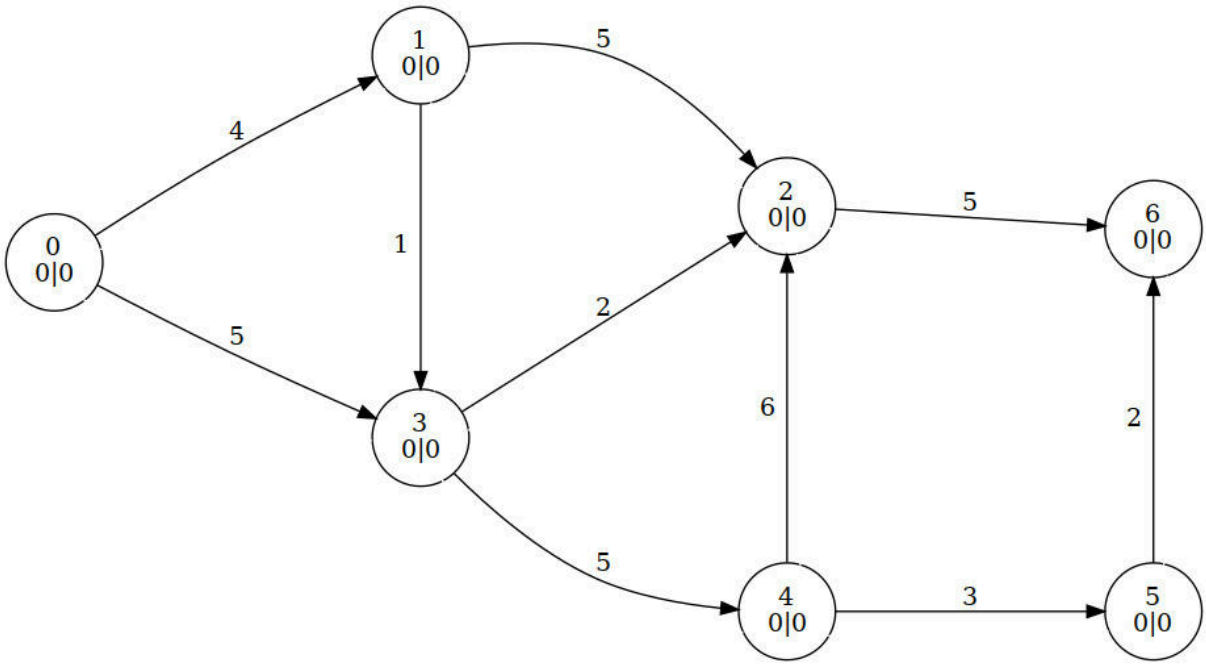
[9] Ahuja R K, Kodialam M, Mishra A K, et al. Computational investigations of maximum flow algorithms[J]. European Journal of Operational Research, 1997, 97(3): 509-542.

```
36      const long long &w = e[i].v;
37      // 初始化时不考虑高度差为1
38      if (!w || (init == false && ht[u] != ht[v] + 1) || ht[v] == INF)
continue;
39      long long k = init ? w : min(w, ex[u]);
40      // 取到剩余容量和超额流的最小值，初始化时可以使源的溢出量为负数.
41      if (v != s && v != t && !ex[v]) B[ht[v]].push(v), level = max(level,
ht[v]);
42      ex[u] -= k, ex[v] += k, e[i].v -= k, e[i ^ 1].v += k;  // push
43      if (!ex[u]) return 0;  // 如果已经推送完就返回
44    }
45    return 1;
46  }
47
48  void relabel(int u) {  // 重贴标签（高度）
49    ht[u] = INF;
50    for (int i = h[u]; i; i = e[i].nex)
51      if (e[i].v) ht[u] = min(ht[u], ht[e[i].t]);
52    if (++ht[u] < n) {  // 只处理高度小于 n 的节点
53      B[ht[u]].push(u);
54      level = max(level, ht[u]);
55      ++gap[ht[u]];  // 新的高度，更新 gap
56    }
57  }
58
59  bool bfs_init() {
60    memset(ht, 0x3f, sizeof(ht));
61    queue<int> q;
62    q.push(t), ht[t] = 0;
63    while (q.size()) {  // 反向 BFS，遇到没有访问过的结点就入队
64      int u = q.front();
65      q.pop();
66      for (int i = h[u]; i; i = e[i].nex) {
67        const int &v = e[i].t;
68        if (e[i ^ 1].v && ht[v] > ht[u] + 1) ht[v] = ht[u] + 1, q.push(v);
69      }
70    }
71    return ht[s] != INF;  // 如果图不连通，返回 0
72  }
73
74  // 选出当前高度最大的节点之一，如果已经没有溢出节点返回 0
75  int select() {
76    while (level > -1 && B[level].size() == 0) level--;
77    return level == -1 ? 0 : B[level].top();
78  }
79
80  long long hlpp() {            // 返回最大流
81    if (!bfs_init()) return 0;  // 图不连通
82    memset(gap, 0, sizeof(gap));
83    for (int i = 1; i <= n; i++)
84      if (ht[i] != INF) gap[ht[i]]++;  // 初始化 gap
85    ht[s] = n;
```

```
  push(s);  // 初始化预流
  int u;
  while ((u = select())) {
    B[level].pop();
    if (push(u)) {  // 仍然溢出
      if (!--gap[ht[u]])
        for (int i = 1; i <= n; i++)
          if (i != s && ht[i] > ht[u] && ht[i] < n + 1)
            ht[i] = n + 1;  // 这里重贴成 n+1 的节点都不是溢出节点
      relabel(u);
    }
  }
  return ex[t];
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  for (int i = 1, u, v, w; i <= m; i++) {
    scanf("%d%d%d", &u, &v, &w);
    add_flow(u, v, w);
  }
  printf("%lld", hlpp());
  return 0;
}
```

感受一下运行过程



pic1-origin

其中 pic13 到 pic14 执行了 Relabel(4)，并进行了 GAP 优化.

脚注

[1] http://pisces.ck.tp.edu.tw/~peng/index.php?action=showfile&file=f6cdf7ef750d7dc79c7d599b942acbaaee86a2e3e

[2] https://people.orie.cornell.edu/dpw/orie633/LectureNotes/lecture9.pdf

11.23.3 最小割

概念

割

对于一个网络流图 $G=(V,E)$，其割的定义为一种 **点的划分方式**：将所有的点划分为 S 和 $T=V-S$ 两个集合，其中源点 $s\in S$，汇点 $t\in T$.

割的容量

我们的定义割 (S,T) 的容量 $c(S,T)$ 表示所有从 S 到 T 的边的容量之和，即 $c(S,T)=\sum_{u\in S,v\in T}c(u,v)$. 当然我们也可以用 $c(s,t)$ 表示 $c(S,T)$.

最小割

最小割就是求得一个割 (S,T) 使得割的容量 $c(S,T)$ 最小.

证明

最大流最小割定理

参见 最大流→11.23.2@p2907 页面最大流最小割定理一节.

代码

最小割

通过 **最大流最小割定理**，我们可以直接得到如下代码：

参考代码

```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>

constexpr int N = 1e4 + 5, M = 2e5 + 5;
int n, m, s, t, tot = 1, lnk[N], ter[M], nxt[M], val[M], dep[N], cur[N];

void add(int u, int v, int w) {
  ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, val[tot] = w;
}

void addedge(int u, int v, int w) { add(u, v, w), add(v, u, 0); }

int bfs(int s, int t) {
  memset(dep, 0, sizeof(dep));
  memcpy(cur, lnk, sizeof(lnk));
  std::queue<int> q;
  q.push(s), dep[s] = 1;
  while (!q.empty()) {
    int u = q.front();
```

```
    q.pop();
    for (int i = lnk[u]; i; i = nxt[i]) {
      int v = ter[i];
      if (val[i] && !dep[v]) q.push(v), dep[v] = dep[u] + 1;
    }
  }
  return dep[t];
}

int dfs(int u, int t, int flow) {
  if (u == t) return flow;
  int ans = 0;
  for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
    int v = ter[i];
    if (val[i] && dep[v] == dep[u] + 1) {
      int x = dfs(v, t, std::min(val[i], flow - ans));
      if (x) val[i] -= x, val[i ^ 1] += x, ans += x;
    }
  }
  if (ans < flow) dep[u] = -1;
  return ans;
}

int dinic(int s, int t) {
  int ans = 0;
  while (bfs(s, t)) {
    int x;
    while ((x = dfs(s, t, 1 << 30))) ans += x;
  }
  return ans;
}

int main() {
  scanf("%d%d%d%d", &n, &m, &s, &t);
  while (m--) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    addedge(u, v, w);
  }
  printf("%d\n", dinic(s, t));
  return 0;
}
```

方案

我们可以通过从源点 s 开始 DFS，每次走残量大于 0 的边，找到所有 S 点集内的点.

```
void dfs(int u) {
  vis[u] = 1;
  for (int i = lnk[u]; i; i = nxt[i]) {
    int v = ter[i];
    if (!vis[v] && val[i]) dfs(v);
  }
```

```
7 }
```

割边数量

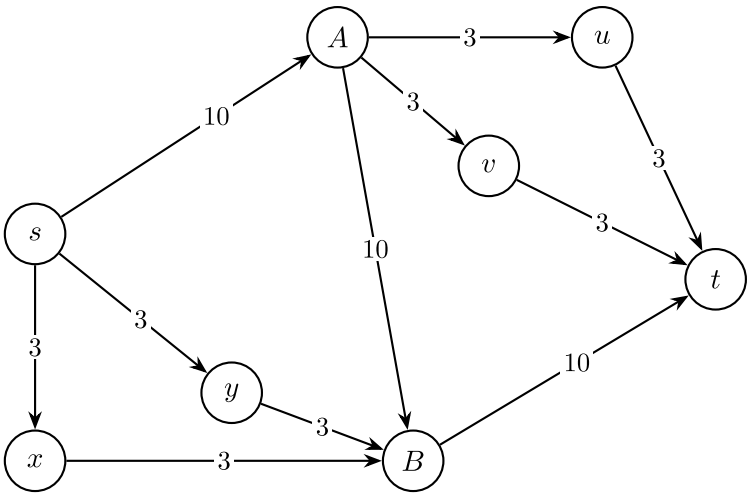
如果要求割边数量最小的割，直接把所有边的容量设成 1，求一遍最小割就好了．如果要求割边数量最小的最小割，那么有两种做法：

1. 将所有边的容量 $c(u,v)$ 更新为 $c(u,v)(|\ E\ |+1)+1$，在新图上求最小割，就得到原图的最少割边最小割．新图最小割的容量除以 $|\ E\ |+1$ 下取整即原图最小割，余数即割边数量．

   这相当于求以割的大小为第一关键字、割边数量为第二关键字的字典序最小割．因为系数 $|\ E\ |+1$ 严格大于总边数，而割边数量不超过总边数，所以无论割边数量如何，都无法抵消割的大小的影响，求出的一定是原图最小割；在原图最小割的前提下，割边数量最小的将成为新图最小割．

   注意，这一做法的正确性建立在所有边容量都是整数的前提下：只有这样，才能保证割边数量 $k\leq|\ E\ |<|\ E\ |+1$ 不会进位到高位．若容量是有理数，可先乘以公分母化为整数；若容量是任意实数，或整型会溢出，则直接将容量取为二元组 $(c(u,v),1)$，加法按分量进行，比较用字典序即可，此时只依赖容量加减与比较的增广类算法（如 Dinic 算法等）都可以原样运行．
2. 首先求出图的最大流，然后建新图：对 **残量图** 中的每条弧（即没有满流的边的正向弧，以及有流量的边的反向弧）连一条容量为 ∞ 的边，对每条满流的边连一条容量为 1 的边，重新跑一遍最小割就可以得到最小割边数量．

   新图最小割的割边一定不包含 ∞ 容量边，即没有残量弧从 S 指向 T．所以，新图最小割中，原图的前向边都满流，反向边都没有流量，这说明它的容量等于最大流，也一定是原图最小割．反过来，原图任一最小割的前向边都满流，它在新图中的容量就是它的割边数量．所以新图最小割就是原图的最少割边最小割．

> ⚠ **常见错误「只将未满流边容量设为 ∞」**
>
> 这种做法有一个常见的错误版本：求出最大流后，将满流边的容量设为 1、未满流边的容量设为 ∞，直接跑最小割．这种做法似乎符合直觉，但其实是错误的．反例如下图所示．
>
> 
>
>
> 如图，最大流为 16（即 s 的所有出边都满流），且最大流唯一，只有边 $A\to B$ 未满流．原图任一最小割的割边数量都是 3，但按错误版本重设容量后，新图最小割是 2，在 $S=\{s,x,y,B\}$ 处取得．这个割的容量是 20，并不是原图最小割：它的前向边 $s\to A$、$B\to t$ 虽然都满流，但有流量的边 $A\to B$ 从 T 侧指回了 S 侧，而错误版本没有排除这种情况．正确做法中，$A\to B$ 有流量，于是会连出容量为 ∞ 的反向边 $B\to A$，阻止了这个错误最小割的出现．

问题模型 1

有 n 个物品和两个集合 A, B，如果一个物品没有放入 A 集合会花费 a_i，没有放入 B 集合会花费 b_i；还有若干个形如 u_i, v_i, w_i 限制条件，表示如果 u_i 和 v_i 同时不在一个集合会花费 w_i．每个物品必须且只能属于一个集合，求最小的代价．

这是一个经典的 **二者选其一** 的最小割题目．我们对于每个集合设置源点 s 和汇点 t，第 i 个点由 s 连一条容量为 a_i 的边、向 t 连一条容量为 b_i 的边．对于限制条件 u, v, w，我们在 u, v 之间连容量为 w 的双向边．

注意到当源点和汇点不相连时，代表这些点都选择了其中一个集合．如果将连向 s 或 t 的边割开，表示不放在 A 或 B 集合，如果把物品之间的边割开，表示这两个物品不放在同一个集合．

最小割就是最小花费．

问题模型 2

最大权值闭合图，即给定一张有向图，每个点都有一个权值（可以为正或负或 0），你需要选择一个权值和最大的子图，使得子图中每个点的后继都在子图中．

做法：建立超级源点 s 和超级汇点 t，若节点 u 权值为正，则 s 向 u 连一条有向边，边权即为该点点权；若节点 u 权值为负，则由 u 向 t 连一条有向边，边权即为该点点权的相反数．原图上所有边权改为 ∞．跑网络最大流，将所有正权值之和减去最大流，即为答案．

几个小结论来证明：

1. 每一个符合条件的子图都对应流量网络中的一个割．因为每一个割将网络分为两部分，与 s 相连的那部分满足没有边指向另一部分，于是满足上述条件．这个命题是充要的．
2. 最小割所去除的边必须与 s 和 t 其中一者相连．因为否则边权是 ∞，不可能成为最小割．
3. 我们所选择的那部分子图，权值和 $=$ 所有正权值之和 $-$ 我们未选择的正权值点的权值之和 $+$ 我们选择的负权值点的权值之和．当我们不选择一个正权值点时，其与 s 的连边会被断开；当我们选择一个负权值点时，其与 t 的连边会被断开．断开的边的边权之和即为割的容量．于是上述式子转化为：权值和 $=$ 所有正权值之和 $-$ 割的容量．
4. 于是得出结论，最大权值和 $=$ 所有正权值之和 $-$ 最小割 $=$ 所有正权值之和 $-$ 最大流．

习题

- 「USACO 4.4」Pollutant Control[1]
- 「USACO 5.4」Telecowmunication[2]
- 「Luogu 1361」小 M 的作物[3]
- 「SHOI 2007」善意的投票[4]
- 太空飞行计划问题[5]

外部参考资料

[1]「USACO 4.4」Pollutant Control

[2]「USACO 5.4」Telecowmunication

[3]「Luogu 1361」小 M 的作物

[4]「SHOI 2007」善意的投票

[5] 太空飞行计划问题

11.23.4 费用流

在看这篇文章前请先看 网络流简介→11.23.1@p2906 这篇 wiki 的定义部分．

费用流

给定一个网络 $G = (V, E)$，每条边除了有容量限制 $c(u, v)$，还有一个单位流量的费用 $w(u, v)$．

当 (u,v) 的流量为 $f(u,v)$ 时，需要花费 $f(u,v)\times w(u,v)$ 的费用.

w 也满足斜对称性，即 $w(u,v)=-w(v,u)$.

则该网络中总花费最小的最大流称为 **最小费用最大流**，即在最大化 $\sum_{(s,v)\in E} f(s,v)$ 的前提下最小化 $\sum_{(u,v)\in E} f(u,v)\times w(u,v)$.

SSP 算法

SSP（Successive Shortest Path）算法是一个贪心的算法. 它的思路是每次寻找单位费用最小的增广路进行增广，直到图上不存在增广路为止.

如果图上存在单位费用为负的圈，SSP 算法无法正确求出该网络的最小费用最大流. 此时需要先使用消圈算法消去图上的负圈.

证明

我们考虑使用数学归纳法和反证法来证明 SSP 算法的正确性.

设流量为 i 的时候最小费用为 f_i. 我们假设最初的网络上 **没有负圈**，这种情况下 $f_0=0$.

假设用 SSP 算法求出的 f_i 是最小费用，我们在 f_i 的基础上，找到一条最短的增广路，从而求出 f_{i+1}. 这时 $f_{i+1}-f_i$ 是这条最短增广路的长度.

假设存在更小的 f_{i+1}，设它为 f'_{i+1}. 因为 $f_{i+1}-f_i$ 已经是最短增广路了，所以 $f'_{i+1}-f_i$ 一定对应一个经过 **至少一个负圈** 的增广路.

这时候矛盾就出现了：既然存在一条经过至少一个负圈的增广路，那么 f_i 就不是最小费用了. 因为只要给这个负圈添加流量，就可以在不增加 s 流出的流量的前提下，使 f_i 对应的费用更小.

综上，SSP 算法可以正确求出无负圈网络的最小费用最大流.

时间复杂度

如果使用 Bellman–Ford 算法→11.9.1@p2735 求解最短路，每次找增广路的时间复杂度为 $O(nm)$. 设该网络的最大流为 f，则最坏时间复杂度为 $O(nmf)$. 事实上，SSP 算法是 伪多项式时间→13.9@p3306 的.

为什么 SSP 算法是伪多项式时间的?

SSP 算法的时间复杂度有 $O(nmf)$ 的上界, 这是一个关于值域的多项式, 所以是伪多项式时间的.

可以构造 $m=n^2, f=2^{n/2}$ 的网络[10]使得 SSP 算法的时间复杂度达到 $O(n^3 2^{n/2})$，所以 SSP 算法不是多项式时间的.

实现

只需将 EK 算法或 Dinic 算法中找增广路的过程，替换为用最短路算法寻找单位费用最小的增广路即可.

基于 EK 算法的实现

```
struct qxx {
  int nex, t, v, c;
};

qxx e[M];
int h[N], cnt = 1;
```

[10]详细构造方法可以参考 min_25 的博客[1].

```

void add_path(int f, int t, int v, int c) {
  e[++cnt] = qxx{h[f], t, v, c}, h[f] = cnt;
}

void add_flow(int f, int t, int v, int c) {
  add_path(f, t, v, c);
  add_path(t, f, 0, -c);
}

int dis[N], pre[N], incf[N];
bool vis[N];

bool spfa() {
  memset(dis, 0x3f, sizeof(dis));
  queue<int> q;
  q.push(s), dis[s] = 0, incf[s] = INF, incf[t] = 0;
  while (q.size()) {
    int u = q.front();
    q.pop();
    vis[u] = false;
    for (int i = h[u]; i; i = e[i].nex) {
      const int &v = e[i].t, &w = e[i].v, &c = e[i].c;
      if (!w || dis[v] <= dis[u] + c) continue;
      dis[v] = dis[u] + c, incf[v] = min(w, incf[u]), pre[v] = i;
      if (!vis[v]) q.push(v), vis[v] = true;
    }
  }
  return incf[t];
}

int maxflow, mincost;

void update() {
  maxflow += incf[t];
  for (int u = t; u != s; u = e[pre[u] ^ 1].t) {
    e[pre[u]].v -= incf[t], e[pre[u] ^ 1].v += incf[t];
    mincost += incf[t] * e[pre[u]].c;
  }
}

// 调用：while(spfa())update();
```

基于 Dinic 算法的实现

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>

constexpr int N = 5e3 + 5, M = 1e5 + 5;
```

```cpp
constexpr int INF = 0x3f3f3f3f;
int n, m, tot = 1, lnk[N], cur[N], ter[M], nxt[M], cap[M], cost[M], dis[N], ret;
bool vis[N];

void add(int u, int v, int w, int c) {
  ter[++tot] = v, nxt[tot] = lnk[u], lnk[u] = tot, cap[tot] = w, cost[tot] = c;
}

void addedge(int u, int v, int w, int c) { add(u, v, w, c), add(v, u, 0, -c); }

bool spfa(int s, int t) {
  memset(dis, 0x3f, sizeof(dis));
  memcpy(cur, lnk, sizeof(lnk));
  std::queue<int> q;
  q.push(s), dis[s] = 0, vis[s] = true;
  while (!q.empty()) {
    int u = q.front();
    q.pop(), vis[u] = false;
    for (int i = lnk[u]; i; i = nxt[i]) {
      int v = ter[i];
      if (cap[i] && dis[v] > dis[u] + cost[i]) {
        dis[v] = dis[u] + cost[i];
        if (!vis[v]) q.push(v), vis[v] = true;
      }
    }
  }
  return dis[t] != INF;
}

int dfs(int u, int t, int flow) {
  if (u == t) return flow;
  vis[u] = true;
  int ans = 0;
  for (int &i = cur[u]; i && ans < flow; i = nxt[i]) {
    int v = ter[i];
    if (!vis[v] && cap[i] && dis[v] == dis[u] + cost[i]) {
      int x = dfs(v, t, std::min(cap[i], flow - ans));
      if (x) ret += x * cost[i], cap[i] -= x, cap[i ^ 1] += x, ans += x;
    }
  }
  vis[u] = false;
  return ans;
}

int mcmf(int s, int t) {
  int ans = 0;
  while (spfa(s, t)) {
    int x;
    while ((x = dfs(s, t, INF))) ans += x;
```

```
  }
  return ans;
}

int main() {
  int s, t;
  scanf("%d%d%d%d", &n, &m, &s, &t);
  while (m--) {
    int u, v, w, c;
    scanf("%d%d%d%d", &u, &v, &w, &c);
    addedge(u, v, w, c);
  }
  int ans = mcmf(s, t);
  printf("%d %d\n", ans, ret);
  return 0;
}
```

Primal-Dual 原始对偶算法

用 Bellman–Ford 求解最短路的时间复杂度为 $O(nm)$，无论在稀疏图上还是稠密图上都不及 Dijkstra 算法[11]．但网络上存在单位费用为负的边，因此无法直接使用 Dijkstra 算法．

Primal-Dual 原始对偶算法的思路与 Johnson 全源最短路径算法→11.9.1@p2735 类似，通过为每个点设置一个势能，将网络上所有边的费用（下面简称为边权）全部变为非负值，从而可以应用 Dijkstra 算法找出网络上单位费用最小的增广路．

首先跑一次最短路，求出源点到每个点的最短距离（也是该点的初始势能）h_i．接下来和 Johnson 算法一样，对于一条从 u 到 v，单位费用为 w 的边，将其边权重置为 $w + h_u - h_v$．

可以发现，这样设置势能后新网络上的最短路径和原网络上的最短路径一定对应．证明在介绍 Johnson 算法时已经给出，这里不再展开．

与常规的最短路问题不同的是，每次增广后图的形态会发生变化，这种情况下各点的势能需要更新．

如何更新呢? 先给出结论，设增广后从源点到 i 号点的最短距离为 d'_i（这里的距离为重置每条边边权后得到的距离），只需给 h_i 加上 d'_i 即可．下面我们证明，这样更新边权后，图上所有边的边权均为非负．

容易发现，在一轮增广后，由于一些 (i,j) 边在增广路上，残量网络上会相应多出一些 (j,i) 边，且一定会满足 $d'_i + (w(i,j) + h_i - h_j) = d'_j$（否则 (i,j) 边就不会在增广路上了）．稍作变形后可以得到 $w(j,i) + (h_j + d'_j) - (h_i + d'_i) = 0$．因此新增的边的边权非负．

而对于原有的边，在增广前，$d'_i + (w(i,j) + h_i - h_j) - d'_j \geq 0$，因此 $w(i,j) + (d'_i + h_i) - (d'_j + h_j) \geq 0$，即用 $h_i + d'_i$ 作为新势能并不会使 (i,j) 的边权变为负．

综上，增广后所有边的边权均非负，使用 Dijkstra 算法可以正确求出图上的最短路．

参考代码

```
#include <algorithm>
#include <cstdio>
#include <cstring>
```

[11]在稀疏图上使用堆优化可以做到 $O(m \log n)$ 的时间复杂度，而在稠密图上不使用堆优化，可以做到 $O(n^2)$ 的时间复杂度．

```cpp
#include <queue>
constexpr int INF = 0x3f3f3f3f;
using namespace std;

struct edge {
  int v, f, c, next;
} e[100005];

struct node {
  int v, e;
} p[10005];

struct mypair {
  int dis, id;

  bool operator<(const mypair& a) const { return dis > a.dis; }

  mypair(int d, int x) { dis = d, id = x; }
};

int head[5005], dis[5005], vis[5005], h[5005];
int n, m, s, t, cnt = 1, maxf, minc;

void addedge(int u, int v, int f, int c) {
  e[++cnt].v = v;
  e[cnt].f = f;
  e[cnt].c = c;
  e[cnt].next = head[u];
  head[u] = cnt;
}

bool dijkstra() {
  priority_queue<mypair> q;
  for (int i = 1; i <= n; i++) dis[i] = INF;
  memset(vis, 0, sizeof(vis));
  dis[s] = 0;
  q.push(mypair(0, s));
  while (!q.empty()) {
    int u = q.top().id;
    q.pop();
    if (vis[u]) continue;
    vis[u] = 1;
    for (int i = head[u]; i; i = e[i].next) {
      int v = e[i].v, nc = e[i].c + h[u] - h[v];
      if (e[i].f && dis[v] > dis[u] + nc) {
        dis[v] = dis[u] + nc;
        p[v].v = u;
        p[v].e = i;
        if (!vis[v]) q.push(mypair(dis[v], v));
      }
    }
  }
```

```
56   return dis[t] != INF;
57 }
58
59 void spfa() {
60   queue<int> q;
61   memset(h, 63, sizeof(h));
62   h[s] = 0, vis[s] = 1;
63   q.push(s);
64   while (!q.empty()) {
65     int u = q.front();
66     q.pop();
67     vis[u] = 0;
68     for (int i = head[u]; i; i = e[i].next) {
69       int v = e[i].v;
70       if (e[i].f && h[v] > h[u] + e[i].c) {
71         h[v] = h[u] + e[i].c;
72         if (!vis[v]) {
73           vis[v] = 1;
74           q.push(v);
75         }
76       }
77     }
78   }
79 }
80
81 int main() {
82   scanf("%d%d%d%d", &n, &m, &s, &t);
83   for (int i = 1; i <= m; i++) {
84     int u, v, f, c;
85     scanf("%d%d%d%d", &u, &v, &f, &c);
86     addedge(u, v, f, c);
87     addedge(v, u, 0, -c);
88   }
89   spfa();  // 先求出初始势能
90   while (dijkstra()) {
91     int minf = INF;
92     for (int i = 1; i <= n; i++) h[i] += dis[i];
93     for (int i = t; i != s; i = p[i].v) minf = min(minf, e[p[i].e].f);
94     for (int i = t; i != s; i = p[i].v) {
95       e[p[i].e].f -= minf;
96       e[p[i].e ^ 1].f += minf;
97     }
98     maxf += minf;
99     minc += minf * h[t];
100  }
101  printf("%d %d\n", maxf, minc);
102  return 0;
103 }
```

习题

- 「Luogu 3381」【模板】最小费用最大流[2]
- 「Luogu 4452」航班安排[3]

- 「SDOI 2009」晨跑[4]
- 「SCOI 2007」修车[5]
- 「HAOI 2010」订货[6]
- 「NOI 2012」美食节[7]

参考资料与注释

[1] min_25 的博客 
[2]「Luogu 3381」【模板】最小费用最大流 
[3]「Luogu 4452」航班安排 
[4]「SDOI 2009」晨跑 
[5]「SCOI 2007」修车 
[6]「HAOI 2010」订货 
[7]「NOI 2012」美食节 

11.23.5 上下界网络流

在阅读这篇文章之前请先阅读 最大流→11.23.2@p2907 并确保自己熟练掌握最大流算法.

概述

上下界网络流本质是给流量网络的每一条边设置了流量上界 $c(u,v)$ 和流量下界 $b(u,v)$. 也就是说，一种可行的流必须满足 $b(u,v)\le f(u,v)\le c(u,v)$. 同时必须满足除了源点和汇点之外的其余点流量平衡.

根据题目要求，我们可以使用上下界网络流解决不同问题.

无源汇上下界可行流

给定无源汇流量网络 G. 询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时每一个点流量平衡.

不妨假设每条边已经流了 $b(u,v)$ 的流量，设其为初始流. 同时我们在新图中加入 u 连向 v 的流量为 $c(u,v)-b(u,v)$ 的边. 考虑在新图上进行调整.

由于最大流需要满足初始流量平衡条件（最大流可以看成是下界为 0 的上下界最大流），但是构造出来的初始流很有可能不满足初始流量平衡. 假设一个点初始流入流量减初始流出流量为 M.

若 $M=0$，此时流量平衡，不需要附加边.

若 $M>0$，此时入流量过大，需要新建附加源点 S'，S' 向其连流量为 M 的附加边.

若 $M<0$，此时出流量过大，需要新建附加汇点 T'，其向 T' 连流量为 $-M$ 的附加边.

如果附加边满流，说明这一个点的流量平衡条件可以满足，否则这个点的流量平衡条件不满足.（因为原图加上附加流之后才会满足原图中的流量平衡.）

在建图完毕之后跑 S' 到 T' 的最大流，若 S' 连出去的边全部满流，则存在可行流，否则不存在.

例题

luogu P14578【模板】无源汇上下界可行流[1]

一个 n 个点、m 条有向边的有向图 G，每条边有流量下界 l_i 和流量上界 r_i.

构造一种方案使得每条边的流量 w_i 满足流量限制 $l_i\le w_i\le r_i$，且每个点流量平衡，即每个点流入流量等于流出流量. 或报告无解.

参考代码

```cpp
#include <iostream>
#include <queue>
using namespace std;
constexpr int MAXN = 3e4 + 10;
constexpr int INF = 1e9 + 10;
int n, m, s = 0, t;
int ecnt = 1, head[MAXN], dep[MAXN], now[MAXN], l[MAXN], d[MAXN], id[MAXN];

struct edge {
  int to, nxt, w;
} e[MAXN];

void add_edge(int u, int v, int w) {
  e[++ecnt] = {v, head[u], w};
  head[u] = ecnt;
}

void init() {
  for (int i = 0; i <= n + 1; i++) dep[i] = 0;
  for (int i = 0; i <= n + 1; i++) now[i] = head[i];
}

int bfs() {
  init();
  queue<int> q;
  q.push(s);
  dep[s] = 1;
  while (!q.empty()) {
    int x = q.front();
    q.pop();
    for (int i = head[x]; i; i = e[i].nxt) {
      int v = e[i].to;
      if (dep[v] || !e[i].w) continue;
      dep[v] = dep[x] + 1;
      q.push(v);
    }
  }
  return dep[t];
}

int dfs(int x, int flow) {
  if (x == t) return flow;
  int ret = 0;
  for (int i = now[x]; i; i = e[i].nxt) {
    now[x] = i;
    int v = e[i].to;
    if (dep[v] != dep[x] + 1 || !e[i].w) continue;
    int w = dfs(v, min(flow - ret, e[i].w));
    e[i].w -= w;
    e[i ^ 1].w += w;
    ret += w;
```

```
    if (ret == flow) return ret;
  }
  return ret;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);
  cin >> n >> m;
  s = 0, t = n + 1;
  for (int i = 1; i <= m; i++) {
    int u, v, r;
    cin >> u >> v >> l[i] >> r;
    add_edge(u, v, r - l[i]);
    add_edge(v, u, 0);
    id[i] = ecnt;
    d[v] += l[i];
    d[u] -= l[i];
  }
  long long sum = 0;
  for (int i = 1; i <= n; i++) {
    if (d[i] > 0) {
      add_edge(s, i, d[i]);
      add_edge(i, s, 0);
      sum += d[i];
    } else if (d[i] < 0) {
      add_edge(i, t, -d[i]);
      add_edge(t, i, 0);
    }
  }
  long long ans = 0;
  while (bfs()) ans += dfs(s, INF);
  if (ans < sum) {
    cout << "No\n";
    return 0;
  }
  cout << "Yes\n";
  for (int i = 1; i <= m; i++) cout << e[id[i]].w + l[i] << "\n";
  return 0;
}

```

有源汇上下界可行流

给定有源汇流量网络 G. 询问是否存在一种标定每条边流量的方式, 使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡.

假设源点为 S，汇点为 T.

则我们可以加入一条 T 到 S 的上界为 ∞，下界为 0 的边转化为无源汇上下界可行流问题.

若有解，则 S 到 T 的可行流流量等于 T 到 S 的附加边的流量.

有源汇上下界最大流

给定有源汇流量网络 G. 询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡. 如果存在，询问满足标定的最大流量.

我们找到网络上的任意一个可行流. 如果找不到解就可以直接结束.

否则我们考虑删去所有附加边之后的残量网络并且在网络上进行调整.

我们在残量网络上再跑一次 S 到 T 的最大流，将可行流流量和最大流流量相加即为答案.

> ⚠ **一个非常易错的问题**
>
> S 到 T 的最大流直接在跑完有源汇上下界可行的残量网络上跑.
>
> 千万不可以在原来的流量网络上跑.

有源汇上下界最小流

给定有源汇流量网络 G. 询问是否存在一种标定每条边流量的方式，使得每条边流量满足上下界同时除了源点和汇点每一个点流量平衡. 如果存在，询问满足标定的最小流量.

类似的，我们考虑将残量网络中不需要的流退掉.

我们找到网络上的任意一个可行流. 如果找不到解就可以直接结束.

否则我们考虑删去所有附加边之后的残量网络.

我们在残量网络上再跑一次 T 到 S 的最大流，将可行流流量减去最大流流量即为答案.

> **AHOI 2014 支线剧情**[2]
>
> 对于每条 x 到 y 花费 v 的剧情边设上界为 ∞, 下界为 1.
>
> 对于每个点，向 T 连边权 c, 上界 ∞, 下界为 1.
>
> S 点为 1 号节点.
>
> 跑一次 上下界带源汇最小费用可行流 即可.
>
> 因为最小费用可行流解法与最小可行流类似，这里不再展开.

外部参考资料

[1] luogu P14578【模板】无源汇上下界可行流 

[2] AHOI 2014 支线剧情 

11.23.6 Stoer–Wagner 算法

Authors: DanJoshua, opsiff, yzy-1, yingqi-z20

定义

由于取消了 **源汇点** 的定义，我们需要对 **割** 的概念进行重定义.

（其实是网络流部分有关割的定义与维基百科不符，只是由于一般接触到的割都是「有源汇的最小割问题」，因此这个概念也就约定俗成了.）

割

去掉其中所有边能使一张网络流图不再连通（即分成两个子图）的边集称为图的割.

即：在无向图 $G=(V,E)$ 中，设 C 为图 G 中一些弧的集合，若从 G 中删去 C 中的所有弧能使图 G 不是连通图，称 C 图 G 的一个割.

有源汇点的最小割问题

同 最小割→11.23.3@p2932 中的定义.

无源汇点的最小割问题

包含的弧的权和最小的割. 也称为全局最小割.

显然，直接跑网络流的复杂度是行不通的.

* * *

Stoer–Wagner 算法

引入

Stoer–Wagner 算法在 1995 年由 *Mechthild Stoer* 与 *Frank Wagner* 提出，是一种通过 **递归** 的方式来解决 **无向正权图** 上的全局最小割问题的算法.

性质

算法复杂度 $O(|V||E|+|V|^2\log|V|)$ 一般可近似看作 $O(|V|^3)$.

它的实现基于以下基本事实：设图 G 中有任意两点 S,T. 那么任意一个图 G 的割 C，或者有 S,T 在同一连通块中，或者有 C 是一个 $S-T$ 割.

过程

1. 在图 G 中任意指定两点 s,t，并且以这两点作为源汇点求出图 G 的 $S-T$ 最小割（定义为 *cut of phase*），更新当前答案.
2. 「合并」点 s,t，如果图 G 中 $|V|$ 大于 1，则回到第一步.
3. 输出所有 *cut of phase* 的最小值.

合并两点 s,t：删除 s,t 之间的连边 (s,t)，对于 $G\setminus\{s,t\}$ 中任意一点 k，删除 (t,k)，并将其边权 $d(t,k)$ 加到 $d(s,k)$ 上

解释：如果 s,t 在同一连通块，对于 $G\setminus\{s,t\}$ 中的一点 k，假如 $(k,s)\in C_{\min}$，那么 $(k,t)\in C_{\min}$ 也一定成立，否则因为 s,t 连通，k,t 连通，导致 s,k 在同一连通块，此时 $C=C_{\min}\setminus\{(t,k)\}$ 将比 $C_{\min}$ 更优. 反之亦然. 所以 s,t 可以看作同一点.

步骤 1 考虑了 s,t 不在同一连通块的情形，步骤 2 考虑了剩余的情况. 由于每次执行步骤 2 都会使 $|V|$ 减小 1，因此算法将在进行 $|V|-1$ 后结束.

S-T 最小割的求法

（显然不是网络流.）

假设进行若干次合并以后，当前图 $G'=(V',E')$，执行步骤 1.

我们构造一个集合 A，初始时令 $A=\emptyset$.

我们每次将 V' 中所有点中，满足 $i\notin A$，且权值函数 $w(A,i)$ 最大的节点加入集合 A，直到 $|A|=|V'|$.

其中权值函数的定义：

$w(A,i)=\sum_{j\in A}d(i,j)$

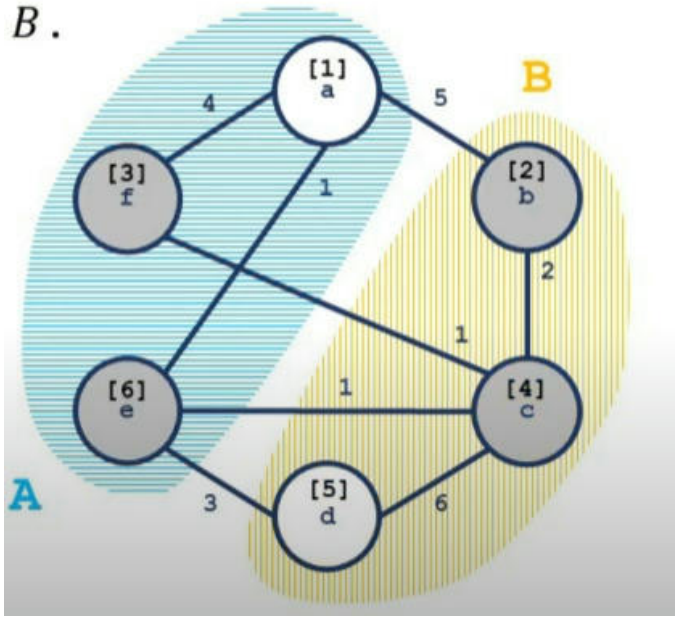
（若 $(i,j)\notin E'$，则 $d(i,j)=0$）.

容易知道所有点加入 A 的顺序是固定的，令 $\mathrm{ord}(i)$ 表示第 i 个加入 A 的点，$t=\mathrm{ord}(|V'|)$；$\mathrm{pos}(v)$ 表示 v 被加入 A 后 $|A|$ 的大小，即 v 被加入的顺序.

则对任意点 s，一个 s 到 t 的割即为 $w(t)$.

证明

定义一个点 v 被激活，当且仅当 v 在加入 A 中时，发现在 A 此时最后一个点 u 早于 v 加入集合，并且在图 $G'' = (V', E'/C)$ 中，u 与 v 不在同一连通块.



如图，蓝色区域和黄色区域为两个不同的连通块，方括号中的数字为加入 A 的顺序. 灰色节点为活跃节点，白色节点则不是活跃节点.

定义 $A_v = \{u \mid \text{pos}(u) < \text{pos}(v)\}$，也就是严格早于 v 加入 A 的点，令 E_v 为 E' 的诱导子图（点集为 $A_v \cup \{v\}$）的边集.（注意包含点 v.）

定义诱导割 C_v 为 $C \cap E_v$. $w(C_v) = \sum_{(i,j)\in C_v} d(i,j)$.

Lemma 1

对于任何被激活的点 v，$w(A_v, v) \le w(C_v)$.

证明：使用数学归纳法.

对于第一个被激活的点 v_0，由定义可知 $w(A_{v_0}, v_0) = w(C_{v_0})$.

对于之后两个被激活的点 u, v，假设 $\text{pos}(v) < \text{pos}(u)$，则有：

$w(A_u, u) = w(A_v, u) + w(A_u - A_v, u)$

又，已知：

$w(A_v, u) \le w(A_v, v)$ 并且 $w(A_v, v) \le w(C_v)$ 联立可得：

$w(A_u, u) \le w(C_v) + w(A_u - A_v, u)$

由于 $w(A_u - A_v, u)$ 对 $w(C_u)$ 有贡献而对 $w(C_v)$ 没有贡献，在所有边均为正权的情况下，可导出：

$w(A_u, u) \le w(C_u)$

由归纳法得证.

由于 $\text{pos}(s) < \text{pos}(t)$，并且 s, t 不在同一连通块，因此 t 会被激活，由此可以得出 $w(A_t, t) \le w(C_t) = w(C)$.

P5632【模板】Stoer–Wagner 算法[1]

```cpp
#include <cstring>
#include <iostream>
using namespace std;
constexpr int N = 601;
int fa[N], siz[N], edge[N][N];

int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }

int dist[N], vis[N], bin[N];
int n, m;

int contract(int &s, int &t) {  // Find s,t
  memset(dist, 0, sizeof(dist));
  memset(vis, false, sizeof(vis));
  int i, j, k, mincut, maxc;
  for (i = 1; i <= n; i++) {
    k = -1;
    maxc = -1;
    for (j = 1; j <= n; j++)
      if (!bin[j] && !vis[j] && dist[j] > maxc) {
        k = j;
        maxc = dist[j];
      }
    if (k == -1) return mincut;
    s = t;
    t = k;
    mincut = maxc;
    vis[k] = true;
    for (j = 1; j <= n; j++)
      if (!bin[j] && !vis[j]) dist[j] += edge[k][j];
  }
  return mincut;
}

constexpr int inf = 0x3f3f3f3f;

int Stoer_Wagner() {
  int mincut, i, j, s, t, ans;
  for (mincut = inf, i = 1; i < n; i++) {
    ans = contract(s, t);
    bin[t] = true;
    if (mincut > ans) mincut = ans;
    if (mincut == 0) return 0;
    for (j = 1; j <= n; j++)
      if (!bin[j]) edge[s][j] = (edge[j][s] += edge[j][t]);
  }
  return mincut;
}

int main() {
  ios::sync_with_stdio(false), cin.tie(nullptr);
```

```
  cin >> n >> m;
  if (m < n - 1) {
    cout << 0;
    return 0;
  }
  for (int i = 1; i <= n; ++i) fa[i] = i, siz[i] = 1;
  for (int i = 1, u, v, w; i <= m; ++i) {
    cin >> u >> v >> w;
    int fu = find(u), fv = find(v);
    if (fu != fv) {
      if (siz[fu] > siz[fv]) swap(fu, fv);
      fa[fu] = fv, siz[fv] += siz[fu];
    }
    edge[u][v] += w, edge[v][u] += w;
  }
  int fr = find(1);
  if (siz[fr] != n) {
    cout << 0;
    return 0;
  }
  cout << Stoer_Wagner();
  return 0;
}
```

* * *

复杂度分析与优化

contract 操作的复杂度为 $O(|E|+|V|\log|V|)$.

一共进行 $O(|V|)$ 次 *contract*，总复杂度为 $O(|E||V|+|V|^2\log|V|)$.

根据 最短路→11.9.1@p2735 的经验，算法瓶颈在于找到权值最大的点.

在一次 *contract* 中需要找 $|V|$ 次堆顶，并递增地修改 $|E|$ 次权值.

斐波那契堆 可以胜任 $O(\log|V|)$ 查找堆顶和 $O(1)$ 递增修改权值的工作，理论复杂度可以达到 $O(|E|+|V|\log|V|)$，但是由于斐波那契堆常数过大，码量高，实际应用价值偏低.

（实际测试中开 O2 还要卡评测波动才能过.）

外部参考资料

[1] P5632【模板】Stoer–Wagner 算法

11.24 图的匹配

11.24.1 图匹配

Authors: 310552025atNYCU, accelsao, Chrogeek, Enter-tainer, iamtwz, mcendu, Shen-Linwood, shuzhouliu, StudyingFather, t4rf9, Tiphereth-A, TrickEye, wlbksy, Xeonacid, yuhuoji, c-forrest, aaron20100919

引入

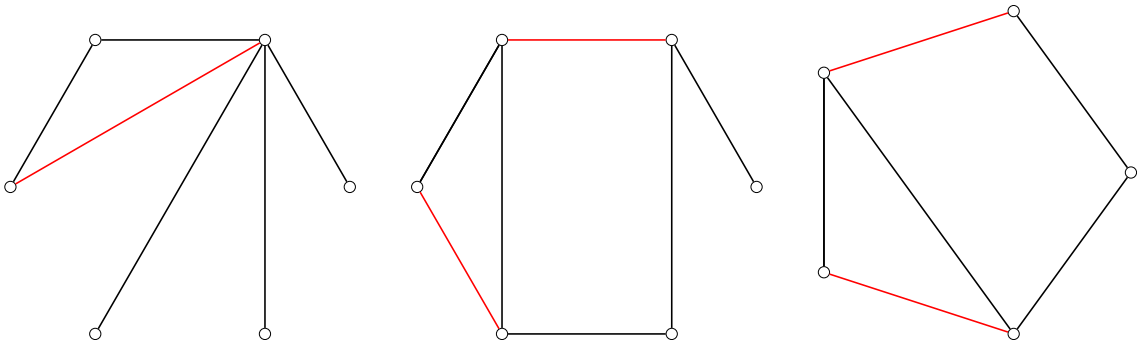
匹配 或是 **独立边集** 是一张图中不具有公共端点的边的集合．图匹配算法是信息学竞赛中常用的算法，大致可以分为最大匹配以及最大权匹配两类．因为 二分图→11.19@p2877 中的匹配等价于网络流问题，性质良好，相对容易处理，所以，此处将先从二分图开始介绍两类算法，再讨论一般图的算法．

图的匹配

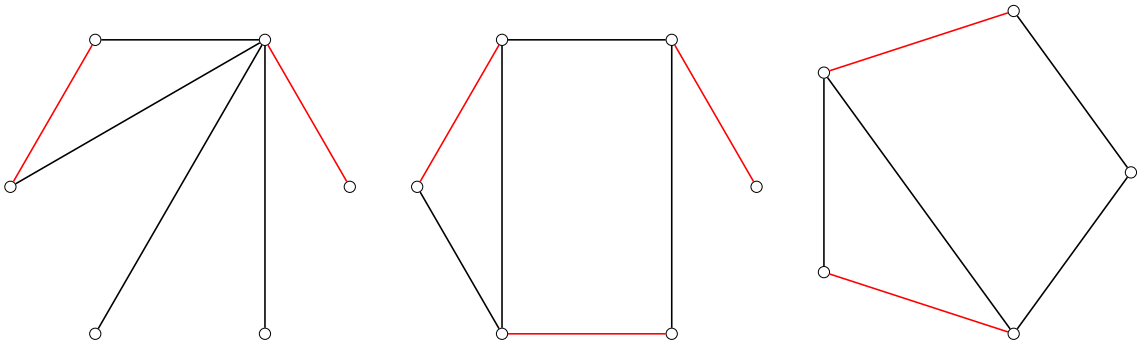
设 $G=(V,E)$ 是一张无向图，其中，V 是顶点集，E 是边集．如果一组边 $M\subseteq E$ 不包含自环，且两两之间没有公共顶点，那么，边集 M 就称为图 G 的一个 **匹配**（matching）或 **独立边集**（independent edge set）．一条边 $e\in E$，如果出现在匹配 M 中，就称为 **匹配边**，否则称为 **非匹配边**．相应地，一个顶点 $v\in V$，如果它是一条匹配边的一个端点，就称为 **匹配点**，否则称为 **未匹配点**．

匹配 M 的大小，就是它包含的边的数量．对于（加权）无向图中的匹配，常常会考虑如下概念：

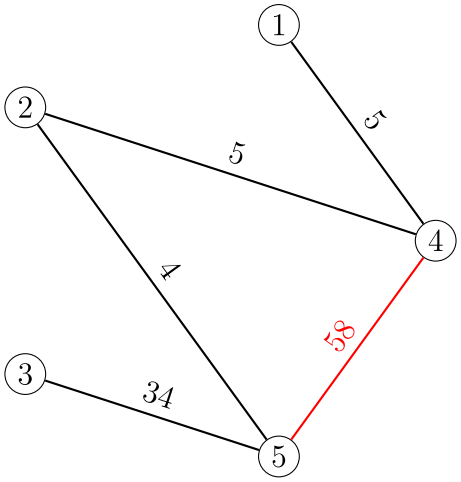
- **极大匹配**（maximal matching）：无法继续增加匹配边的匹配．极大匹配不一定是最大匹配．



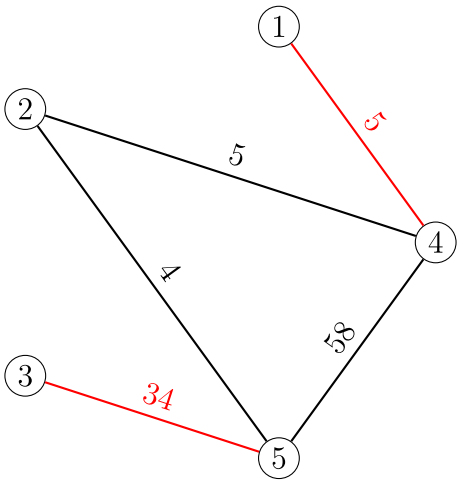
- **最大匹配**（maximum matching or maximum cardinality matching）：匹配边数最多的匹配．最大匹配可能有不止一个，但最大匹配的边数是确定的，而且不可能超过图中顶点数的一半．



- **最大权匹配**（maximum weight matching）：加权图中，边权和最大的匹配．



- **最大权最大匹配**（maximum weight maximum cardinality matching）：匹配数最多的前提下，边权和最大的匹配．即所有最大匹配中，边权和最大的匹配．



- **完美匹配**（perfect matching）：每个顶点都是匹配点的匹配．完美匹配一定是最大匹配．顶点数为偶数的完全图，必然存在完美匹配．
- **近完美匹配**（near-perfect matching）：有且只有一个未匹配点的匹配．这只能发生在图的顶点数是奇数时．近完美匹配也一定是最大匹配．顶点数为奇数的完全图，必然存在近完美匹配．

算法竞赛中涉及的图的匹配问题，主要指的是图的最大匹配或最大权匹配．

增广路

图匹配算法中，增广路是用于改进匹配的核心结构．

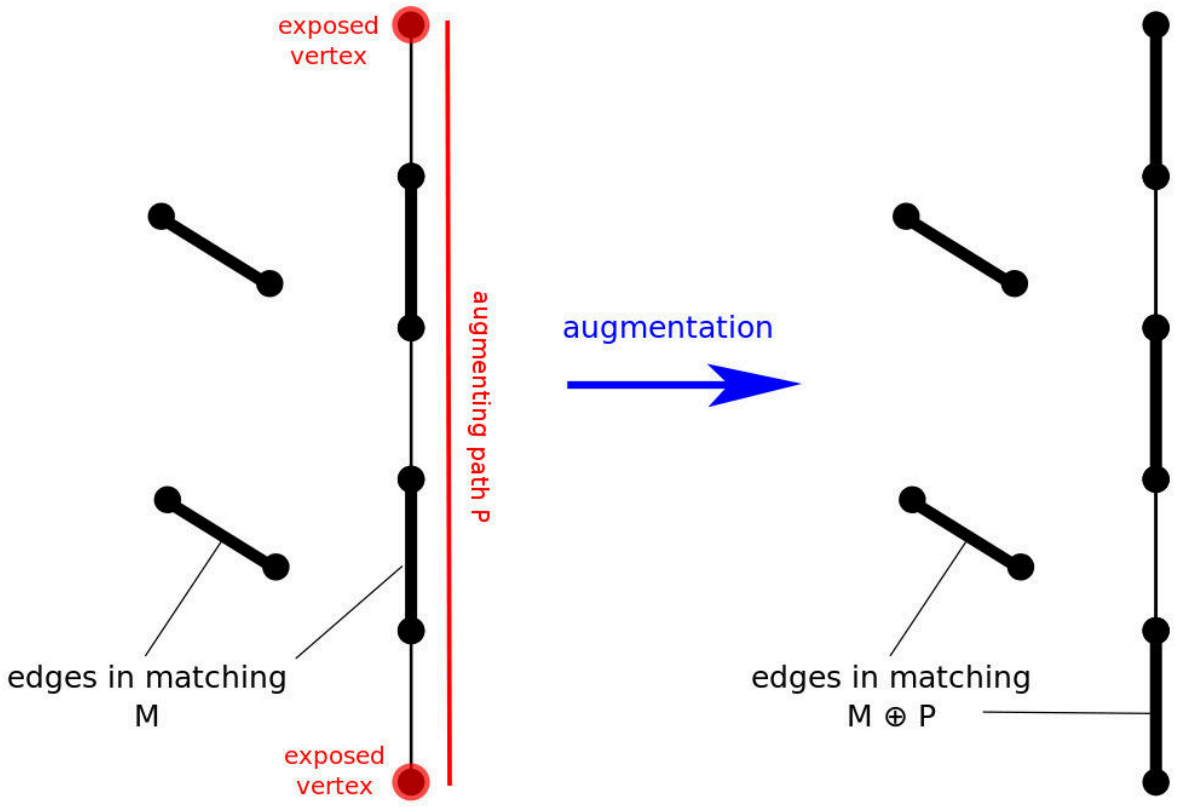
定义

对于图 $G=(V,E)$ 和它的一个匹配 M，可以定义如下两种（简单）路径：

- **交错路**（alternating path）是由匹配边与非匹配边交错而成的路径；
- **增广路**（augmenting path）是始于未匹配点且终于未匹配点的交错路．

因为增广路上非匹配边比匹配边数量多 1，所以增广路中边的数量一定是奇数．如果将增广路上的匹配边和非匹配边反转，那么它依然是交错路，而且匹配数量会增加 1．寻找增广路并反转它以增加匹配大小的过程，就称为 **增广**（augmentation）．用数学语言说，增广相当于将匹配 M 与增广路 P 取对称差，得到新的匹配 $M\oplus P$．

下图展示了在一次增广操作后，匹配数量由 2 增加为 3 的过程．



Berge 引理

Berge 引理说明，利用增广路改进匹配的方法是充分的．也就是说，当找不到增广路时，就说明已经得到了最大匹配．

Berge 引理

对于图 $G=(V,E)$ 和它的一个匹配 M，匹配 M 是最大匹配，当且仅当不存在相对于匹配 M 的增广路．

证明

前文已经说明，存在增广路 P 时，匹配 $M\oplus P$ 是比 M 更大的匹配，因此 M 一定不是最大匹配．

反过来，需要说明，如果存在比匹配 M 更大的匹配 M'，那么，一定存在相对于 M 的增广路 P．为此，考察对称差 $M\oplus M'$．图 $(V,M\oplus M')$ 中顶点的度数只能是 0、1 或 2；这样的图的连通分量必然是路径、环路、孤立点之一．而且，与度数为 2 的顶点相邻的两条边必定来自不同的匹配，所以，这些环路中来自 M 和 M' 的边的数量是一样的．因为 M' 比 M 大，所以存在至少一条路径中来自 M' 的边的数量多于 M；记该路径为 P．那么，P 的起点和终点都是 M 的未匹配点，且 P 是相对于 M 的交错路，所以 P 一定是相对于 M 的增广路．这就完成了证明．

由此定理可知我们求最大匹配的核心思路：

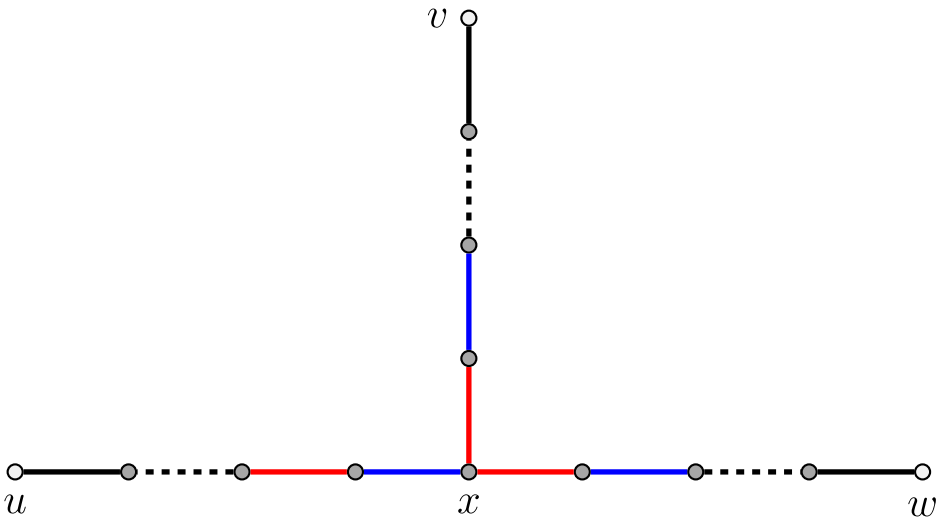
- 枚举所有未匹配点，找增广路径，直到找不到增广路径．

事实上，每次增广操作结束后，不需要重新遍历一次所有未匹配点．在整个求最大匹配的过程中，每个顶点只需要遍历一次就好．

证明

只需要说明，如果在枚举进行到顶点 v 时，不存在以 v 为起点的增广路，那么在若干轮增广之后，也不存在以 v 为起点的增广路．这就说明，即使增广引起了匹配的改变，也不需要再次检查之前已经枚举过的未匹配点．

假设不然．也就是说，假设 v 是已经枚举过的未匹配点，在某一轮沿着自 u 至 w 的增广路 P 增广后，新增了之前不存在的以 v 为起点的增广路 P'．那么，路径 P' 必然与 P 有公共边；否则，沿着 P 增广不会引起 P' 中的边的匹配状态改变，P' 也就不是因为此次增广而新增的增广路．



（图中黑色表示非匹配边，红色和蓝色表示不同的匹配状态）

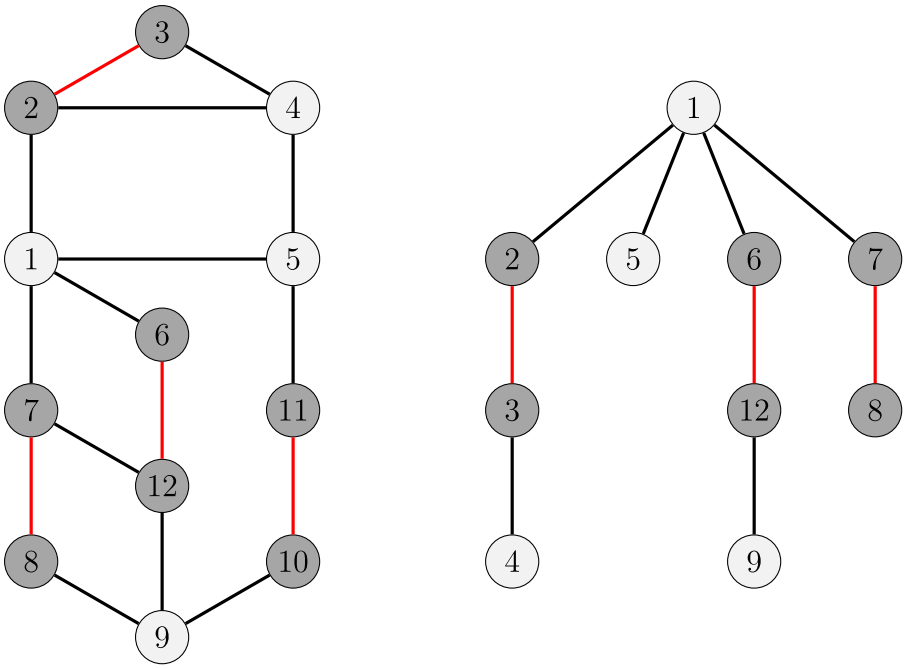
设 x 是自 v 出发，沿着路径 P' 最先到达的 P 中的顶点．因为在本次增广前，就存在自 v 到 x 的交错路，而此时并不存在以 v 为起点的增广路，所以 x 必然是匹配点，也就不能是顶点 u 或顶点 w 中的一个．因此，在增广路 P 上，与 x 相邻的边有两条，且它们的匹配状态相反．这说明，无论沿着以 v 为起点的交错路到达顶点 x 时，边的匹配状态如何，都可以沿着路径 P 将交错路延长至 u 或 w 中的一个．这说明，增广之前就已经存在以 v 为起点的增广路，与假设矛盾．

交错树

另一个与增广路紧密相关的概念是交错树．它是自未匹配点 r 出发进行 DFS 或 BFS 寻找增广路的过程中产生的树．

对于图 $G=(V,E)$ 和它的一个匹配 M，如果子图 $H\subseteq G$ 是以未匹配点 r 为根的树，且连接 r 和任意 $v\in V(H)$ 的路径都是交错路，那么，就称 H 是一个 **交错树**（alternating tree）．其中，树上深度为偶数的点称为偶点，树上深度为奇数的点称为奇点．

下图展示了自未匹配点 1 开始进行 BFS 可能得到的一个交错树．（图中，红边为匹配边，黑边为非匹配边；深色顶点为匹配点，浅色顶点为未匹配点．）



完美匹配的存在性

图匹配理论中，有两个重要的存在性定理，可以用于判定二分图或一般图中完美匹配是否存在.

Hall 定理

假设 $G=(X,Y,E)$ 是二分图，且 $|X|\leq|Y|$. 对于图 G 的一个匹配 M，如果 X 中的所有顶点都是匹配点，那么就称 M 是一个 X-**完美匹配**，有时也简称作（二分图 G 的）完美匹配. 这是二分图中可能达成的最大的匹配. Hall 定理提供了判断这种匹配是否存在的充要条件.

Hall 定理说明, 只要保证对于 X 的任何子集, Y 中都有足够多的顶点可以与它匹配, 就一定存在 X-完美匹配.

Hall 定理

假设 $G=(X,Y,E)$ 是二分图，且 $|X|\leq|Y|$. 对于任何 $W\subseteq X$，记 $N_G(W)$ 为图 G 中所有与 W 中的顶点相邻的顶点集合. 那么，X-完美匹配存在，当且仅当 $|W|\leq|N_G(W)|$ 对于所有 $W\subseteq X$ 都成立.

证明

条件显然是必要的. 假设 X-完美匹配 M 存在，那么 X 中的每个顶点都匹配到了 Y 中的不同顶点. 集合 $N_G(W)$ 至少包括了与 W 中顶点匹配的那些顶点，所以它的大小至少为 $|W|$.

条件也是充分的. 假设 X-完美匹配不存在, 那么一定有最大匹配 M 使得顶点 $v\in X$ 仍然是未匹配点. 设 Z 是所有可以通过由 v 出发的交错路到达的顶点集合，并设 $S=Z\cap X$，$T=Z\cap Y$. 集合 $S\setminus\{v\}$ 一定全部是匹配点，因为 G 是二分图，自 v 出发到达 X 中顶点的交错路长度必然是偶数，其最后一条边必然是匹配边；集合 T 也一定全部是匹配点，否则就存在增广路，根据 Berge 引理，这与 M 是最大匹配矛盾. 因为全部是匹配点，且匹配只能发生在 X 和 Y 之间，所以集合 $S\setminus\{v\}$ 和集合 T 中的顶点一一对应，也就是说，$|T|=|S|-1$. 同时，因为 T 中的顶点已经和 S 中顶点匹配，所以至少有 $T\subseteq N_G(S)$；但是，对于任意 $u\in N_G(S)$，设它与 S 中的 v' 相邻，那么，必然可以通过延长到达 v' 的交错路得到到达 u 的交错路，因而 $u\in T$：这说明 $T=N_G(S)$. 这些论证说明 $|N_G(S)|<|S|$，这与 Hall 定理所设条件矛盾. 这就说明了 X-完美匹配存在.

推论

所有 k-正则（$k \geq 1$）的二分图都有完美匹配.

证明

正则二分图中，所有顶点的度数都相同，设为 $k \geq 1$. 首先验证 Hall 条件成立，即对于任何 $W \subseteq X$，都有 $| N_G(W)| \geq | W |$. 因为与 W 中顶点相邻的边的数量是 $k | W |$，而集合 $N_G(W)$ 中每个顶点至多只能与其中 k 条边相邻，所以必然有 $k | W | \leq k | N_G(W)|$，也就是 $| W | \leq | N_G(W)|$. 特别地，有 $| X | \leq | Y |$；因为 X 和 Y 是对称的，所以有 $| X | = | Y |$. 这说明，正则二分图中，X-完美匹配也一定是完美匹配. 由于 Hall 定理保证了 X-完美匹配存在，那么完美匹配也一定存在.

Tutte 定理

Tutte 定理提供了判断一般图中是否存在完美匹配的充要条件. 这个条件源于一个直接的观察：顶点数量为奇数的图中一定不存在完美匹配.

Tutte 定理

图 $G = (V, E)$ 存在完美匹配，当且仅当对于任何 $U \subseteq V$，都有 $\text{odd}(G - U) \leq | U |$，其中，$G - U$ 表示从图 G 中删去 U 中的顶点以及与之相邻的边得到的子图，而 $\text{odd}(G - U)$ 表示子图 $G - U$ 中顶点数量为奇数的连通分量的数量.

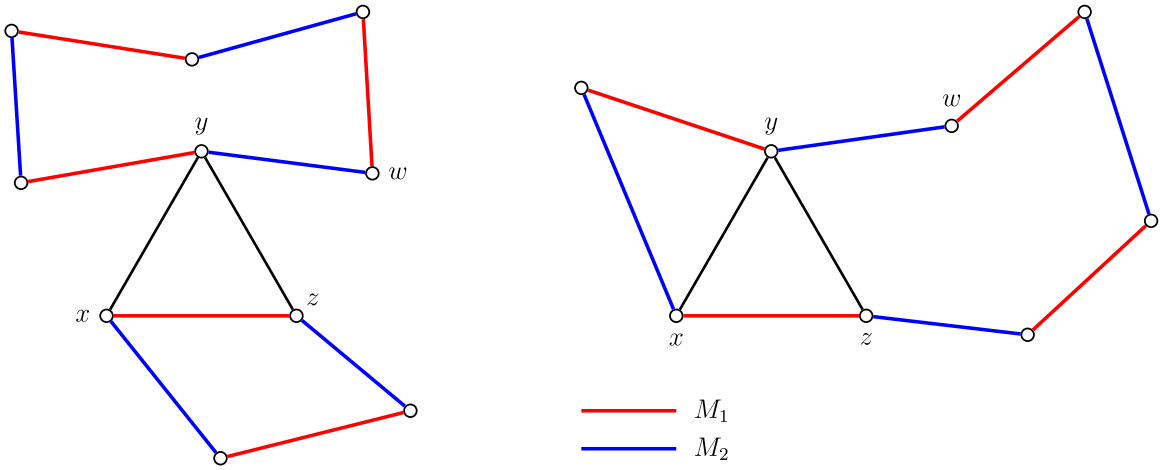
证明

只需要考虑简单图即可，因为重边和自环不影响 Tutte 条件和完美匹配的存在性.

条件的必要性相对容易. 假设存在完美匹配 M. 对于任意 $U \subseteq V$，从图 G 中删去 U 中的顶点之后，每一个顶点数量为奇数的连通分量都至少有一个无法与同一个连通分量顶点匹配的顶点，这些顶点只能寻求与 U 中的顶点匹配. 这样的匹配存在，至少要求 $\text{odd}(G - U) \leq | U |$. 这就是 Tutte 条件.

条件的充分性较为复杂. 假设 G 满足 Tutte 条件，但是没有完美匹配. 因为向 G 中添加任何边都会使得 Tutte 条件依然成立，所以不妨设 G 是一个极大的这样的图，也就是说，G 不存在完美匹配，但是，向 G 中添加任何尚不存在的边 e，都会使得 $G + e$ 存在完美匹配. 设 $U \subseteq V$ 是所有度数为 $| V | - 1$ 的顶点的集合. 可以证明，$G - U$ 的每一个连通分量都是一个完全图. 由此，可以构造出一个 G 的完美匹配：首先取每个 $G - U$ 的连通分量的最大匹配，这样只有连通分量顶点数量为奇数时，才会出现一个未匹配点；将这些未匹配点匹配到 U 中顶点；因为 G 的顶点数量为偶数（在 Tutte 条件中取 $U = \emptyset$），所以，U 中剩余的尚未匹配的顶点数量也是偶数，将它们两两配对即可. 这一矛盾就说明，不存在满足 Tutte 条件但是没有完美匹配的 G.

关键是要证明 $G - U$ 的每一个连通分量都是一个完全图. 假设不然. 不妨设顶点 x, y, z 属于这样的一个连通分量，且 $(x, y) \in E$，$(y, z) \in E$，$(x, z) \notin E$. 而且，因为 $y \notin U$，必然存在 $w \in V \setminus U$ 使得 $(y, w) \notin E$. 由于 G 的极大性，图 $G + (x, z)$ 和图 $G + (y, w)$ 中分别存在完美匹配 M_1 和 M_2. 考察它们的对称差 $M_1 \oplus M_2$. 因为图 $(V, M_1 \oplus M_2)$ 中所有顶点的度数要么是 0 要么是 2，所以 $M_1 \oplus M_2$ 实际上是若干个偶环的不交并，且每个偶环都是 M_1 和 M_2 中的匹配边交错组成的. 不妨设 $(x, z) \in M_1$ 且 $(y, w) \in M_2$，否则 M_1 或 M_2 已经是 G 的完美匹配；因此，这两条边都出现在 $M_1 \oplus M_2$ 中.



如图所示，可以分两种情形：

- (x,z) 和 (y,w) 位于不同的环路（如图左所示）：设 (y,w) 所在环路为 C，那么，边集 $M_2 \oplus C$ 就是图 G 的完美匹配；
- (x,z) 和 (y,w) 位于相同的环路（如图右所示）：由对称性，不妨设环路依次经过 x,y,w,z，因此可以取环路上从 y 经过 w 到达 z 的路径 P，记 $\{(y,z)\} \cup P$ 为环路 C，那么，边集 $M_2 \oplus C$ 同样是图 G 的完美匹配.

无论是哪种情形，都与 G 的选取矛盾．这个矛盾说明了 $G-U$ 的每一个连通分量都是一个完全图.

推论

无桥 3-正则图都有完美匹配.

证明

为了验证 Tutte 条件成立，任取 $U \subseteq V$，需要证明 $\mathrm{odd}(G-U) \leq |\ U\ |$．设 $G_1,\cdots,G_n$ 是 $G-U$ 的顶点数为奇数的所有连通分量．设 m_i 是连接 G_i 中顶点和 U 中顶点的边的数量．简单的计数可知

$$3\ |\ V(G_i)| = \sum_{v \in V(G_i)} d(v) = 2\ |\ E(G_i)| + m_i.$$

因此，m_i 必然是奇数．因为 G 中没有桥（即割边），所以 $m_i \geq 3$．这就说明

$$\mathrm{odd}(G-U) = n \leq \frac{1}{3}\sum_{i=1}^{n} m_i \leq \frac{1}{3}\sum_{v \in U} d(v) = |\ U\ |\ .$$

因此，Tutte 条件成立，图 G 必然有完美匹配.

常见算法

组合优化中的一个基本问题是求图的最大匹配和最大权匹配.

二分图最大匹配

详见 二分图最大匹配→11.24.2@p2959 页面.

在无权二分图中，可以使用 Kuhn 算法在 $O(|\ V\ |\ |\ E\ |)$ 时间内解决，也可以使用 Hopcroft–Karp 算法在 $O(|\ V\ |^{1/2}|\ E\ |)$ 时间内解决.

二分图最大权匹配

详见 二分图最大权匹配→11.24.3@p2983 页面.

在加权二分图中，可用 Hungarian 算法解决．如果在寻找最短路时使用 Bellman–Ford 算法，时间复杂度为 $O(|V||E|)$；如果使用 Dijkstra 算法配合 Fibonacci 堆，可在 $O(|V|^2 \log|V|+|V||E|)$ 时间内解决．

一般图最大匹配

详见 一般图最大匹配→11.24.4@p2996 页面．

无权一般图中，可以使用 Edmonds' blossom 算法在 $O(|V||^2|E|)$ 时间内解决．

一般图最大权匹配

详见 一般图最大权匹配→11.24.5@p3012 页面．

加权一般图中，可以使用 Edmonds' blossom 算法在 $O(|V||^2|E|)$ 时间内解决．

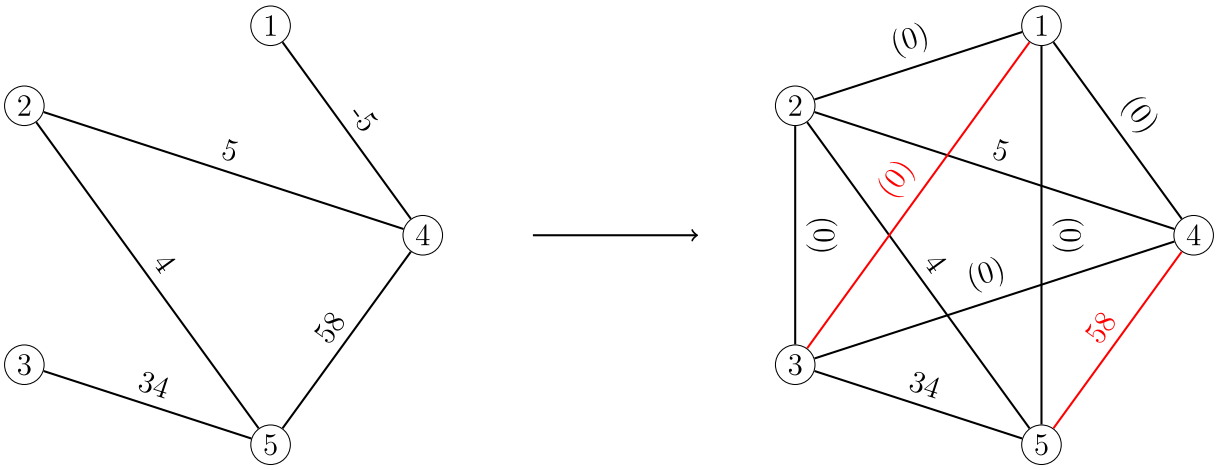
相关问题

最大（权）匹配与其它图论问题有着紧密的联系．本节仅讨论一般图，关于二分图的结论可以参考 二分图最大匹配→11.24.2@p2959 页面．

最大权最大匹配

最大权最大匹配问题和最大权匹配问题可以相互归约．它们之间一个很显著的区别是，最大权最大匹配中可能存在负权边，但是最大权匹配中不会存在负权边．

首先，最大权匹配问题可以归约为最大权最大匹配问题．为此，先将图 G 的所有负权边的权重设为 0；然后，通过连接若干权重为 0 的边，将图扩充为完全图 G'．注意到，边权非负的完全图中，最大权最大匹配和最大权匹配是一致的．所以，只需要计算 G' 的最大权最大匹配 M'，再删去 M' 中的所有零权边，得到的边集 M 就是图 G 的最大权匹配．[1]



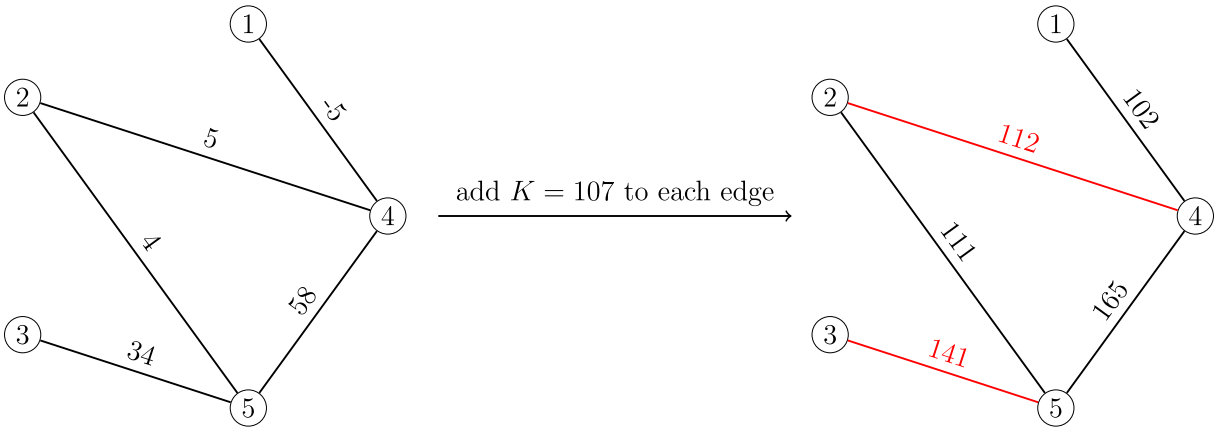
反过来，最大权最大匹配问题也可以归约为最大权匹配问题．只需要对图 G 所有边的边权都加一个足够大的正数 K，就可以保证得到的图 G' 的最大权匹配也一定是最大匹配，故而必然是最大权最大匹配．这是因为计算图 G' 的最大权匹配相当于在图 G 的所有匹配中最大化

$$K|M|+\sum_{e\in M} w(e).$$

[1]当然，这并不是唯一的归约方式．对于图 $G=(V,E)$，还可以将它的一份拷贝 $\tilde{G}=(\tilde{V},\tilde{E})$ 逐点地连接到原来的图上，并将所有相对于原图 G 新加的边（包括拷贝中的边）的权重设为 0，得到图 $G'=(V',E')$．换句话说，新图 G' 的顶点集是 $V\cup\tilde{V}$，而它的边集除了图 G 中的边之外，还将所有顶点 $v\in V$ 都与它的拷贝 $\tilde{v}\in\tilde{V}$ 用零权边连接，且对于所有边 $(u,v)\in E$，都将 $\tilde{u}$ 与 $\tilde{v}$ 用零权边连接．图 G 中的所有匹配 M 都对应着图 G' 中的一个权值和相同的完美匹配：只需要将 G 中所有未匹配点 v 都与它的拷贝 $\tilde{v}$ 匹配，而对于所有匹配边 (u,v)，都将 $\tilde{u}$ 与 $\tilde{v}$ 匹配．因此，图 G' 中的最大权最大匹配，也就是最大权完美匹配，限制到 E 上，就得到图 G 的最大权匹配．这样归约的好处是，如果图 G 是二分图或稀疏图，那么扩充得到的图 G' 也分别是二分图或稀疏图．

当 K 充分大时，多匹配一条边带来的增益 K，会超过后面一项的权值和的变化．因此，算法会首先尽可能地多匹配边，然后才会最大化匹配边的边权和．常数 K 的选择，只要保证它严格大于任意两个匹配的权值和之差即可．一个显然的选择是

$$K = \sum_{e \in E} |\, w(e)| + 1.$$

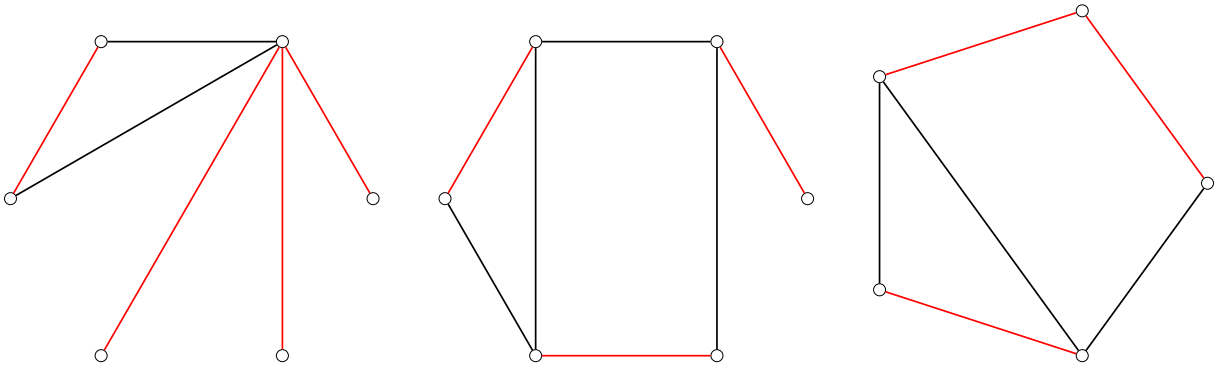


最小（权）边覆盖

另一个与最大（权）匹配紧密相关的问题是最小（权）边覆盖．边覆盖与匹配（又称边独立集）的关系，和点覆盖与独立集的关系相似．

图 $G = (V, E)$ 中的一组边 $C \subseteq E$，如果任意顶点 $v \in V$ 都是 C 中某条边的端点，那么，就称 C 是图 G 的一组 **边覆盖**（edge cover）．讨论边覆盖时，总是假设图 G 没有孤立点．

对于无权图，最小边覆盖问题几乎就是最大匹配问题．对于图 G 的任何最大匹配 M，只要为每个未匹配点都加入一条相连的边，就可以得到一个最小边覆盖 C．它们的大小满足简单的数量关系：$|\, M\,| + |\, C\,| = |\, V\,|$．下图是一些最小边覆盖的例子：



对于加权图，最小权边覆盖问题可以归约为一个 **最小权完美匹配** 问题．首先，将图 $G = (V, E)$ 拷贝一份得到 $\tilde{G} = (\tilde{V}, \tilde{E})$，边权与原图一致；然后，将每个顶点 $v \in V$ 都与它的拷贝 $\tilde{v} \in \tilde{V}$ 连接起来，边权为图 G 中与 v 关联的边的权值最小值．这样得到的图记作 $G' = (V', E')$．如果图 G 是二分图或稀疏图，那么图 G' 同样分别是二分图或稀疏图．而且，图 G 的最小权边覆盖问题，就归约为图 G' 的最小权完美匹配问题[2]：对于图 G' 的最小权完美匹配 M'，只要保留 E 中的边，再将所有匹配到的 $(v, \tilde{v})$ 替换成图 G 中与 v 关联的权值最小的边，就得到图 G 的最小权边覆盖．

[2]对于图 G' 的每个完美匹配 M'，都可以按照此处描述的方式得到一个图 G 的边覆盖 C，且后者的权值和为前者的一半；将这个构造过程反过来，对于图 G 的每个边覆盖 C，都可以构造出一个图 G' 的完美匹配 M'，且后者的权值和不超过前者的二倍．这就说明归约是成立的．

参考资料

1. Wikiwand - Matching (graph theory)[1]
2. Wikiwand - Blossom algorithm[2]
3. 2015 年《浅谈图的匹配算法及其应用》- 陈胤伯
4. 演算法笔记 - Matching[3]
5. the-tourist/algo[4]
6. Bill Yang's Blog - 带花树学习笔记[5]
7. 二分图的最大匹配、完美匹配和匈牙利算法[6]
8. Wikiwand - Hopcroft–Karp algorithm[7]
9. Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. Graph theory with applications. Vol. 290. London: Macmillan, 1976.

[1] Wikiwand - Matching (graph theory)

[2] Wikiwand - Blossom algorithm

[3] 演算法笔记 - Matching

[4] the-tourist/algo

[5] Bill Yang's Blog - 带花树学习笔记

[6] 二分图的最大匹配、完美匹配和匈牙利算法

[7] Wikiwand - Hopcroft–Karp algorithm

11.24.2 二分图最大匹配

> **Authors:** accelsao, thallium, Chrogeek, Enter-tainer, ksyx, StudyingFather, H-J-Granger, Henry-ZHR, countercurrent-time, william-song-shy, 5ab-juruo, XiaoQuQuSD, hhc0001, GCVillager

前置知识：二分图→11.19@p2877、图匹配→11.24.1@p2949

引入

本文讨论二分图 $G=(X,Y,E)$ 的最大匹配问题.

生活中一个典型的二分图匹配的例子是男女配对. 设有若干男生（X）和女生（Y），每个人只能配对一次，且允许配对的组合已由某个列表（E）限定. 此时，二分图最大匹配算法的任务就是在这些限制下，找到最多的配对对数，使尽可能多的人成功配对.

> **提示**
>
> 本文假设已知二分图顶点集 V 的一种划分（染色）方式：$V=X\cup Y$. 如果事先不清楚二分图的顶点集 V 的划分方法，可以通过 二分图的染色算法→11.19@p2877 在 $O(|V|+|E|)$ 时间内求出这样一个划分.

Kuhn 算法

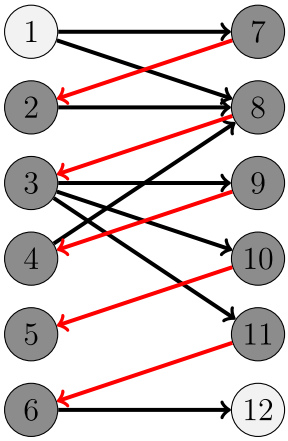
Kuhn 算法是 Berge 引理→11.24.1@p2949 的直接应用. 它也是 匈牙利算法→11.24.3@p2983 的一部分.

过程

为了求出最大匹配，算法依次枚举所有顶点，尝试求出从它出发的一条增广路，并进行增广. 因为增广路的长度总是奇数，所以在二分图中，它的端点必然分别位于左右两个部分. 这说明，只需要考虑从左部出发的增广路即可.

为了寻找增广路，可以依据当前的匹配 M 给二分图定向. 从左部的未匹配点出发的增广路（或者任何一条交错路）中，只能沿着非匹配边从左部点到达右部点，再沿着匹配边从右部点到达左部点. 因此，可以规定所有非匹配边都指向右部点，而所有匹配边都指向左部点. 寻找增广路的问题就转换为从某个未

匹配的左部点出发，在有向图中寻找一条简单路径通向某个未匹配的右部点．这一问题很容易通过 DFS→11.4@p2612 或 BFS→11.5@p2616 在 $O(|E|)$ 时间内解决．



（图中，深色点为匹配点，浅色点为未匹配点，红边为匹配边，黑边为非匹配边，箭头表示当前匹配对应的定向．由图可知，路径 $1 \to 8 \to 3 \to 11 \to 6 \to 12$ 是相对于当前匹配的一条增广路．）

算法开始时，所有边都指向右部点．每次寻找到增广路后，都需要沿着增广路将经过的所有边都反向，以表示它们的匹配状态已经反转．算法结束时，所有指向左部点的边就是匹配边．

因为至多只需要枚举 $O(|V|)$ 个左部点 各一次→11.24.1@p2949，所以，算法总的时间复杂度为 $O(|V||E|)$．

优化

有一些简单的技巧，可以优化 Kuhn 算法的常数：

1. Kuhn 算法基于 Berge 引理，而后者并不要求事先给出二分图的左右部分．因此，即使在左右两部分未明确划分的情况下，Kuhn 算法也能正确运行，只要图本身是二分图．但是，先将二分图染色，确定好它的左部和右部，往往效率更高．
2. 因为上文描述的 Kuhn 算法的时间复杂度实际上是 $O(|X||E|)$ 的，所以，可以选取二分图的两个部分中较小的那个作为左部 X．
3. 在寻找增广路时，用于避免重复查找的标记无需每次 DFS 都清空．可以在清空标记前，尝试为所有未匹配的左部点都寻找增广路．在一轮这样的查找中，所有边至多访问一次，复杂度仍然是 $O(|E|)$ 的；但是，在一轮查找中，可能找到多条增广路，因此总的轮数 k 不会超过 $|M|+1$，其中，M 为最大匹配．相应地，算法整体复杂度降低到了 $O(k|E|)$．
4. 寻找增广路时，优先考虑未匹配的右部点，因为这意味着更短的增广路．
5. 因为 Berge 引理并不要求初始匹配为空，所以，Kuhn 算法开始时，可以随机地选取一些互不相交的边作为初始匹配，以减少后续搜索的次数．如果已经应用优化 3，本优化可以忽略．

虽然最差复杂度仍然是 $O(|V||E|)$，但是，充分优化的 Kuhn 算法效率并不差．不过，为了避免个别数据将它卡到最差复杂度，在匹配前需要首先随机打乱边或顶点的顺序．

参考实现

实现时，不需要真正维护定向，只需要为每个顶点都维护与它相匹配的顶点即可．

模板题 Library Checker - Matching on Bipartite Graph[1]

```cpp
#include <algorithm>
#include <ctime>
#include <iostream>
```

```cpp
#include <random>
#include <tuple>
#include <vector>

std::mt19937_64 rng(
    static_cast<std::mt19937_64::result_type>(std::time(nullptr)));

struct BipartiteGraph {
  int n1, n2;                          // number of vertices in X and Y, resp.
  std::vector<std::vector<int>> g;  // edges from X to Y
  std::vector<int> ma, mb;  // matches from X to Y and from Y to X, resp.
  std::vector<bool> vis;    // visiting marks for DFS.

  BipartiteGraph(int n1, int n2)
      : n1(n1), n2(n2), g(n1), ma(n1, -1), mb(n2, -1) {}

  // Add an edge from u in X to v in Y.
  void add_edge(int u, int v) { g[u].emplace_back(v); }

  // Find an augmenting path starting at u.
  bool dfs(int u) {
    vis[u] = true;
    // Heuristic: find unsaturated vertices whenever possible.
    for (int v : g[u]) {
      if (mb[v] == -1) {
        ma[u] = v;
        mb[v] = u;
        return true;
      }
    }
    for (int v : g[u]) {
      if (!vis[mb[v]] && dfs(mb[v])) {
        ma[u] = v;
        mb[v] = u;
        return true;
      }
    }
    return false;
  }

  // Kuhn's maximum matching algorithm.
  std::vector<std::pair<int, int>> kuhn_maximum_matching() {
    // Randomly shuffle the edges.
    for (int u = 0; u < n1; ++u) {
      std::shuffle(g[u].begin(), g[u].end(), rng);
    }
    // Find a maximal set of vertex-disjoint augmenting paths in each round.
    while (true) {
      bool succ = false;
      vis.assign(n1, false);
      for (int u = 0; u < n1; ++u) {
        succ |= ma[u] == -1 && dfs(u);
```

```
      }
      if (!succ) break;
    }
    // Collect the matched pairs.
    std::vector<std::pair<int, int>> matches;
    matches.reserve(n1);
    for (int u = 0; u < n1; ++u) {
      if (ma[u] != -1) {
        matches.emplace_back(u, ma[u]);
      }
    }
    return matches;
  }
};

int main() {
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n1, n2, m;
  std::cin >> n1 >> n2 >> m;
  BipartiteGraph gr(n1, n2);
  for (int i = 0; i < m; ++i) {
    int u, v;
    std::cin >> u >> v;
    gr.add_edge(u, v);
  }
  auto res = gr.kuhn_maximum_matching();
  std::cout << res.size() << '\n';
  for (int i = 0; i < res.size(); ++i) {
    std::cout << res[i].first << ' ' << res[i].second << '\n';
  }
  return 0;
}

```

Hopcroft–Karp 算法

Hopcroft–Karp 算法进一步优化了 Kuhn 算法查找增广路的过程，将总的轮数降低到了 $O(|V|^{1/2})$，从而获得了 $O(|V|^{1/2}|E|)$ 的时间复杂度．这一算法实际上是 Dinic 算法→11.23.2@p2907 的一种特殊情形.

过程

算法依然是在寻找增广路，但是为了在更少的轮次内完成匹配，算法在每一轮中都采取了如下策略：

1. 将匹配边定向为指向左部点，非匹配边定向为指向右部点.
2. 从所有未匹配的左部点出发，在有向图上进行 BFS，记录每个访问到的顶点所在的层数 $d(v)$，直到某一层出现未匹配的右部点为止．若 BFS 结束仍未找到未匹配的右部点，说明当前匹配已是最大匹配.
3. 依次从每个未匹配的左部点出发进行 DFS，寻找增广路并进行增广．DFS 时沿着满足层数连续且严格递增（即 $d(v')=d(v)+1$）的边向前扩展，且只访问尚未在本轮 DFS 中访问过的顶点．特别地，DFS 中不会访问到前一步 BFS 中尚未访问到的顶点.

用网络流的术语说，步骤 2 是构造了一个层次图，而步骤 3 则是找到了层次图上的阻塞流．所谓层次图，是指它的每条边都必然从这一层指向下一层；而所谓阻塞流，在当前语境下，就是指一组极大的、两

两之间没有公共顶点的增广路．步骤 3 得到的这一组增广路必然是极大的：假设不然，存在一条新的增广路，那么在当初枚举到它的起点时，就应当已经找到这样一条路．

比起前文的 Kuhn 算法，Hopcroft–Karp 算法最关键的改变就是在求阻塞流之前，添加了求层次图这一步骤．基于层次图进行 DFS，相当于限制了算法总是沿着最短路径到达各个顶点．这样做的好处是，在算法的不同轮次间，算法找到的增广路的长度是严格递增的．而且，可以证明的是，直到求出最大匹配为止，增广路的长度至多增加 $3\mid M\mid^{1/2}$ 次，其中，$\mid M\mid$ 为最大匹配的大小．所以，这就将总的增广的轮次控制在 $O(\mid M\mid^{1/2})$，进而得到 $O(\mid M\mid^{1/2}\mid E\mid)$ 的时间复杂度．由于 $2\mid M\mid\leq\mid V\mid$，所以，时间复杂度也可以写成更宽松的上界 $O(\mid V\mid^{1/2}\mid E\mid)$．

证明

首先说明，在算法的不同轮次之间，算法找到的增广路的长度是严格递增的．

假设当前轮次的 BFS 中，顶点向前延伸了 ℓ 层，那么，因为 BFS 中找到的未匹配的右部点都位于同一层，所以本轮 DFS 能够找到的所有增广路的长度就都是 ℓ．需要证明的是，沿着本轮找到的这组增广路 $\{P_i\}$ 增广结束后，重新定向得到的有向图中，将不再存在长度不超过 ℓ 的增广路．

实际上，如果 P 是相对于 M 的最短的增广路，而 P' 是相对于 $M\oplus P$ 的增广路，都有 $\mid P'\mid\geq\mid P\mid+2\mid P\cap P'\mid$ 成立．这是因为 $N=(M\oplus P)\oplus P'$ 相对于 M 增广了两次，故而类似 Berge 引理的证明→11.24.1@p2949，可以说明对称差 $M\oplus N=P\oplus P'$ 中至少包含两条相对于 M 的不交的增广路 P_1 和 P_2．由于 P 的最短性，有

$$2\mid P\mid\leq\mid P_1\mid+\mid P_2\mid\leq\mid P\oplus P'\mid=\mid P\mid+\mid P'\mid-2\mid P\cap P'\mid.$$

这就说明 $\mid P'\mid\geq\mid P\mid+2\mid P\cap P'\mid$．因此，如果添加完增广路 $\{P_i\}$ 之后，新的增广路 P' 仍然和它们一样长，就必然与它们两两不交，这与 $\{P_i\}$ 的极大性相矛盾．这一矛盾说明，增广完阻塞流后，新的增广路一定严格更长．

最后，需要说明，增广路的长度至多增加 $3\mid M\mid^{1/2}$ 次．

记 $p=\lfloor\mid M\mid^{1/2}\rfloor$．在前 p 个轮次结束后，剩下的增广路的长度至少是 $\mid M\mid^{1/2}$．设当前匹配为 M_p，那么，与前文情形类似，可以说明图 $(V,M\oplus M_p)$ 中有 $\mid M\mid-\mid M_p\mid$ 条两两之间没有公共顶点的相对于 M_p 的增广路．每条增广路至少用到 $\mid M\mid^{1/2}/2$ 条 M 中的匹配边，因此，这些增广路的总数不会超过 $2\mid M\mid^{1/2}$，也就是说，$\mid M\mid-\mid M_p\mid\leq 2\mid M\mid^{1/2}$．这说明从 M_p 开始，至多只能再增广 $2\mid M\mid^{1/2}$ 次，这同样意味着算法至多再进行 $2\mid M\mid^{1/2}$ 轮增广．所以，增广路的长度总共至多增加 $3\mid M\mid^{1/2}$ 次．

这仅仅是对 Hopcroft–Karp 算法的最差复杂度的估计．实际上，在随机图中，Hopcroft–Karp 算法的时间复杂度有很大概率是 $O(\mid E\mid\log\mid V\mid)$ 的[3].

优化

在建立层次图时，和一般的 Dinic 算法一样，Hopcroft–Karp 算法在到达未匹配的右部点时就终止．但是，仅仅就二分图匹配问题来说，这样做是没有必要的．而且，因为 BFS 过早终止，限制了后续 DFS 的范围，会导致每轮找到的增广路数目有限，从而拖慢整体匹配效率．在有些图上，它的效率甚至不如经过优化的 Kuhn 算法．所以，一个简单的改进是，不提前终止 BFS，而是为所有可以到达的顶点建立层次图．

[3]Bast, Holger; Mehlhorn, Kurt; Schäfer, Guido; Tamaki, Hisao (2006), "Matching algorithms are fast in sparse random graphs", Theory of Computing Systems, 39 (1): 3–14.

正确性证明

在优化后的算法中，阻塞流中增广路的长度将不再相同，因此，前文中关于复杂度的证明也不再成立．但是，可以说明的是，通过为算法的每个轮次建立辅助图，同样可以建立最短增广路长度严格递增的结论，进而保证最差复杂度依然是正确的．

设二分图为 $G=(X,Y,E)$，当前的匹配为 M．设通过 BFS 可以访问到的未匹配的右部点的集合为 $W\subseteq Y$，且到达 $y\in W$ 的最短的增广路的长度为 $d(y)$．记 $d_{\max}=\max_{y\in W}d(y)$，那么，对于每个 $y\in W$，都可以新建一条从 y 开始且长度为 $d_{\max}-d(y)$ 的链，并将新建的顶点依次标记为左部点和右部点，将新建的边依次标记为匹配边和非匹配边．设这样得到的图为 $G'=(X',Y',E')$，匹配为 M'，最短增广路的长度为 $d_{\max}$．那么，图 G 中沿着层次图能够找到的相对于 M 的增广路——也就是那些到达相应顶点的最短的增广路——与图 G' 中相对于 M' 的全局最短的增广路之间存在双射．因此，在图 G 的层次图中找到阻塞流并进行增广，就相当于在图 G' 的层次图中找到阻塞流并进行增广．按照前文的证明，增广后，图 G' 中将不再存在长度为 $d_{\max}$ 的增广路．所以，图 G 中也不再存在长度为 $d_{\min}=\min_{y\in W}d(y)$ 的增广路：因为这样的增广路通过新延长的交错路，必然对应着一条图 G' 中长度为 $d_{\max}$ 的增广路．这样就再次得到了算法的不同轮次间，最短增广路的长度严格递增的结论．因此，整体复杂度也仍然是 $O(|M|^{1/2}|E|)$ 的．

参考实现

模板题 Library Checker - Matching on Bipartite Graph[2]

```cpp
#include <algorithm>
#include <iostream>
#include <queue>
#include <tuple>
#include <vector>

struct BipartiteGraph {
  int n1, n2;                          // number of vertices in X and Y, resp.
  std::vector<std::vector<int>> g;  // edges from X to Y
  std::vector<int> ma, mb;  // matches from X to Y and from Y to X, resp.
  std::vector<int> dist;    // distance from unsaturated vertices in X.

  BipartiteGraph(int n1, int n2)
      : n1(n1), n2(n2), g(n1), ma(n1, -1), mb(n2, -1) {}

  // Add an edge from u in X to v in Y.
  void add_edge(int u, int v) { g[u].emplace_back(v); }

  // Build the level graph.
  bool bfs() {
    dist.assign(n1, -1);
    std::queue<int> q;
    for (int u = 0; u < n1; ++u) {
      if (ma[u] == -1) {
        dist[u] = 0;
        q.emplace(u);
      }
    }
    // Build the level graph for all reachable vertices.
```

```
    bool succ = false;
    while (!q.empty()) {
      int u = q.front();
      q.pop();
      for (int v : g[u]) {
        if (mb[v] == -1) {
          succ = true;
        } else if (dist[mb[v]] == -1) {
          dist[mb[v]] = dist[u] + 1;
          q.emplace(mb[v]);
        }
      }
    }
    return succ;
  }

  // Find an augmenting path starting at u.
  bool dfs(int u) {
    for (int v : g[u]) {
      if (mb[v] == -1 || (dist[mb[v]] == dist[u] + 1 && dfs(mb[v]))) {
        ma[u] = v;
        mb[v] = u;
        return true;
      }
    }
    dist[u] = -1;  // Mark this point as inreachable after one visit.
    return false;
  }

  // Hopcroft-Karp maximum matching algorithm.
  std::vector<std::pair<int, int>> hopcroft_karp_maximum_matching() {
    // Build the level graph and then find a blocking flow.
    while (bfs()) {
      for (int u = 0; u < n1; ++u) {
        if (ma[u] == -1) {
          dfs(u);
        }
      }
    }
    // Collect the matched pairs.
    std::vector<std::pair<int, int>> matches;
    matches.reserve(n1);
    for (int u = 0; u < n1; ++u) {
      if (ma[u] != -1) {
        matches.emplace_back(u, ma[u]);
      }
    }
    return matches;
  }
};

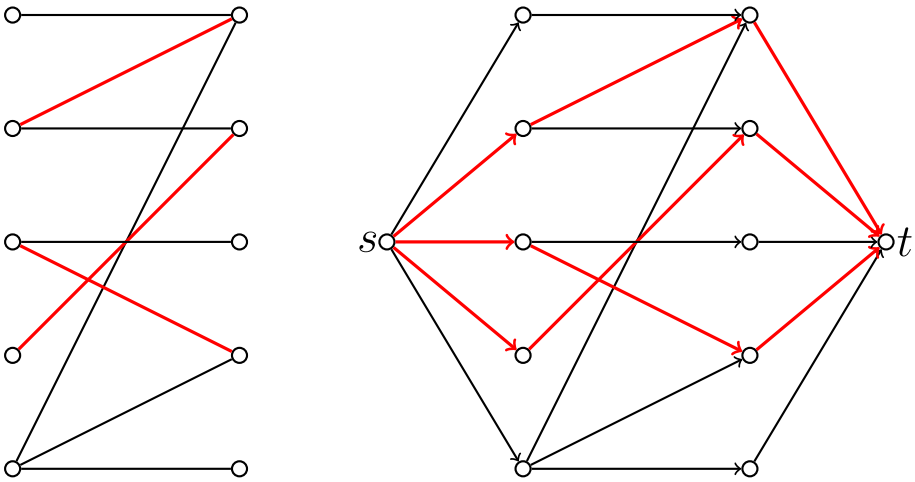
int main() {
```

```
  std::ios::sync_with_stdio(false), std::cin.tie(nullptr);
  int n1, n2, m;
  std::cin >> n1 >> n2 >> m;
  BipartiteGraph gr(n1, n2);
  for (int i = 0; i < m; ++i) {
    int u, v;
    std::cin >> u >> v;
    gr.add_edge(u, v);
  }
  auto res = gr.hopcroft_karp_maximum_matching();
  std::cout << res.size() << '\n';
  for (int i = 0; i < res.size(); ++i) {
    std::cout << res[i].first << ' ' << res[i].second << '\n';
  }
  return 0;
}

```

归约为最大流问题

二分图最大匹配问题可以归约为最大流问题.



如图所示，添加两个顶点分别作为源点和汇点．从源点出发，向每个左部点连接一条边；从每个右部点出发，向汇点连接一条边；并为二分图中的每条无向边，都连接一条从左部点指向右部点的边．所有边的容量都是 1．这样得到的有向图中的每个整数流，都和二分图中的一组匹配一一对应，且网络流的流量就是相应的匹配的大小．因此，求解二分图最大匹配，就相当于求解相应的有向图中的最大流．

任何可以解决最大流问题的算法都可以用于解决二分图最大匹配问题．容易发现，Kuhn 算法和 Hopcroft–Karp 算法都是最大流问题中相应算法的特例．同样地，预流推进算法→11.23.2@p2907 等同样可以用于解决二分图最大匹配问题．但是，应当注意的是，任何最大流算法，在应用于二分图最大匹配问题时，都需要有针对性地进行相应的优化，以避免过大的常数．

线性规划形式

和其他最大流问题一样，二分图 $G = (X, Y, E)$（记 $V = X \cup Y$）的最大匹配问题可以写作线性规划问题．如果用 $x_e \in \{0, 1\}$ 表示边 e 是否属于匹配，那么，可以得到如下的线性规划问题：

$$\max_{\{x_e\}} \sum_{e\in E} x_e$$
$$\text{subject to} \sum_{e\sim v} x_e \le 1,\ \forall v\in V,$$
$$x_e \ge 0,\ \forall e\in E.$$

其中，$e\sim v$ 表示关联关系，即顶点 v 是边 e 的端点之一．除了非负限制之外，问题的约束还要求每个顶点 $v\in V$ 处至多与一条边关联，这正是匹配的定义．因此，所有的匹配都对应于该线性规划可行域中的某些整点．

反过来却不然．在可行解中，x_e 可能是小数，这并不代表任何实际的匹配．尽管如此，对于二分图 G，上述线性规划的所有极点解都是整点．这意味着，目标函数的最优值总能在整点处取得，而无需考虑非整数的情形．这一性质对一般图并不成立，因此，上述线性规划在一般图中并不等价于最大匹配问题．

这一线性规划问题的对偶问题可以写作如下形式：

$$\min_{\{y_v\}} \sum_{v\in V} y_v$$
$$\text{subject to } y_u + y_v \ge 1,\ \forall (u,v)\in E,$$
$$y_v \ge 0,\ \forall v\in V.$$

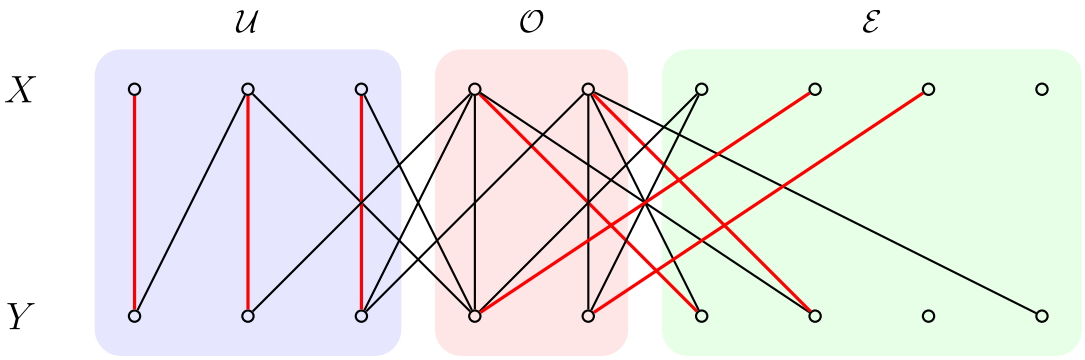
随后就会看到，这正是二分图的最小点覆盖问题．

Dulmage–Mendelsohn 分解

利用二分图的最大匹配，可以将顶点划分为若干互不相交的子集，从而完整刻画该二分图所有最大匹配的分布及结构特征．这就是 Dulmage–Mendelsohn 分解．算法竞赛中，利用这一分解，可以识别最大匹配中的关键点和关键边，进而判断最大匹配的唯一性或求解二分图博弈等问题．

构造方法

设二分图 $G=(X,Y,E)$ 的一个最大匹配为 M．



如图所示，可以为所有顶点 $V=X\cup Y$ 定义如下三个子集：

- 偶可达点 $\mathcal{E}$，即所有可以从一个未匹配点出发，沿着偶数长度的交错路可以到达的顶点集合；
- 奇可达点 $\mathcal{O}$，即所有可以从一个未匹配点出发，沿着奇数长度的交错路可以到达的顶点集合；
- 不可达点 $\mathcal{U}$，即所有无法从一个未匹配点出发，沿着交错路到达的顶点集合．

可以证明，这样得到的三个顶点集合 $\mathcal{E},\mathcal{O},\mathcal{U}$ 有如下性质：

性质

1. 集合 $\mathcal{E},\mathcal{O},\mathcal{U}$ 构成顶点集合的一个划分，且这一划分和最大匹配 M 的选取无关．
2. 图 G 的任一最大匹配都包含 $\mathcal{U}$ 的顶点之间的一个完美匹配，且将 $\mathcal{O}$ 中的每一个顶点都匹配到 $\mathcal{E}$ 中的一个顶点．也就是说，图 G 的最大匹配的大小等于 $|\mathcal{O}|+|\mathcal{U}|/2$．
3. 图 G 中不包含连接 $\mathcal{E}$ 中顶点和 $\mathcal{E}\cup\mathcal{U}$ 中顶点的边．

证明

1. 按照定义，$\mathcal{U}$ 和 $\mathcal{E} \cup \mathcal{O}$ 不交．只需要证明 $\mathcal{E}$ 和 $\mathcal{O}$ 不交．假若不然，对于顶点 $v \in \mathcal{E} \cap \mathcal{O}$，存在一条从未匹配点 a 到 v 的长度为偶数的交错路，也存在一条从未匹配点 b 到 v 的长度为奇数的交错路．由于图 G 是二分图，$a \neq b$，且两条路径到达 v 时的边分别是匹配边和非匹配边．因此，将两条路连接起来，就得到一条从 a 经 v 到 b 的交错路．这是一条增广路．这与 M 是最大匹配这一点相矛盾．因此，$\mathcal{E} \cap \mathcal{O} = \emptyset$.

   设 M' 是一个与 M 不同的最大匹配．重复 Berge 引理的证明→11.24.1@p2949 可以说明，$M' \oplus M$ 仅由偶数长度的路径和偶环组成．从最大匹配 M 出发，可以逐个将这些连通块（路径和环）中的边翻转（匹配边和非匹配边交换），就能得到最大匹配 M'．翻转偶环时，未匹配点仍然是未匹配点，从它出发的交错路的长度奇偶性也不会改变；翻转偶数长度路径时，路径两个端点的匹配状态互换，但是从它们出发到达路径中的任何一个点的路径长度的奇偶性也是一致的．因此，在翻转过程中，集合 $\mathcal{E}, \mathcal{O}, \mathcal{U}$ 均始终保持不变．这就说明这一分解与最大匹配 M 的选取无关．
2. 如果一条匹配边出现在某条从未匹配点 v 出发的交错路中，那么它的两个端点与点 v 的距离的奇偶性必然不同，因而分别属于集合 $\mathcal{E}$ 和 $\mathcal{O}$；否则，它的两个端点必然都在 $\mathcal{U}$ 中．这说明，最大匹配中的匹配边必然是 $\mathcal{EO}$ 边或 $\mathcal{UU}$ 边．反过来说，未匹配点可以沿着从它自身出发、长度为零的交错路到达，所以只会出现在集合 $\mathcal{E}$ 中，这说明，集合 $\mathcal{O}$ 和 $\mathcal{U}$ 中都是匹配点．简单计数可知，最大匹配的大小就是 $|\mathcal{O}| + |\mathcal{U}|/2$.
3. 按照定义，$\mathcal{E}$ 中的任一顶点 a 可以从未匹配点 v 出发沿偶数长度交错路到达，也就是说，$\mathcal{E}$ 中的顶点要么是未匹配点，要么到达该点的交错路 P 以匹配边结束．如果图 G 中存在连接 a 和 $\mathcal{E} \cup \mathcal{U}$ 中某个顶点 b 的一条边，那么根据上一段的讨论，这条边必然是非匹配边，可以沿着它延长交错路 P．这说明顶点 b 也属于集合 $\mathcal{O}$，这就与第一条性质矛盾．因此，图 G 中不存在连接 $\mathcal{E}$ 中顶点和 $\mathcal{E} \cup \mathcal{U}$ 中顶点的边．

由此得到的顶点集合的分解 $V = \mathcal{E} \cup \mathcal{O} \cup \mathcal{U}$ 就称为 **Dulmage–Mendelsohn 分解**．在利用前文所述算法求得最大匹配之后，可以通过 BFS 在 $O(|V| + |E|)$ 的时间内求出 Dulmage–Mendelsohn 分解．

最大匹配关键点

如果一个顶点 v 在二分图 G 的每一个最大匹配中都是匹配点，那么它就称为最大匹配的关键点．下面的结论说明：一个顶点是关键点，当且仅当在一个最大匹配中，不存在从未匹配点出发到达该顶点的偶数长度的交错路．

定理

设二分图 $G = (X, Y, E)$ 的 Dulmage–Mendelsohn 分解为 $V = \mathcal{E} \cup \mathcal{O} \cup \mathcal{U}$．那么，顶点 $v \in V$ 是关键点，当且仅当 $v \in \mathcal{O} \cup \mathcal{U}$.

证明

根据 Dulmage–Mendelsohn 分解的性质可知，在图 G 的任一最大匹配中，$\mathcal{O}$ 和 $\mathcal{U}$ 中的顶点都必然是匹配点．因此，$\mathcal{O} \cup \mathcal{U}$ 中的顶点必然是关键点．然后，需要说明集合 $\mathcal{E}$ 中一定没有关键点．如果在最大匹配 M 中，顶点 $a \in \mathcal{E}$ 是关键点，那么存在一条偶数长度的交错路 P 连接顶点 a 和某个未匹配点 $b \in \mathcal{E}$．将这条路上的所有边翻转，得到的最大匹配 $M \oplus P$ 中，顶点 a 就变成未匹配点．因此，集合 $\mathcal{E}$ 中没有关键点．

因此，要求出最大匹配的关键点，只需要求出 Dulmage–Mendelsohn 分解即可．

最大匹配关键边

类似地，如果一条边 e 在二分图 G 的每一个最大匹配中都是匹配边，那么它就称为最大匹配的关键边．二分图的最大匹配是唯一的，当且仅当它的一个最大匹配中，所有匹配边都是关键边．

定理

设二分图 $G=(X,Y,E)$ 的 Dulmage–Mendelsohn 分解为 $V=\mathcal{E}\cup\mathcal{O}\cup\mathcal{U}$，且 M 是它的一个最大匹配．那么，边 $e\in E$ 是关键边，当且仅当 e 的端点都在 $\mathcal{U}$ 中，边 e 是 M 中匹配边，且相对于 M 不存在一个包含边 e 的交错环．

证明

关键边的端点必须是关键点．根据 Dulmage–Mendelsohn 分解的性质，最大匹配的边只能是 $\mathcal{EO}$ 边或 $\mathcal{UU}$ 边．但是，$\mathcal{E}$ 中没有关键点，所以，关键边只能是 $\mathcal{UU}$ 边．当然，关键边也必须是 M 中的匹配边．设 $e\in M$ 是一条 $\mathcal{UU}$ 边．它不是关键边，当且仅当存在另一个最大匹配 $M'\neq M$ 使得 $e\in M\oplus M'$．重复 Berge 引理的证明→11.24.1@p2949 可以说明，$M'\oplus M$ 仅由偶数长度的路径和偶环组成．这些路径的端点之一是相对于 M 的未匹配点，所以路径中的顶点都不是 $\mathcal{U}$ 中的点，这与边 e 的选取矛盾．因此，边 e 只能出现在偶环中．因此，一条 $\mathcal{UU}$ 边 $e\in M$ 不是关键边，当且仅当相对于 M 存在一个包含边 e 的交错环．这就是所要求证的．

因此，要求出最大匹配的关键边，需要按照如下步骤进行：

1. 求出图 G 的最大匹配 M；
2. 按照 M 将图 G 的边定向，得到有向图 G_M；
3. 进行 BFS 求出 Dulmage–Mendelsohn 分解中的集合 $\mathcal{U}$，即无法通过未匹配点沿着交错路到达的顶点集合；
4. 利用 Tarjan 算法→11.13.1@p2806 求出有向图 G_M 的全部强连通分量；
5. 遍历匹配 M 中的边，如果它的端点都在 $\mathcal{U}$ 中，但是不在同一个强连通分量中，它就是一条关键边．

得到最大匹配后，后续步骤的时间复杂度为 $O(|V|+|E|)$．

相关问题

利用二分图最大匹配的算法，可以解决其它组合优化问题．

二分图最小点覆盖

最小点覆盖问题是指，在一张无向图中选择最少的顶点，满足每条边至少有一个端点被选．

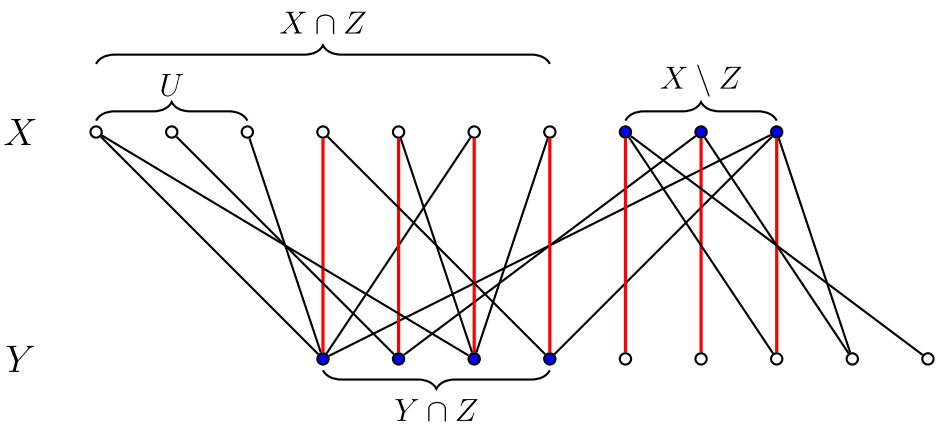
一般图的最小点覆盖问题是 NP 困难的，但是对于二分图，Kőnig 定理说明它可以归约为最大匹配问题，从而高效求解．定理的证明同时也给出了最小点覆盖的构造．

Kőnig 定理

二分图中，最小点覆盖中的顶点数量等于最大匹配中的边数量．

证明

设二分图 $G=(X,Y,E)$ 的一个最大匹配为 M．设 U 是未匹配的左部点的集合，图 G 中可以由 U 中顶点出发，沿着某条交错路到达的顶点集合为 Z．于是，顶点集合 $C=(X\setminus Z)\cup(Y\cap Z)$ 就是所求的最小点覆盖．



首先，集合 C 是点覆盖．假设不然，存在边 $(u,v) \in E$ 使得 $u \in X \cap Z$ 且 $v \in Y \setminus Z$．设 P_u 是到达 u 的一条交错路．如果边 (u,v) 是匹配边，那么，路径 P_u 中的最后一条边就是 (v,u)，这与 $v \notin Z$ 矛盾；如果边 (u,v) 不是匹配边，那么，可以沿着边 (u,v) 延长 P_u 得到一条到达 v 的交错路，同样与 $v \notin Z$ 矛盾．这些矛盾说明所有边都至少包含一个 C 中的端点，所以 C 是点覆盖．

然后，需要说明 C 是最小点覆盖．为了覆盖最大匹配 M 的所有边，任何点覆盖都至少需要 $|M|$ 个顶点．因此，只需要证明 $|C| = |M|$，它就一定是最小点覆盖．这等价于证明，除了包含每条匹配边各一个端点之外，C 再不包含其它顶点；也就是说，C 不包含未匹配点．假设不然，存在未匹配点 $v \in C$．如果 $v \in X$，就一定有 $v \in U \subseteq Z$，这与 C 的构造矛盾；如果 $v \in Y$，那么，到达 v 的一条交错路是相对于 M 的一条增广路，由 Berge 引理，这与 M 是最大匹配矛盾．这些矛盾说明不存在这样的未匹配点，进而 C 是最小点覆盖．

从网络流的角度看，最小点覆盖问题就是最小割问题：选择左部点，相当于切割它与源点的连边；选择右部点，相当于切割它与汇点的连边．从线性规划的角度看，最小点覆盖问题就是最大匹配问题的对偶问题．因此，Kőnig 定理可以看作是 最大流最小割定理→11.23.2@p2907 的特殊情形，或者更一般地，线性规划的强对偶定理的特殊情形．

二分图最大独立集

最大独立集问题是指，在一张无向图中选择最多的顶点，满足两两之间互不相邻．

对于一般图，成立如下定理：

定理

图 $G = (V, E)$ 中，点集 $C \subseteq V$ 是点覆盖，当且仅当它的补集 $V \setminus C$ 是独立集．

证明

点集 C 是点覆盖，当且仅当 E 中任何一条边 e 的两个端点至少有一个出现在集合 C 中，当且仅当 E 中没有一条边的两个端点都出现在集合 $V \setminus C$ 中，当且仅当，$V \setminus C$ 是独立集．

推论

图 $G = (V, E)$ 中，最小点覆盖与最大独立集的大小之和等于顶点数目．

因此，与最小点覆盖问题一样，最大独立集问题对于一般图是 NP 困难的，但是对于二分图它可以归约为最大匹配问题，从而高效求解．

有向无环图最小路径覆盖

最小路径覆盖问题是指，在一张有向图中，选择最少数量的简单路径，使得所有顶点都恰好出现在一条路径中.

一般的有向图上的最小路径覆盖问题是 NP 困难的，但是对于有向无环图，该问题可以归约为二分图最大匹配问题. 对于有向无环图 $G=(V,E)$，可以构造二分图 $G'=(V^{\text{in}},V^{\text{out}},E')$ 如下：

- 为每个顶点 $v\in V$，分别建立一个入点 v^{in} 和一个出点 v^{out}. 设全体入点和出点的集合分别为 V^{in} 和 V^{out}. 它们分别成为新图的左部和右部.
- 为每条有向边 $(u,v)\in E$，建立无向边 $(u^{\text{out}},v^{\text{in}})$. 全体无向边的集合就是 E'.

据此，有如下结论：

定理

有向无环图 $G=(V,E)$ 的最小路径覆盖与相应的二分图 $G'=(V^{\text{in}},V^{\text{out}},E')$ 的最大匹配的大小之和等于顶点数量.

证明

二分图 G' 的每个匹配 M'，都对应着图 G 的一张子图 F，且子图 F 中每个顶点的入度和出度都至多为一，也就是说，子图 F 实际上是有向图 G 中若干互不相交的路径或环的集合. 但是，已经假设 G 中不存在环路，所以 F 只包含若干不交的路径. 反过来，对于每个这样的子图 F，都能构造出相应的匹配. 因为匹配 M' 的大小，就是顶点数量与 F 中路径数量的差值，所以，图 G 的最小路径覆盖问题，就对应着图 G' 的最大匹配问题.

证明是构造性的，因此，很容易根据得到的最大匹配构造出相应的最小路径覆盖. 而且，这个构造说明，对于一般的有向图，这个归约不再成立，正是因为二分图中的匹配可能对应着有向图中的环.

特别地，对于集合 X 和它上面的偏序关系 P，可以建立有向无环图 $G=(X,P)$. 此时，根据 Dilworth 定理→9.20@p1951，图 G 的最小路径覆盖的大小，就等于它的最长反链的长度，也就是偏序集 (X,P) 的宽度. 因此，本节实际上给出了任意偏序集的宽度的高效计算方法.

例题

应用二分图匹配的难点在于建图，本节通过一些例题展示建图的技巧.

Luogu P1129 矩阵游戏[3]

有一个 01 方阵，每一次可以交换两行或两列，问是否可以交换使得主对角线（左上到右下）全都是 1.

解法

注意到，如果存在 n 个 1，使得它们两两不在同一行、同一列，那么必然有解，否则必然无解. 问题转化成了能否找到这 n 个 1.

考虑对于一个 1 而言，最终的方案中选了这个 1 代表这个 1 的行、列被占用. 于是可以建出一个 n 个左部点、n 个右部点的二分图，其中对于某个为 1 的元素，我们建一条连接它的行的左部点和它的列的右部点. 于是就可以二分图匹配了.

代码

```cpp
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int ct, n, t[100010], x, r[100010], ans, vis[100010], dis[100010];
vector<int> to[100010];
queue<int> q;

int DFS(int x) {
  if (vis[x]) {
    return 0;
  }
  vis[x] = 1;
  for (auto i : to[x]) {
    if (!r[i]) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
    if (dis[r[i]] == dis[x] + 1 && DFS(r[i])) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
  }
  return 0;
}

int BFS() {
  fill(vis + 1, vis + n + 1, 0);
  fill(dis + 1, dis + n + 1, 0);
  for (int i = 1; i <= n; i++) {
    if (!t[i]) {
      q.push(i);
      dis[i] = 1;
    }
  }
  int f = 0;
  for (; q.size(); q.pop()) {
    int tmp = q.front();
    for (auto i : to[tmp]) {
      if (!r[i]) {
        f = 1;
      }
      if (r[i]) {
        if (!dis[r[i]]) {
          dis[r[i]] = dis[tmp] + 1;
          q.push(r[i]);
        }
      }
```

```
    }
  }
  return f;
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr), cout.tie(nullptr);
  for (cin >> ct; ct--;) {
    cin >> n;
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        cin >> x;
        if (x) {
          to[i].push_back(j);
        }
      }
    }
    for (; BFS();) {
      for (int i = 1; i <= n; i++) {
        if (!t[i] && DFS(i)) {
          ans++;
        }
      }
    }
    cout << (ans == n ? "Yes" : "No") << '\n';
    for (int i = 1; i <= n; i++) {
      t[i] = r[i] = 0;
      to[i].clear();
    }
    ans = 0;
  }
  return 0;
}

```

Gym 104427B Lawyers[4]

有 n 个律师，都被指控有欺诈罪．于是，他们需要互相辩护，确保每一名律师都被释放．这 n 个律师有 m 对信任关系，一个信任关系 (a,b) 表示 a 可以为 b 辩护．任何一个受到辩护的律师都会被无罪释放，除了一个例外：如果 a 和 b 互相辩护，他们都会被判有罪．

求是否可以使得每一名律师都被释放．

解法

对于每一个 **无序对** (a,b)，当 a 可以辩护 b，连这个无序对向 b 的边，反之亦然．

只保存有边相连的 (a,b)，问题被转化成了一个 m 个左部点、n 个右部点的二分图最大匹配．

代码

```cpp
#include <algorithm>
#include <iostream>
#include <map>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

int n, k, u, v, t[200020], r[200020], ans, vis[200020], dis[200020], c;
map<pair<int, int>, int> mp;
vector<int> to[200020];
queue<int> q;

int DFS(int x) {
  if (vis[x]) {
    return 0;
  }
  vis[x] = 1;
  for (auto i : to[x]) {
    if (!r[i]) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
    if (dis[r[i]] == dis[x] + 1 && DFS(r[i])) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
  }
  return 0;
}

int BFS() {
  fill(vis + 1, vis + n + 1, 0);
  fill(dis + 1, dis + n + 1, 0);
  for (int i = 1; i <= n; i++) {
    if (!t[i]) {
      q.push(i);
      dis[i] = 1;
    }
  }
  int f = 0;
  for (; q.size(); q.pop()) {
    int tmp = q.front();
    for (auto i : to[tmp]) {
      if (!r[i]) {
        f = 1;
      }
      if (r[i]) {
        if (!dis[r[i]]) {
          dis[r[i]] = dis[tmp] + 1;
```

```
53            q.push(r[i]);
54          }
55        }
56      }
57    }
58    return f;
59  }
60
61  void mxf() {
62    for (; BFS();) {
63      for (int i = 1; i <= n; i++) {
64        if (!t[i] && DFS(i)) {
65          ans++;
66        }
67      }
68    }
69  }
70
71  int main() {
72    ios::sync_with_stdio(false);
73    cin.tie(nullptr), cout.tie(nullptr);
74    cin >> n >> k;
75    for (int i = 1; i <= k; i++) {
76      cin >> u >> v;
77      if (mp.find({u, v}) == mp.end()) {
78        mp[{u, v}] = mp[{v, u}] = ++c;
79      }
80      to[v].push_back({mp[{u, v}]});
81    }
82    mxf();
83    cout << (ans == n ? "YES" : "NO") << '\n';
84    ans = 0;
85    return 0;
86  }
87
```

Codeforces 1404E Bricks[5]

用一些 $1 \times x$ 的砖精确覆盖一个 $n \times m$ 的网格，砖可以旋转，其中有一些格子不能覆盖.

解法

考虑最终的方案是如何构成的:

先在所有能覆盖的网格上全部铺上 1×1 的砖. 对于一个 $1 \times x$ 的砖，可以由同一行的 x 个连续的 1×1 砖依次「行合并」形成. 同理，对于一个 $x \times 1$ 的砖. 可以由同一列的 x 个连续的 1×1 砖依次「列合并」形成.

显然，一次行合并和一次列合并不能干涉到同一个砖，而且合并的次数越多，砖块数量越少. 于是，可以以行合并作为左部点，列合并作为右部点，以前面的冲突作为边，建出一个二分图. 从而，原问题变成了一个二分图最大独立集问题.

代码

```cpp
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

int n, m, belr[220][220], beld[220][220], dcnt, rcnt, vis[100010],
dis[100010],
    t[100010], r[100010], cnt;
char c[220][220];
vector<int> to[100010];
queue<int> q;

int DFS(int x) {
  if (vis[x]) {
    return 0;
  }
  vis[x] = 1;
  for (auto i : to[x]) {
    if (!r[i]) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
    if (dis[r[i]] == dis[x] + 1 && DFS(r[i])) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
  }
  return 0;
}

int BFS() {
  fill(vis + 1, vis + rcnt + 1, 0);
  fill(dis + 1, dis + rcnt + 1, 0);
  for (int i = 1; i <= rcnt; i++) {
    if (!t[i]) {
      q.push(i);
      dis[i] = 1;
    }
  }
  int f = 0;
  for (; q.size(); q.pop()) {
    int tmp = q.front();
    for (auto i : to[tmp]) {
      if (!r[i]) {
        f = 1;
      }
      if (r[i]) {
        if (!dis[r[i]]) {
```

```
51            dis[r[i]] = dis[tmp] + 1;
52            q.push(r[i]);
53          }
54        }
55      }
56    }
57    return f;
58  }
59
60  int solve() {
61    int rt = 0;
62    for (; BFS();) {
63      for (int i = 1; i <= rcnt; i++) {
64        if (!t[i] && DFS(i)) {
65          rt++;
66        }
67      }
68    }
69    return rt;
70  }
71
72  int main() {
73    ios::sync_with_stdio(false);
74    cin.tie(nullptr), cout.tie(nullptr);
75    cin >> n >> m;
76    for (int i = 1; i <= n; i++) {
77      for (int j = 1; j <= m; j++) {
78        cin >> c[i][j];
79      }
80    }
81    for (int i = 1; i <= n; i++) {
82      for (int j = 1; j <= m; j++) {
83        if (c[i][j] == '#') {
84          cnt++;
85          if (c[i + 1][j] == '#') {
86            beld[i][j] = ++dcnt;
87          }
88          if (c[i][j + 1] == '#') {
89            belr[i][j] = ++rcnt;
90          }
91        }
92      }
93    }
94    for (int i = 1; i <= n; i++) {
95      for (int j = 1; j <= m; j++) {
96        if (c[i][j] == '#') {
97          if (c[i - 1][j] == '#' && c[i][j - 1] == '#') {
98            to[belr[i][j - 1]].push_back(beld[i - 1][j]);
99          }
100         if (c[i + 1][j] == '#' && c[i][j + 1] == '#') {
101           to[belr[i][j]].push_back(beld[i][j]);
102         }
```

```
        if (c[i + 1][j] == '#' && c[i][j - 1] == '#') {
          to[belr[i][j - 1]].push_back(beld[i][j]);
        }
        if (c[i - 1][j] == '#' && c[i][j + 1] == '#') {
          to[belr[i][j]].push_back(beld[i - 1][j]);
        }
      }
    }
  }
  cout << cnt - rcnt - dcnt + solve() << '\n';
  return 0;
}
```

Codeforces 1139E - Maximize Mex[6]

有 m 个共有 n 个元素的可重集，每一次从某一个可重集里面删除一个元素，然后查询「在每一个可重集里面选至多一个元素，可以达到的最大 mex」.

解法

先考虑如果没有删除元素时怎么做.

对于每一个可重集，开一个新点；对于每一个可能的答案，开一个新点. 然后，对于某一个对应点 l_i 的可重集的一个元素 a，连一条 l_i 至 r_a 的边. 此时这个弱化版本变成了一个二分图最大匹配.

现在加回来删除元素的操作，发现根本搞不了：删了一条边可能引起匹配的巨变，复杂度无法接受. 于是，不如反过来，我们每一次加一条边，然后顺过去重新增广. 所以本题只能使用 Kuhn 算法.

代码

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

struct node {
  int u, v;
} arr[5050];

int n, m, t[5050], r[5050], d, ot[5050], kil[5050], vis[5050], ans,
out[5050];
vector<int> to[5050];

int DFS(int x) {
  if (vis[x]) {
    return 0;
  }
  vis[x] = 1;
  for (auto i : to[x]) {
    if (r[i] == -1) {
```

```
      r[i] = x;
      t[x] = i;
      return 1;
    }
    if (DFS(r[i])) {
      r[i] = x;
      t[x] = i;
      return 1;
    }
  }
  return 0;
}

int main() {
  cin >> n >> m;
  for (int i = 1; i <= n; i++) {
    cin >> arr[i].u;
    if (arr[i].u > m) {
      arr[i].u = m + 1;
    }
  }
  for (int i = 1; i <= n; i++) {
    cin >> arr[i].v;
  }
  cin >> d;
  for (int i = 1; i <= d; i++) {
    cin >> ot[i];
    kil[ot[i]] = 1;
  }
  for (int i = 1; i <= n; i++) {
    if (!kil[i]) {
      to[arr[i].u].push_back(arr[i].v);
    }
  }
  fill(r + 1, r + m + 1, -1);
  for (; ans <= m;) {
    fill(vis, vis + m + 2, 0);
    if (DFS(ans)) {
      ans++;
    } else {
      break;
    }
  }
  out[d] = ans;
  for (int i = d; i > 1; i--) {
    to[arr[ot[i]].u].push_back(arr[ot[i]].v);
    for (; ans <= m;) {
      fill(vis, vis + m + 2, 0);
      if (DFS(ans)) {
        ans++;
      } else {
        break;
```

```
72        }
73      }
74      out[i - 1] = ans;
75    }
76    for (int i = 1; i <= d; i++) {
77      cout << out[i] << '\n';
78    }
79    return 0;
80  }
81
```

Luogu P3355 - 骑士共存问题[7]

有一个 $n \times n$ 的国际象棋棋盘，其中一些位置不能放棋子，问最多可以放多少个马使得这些马不会互相攻击.

解法

可以发现，如果对整个棋盘染色使得所有黑格、白格均不相邻，那么马只能够攻击到与其异色的格子.

然后就可以直接二分图最大独立集了.

代码

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

const int kD[8][2] = {{-2, -1}, {-1, -2}, {1, -2}, {2, -1},
                      {2, 1},   {1, 2},   {-1, 2}, {-2, 1}};

int n, m, ib;
int vis[200020], dis[200020], t[200020], r[200020], ans, u, v, cnta,
    x[220][220], cl[220][220], idx, bel[220][220];
vector<int> to[200020];
queue<int> q;

int DFS(int x) {
  if (vis[x]) {
    return false;
  }
  vis[x] = true;
  for (auto i : to[x]) {
    if (!r[i]) {
      r[i] = x;
      t[x] = i;
      return true;
```

```
    }
    if (dis[r[i]] == dis[x] + 1 && DFS(r[i])) {
      r[i] = x;
      t[x] = i;
      return true;
    }
  }
  return false;
}

int BFS() {
  fill(vis + 1, vis + cnta + 1, false);
  fill(dis + 1, dis + cnta + 1, 0);
  for (int i = 1; i <= cnta; i++) {
    if (!t[i]) {
      q.push(i);
      dis[i] = 1;
    }
  }
  int f = 0;
  for (; q.size(); q.pop()) {
    int tmp = q.front();
    for (auto i : to[tmp]) {
      if (!r[i]) {
        f = 1;
      }
      if (r[i]) {
        if (!dis[r[i]]) {
          dis[r[i]] = dis[tmp] + 1;
          q.push(r[i]);
        }
      }
    }
  }
  return f;
}

void din() {
  for (; BFS();) {
    for (int i = 1; i <= cnta; i++) {
      if (!t[i] && DFS(i)) {
        ans++;
      }
    }
  }
}

int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr), cout.tie(nullptr);
  cin >> n >> m;
  for (int i = 1; i <= m; i++) {
```

```
    cin >> u >> v;
    x[u][v] = true;
  }
  cl[1][1] = 1;
  if (!x[1][1]) {
    bel[1][1] = ++idx;
  }
  for (int i = 2; i <= n; i++) {
    cl[1][i] = cl[1][i - 1] ^ 1;
    if (!x[1][i] && cl[1][i]) {
      bel[1][i] = ++idx;
    } else {
      if (!x[1][i]) {
        bel[1][i] = ++ib;
      }
    }
  }
  for (int i = 2; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      cl[i][j] = cl[i - 1][j] ^ 1;
      if (!x[i][j] && cl[i][j]) {
        bel[i][j] = ++idx;
      } else {
        if (!x[i][j]) {
          bel[i][j] = ++ib;
        }
      }
    }
  }
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (!x[i][j]) {
        cnta += cl[i][j];
        for (int k = 0; k < 8; k++) {
          if (i + kD[k][0] > 0 && j + kD[k][1] > 0 &&
              bel[i + kD[k][0]][j + kD[k][1]] &&
              !x[i + kD[k][0]][j + kD[k][1]]) {
            if (cl[i + kD[k][0]][j + kD[k][1]]) {
              to[bel[i + kD[k][0]][j + kD[k][1]]].push_back(bel[i][j]);
            } else {
              to[bel[i][j]].push_back(bel[i + kD[k][0]][j + kD[k][1]]);
            }
          }
        }
      }
    }
  }
  din();
  cout << n * n - m - ans << '\n';
  return 0;
}

```

习题

- Codeforces 1765A - Access Levels[8]
- AtCoder abc274G - Security Camera 3[9]
- Codeforces 1773D - Dominoes[10]
- Luogu P5030 - 长脖子鹿放置[11]
- Luogu P2071 - 座位安排[12]
- LibreOJ 6002 - 最小路径覆盖[13]

参考资料

- Kuhn's Algorithm - Maximum Bipartite Matching[14]
- 二分图最大匹配的 König 定理及其证明[15]
- Implementing Dinitz on bipartite graphs by adamant - Codeforces blogs[16]
- Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. Graph theory with applications. Vol. 290. London: Macmillan, 1976.
- 陈胤伯. 浅谈图的匹配算法及其应用. 2015 年信息学奥林匹克中国国家队候选队员论文集.
- Dulmage–Mendelsohn decomposition - Wikipedia[17]
- Notes on Dulmage–Mendelsohn decomposition[18]

[1] Library Checker - Matching on Bipartite Graph

[2] Library Checker - Matching on Bipartite Graph

[3] Luogu P1129 矩阵游戏

[4] Gym 104427B Lawyers

[5] Codeforces 1404E Bricks

[6] Codeforces 1139E - Maximize Mex

[7] Luogu P3355 - 骑士共存问题

[8] Codeforces 1765A - Access Levels

[9] AtCoder abc274G - Security Camera 3

[10] Codeforces 1773D - Dominoes

[11] Luogu P5030 - 长脖子鹿放置

[12] Luogu P2071 - 座位安排

[13] LibreOJ 6002 - 最小路径覆盖

[14] Kuhn's Algorithm - Maximum Bipartite Matching

[15] 二分图最大匹配的 König 定理及其证明

[16] Implementing Dinitz on bipartite graphs by adamant - Codeforces blogs

[17] Dulmage–Mendelsohn decomposition - Wikipedia

[18] Notes on Dulmage–Mendelsohn decomposition

11.24.3 二分图最大权匹配

Authors: accelsao, Enter-tainer, guodong2005, StudyingFather, Backl1ght, Chrogeek, H-J-Granger, Henry-ZHR

二分图的最大权匹配是指二分图中边权和最大的匹配.

Hungarian Algorithm（Kuhn–Munkres Algorithm）

匈牙利算法又称为 KM 算法，可以在 $O(n^3)$ 时间内求出二分图的 **最大权完美匹配**.

考虑到二分图中两个集合中的点并不总是相同，为了能应用 KM 算法解决二分图的最大权匹配，需要先作如下处理：将两个集合中点数比较少的补点，使得两边点数相同，再将不存在的边权重设为 0，这种情况下，问题就转换成求 **最大权完美匹配问题**，从而能应用 KM 算法求解.

可行顶标

给每个节点 i 分配一个权值 $l(i)$，对于所有边 (u,v) 满足 $w(u,v) \le l(u)+l(v)$.

相等子图

在一组可行顶标下原图的生成子图，包含所有点但只包含满足 $w(u,v)=l(u)+l(v)$ 的边 (u,v).

定理 1：对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配.

证明 1.

考虑原二分图任意一组完美匹配 M，其边权和为

$val(M)=\sum_{(u,v)\in M}w(u,v)\le\sum_{(u,v)\in M}l(u)+l(v)\le\sum_{i=1}^{n}l(i)$

任意一组可行顶标的相等子图的完美匹配 M' 的边权和

$val(M')=\sum_{(u,v)\in M}l(u)+l(v)=\sum_{i=1}^{n}l(i)$

即任意一组完美匹配的边权和都不会大于 $val(M')$，那个 M' 就是最大权匹配.

有了定理 1，我们的目标就是透过不断的调整可行顶标，使得相等子图是完美匹配.

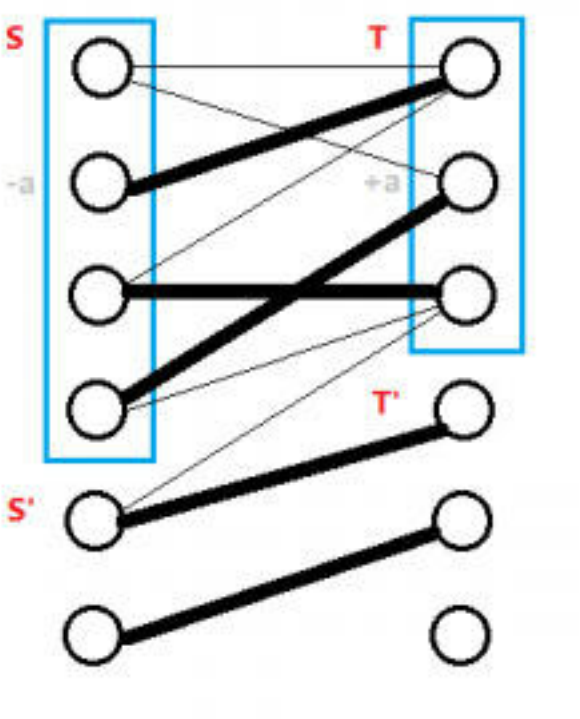
因为两边点数相等，假设点数为 n，$lx(i)$ 表示左边第 i 个点的顶标，$ly(i)$ 表示右边第 i 个点的顶标，$w(u,v)$ 表示左边第 u 个点和右边第 v 个点之间的权重.

首先初始化一组可行顶标，例如

$lx(i)=\max_{1\le j\le n}\{w(i,j)\},ly(i)=0$

然后选一个未匹配点，如同最大匹配一样求增广路. 找到增广路就增广，否则，会得到一个交错树.

令 S，T 表示二分图左边右边在交错树中的点，S'，T' 表示不在交错树中的点.



在相等子图中：

- $S-T'$ 的边不存在，否则交错树会增长.
- $S'-T$ 一定是非匹配边，否则他就属于 S.

假设给 S 中的顶标 $-a$，给 T 中的顶标 $+a$，可以发现

- $S-T$ 边依然存在相等子图中.
- $S'-T'$ 没变化.
- $S-T'$ 中的 $lx+ly$ 有所减少，可能加入相等子图.
- $S'-T$ 中的 $lx+ly$ 会增加，所以不可能加入相等子图.

所以这个 a 值的选择，显然得是 $S-T'$ 当中最小的边权，

$$a=\min\{lx(u)+ly(v)-w(u,v)\mid u\in S,v\in T'\}.$$

当一条新的边 (u,v) 加入相等子图后有两种情况

- v 是未匹配点，则找到增广路
- v 和 S' 中的点已经匹配

这样至多修改 n 次顶标后，就可以找到增广路.

每次修改顶标的时候，交错树中的边不会离开相等子图，那么我们直接维护这棵树.

我们对 T 中的每个点 v 维护

$$slack(v)=\min\{lx(u)+ly(v)-w(u,v)\mid u\in S\}.$$

所以可以在 $O(n)$ 算出顶标修改值 a

$$a=\min\{slack(v)\mid v\in T'\}$$

交错树新增一个点进入 S 的时候需要 $O(n)$ 更新 $slack(v)$. 修改顶标需要 $O(n)$ 给每个 $slack(v)$ 减去 a. 只要交错树找到一个未匹配点，就找到增广路.

一开始枚举 n 个点找增广路，为了找增广路需要延伸 n 次交错树，每次延伸需要 n 次维护，共 $O(n^3)$.

参考代码

```cpp
template <typename T>
struct hungarian {  // km
  int n;
  vector<int> matchx;  // 左集合对应的匹配点
  vector<int> matchy;  // 右集合对应的匹配点
  vector<int> pre;     // 连接右集合的左点
  vector<bool> visx;   // 拜访数组 左
  vector<bool> visy;   // 拜访数组 右
  vector<T> lx;
  vector<T> ly;
  vector<vector<T>> g;
  vector<T> slack;
  T inf;
  T res;
  queue<int> q;
  int org_n;
  int org_m;

  hungarian(int _n, int _m) {
    org_n = _n;
    org_m = _m;
    n = max(_n, _m);
    inf = numeric_limits<T>::max();
    res = 0;
    g = vector<vector<T>>(n, vector<T>(n));
    matchx = vector<int>(n, -1);
```

```
    matchy = vector<int>(n, -1);
    pre = vector<int>(n);
    visx = vector<bool>(n);
    visy = vector<bool>(n);
    lx = vector<T>(n, -inf);
    ly = vector<T>(n);
    slack = vector<T>(n);
  }

  void addEdge(int u, int v, int w) {
    g[u][v] = max(w, 0);  // 负值还不如不匹配 因此设为0不影响
  }

  bool check(int v) {
    visy[v] = true;
    if (matchy[v] != -1) {
      q.push(matchy[v]);
      visx[matchy[v]] = true;  // in S
      return false;
    }
    // 找到新的未匹配点 更新匹配点 pre 数组记录着"非匹配边"上与之相连的点
    while (v != -1) {
      matchy[v] = pre[v];
      swap(v, matchx[pre[v]]);
    }
    return true;
  }

  void bfs(int i) {
    while (!q.empty()) {
      q.pop();
    }
    q.push(i);
    visx[i] = true;
    while (true) {
      while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < n; v++) {
          if (!visy[v]) {
            T delta = lx[u] + ly[v] - g[u][v];
            if (slack[v] >= delta) {
              pre[v] = u;
              if (delta) {
                slack[v] = delta;
              } else if (check(v)) {  // delta=0 代表有机会加入相等子图 找增广路
                                      // 找到就return 重建交错树
                return;
              }
            }
          }
        }
```

```
      }
      // 没有增广路 修改顶标
      T a = inf;
      for (int j = 0; j < n; j++) {
        if (!visy[j]) {
          a = min(a, slack[j]);
        }
      }
      for (int j = 0; j < n; j++) {
        if (visx[j]) {  // S
          lx[j] -= a;
        }
        if (visy[j]) {  // T
          ly[j] += a;
        } else {  // T'
          slack[j] -= a;
        }
      }
      for (int j = 0; j < n; j++) {
        if (!visy[j] && slack[j] == 0 && check(j)) {
          return;
        }
      }
    }
  }

  void solve() {
    // 初始顶标
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        lx[i] = max(lx[i], g[i][j]);
      }
    }

    for (int i = 0; i < n; i++) {
      fill(slack.begin(), slack.end(), inf);
      fill(visx.begin(), visx.end(), false);
      fill(visy.begin(), visy.end(), false);
      bfs(i);
    }

    // custom
    for (int i = 0; i < n; i++) {
      if (g[i][matchx[i]] > 0) {
        res += g[i][matchx[i]];
      } else {
        matchx[i] = -1;
      }
    }
    cout << res << "\n";
    for (int i = 0; i < org_n; i++) {
      cout << matchx[i] + 1 << " ";
```

```
131     }
132     cout << "\n";
133   }
134 };
```

Dynamic Hungarian Algorithm

原论文 The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs[1]

伪代码更清晰的论文 A Fast Dynamic Assignment Algorithm for Solving Resource Allocation Problems[2]

相关 OJ 问题 DAP[3]

算法思路

1. 修改单点 u_i 和所有 v_j 之间的权重，即权重矩阵中的一行
   - 修改顶标 $lx(u_i) = max(w_{ij} - v_j), \forall j$
   - 删除 u_i 相关的匹配
2. 修改所有 u_i 和单点 v_j 之间的权重，即权重矩阵中的一列
   - 修改顶标 $ly(v_j) = max(w_{ij} - u_i), \forall i$
   - 删除 v_j 相关的匹配
3. 修改单点 u_i 和单点 v_j 之间的权重，即权重矩阵中的单个元素
   - 做 1 或 2 两种操作之一即可
4. 添加某一单点 u_i，或者某一单点 v_j，即在权重矩阵中添加或者删除一行或者一列
   - 对应地做 1 或 2 即可，注意此处加点操作仅为加点，不额外设定权重值，新加点与其他点的权重为 0.

算法证明

- 设原图为 G，左右两边的顶标为 α^i 和 β^j，可行顶标为 l，那 G_l 是 G 的一个子图，包含图 G 中满足 $w_{ij} = alpha_i + beta_j$ 的点和边.
- 在上面匈牙利算法的部分，定理一证明了：对于某组可行顶标，如果其相等子图存在完美匹配，那么，该匹配就是原二分图的最大权完美匹配.
- 假设原来的最优匹配是 M^*，当一个修改发生的时候，我们会根据规则更新可行顶标，更新后的顶标设为 α^{i^*}，或者 β^{j^*}，会出现以下情况：
  1. 权重矩阵的一整行被修改了，设被修改的行为 i^* 行，即 v_{i^*} 的所有边被修改了，所以 v_{i^*} 原来的顶标可能不满足条件，因为我们需要 $w_{i^*j} \leq alpha_{i^*} + beta_j$，但对于其他的 u_j 来说，除了 i^* 相关的边，他们的边权是不变的，因此他们的顶标都是合法的，所以算法中修改了 v_{i^*} 相关的顶标使得这组顶标是一组可行顶标.
  2. 权重矩阵的一整列被修改了，同理可得算法修改顶标使得这组顶标是一组可行顶标.
  3. 修改权重矩阵某一元素，任意修改其中一个顶标即可满足顶标条件
- 每一次权重矩阵被修改，都关系到一个特定节点，这个节点可能是左边的也可能是右边的，因此我们直接记为 x，这个节点和某个节点 y 在原来的最优匹配中匹配上了. 每一次修改操作，最多让这一对节点 unpair，因此我们只要跑一轮匈牙利算法中的搜索我们就能得到一个新的 match，而根据定理一，新跑出来的 match 是最优的.

以下代码应该为论文 2 作者提交的代码（以下代码为最大化权重版本，原始论文中为最小化 cost）

动态匈牙利算法参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
#include <list>
using namespace std;
using LL = long long;
constexpr LL INF = (LL)1e15;
constexpr int MAXV = 105;

int N, mateS[MAXV], mateT[MAXV], p[MAXV];
LL u[MAXV], v[MAXV], slack[MAXV];
LL W[MAXV][MAXV];
bool m[MAXV];
list<int> Q;

void readMatrix() {
  cin >> N;
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) cin >> W[i][j];
}

void initHungarian() {
  memset(mateS, -1, sizeof(mateS));
  memset(mateT, -1, sizeof(mateT));
  for (int i = 0; i < N; i++) {
    u[i] = -INF;
    for (int j = 0; j < N; j++) u[i] = max(u[i], W[i][j]);
    v[i] = 0;
  }
}

void augment(int j) {
  int i, next;
  do {
    i = p[j];
    mateT[j] = i;
    next = mateS[i];
    mateS[i] = j;
    if (next != -1) j = next;
  } while (next != -1);
}

LL hungarian() {
  int nres = 0;
  for (int i = 0; i < N; i++)
    if (mateS[i] == -1) nres++;

  while (nres > 0) {
    for (int i = 0; i < N; i++) {
      m[i] = false;
      p[i] = -1;
```

```
52        slack[i] = INF;
53      }
54      bool aug = false;
55      Q.clear();
56      for (int i = 0; i < N; i++)
57        if (mateS[i] == -1) {
58          Q.push_back(i);
59          break;
60        }
61
62      do {
63        int i, j;
64        i = Q.front();
65        Q.pop_front();
66        m[i] = true;
67        j = 0;
68
69        while (!aug && j < N) {
70          if (mateS[i] != j) {
71            LL minSlack = u[i] + v[j] - W[i][j];
72            if (minSlack < slack[j]) {
73              slack[j] = minSlack;
74              p[j] = i;
75              if (slack[j] == 0) {
76                if (mateT[j] == -1) {
77                  augment(j);
78                  aug = true;
79                  nres--;
80                } else
81                  Q.push_back(mateT[j]);
82              }
83            }
84          }
85          j++;
86        }
87
88        if (!aug && Q.size() == 0) {
89          LL minSlack = INF;
90          for (int k = 0; k < N; k++)
91            if (slack[k] > 0) minSlack = min(minSlack, slack[k]);
92          for (int k = 0; k < N; k++)
93            if (m[k]) u[k] -= minSlack;
94
95          int x = -1;
96          bool X[MAXV];
97          for (int k = 0; k < N; k++)
98            if (slack[k] == 0)
99              v[k] += minSlack;
100           else {
101             slack[k] -= minSlack;
102             if (slack[k] == 0 && mateT[k] == -1) x = k;
103             if (slack[k] == 0)
```

```
104               X[k] = true;
105             else
106               X[k] = false;
107           }
108
109         if (x == -1) {
110           for (int k = 0; k < N; k++)
111             if (X[k]) Q.push_back(mateT[k]);
112         } else {
113           augment(x);
114           aug = true;
115           nres--;
116         }
117       }
118     } while (!aug);
119   }
120
121   LL ans = 0;
122   for (int i = 0; i < N; i++) ans += (u[i] + v[i]);
123   return ans;
124 }
125
126 void dynamicHungarian() {
127   char type[2];
128   LL w;
129   int i, j;
130
131   cin >> type;
132   if (type[0] == 'C') {
133     cin >> i >> j >> w;
134     if ((w < W[i][j]) && (mateS[i] == j)) {
135       W[i][j] = w;
136       if (mateS[i] != -1) {
137         mateT[mateS[i]] = -1;
138         mateS[i] = -1;
139       }
140     } else if ((w > W[i][j]) && (u[i] + v[j] < w)) {
141       W[i][j] = w;
142       u[i] = -INF;
143       for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
144       if (mateS[i] != j) {
145         mateT[mateS[i]] = -1;
146         mateS[i] = -1;
147       }
148     } else
149       W[i][j] = w;
150   } else if (type[0] == 'X') {
151     cin >> i;
152     for (int c = 0; c < N; c++) cin >> W[i][c];
153     if (mateS[i] != -1) {
154       mateT[mateS[i]] = -1;
155       mateS[i] = -1;
```

```
156     }
157     u[i] = -INF;
158     for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
159   } else if (type[0] == 'Y') {
160     cin >> j;
161     for (int r = 0; r < N; r++) cin >> W[r][j];
162     if (mateT[j] != -1) {
163       mateS[mateT[j]] = -1;
164       mateT[j] = -1;
165     }
166     v[j] = -INF;
167     for (int r = 0; r < N; r++) v[j] = max(v[j], W[r][j] - u[r]);
168   } else if (type[0] == 'A') {
169     i = j = N++;
170     u[i] = -INF;
171     for (int c = 0; c < N; c++) u[i] = max(u[i], W[i][c] - v[c]);
172     v[j] = -INF;
173     for (int r = 0; r < N; r++) v[j] = max(v[j], W[r][j] - u[r]);
174   } else if (type[0] == 'Q')
175     cout << hungarian() << '\n';
176 }
177
178 int main() {
179   cin.tie(nullptr)->sync_with_stdio(false);
180   readMatrix();
181   initHungarian();
182   LL ans = hungarian();
183   int M;
184   cin >> M;
185   while (M--) dynamicHungarian();
186   return 0;
187 }
```

转化为费用流模型

与 二分图最大匹配→11.24.2@p2959 类似，二分图的最大权匹配也可以转化为网络流问题来求解.

首先，在图中新增一个源点和一个汇点.

从源点向二分图的每个左部点连一条流量为 1，费用为 0 的边，从二分图的每个右部点向汇点连一条流量为 1，费用为 0 的边.

接下来对于二分图中每一条连接左部点 u 和右部点 v，边权为 w 的边，则连一条从 u 到 v，流量为 1，费用为 w 的边.

另外，考虑到最大权匹配下，匹配边的数量不一定与最大匹配的匹配边数量相等，因此对于每个左部点，还需向汇点连一条流量为 1，费用为 0 的边.

求这个网络的 最大费用最大流→11.23.4@p2935 即可得到答案. 此时，该网络的最大流量一定为左部点的数量，而最大流量下的最大费用即对应一个最大权匹配方案.

习题

UOJ #80. 二分图最大权匹配[4]

模板题

```cpp
#include <iostream>
#include <limits>
#include <queue>
#include <vector>
using namespace std;

template <typename T>
struct hungarian {  // km
  int n;
  vector<int> matchx, matchy, pre;
  vector<bool> visx, visy;
  vector<T> lx, ly;
  vector<vector<T>> g;
  vector<T> slack;
  T inf, res;
  queue<int> q;
  int org_n, org_m;

  hungarian(int _n, int _m) {
    org_n = _n;
    org_m = _m;
    n = max(_n, _m);
    inf = numeric_limits<T>::max();
    res = 0;
    g = vector<vector<T>>(n, vector<T>(n));
    matchx = vector<int>(n, -1);
    matchy = vector<int>(n, -1);
    pre = vector<int>(n);
    visx = vector<bool>(n);
    visy = vector<bool>(n);
    lx = vector<T>(n, -inf);
    ly = vector<T>(n);
    slack = vector<T>(n);
  }

  void addEdge(int u, int v, int w) {
    g[u][v] = max(w, 0);  // 负值还不如不匹配 因此设为0不影响
  }

  bool check(int v) {
    visy[v] = true;
    if (matchy[v] != -1) {
      q.push(matchy[v]);
      visx[matchy[v]] = true;
      return false;
    }
    while (v != -1) {
      matchy[v] = pre[v];
```

```
      swap(v, matchx[pre[v]]);
    }
    return true;
  }

  void bfs(int i) {
    while (!q.empty()) {
      q.pop();
    }
    q.push(i);
    visx[i] = true;
    while (true) {
      while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < n; v++) {
          if (!visy[v]) {
            T delta = lx[u] + ly[v] - g[u][v];
            if (slack[v] >= delta) {
              pre[v] = u;
              if (delta) {
                slack[v] = delta;
              } else if (check(v)) {
                return;
              }
            }
          }
        }
      }
      // 没有增广路 修改顶标
      T a = inf;
      for (int j = 0; j < n; j++) {
        if (!visy[j]) {
          a = min(a, slack[j]);
        }
      }
      for (int j = 0; j < n; j++) {
        if (visx[j]) {  // S
          lx[j] -= a;
        }
        if (visy[j]) {  // T
          ly[j] += a;
        } else {  // T'
          slack[j] -= a;
        }
      }
      for (int j = 0; j < n; j++) {
        if (!visy[j] && slack[j] == 0 && check(j)) {
          return;
        }
      }
    }
```

```
  }

  void solve() {
    // 初始顶标
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        lx[i] = max(lx[i], g[i][j]);
      }
    }

    for (int i = 0; i < n; i++) {
      fill(slack.begin(), slack.end(), inf);
      fill(visx.begin(), visx.end(), false);
      fill(visy.begin(), visy.end(), false);
      bfs(i);
    }

    // custom
    for (int i = 0; i < n; i++) {
      if (g[i][matchx[i]] > 0) {
        res += g[i][matchx[i]];
      } else {
        matchx[i] = -1;
      }
    }
    cout << res << "\n";
    for (int i = 0; i < org_n; i++) {
      cout << matchx[i] + 1 << " ";
    }
    cout << "\n";
  }
};

int main() {
  ios::sync_with_stdio(false), cin.tie(nullptr);
  int n, m, e;
  cin >> n >> m >> e;

  hungarian<long long> solver(n, m);

  int u, v, w;
  for (int i = 0; i < e; i++) {
    cin >> u >> v >> w;
    u--, v--;
    solver.addEdge(u, v, w);
  }
  solver.solve();
}
```

外部参考资料

[1] The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs

[2] A Fast Dynamic Assignment Algorithm for Solving Resource Allocation Problems

[3] DAP

[4] UOJ #80. 二分图最大权匹配

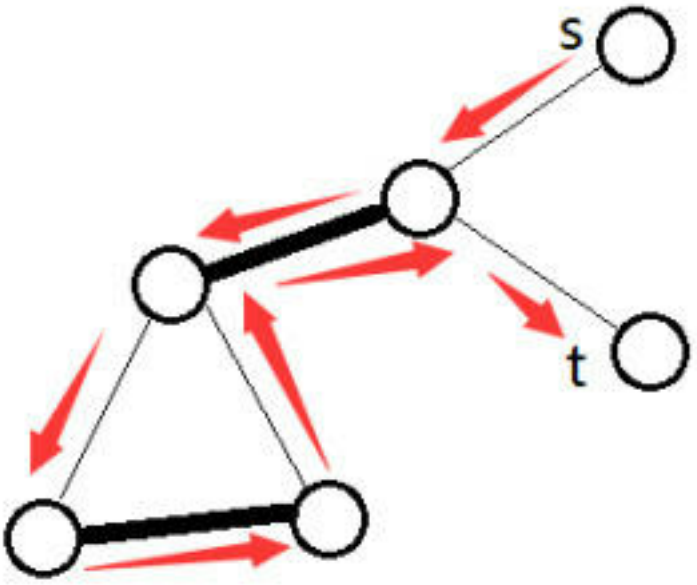
11.24.4 一般图最大匹配

Authors: H-J-Granger, accelsao, Ir1d, Early0v0, Henry-ZHR, HeliumOI, AntiLeaf, ShizuhaAki, pukui

带花树算法（Blossom Algorithm）

开花算法（Blossom Algorithm，也被称做带花树）可以解决一般图最大匹配问题（maximum cardinality matchings）. 此算法由 Jack Edmonds 在 1961 年提出. 经过一些修改后也可以解决一般图最大权匹配问题. 此算法是第一个给出证明说最大匹配有多项式复杂度.

一般图匹配和二分图匹配（bipartite matching）不同的是，图可能存在奇环.



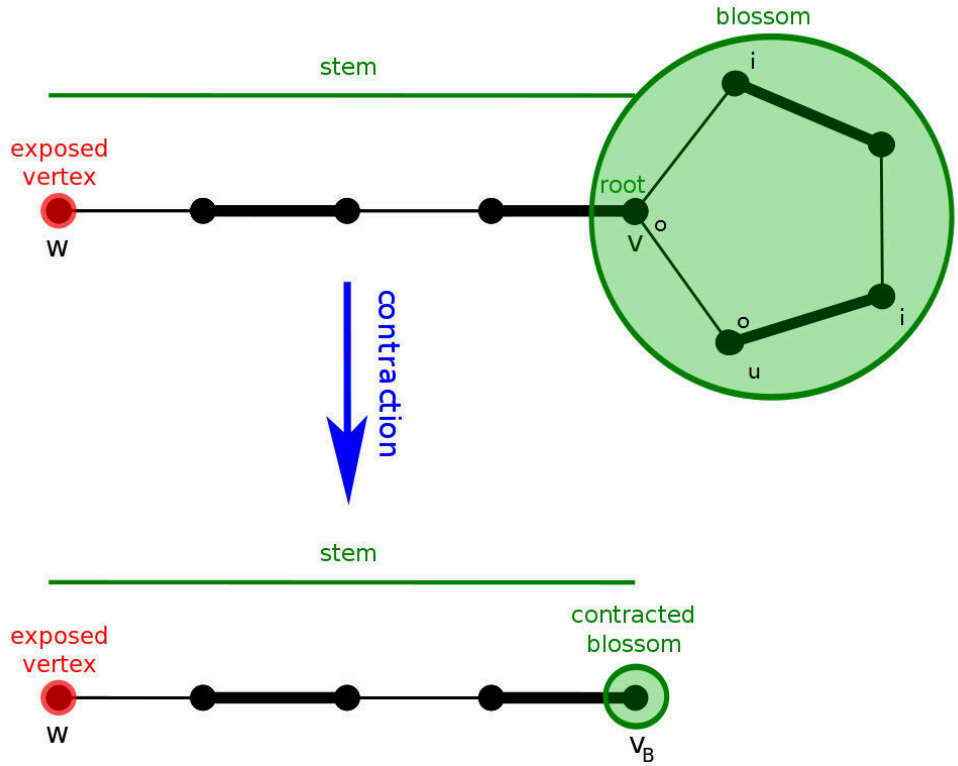
以此图为例，若直接取反（匹配边和未匹配边对调），会使得取反后的 M 不合法，某些点会出现在两条匹配上，而问题就出在奇环.

下面考虑一般图的增广算法. 从二分图的角度出发，每次枚举一个未匹配点，设出发点为根，标记为「o」，接下来交错标记「o」和「i」，不难发现「i」到「o」这段边是匹配边.

假设当前点是 v，相邻点为 u，可以分为以下两种情况：

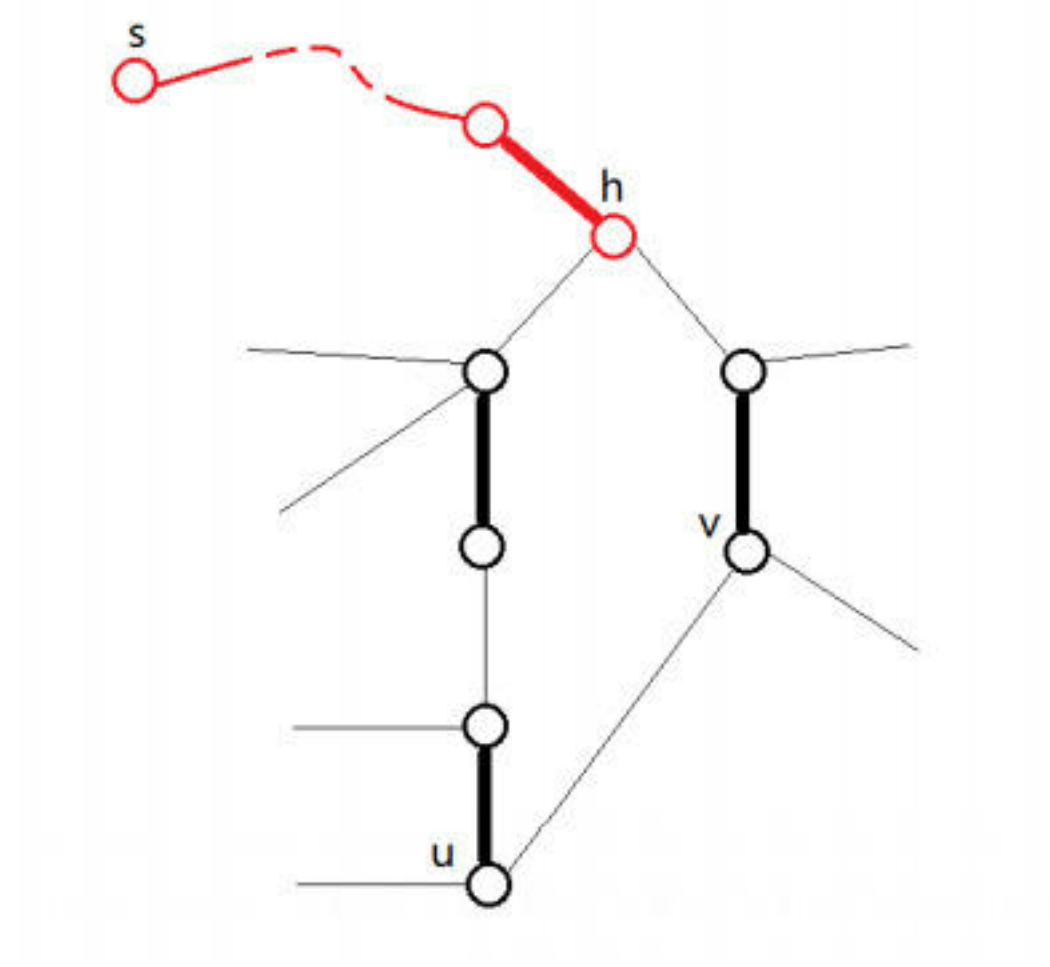
1. u 未拜访过，当 u 是未匹配点，则找到增广路径，否则从 u 的配偶找增广路.
2. u 已拜访过，遇到标记「o」代表需要 **缩花**，否则代表遇到偶环，跳过.

遇到偶环的情况，将他视为二分图解决，故可忽略. **缩花** 后，再新图中继续找增广路.



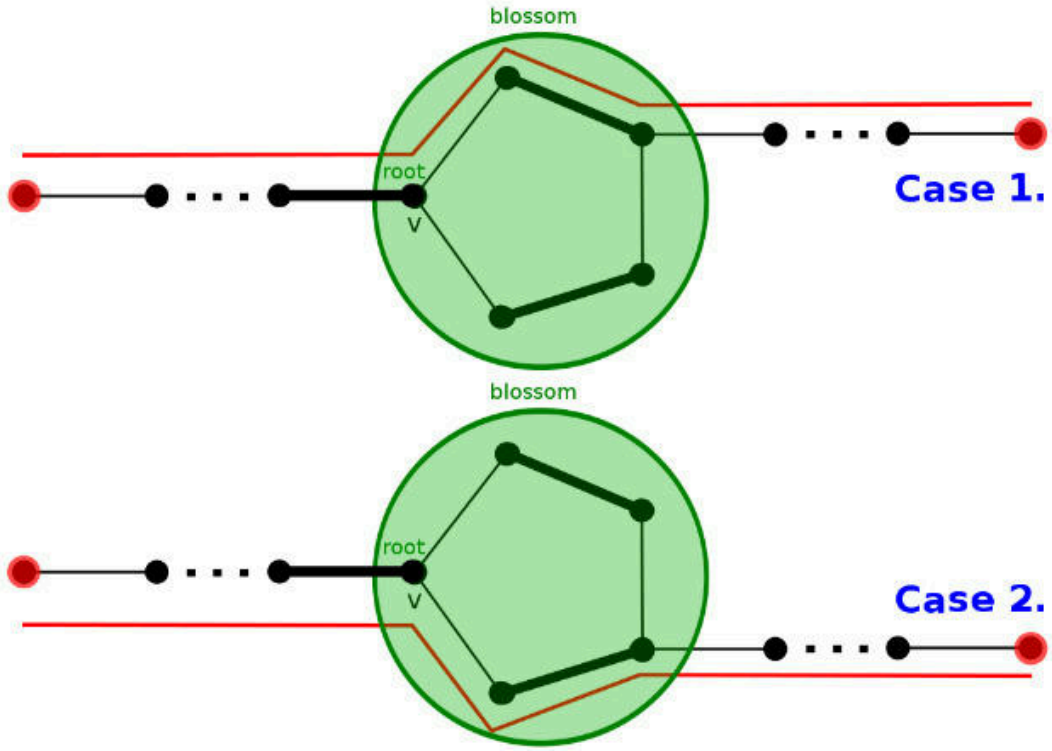
设原图为 G，**缩花** 后的图为 G'，我们只需要证明：

1. 若 G 存在增广路，G' 也存在.
2. 若 G' 存在增广路，G 也存在.



设非树边（形成环的那条边）为 (u,v)，定义花根 $h=LCA(u,v)$. 奇环是交替的，有且仅有 h 的两条邻边类型相同，都是非匹配边. 那么进入 h 的树边肯定是匹配边，环上除了 h 以外其他点往环外的边都是非匹配边.

观察可知，从环外的边出去有两种情况，顺时针或逆时针.



于是 **缩花** 与 **不缩花** 都不影响正确性.

实作上找到 **花** 以后我们不需要真的 **缩花**，可以用数组纪录每个点在以哪个点为根的那朵花中.

复杂度分析 Complexity Analysis

每次找增广路，遍历所有边，遇到 **花** 会维护 **花** 上的点，$O(|E|^2)$.

枚举所有未匹配点做增广路，总共 $O(|V||E|^2)$.

参考代码

参考代码

```cpp
// graph
template <typename T>
class graph {
 public:
  struct edge {
    int from;
    int to;
    T cost;
  };

  vector<edge> edges;
  vector<vector<int>> g;
  int n;

  graph(int _n) : n(_n) { g.resize(n); }

  virtual int add(int from, int to, T cost) = 0;
};
```

```cpp

// undirectedgraph
template <typename T>
class undirectedgraph : public graph<T> {
 public:
  using graph<T>::edges;
  using graph<T>::g;
  using graph<T>::n;

  undirectedgraph(int _n) : graph<T>(_n) {}

  int add(int from, int to, T cost = 1) {
    assert(0 <= from && from < n && 0 <= to && to < n);
    int id = (int)edges.size();
    g[from].push_back(id);
    g[to].push_back(id);
    edges.push_back({from, to, cost});
    return id;
  }
};

// blossom / find_max_unweighted_matching
template <typename T>
vector<int> find_max_unweighted_matching(const undirectedgraph<T> &g) {
  std::mt19937 rng(std::random_device{}());
  vector<int> match(g.n, -1);   // 匹配
  vector<int> aux(g.n, -1);     // 时间戳记
  vector<int> label(g.n);       // 「o」或「i」
  vector<int> orig(g.n);        // 花根
  vector<int> parent(g.n, -1);  // 父节点
  queue<int> q;
  int aux_time = -1;

  auto lca = [&](int v, int u) {
    aux_time++;
    while (true) {
      if (v != -1) {
        if (aux[v] == aux_time) {  // 找到拜访过的点 也就是LCA
          return v;
        }
        aux[v] = aux_time;
        if (match[v] == -1) {
          v = -1;
        } else {
          v = orig[parent[match[v]]];  // 以匹配点的父节点继续寻找
        }
      }
      swap(v, u);
    }
  };  // lca

  auto blossom = [&](int v, int u, int a) {
```