图论及其应用

hzwer,miskcoo

北京大学, 清华大学

2016 年 8 月 18 日

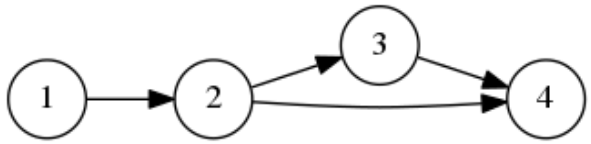
图论及其应用

hzwer,miskcoo

北京大学, 清华大学

2016 年 8 月 18 日

图的矩阵存储



图可以直接用二维数组来存储。具体来说，如果一张图有 n 个结点，那么就使用 n × n 的二维数组来存储。

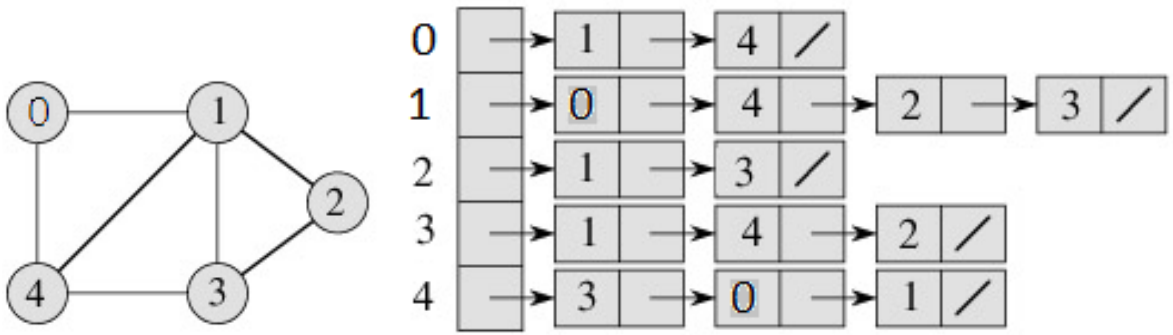
数组的每个元素的值代表了对应的边的数量。例如上面这张图就可以存储如下：

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

这种方式通常用来存储十分稠密的图，可以比较快地定位到某一条边。

图的链表存储

图还可以用邻接表来存储。也就是每个结点维护一个链表，这个链表存储着以当前结点为开头的边。



通常情况下我们都是用这种方法来存储图的。

一些概念

一些概念

- **顶点的度**：在无向图中，某个顶点的度是与它相关联的边的数目。在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。

一些概念

- **顶点的度**：在无向图中，某个顶点的度是与它相关联的边的数目。在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。
- **简单路径**：顶点不重复的路径。

一些概念

- **顶点的度**：在无向图中，某个顶点的度是与它相关联的边的数目。在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。
- **简单路径**：顶点不重复的路径。
- **自环**：从某个顶点出发连向它自身的边。

一些概念

- **顶点的度**：在无向图中，某个顶点的度是与它相关联的边的数目。在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。
- **简单路径**：顶点不重复的路径。
- **自环**：从某个顶点出发连向它自身的边。
- **环**：从某个顶点出发再回到自身的路径，又称**回路**。

一些概念

- **顶点的度**：在无向图中，某个顶点的度是与它相关联的边的数目。在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。
- **简单路径**：顶点不重复的路径。
- **自环**：从某个顶点出发连向它自身的边。
- **环**：从某个顶点出发再回到自身的路径，又称**回路**。
- **重边**：从一个顶点到另一个顶点有两条边直接相连。

一些概念

在无向图中，若从顶点 u 到 v 存在路径，那么称顶点 u 和 v 是**连通的**。如果无向图中任意一对顶点都是连通的，那么称此图为**连通图**。如果一个无向图不是连通的，则称它的一个极大连通子图为**连通分量**。这里的极大是指顶点个数极大。

一些概念

在无向图中，若从顶点 u 到 v 存在路径，那么称顶点 u 和 v 是**连通的**。如果无向图中任意一对顶点都是连通的，那么称此图为**连通图**。如果一个无向图不是连通的，则称它的一个极大连通子图为**连通分量**。这里的极大是指顶点个数极大。

在有向图中，如果每一对顶点 u 和 v，既存在从 u 到 v 的路径，又存在从 v 到 u 的路径，那么称此图为**强连通图**。对于非强连通图，其极大强连通子图成为其**强连通分量**。

一些概念

在无向图中，若从顶点 u 到 v 存在路径，那么称顶点 u 和 v 是**连通的**。如果无向图中任意一对顶点都是连通的，那么称此图为**连通图**。如果一个无向图不是连通的，则称它的一个极大连通子图为**连通分量**。这里的极大是指顶点个数极大。

在有向图中，如果每一对顶点 u 和 v，既存在从 u 到 v 的路径，又存在从 v 到 u 的路径，那么称此图为**强连通图**。对于非强连通图，其极大强连通子图成为其**强连通分量**。

在有向图中，若不考虑边的方向时图才为连通图，那么称原有向图为**弱连通**。

拓扑排序

现在来考虑一个问题：现在有一个工程，这个工程被分成了很多部分。有一些部分要求前面某些部分完成后才可以开始进行。有些部分则可以同时进行。

拓扑排序

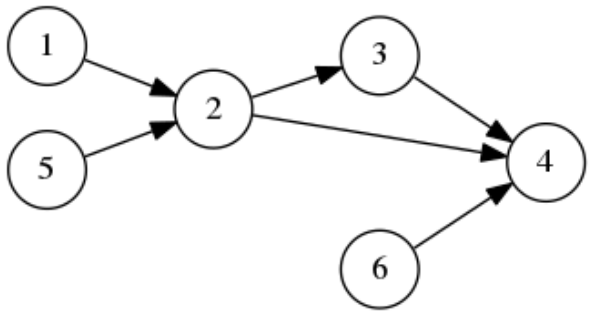
现在来考虑一个问题：现在有一个工程，这个工程被分成了很多部分。有一些部分要求前面某些部分完成后才可以开始进行。有些部分则可以同时进行。

我们可以把每个部分看作一个结点，刚刚这些限制看成是有向边。

拓扑排序

现在来考虑一个问题：现在有一个工程，这个工程被分成了很多部分。有一些部分要求前面某些部分完成后才可以开始进行。有些部分则可以同时进行。

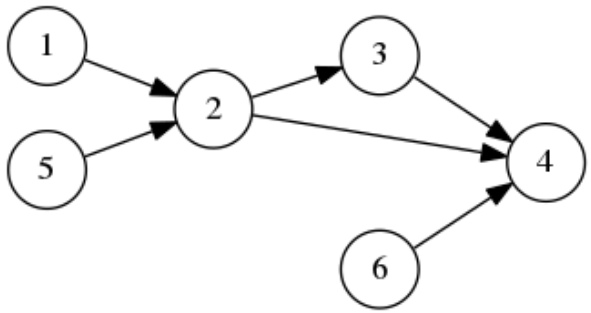
我们可以把每个部分看作一个结点，刚刚这些限制看成是有向边。



拓扑排序

现在来考虑一个问题：现在有一个工程，这个工程被分成了很多部分。有一些部分要求前面某些部分完成后才可以开始进行。有些部分则可以同时进行。

我们可以把每个部分看作一个结点，刚刚这些限制看成是有向边。



比如说这张图就可以看成是这样一个限制。这样的图有个特点：没有环！

因此这样的图也被成为 Directed Acyclic Graph（有向无环图）。

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序的过程大概是这样的：

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序的过程大概是这样的：

1. 选择一个入度为 0 的结点并直接输出。

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序的过程大概是这样的：

1. 选择一个入度为 0 的结点并直接输出。
2. 删除这个结点以及与它关联的所有边。

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序的过程大概是这样的：

1. 选择一个入度为 0 的结点并直接输出。
2. 删除这个结点以及与它关联的所有边。
3. 重复步骤 (1) 和 (2)，直到找不到入度为 0 的结点。

拓扑排序

求出一个这个工程的工作序列的算法被成为**拓扑排序**。

比如说 1，5，2，3，6，4 就可以算作一个工作序列。

拓扑排序的过程大概是这样的：

1. 选择一个入度为 0 的结点并直接输出。
2. 删除这个结点以及与它关联的所有边。
3. 重复步骤 (1) 和 (2)，直到找不到入度为 0 的结点。

通常情况下，在实现的时候会维护一个队列以及每个结点的入度。在删除边的时候顺便把相应结点的入度减去，当这个结点入度为 0 的时候直接将其加入队列。

例题 1

小明要去一个国家旅游。这个国家有 n 个城市，编号为 1 n，并且有 m 条道路连接着，小明准备从其中一个城市出发，并只往东走到城市 i 停止。

所以他就需要选择最先到达的城市，并制定一条路线以城市 i 为终点，使得线路上除了第一个城市，每个城市都在路线前一个城市东面，并且满足这个前提下还希望游览的城市尽量多。

现在，你只知道每一条道路所连接的两个城市的相对位置关系，但并不知道所有城市具体的位置。现在对于所有的 i，都需要你为小明制定一条路线，并求出以城市 i 为终点最多能够游览多少个城市。

其中 $1 \leq n \leq 100000, 1 \leq m \leq 200000$。

例题 1

最长路

例题 1

最长路

DAG

例题 1

最长路

DAG

拓扑排序的过程中直接 DP

例题 2 CF698B

给出 n 个结点的父亲，问至少修改多少个结点的父亲，能使整张图变成一棵树（根的父亲为自己），要求输出任一方案。

其中 $1 \leq n \leq 200000$。

例题 2 CF698B

思考环和链的答案

例题 2 CF698B

思考环和链的答案

图的各个弱连通块是环 + 内向树，或者树/环。

例题 2 CF698B

思考环和链的答案

图的各个弱连通块是环 + 内向树，或者树/环。

先用拓扑排序把内向树消掉

例题 2 CF698B

思考环和链的答案

图的各个弱连通块是环 + 内向树，或者树/环。

先用拓扑排序把内向树消掉

剩下来的是一些环，每个环随便选一个结点当根，然后再把所有的根连在一起。

例题 2 CF698B

思考环和链的答案

图的各个弱连通块是环 + 内向树，或者树/环。

先用拓扑排序把内向树消掉

剩下来的是一些环，每个环随便选一个结点当根，然后再把所有的根连在一起。

答案是环数-（是否存在自环）

最小生成树

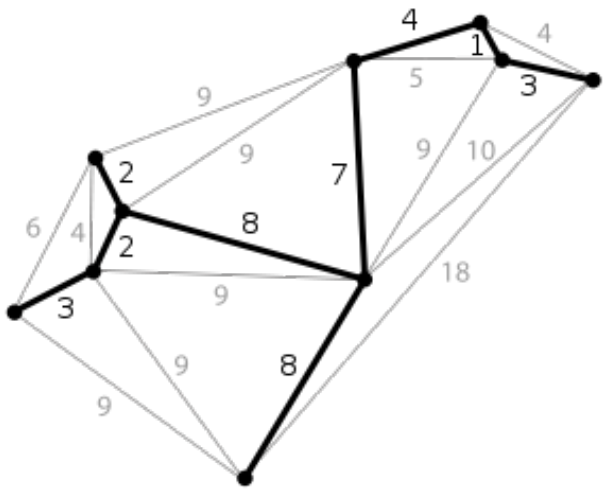
生成树：无向连通图 G 的一个子图如果是包含 G 的所有顶点的树，那么就称这个子图为 G 的生成树。

我们称生成树各边权值和为该树的**权**。对于无向连通图来说，权最小的生成树被成为**最小生成树**。

最小生成树

生成树：无向连通图 G 的一个子图如果是包含 G 的所有顶点的树，那么就称这个子图为 G 的生成树。

我们称生成树各边权值和为该树的**权**。对于无向连通图来说，权最小的生成树被成为**最小生成树**。



Kruskal 算法

Kruskal 算法是能够在 $\mathcal{O}(\mathrm{m}\log\mathrm{m})$ 的时间内得到一个最小生成树的算法。它主要是基于贪心的思想：

Kruskal 算法

Kruskal 算法是能够在 $\mathcal{O}(m \log m)$ 的时间内得到一个最小生成树的算法。它主要是基于贪心的思想：

1. 将边按照边权从小到大排序，并建立一个没有边的图 T。

Kruskal 算法

Kruskal 算法是能够在 $\mathcal{O}(m \log m)$ 的时间内得到一个最小生成树的算法。它主要是基于贪心的思想：

1. 将边按照边权从小到大排序，并建立一个没有边的图 T。
2. 选出一条没有被选过的边权最小的边。

Kruskal 算法

Kruskal 算法是能够在 $\mathcal{O}(m \log m)$ 的时间内得到一个最小生成树的算法。它主要是基于贪心的思想：

1. 将边按照边权从小到大排序，并建立一个没有边的图 T。
2. 选出一条没有被选过的边权最小的边。
3. 如果这条边两个顶点在 T 中所在的连通块不相同，那么将它加入图 T。

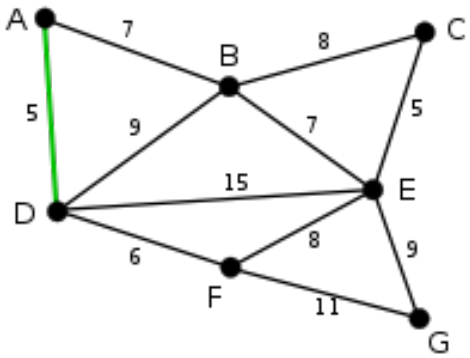
Kruskal 算法

Kruskal 算法是能够在 $\mathcal{O}(m \log m)$ 的时间内得到一个最小生成树的算法。它主要是基于贪心的思想：

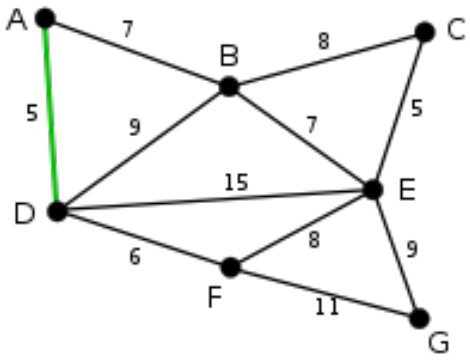
1. 将边按照边权从小到大排序，并建立一个没有边的图 T。
2. 选出一条没有被选过的边权最小的边。
3. 如果这条边两个顶点在 T 中所在的连通块不相同，那么将它加入图 T。
4. 重复 (2) 和 (3) 直到图 T 连通为止。

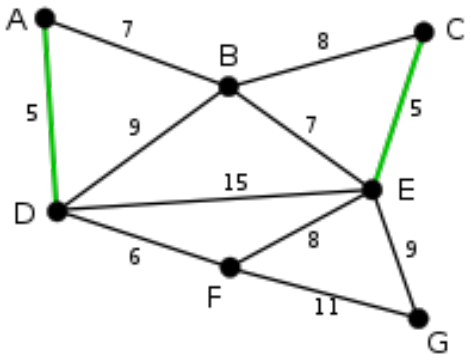
由于只需要维护连通性，可以不需要真正建立图 T，可以用并查集来维护。

Kruskal 算法

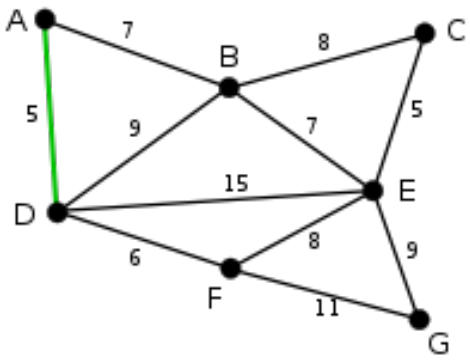


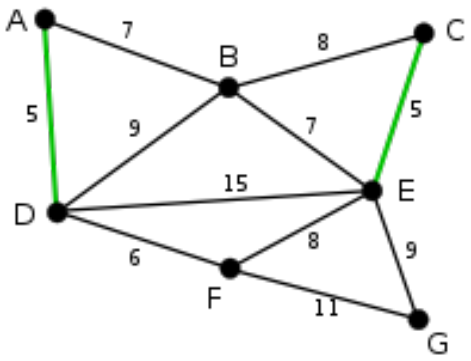
Kruskal 算法

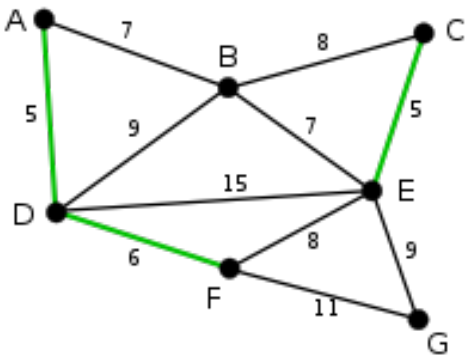




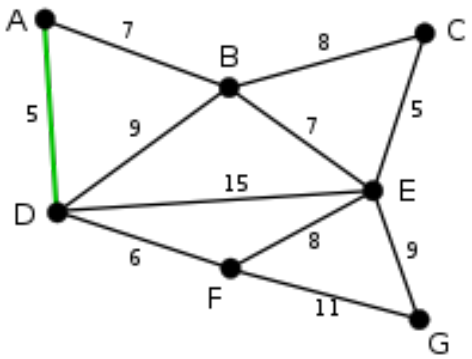
Kruskal 算法

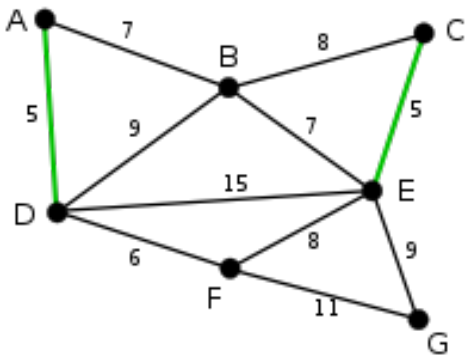

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11

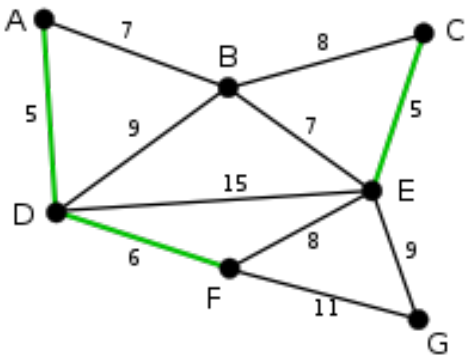

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11

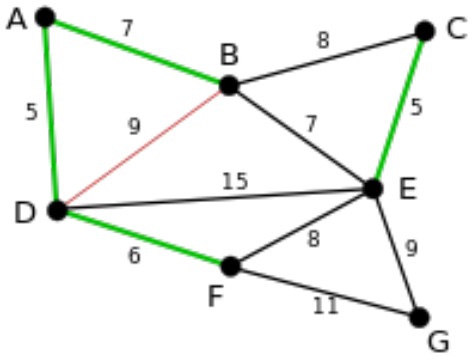

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11

Kruskal 算法

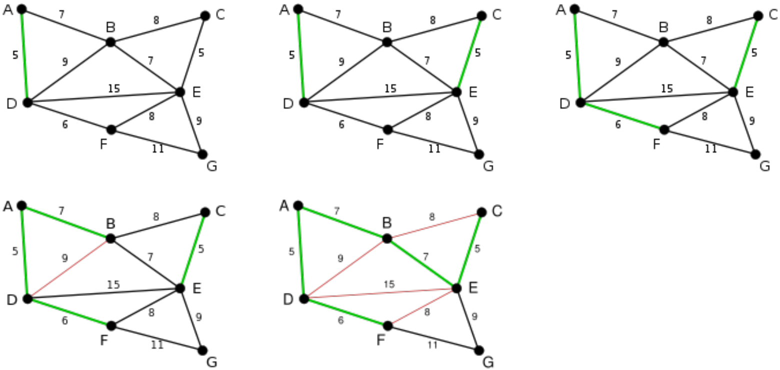

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11

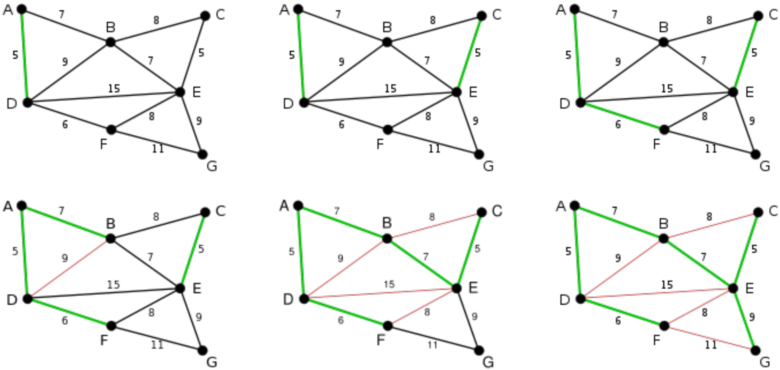

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11

Kruskal 算法



Kruskal 算法


A B C D E F G 7 8 5 9 7 15 5 6 8 9 11
A B C D E F G 7 8 5 9 7 15 5 6 8 9 11
A B C D E F G 7 8 5 9 7 15 5 6 8 9 11
A B C D E F G 7 8 5 9 7 15 5 6 8 9 11
A B C D E F G 7 8 5 9 7 15 5 6 8 9 11
A B C D E F G 7 8 5 9 7 15 5 6 8 9 11

Prim 算法

Prim 算法和 Kruskal 算法一样也是寻找最小生成树的一种方法。

Prim 算法

Prim 算法和 Kruskal 算法一样也是寻找最小生成树的一种方法。

1. 先建立一个只有一个结点的树，这个结点可以是原图中任意的一个结点。

Prim 算法

Prim 算法和 Kruskal 算法一样也是寻找最小生成树的一种方法。

1. 先建立一个只有一个结点的树，这个结点可以是原图中任意的一个结点。
2. 使用一条边扩展这个树，要求这条边一个顶点在树中另一个顶点不在树中，并且这条边的权值要求最小。

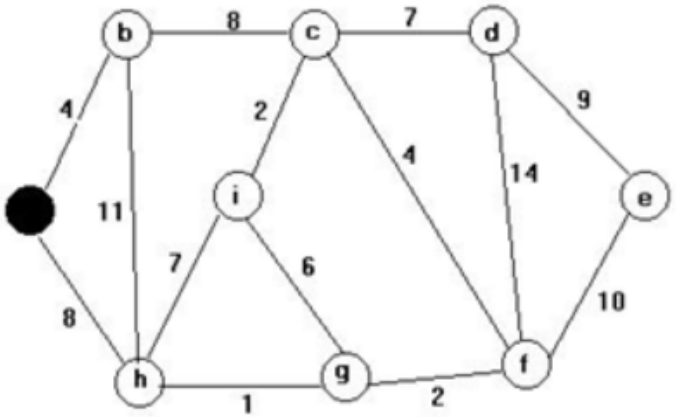
Prim 算法

Prim 算法和 Kruskal 算法一样也是寻找最小生成树的一种方法。

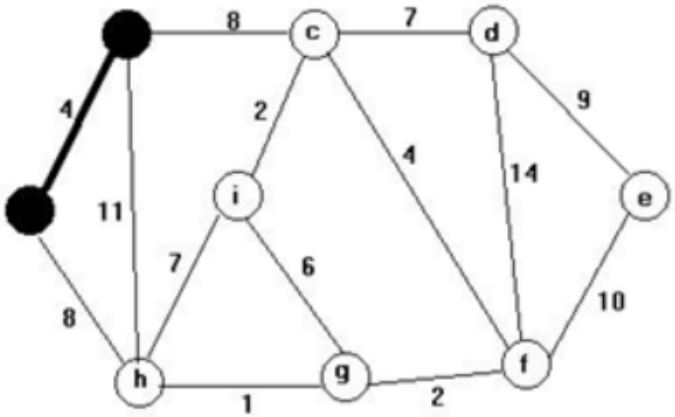
1. 先建立一个只有一个结点的树，这个结点可以是原图中任意的一个结点。
2. 使用一条边扩展这个树，要求这条边一个顶点在树中另一个顶点不在树中，并且这条边的权值要求最小。
3. 重复步骤 (2) 直到所有顶点都在树中。

Prim 算法

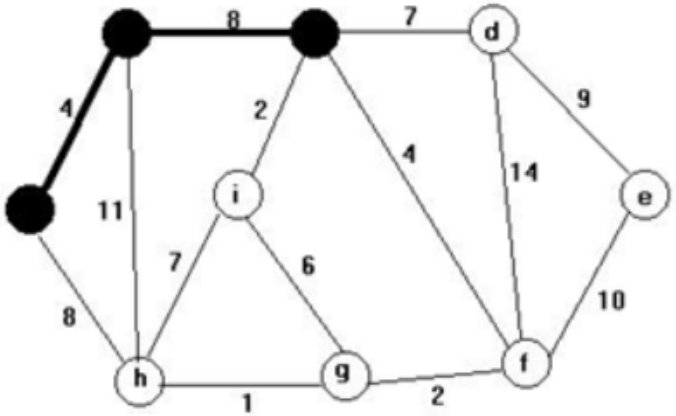
Iteration 0:



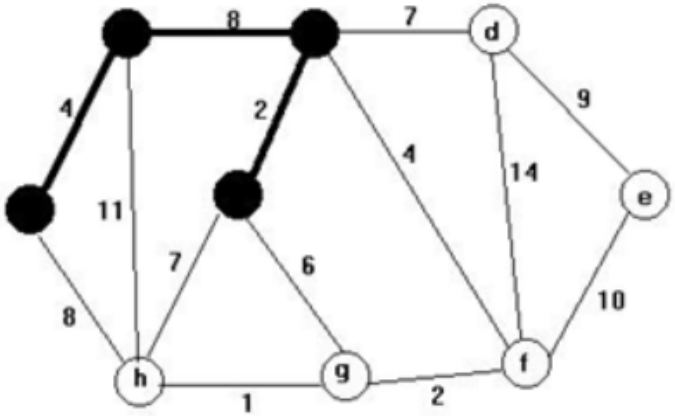
Iteration 1:



Iteration 2:

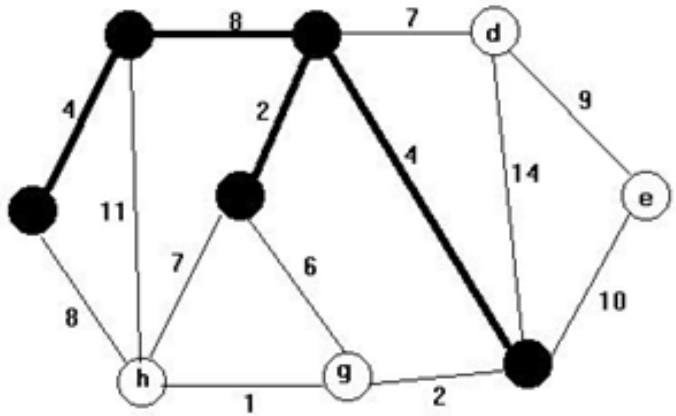


Iteration 3:

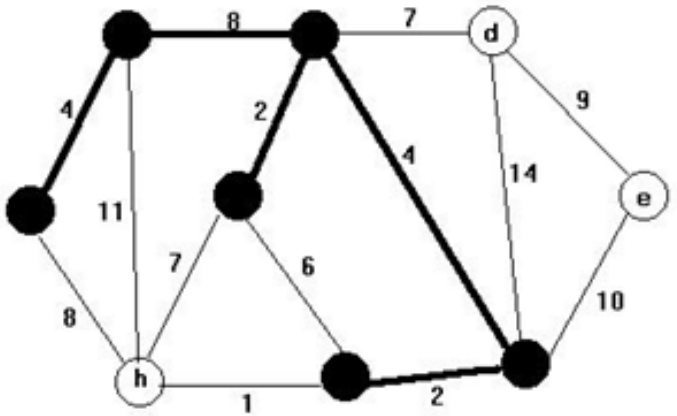


Prim 算法

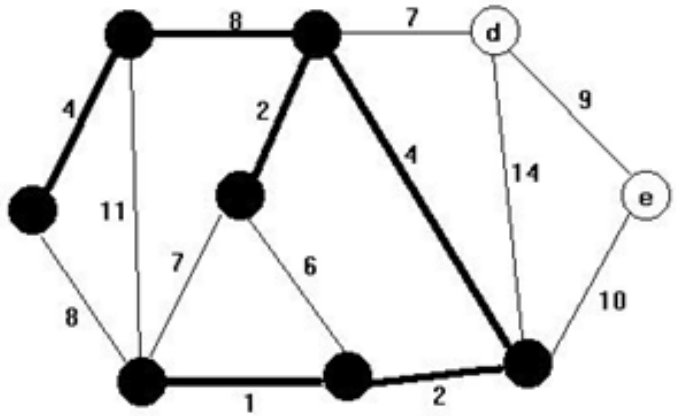
Iteration 4:



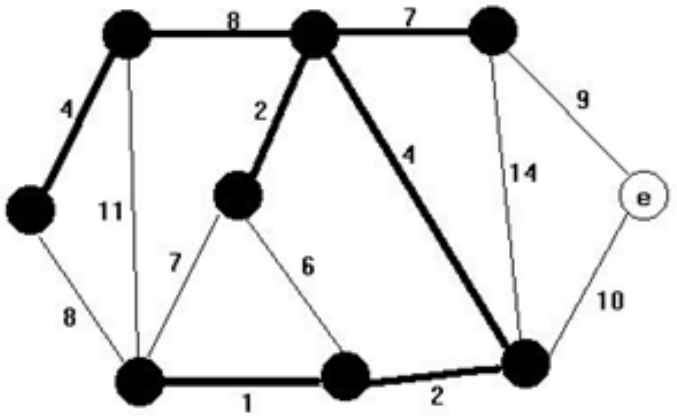
Iteration 5:



Iteration 6:

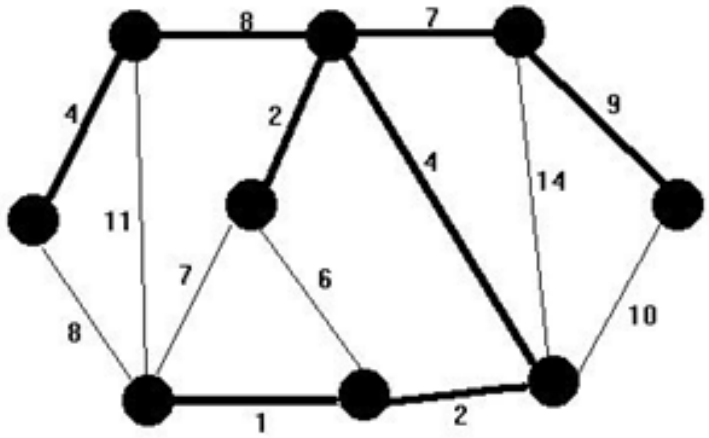


Iteration 7:

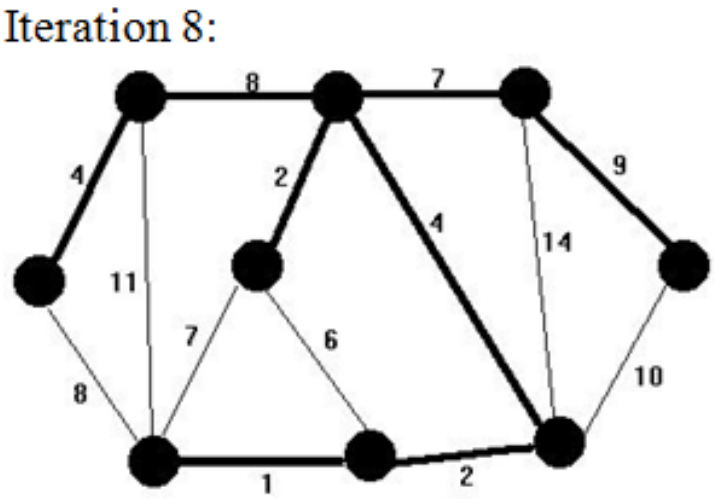


Prim 算法

Iteration 8:

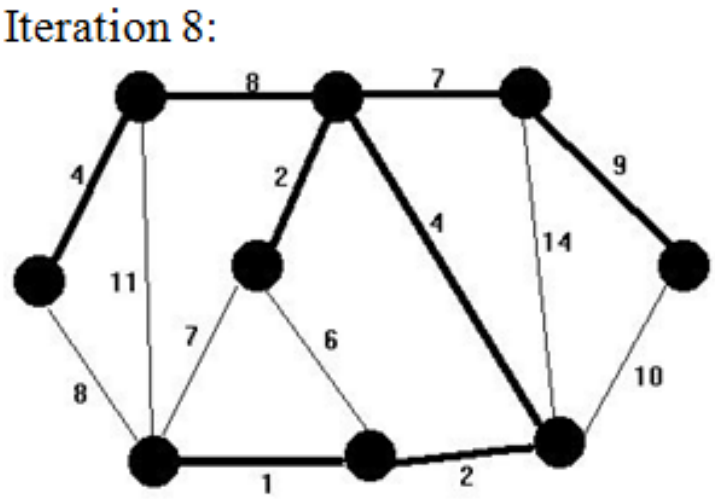


Prim 算法



具体实现的时候，可以为每个结点维护一个 key 值，表示它到已经选中的顶点中权值最小的边的权值。每次就在所有没有选中的顶点中找到 key 值最小的顶点，以及与它相关联的那条边加入树中并且更新与这个顶点相关联的所有顶点的 key 值。

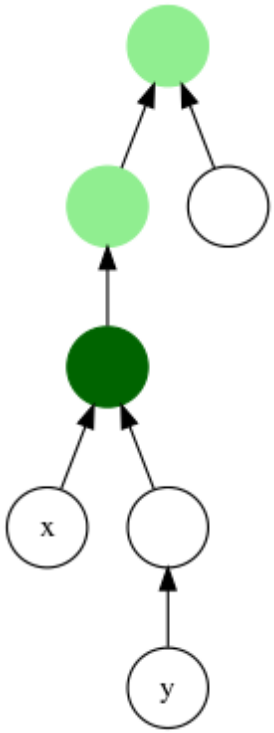
Prim 算法



具体实现的时候，可以为每个结点维护一个 key 值，表示它到已经选中的顶点中权值最小的边的权值。每次就在所有没有选中的顶点中找到 key 值最小的顶点，以及与它相关联的那条边加入树中并且更新与这个顶点相关联的所有顶点的 key 值。

这是可以用堆维护的，它的复杂度是 $\mathcal{O}(m \log n)$。

最近公共祖先



在一棵树上，两个结点的**最近公共祖先**（LCA）是它们的公共祖先中深度最大的那个顶点。

比如说，在左边的树中，顶点 x 和 y 的公共祖先用绿色标出，其中深绿色的顶点就是它们的最近公共祖先。

倍增算法

常见的求最近公共祖先的算法是**倍增算法**。

倍增算法

常见的求最近公共祖先的算法是**倍增算法**。

首先对于每个结点先进行 DFS 预处理出它的深度，再记录下它们往父亲方向走 $2^0, 2^1, 2^2, \cdots, 2^k$ 步所到达的结点。在这里 2^k 大于整棵树的最大深度。

倍增算法

常见的求最近公共祖先的算法是**倍增算法**。

首先对于每个结点先进行 DFS 预处理出它的深度，再记录下它们往父亲方向走 $2^0, 2^1, 2^2, \cdots, 2^k$ 步所到达的结点。在这里 2^k 大于整棵树的最大深度。

预处理完后，需要查询两个点 u 和 v 的 LCA 的时候，先将 u 和 v 中深度较大的一个利用先前处理出的数组走到和另一个结点相同的深度，这的所需要的操作次数不会超过 $\log_2 \left|\mathrm{depth}(u) - \mathrm{depth}(v)\right|$。

倍增算法

常见的求最近公共祖先的算法是**倍增算法**。

首先对于每个结点先进行 DFS 预处理出它的深度，再记录下它们往父亲方向走 $2^0, 2^1, 2^2, \cdots, 2^k$ 步所到达的结点。在这里 2^k 大于整棵树的最大深度。

预处理完后，需要查询两个点 u 和 v 的 LCA 的时候，先将 u 和 v 中深度较大的一个利用先前处理出的数组走到和另一个结点相同的深度，这的所需要的操作次数不会超过 $\log_2 \left|\mathrm{depth}(u) - \mathrm{depth}(v)\right|$。

接下来从 k 开始往下枚举，如果 u 和 v 如果往上走 2^i 后不相同，那么就将它们一起往上走这么多步。

倍增算法

常见的求最近公共祖先的算法是**倍增算法**。

首先对于每个结点先进行 DFS 预处理出它的深度，再记录下它们往父亲方向走 $2^0, 2^1, 2^2, \cdots, 2^k$ 步所到达的结点。在这里 2^k 大于整棵树的最大深度。

预处理完后，需要查询两个点 u 和 v 的 LCA 的时候，先将 u 和 v 中深度较大的一个利用先前处理出的数组走到和另一个结点相同的深度，这的所需要的操作次数不会超过 $\log_2 \left|\mathrm{depth}(u) - \mathrm{depth}(v)\right|$。

接下来从 k 开始往下枚举，如果 u 和 v 如果往上走 2^i 后不相同，那么就将它们一起往上走这么多步。

结束后如果 u 和 v 仍然不相等，再往上走一步。最后的顶点就是它们的 LCA。

倍增算法

miskcoo@bw:~/Code/oicode

```
4  int get_lca(int u, int v)
5  {
6      if(depth[u] < depth[v])
7          swap(u, v);
8      int d = depth[u] - depth[v];
9      for(int l = 0; d; ++l, d >>= 1)
10         if(d & 1) u = dist[l][u];
11
12     for(int p = MaxL - 1; u != v; p ? --p : 0)
13     {
14         if(dist[p][u] != dist[p][v] || p == 0)
15         {
16             u = dist[p][u];
17             v = dist[p][v];
18         }
19     }
20
21     return u;
22 }
```

倍增算法

倍增算法预处理的复杂度预处理是 $\mathcal{O}(n \log n)$，每次查询都是 $\mathcal{O}(\log n)$。

倍增算法

倍增算法预处理的复杂度预处理是 $\mathcal{O}(n \log n)$，每次查询都是 $\mathcal{O}(\log n)$。

不仅如此，我们可以动态地给树增加一些叶子结点。

倍增算法

倍增算法预处理的复杂度预处理是 $\mathcal{O}(n \log n)$，每次查询都是 $\mathcal{O}(\log n)$。

不仅如此，我们可以动态地给树增加一些叶子结点。

此外，在预处理的时候还可以顺便记录下这段路径的权值最大值，最小值或者权值和之类的信息，这样就可以在 $\mathcal{O}(\log n)$ 的时间内求出树上两点间路径权值的最大值、最小值还有权值和。

NOIP2013. 货车运输

A 国有 n 座城市，编号从 1 到 n，城市之间有 m 条双向道路。每一条道路对车辆都有重量限制，简称限重。现在有 q 辆货车在运输货物，每辆货车需要从一个城市运输到另一个城市。司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

其中 $0 < \mathrm{n} < 10^4, 0 < \mathrm{m} < 5 \times 10^4, 0 < \mathrm{q} < 3 \times 10^4$

NOIP2013. 货车运输

A 国有 n 座城市，编号从 1 到 n，城市之间有 m 条双向道路。每一条道路对车辆都有重量限制，简称限重。现在有 q 辆货车在运输货物，每辆货车需要从一个城市运输到另一个城市。司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

其中 $0 < \mathrm{n} < 10^4, 0 < \mathrm{m} < 5 \times 10^4, 0 < \mathrm{q} < 3 \times 10^4$

最大生成树

NOIP2013. 货车运输

A 国有 n 座城市，编号从 1 到 n，城市之间有 m 条双向道路。每一条道路对车辆都有重量限制，简称限重。现在有 q 辆货车在运输货物，每辆货车需要从一个城市运输到另一个城市。司机们想知道每辆车在不超过车辆限重的情况下，最多能运多重的货物。

其中 $0 < \mathrm{n} < 10^4, 0 < \mathrm{m} < 5 \times 10^4, 0 < \mathrm{q} < 3 \times 10^4$

最大生成树、倍增算路径最小值

例题 1

给出一个有 n 个结点，m 条边的无向图，判断其最小生成树是否唯一。

其中 $0 < n < 10^4, 0 < m < 10^6$。

例题 1

给出一个有 n 个结点，m 条边的无向图，判断其最小生成树是否唯一。

其中 $0 < \mathrm{n} < 10^4, 0 < \mathrm{m} < 10^6$。

MST 什么时候唯一？

例题 1

给出一个有 n 个结点，m 条边的无向图，判断其最小生成树是否唯一。

其中 $0 < \mathrm{n} < 10^4, 0 < \mathrm{m} < 10^6$。

MST 什么时候唯一？

加入一条非树边会形成环

例题 1

给出一个有 n 个结点，m 条边的无向图，判断其最小生成树是否唯一。

其中 $0 < n < 10^4, 0 < m < 10^6$。

MST 什么时候唯一？

加入一条非树边会形成环

找到环上除了新加入的边外权值最大的边，该边权值小于这条非树边

例题 2 BZOJ1977

给出一个有 n 个结点，m 条边的无向图，求其严格次小生成树。

其中 $0 < \mathrm{n} < 10^5, 0 < \mathrm{m} < 3 * 10^5$。

例题 2 BZOJ1977

给出一个有 n 个结点，m 条边的无向图，求其严格次小生成树。

其中 $0 < \mathrm{n} < 10^5, 0 < \mathrm{m} < 3 * 10^5$。

最小生成树和严格次小的区别？

例题 2 BZOJ1977

给出一个有 n 个结点，m 条边的无向图，求其严格次小生成树。

其中 $0 < \mathrm{n} < 10^5, 0 < \mathrm{m} < 3 * 10^5$。

最小生成树和严格次小的区别？

用非树边替换最小生成树的一条边

例题 2 BZOJ1977

给出一个有 n 个结点，m 条边的无向图，求其严格次小生成树。

其中 $0 < \mathrm{n} < 10^5, 0 < \mathrm{m} < 3 * 10^5$。

最小生成树和严格次小的区别？

用非树边替换最小生成树的一条边

枚举每一条非树边找俩顶点树链上的最大边（如果最大边相同与非树边边权相同则找次大边）然后更新最小增量

例题 2 BZOJ1977

给出一个有 n 个结点，m 条边的无向图，求其严格次小生成树。

其中 $0 < n < 10^5, 0 < m < 3 * 10^5$。

最小生成树和严格次小的区别？

用非树边替换最小生成树的一条边

枚举每一条非树边找俩顶点树链上的最大边（如果最大边相同与非树边边权相同则找次大边）然后更新最小增量

最大边和次大边可以通过树上倍增求出

Bellman-Ford 算法

给定了一个边带有权值（可以为负数）的有向图（不包含负环）和一个指定的顶点 s。要求求出从 s 到其余各点的最短路径长度。

Bellman-Ford 算法

给定了一个边带有权值（可以为负数）的有向图（不包含负环）和一个指定的顶点 s。要求求出从 s 到其余各点的最短路径长度。

Bellman-Ford 算法是一个比较直观的求解单源最段路问题的算法。

Bellman-Ford 算法

给定了一个边带有权值（可以为负数）的有向图（不包含负环）和一个指定的顶点 s。要求求出从 s 到其余各点的最短路径长度。

Bellman-Ford 算法是一个比较直观的求解单源最段路问题的算法。

我们可以肯定最短路径包含的边的条数不会超过 n-1 个，如果超过这个数，那么肯定形成了一个环，又因为这个环权值是正的，我们可以将路径上这个环删除，路径长度就会变小。

Bellman-Ford 算法

这个算法主要是构造一个最短路径长度数组的序列：dist$[1][u]$, dist$[2][u]$, $\cdots$, dist$[n-1][u]$。

其中 dist$[k][u]$ 表示从源 s 到 u **至多**经过 k 条边的最短路径的长度。

Bellman-Ford 算法

这个算法主要是构造一个最短路径长度数组的序列：dist$[1][u]$, dist$[2][u]$, $\cdots$, dist$[n-1][u]$。

其中 dist$[k][u]$ 表示从源 s 到 u **至多**经过 k 条边的最短路径的长度。

显然我们可以得到这样的关系：

Bellman-Ford 算法

这个算法主要是构造一个最短路径长度数组的序列：dist$[1][u]$, dist$[2][u]$, $\cdots$, dist$[n-1][u]$。

其中 dist$[k][u]$ 表示从源 s 到 u **至多**经过 k 条边的最短路径的长度。

显然我们可以得到这样的关系：

$$\text{dist}[1][u] = w[s][u]$$

Bellman-Ford 算法

这个算法主要是构造一个最短路径长度数组的序列：dist$[1][u]$, dist$[2][u]$, $\cdots$, dist$[n-1][u]$。

其中 dist$[k][u]$ 表示从源 s 到 u **至多**经过 k 条边的最短路径的长度。

显然我们可以得到这样的关系：

$$dist[1][u] = w[s][u]$$

$$dist[k][u] = \min(dist[k-1][u], \min_{(v,u)\in E}(dist[k-1][v] + w[v][u]))$$

Bellman-Ford 算法

这个算法主要是构造一个最短路径长度数组的序列：dist$[1][u]$, dist$[2][u]$, $\cdots$, dist$[n-1][u]$。

其中 dist$[k][u]$ 表示从源 s 到 u **至多**经过 k 条边的最短路径的长度。

显然我们可以得到这样的关系：

$$\text{dist}[1][u] = w[s][u]$$

$$\text{dist}[k][u] = \min(\text{dist}[k-1][u], \min_{(v,u)\in E}(\text{dist}[k-1][v] + w[v][u]))$$

由于每次计算 dist 数组复杂度都是 $\mathcal{O}(|E|)$ 的，总的复杂度就是 $\mathcal{O}(|V||E|)$。

SPFA 算法

事实上，你会发现我们每次进行的计算可以看成是一次“松弛”。假设我们现在已经得到了 Bellman-Ford 算法某个阶段的 dist 数组，然后我们发现了一条 s 到 u 的距离比 dist[u] 更加短的路径。我们更新了 dist[u]。

SPFA 算法

事实上，你会发现我们每次进行的计算可以看成是一次“松弛”。假设我们现在已经得到了 Bellman-Ford 算法某个阶段的 dist 数组，然后我们发现了一条 s 到 u 的距离比 dist[u] 更加短的路径。我们更新了 dist[u]。

那么接下来直接受到影响的就是与 u 直接关联的顶点 v，也就是如果 $\mathrm{dist}[u] + w[u][v] < \mathrm{dist}[v]$ 的话，s 到 v 的最短路就可以利用 s 到 u 的最短路加上 u 到 v 的边来更新。这样的话与 v 直接关联的顶点又会受到影响……不断这样持续下去直到最后没有顶点能被影响。

SPFA 算法

事实上，你会发现我们每次进行的计算可以看成是一次“松弛”。假设我们现在已经得到了 Bellman-Ford 算法某个阶段的 dist 数组，然后我们发现了一条 s 到 u 的距离比 dist[u] 更加短的路径。我们更新了 dist[u]。

那么接下来直接受到影响的就是与 u 直接关联的顶点 v，也就是如果 $\mathrm{dist}[u] + w[u][v] < \mathrm{dist}[v]$ 的话，s 到 v 的最短路就可以利用 s 到 u 的最短路加上 u 到 v 的边来更新。这样的话与 v 直接关联的顶点又会受到影响……不断这样持续下去直到最后没有顶点能被影响。

那么一个优化就是我们利用队列存储这些需要更新的结点，每次从队列中取出一个结点，计算是否有结点需要更新，如果有，并且这个结点不在队列中，那么就将它加入队列。

SPFA 算法

事实上，你会发现我们每次进行的计算可以看成是一次“松弛”。假设我们现在已经得到了 Bellman-Ford 算法某个阶段的 dist 数组，然后我们发现了一条 s 到 u 的距离比 dist[u] 更加短的路径。我们更新了 dist[u]。

那么接下来直接受到影响的就是与 u 直接关联的顶点 v，也就是如果 $\mathrm{dist}[u] + w[u][v] < \mathrm{dist}[v]$ 的话，s 到 v 的最短路就可以利用 s 到 u 的最短路加上 u 到 v 的边来更新。这样的话与 v 直接关联的顶点又会受到影响……不断这样持续下去直到最后没有顶点能被影响。

那么一个优化就是我们利用队列存储这些需要更新的结点，每次从队列中取出一个结点，计算是否有结点需要更新，如果有，并且这个结点不在队列中，那么就将它加入队列。

这样的算法被称为 SPFA——一种优化的 Bellman-Ford 算法。

SPFA 算法

在竞赛中大多数人会选择用 SPFA 作为单源最段路的算法，主要原因在于它比较好写，而且通常情况下跑得比较快。但是 SPFA 的复杂度实际上是没有保证的，最坏情况基本和 Bellman-Ford 相同，但是最好的时候可以到达 $\mathcal{O}(|\mathsf{V}| + |\mathsf{E}|)$。

SPFA 算法

在竞赛中大多数人会选择用 SPFA 作为单源最段路的算法，主要原因在于它比较好写，而且通常情况下跑得比较快。但是 SPFA 的复杂度实际上是没有保证的，最坏情况基本和 Bellman-Ford 相同，但是最好的时候可以到达 $\mathcal{O}(|V| + |E|)$。

非常重要的一点就是 SPFA 的队列需要使用**循环队列**，虽然最多队列里只会有 n 各点，但是每个点可能会入队多次。

SPFA 算法

miskcoo@bw:~/Code/oicode

```
34 void spfa()
35 {
36     std::memset(dist, 77, sizeof(dist));
37     int qhead = 0, qtail = 0;
38     dist[1] = 0, mark[1] = 1;
39     que[qtail++] = 1;
40     while(qhead != qtail)
41     {
42         int u = que[qhead++];
43         if(qhead == MaxN) qhead = 0;
44         for(int k = gp.head[u]; k; k = gp.next[k])
45         {
46             int v = gp.point[k];
47             int d = dist[u] + gp.weight[k];
48             if(d < dist[v])
49             {
50                 dist[v] = d;
51                 if(!mark[v])
52                 {
53                     mark[v] = 1;
54                     que[qtail++] = v;
55                     if(qtail == MaxN) qtail = 0;
56                 }
57             }
58         }
59
60         mark[u] = 0;
61     }
```

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|\mathrm{E}|\log|\mathrm{V}|)$。

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|\mathrm{E}|\log|\mathrm{V}|)$。

事实上，Dijkstra 算法的思想和 Prim 有很多类似之处。Dijkstra 算法维护了一个未访问的结点集合 T 以及一个从 s 到结点 u 的当前距离 dist[u]。

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|E| \log |V|)$。

事实上，Dijkstra 算法的思想和 Prim 有很多类似之处。Dijkstra 算法维护了一个未访问的结点集合 T 以及一个从 s 到结点 u 的当前距离 dist[u]。

1. 将除源外所有结点当前距离设置为 ∞，将源 s 的当前距离设置为 0，将当前节点设置为源 s。

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|E| \log |V|)$。

事实上，Dijkstra 算法的思想和 Prim 有很多类似之处。Dijkstra 算法维护了一个未访问的结点集合 T 以及一个从 s 到结点 u 的当前距离 dist[u]。

1. 将除源外所有结点当前距离设置为 ∞，将源 s 的当前距离设置为 0，将当前节点设置为源 s。
2. 从当前结点 u 开始，找出所有在未访问集合 T 中与 u 有边 (u,v) 的结点 v。如果 dist[u]+w[u][v]<dist[v]，那么就更新 dist[v] 的值。

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|E| \log |V|)$。

事实上，Dijkstra 算法的思想和 Prim 有很多类似之处。Dijkstra 算法维护了一个未访问的结点集合 T 以及一个从 s 到结点 u 的当前距离 dist[u]。

1. 将除源外所有结点当前距离设置为 ∞，将源 s 的当前距离设置为 0，将当前节点设置为源 s。
2. 从当前结点 u 开始，找出所有在未访问集合 T 中与 u 有边 (u,v) 的结点 v。如果 dist[u]+w[u][v]<dist[v]，那么就更新 dist[v] 的值。
3. 将当前节点从 T 中删除，并且找到在 T 中 dist 最小的结点设置为新的当前节点。

Dijkstra 算法

如果有向图的边权值全为正数，那么有一种复杂度有保证的单源最段路算法——Dijkstra 算法。它的复杂度是 $\mathcal{O}(|E| \log |V|)$。

事实上，Dijkstra 算法的思想和 Prim 有很多类似之处。Dijkstra 算法维护了一个未访问的结点集合 T 以及一个从 s 到结点 u 的当前距离 dist[u]。

1. 将除源外所有结点当前距离设置为 ∞，将源 s 的当前距离设置为 0，将当前节点设置为源 s。
2. 从当前结点 u 开始，找出所有在未访问集合 T 中与 u 有边 (u,v) 的结点 v。如果 dist[u]+w[u][v]<dist[v]，那么就更新 dist[v] 的值。
3. 将当前节点从 T 中删除，并且找到在 T 中 dist 最小的结点设置为新的当前节点。
4. 重复 (2) 和 (3) 直到 T 成为空集。

Dijkstra 算法

它的正确性在于，在未访问集合 T 中结点的 dist 是从 s 开始经过已经访问集合中的结点到达它的最短路。

Dijkstra 算法

它的正确性在于，在未访问集合 T 中结点的 dist 是从 s 开始经过已经访问集合中的结点到达它的最短路。

如果选出的当前结点 u 的 dist 不是最终的最小值，那么它最终的最短路一定是要经过一个此时 T 中的其它结点再到 u。这时那个结点的 dist 肯定要小于 u 的 dist，这就和 u 是 dist 最小的结点矛盾了！

Floyd 算法

如果你要求所有顶点间的最短路，当然可以对每个顶点跑单源最短路。但是，有一个更为直接的 Floyd 算法。

Floyd 算法

如果你要求所有顶点间的最短路，当然可以对每个顶点跑单源最短路。但是，有一个更为直接的 Floyd 算法。

我们设 d[k][i][j] 为除了 i 和 j 外只经过前 k 个结点，从 i 到 j 的最短路。

Floyd 算法

如果你要求所有顶点间的最短路，当然可以对每个顶点跑单源最短路。但是，有一个更为直接的 Floyd 算法。

我们设 d[k][i][j] 为除了 i 和 j 外只经过前 k 个结点，从 i 到 j 的最短路。

显然可以知道 d[0][i][j] = w[i][j]。

Floyd 算法

如果你要求所有顶点间的最短路，当然可以对每个顶点跑单源最短路。但是，有一个更为直接的 Floyd 算法。

我们设 d[k][i][j] 为除了 i 和 j 外只经过前 k 个结点，从 i 到 j 的最短路。

显然可以知道 d[0][i][j] = w[i][j]。

那么当加入了一个顶点 k 之后，最短路如果有变化的话一定是以 k 为中间顶点，那么可以得到

$$d[k][i][j] = \min(d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j])$$

Floyd 算法

如果你要求所有顶点间的最短路，当然可以对每个顶点跑单源最短路。但是，有一个更为直接的 Floyd 算法。

我们设 d[k][i][j] 为除了 i 和 j 外只经过前 k 个结点，从 i 到 j 的最短路。

显然可以知道 d[0][i][j] = w[i][j]。

那么当加入了一个顶点 k 之后，最短路如果有变化的话一定是以 k 为中间顶点，那么可以得到

$$\mathrm{d[k][i][j]} = \min(\mathrm{d[k-1][i][j]}, \mathrm{d[k-1][i][k]} + \mathrm{d[k-1][k][j]})$$

这个算法的复杂度是 $\mathcal{O}(|\mathrm{V}|^3)$。

例题 1 虫洞

给出由 n 个虫洞的质量，其由 m 条单向边连接。

虫洞之间跃迁需要一定燃料和 1 个单位时间，每过 1 单位时间虫洞反色。

从白洞到黑洞，消耗的燃料减少 delta，反之燃料增加 delta（delta 为两虫洞质量差）。

可以选择在一个结点花费 s_i 的时间停留一个单位时间。

求从虫洞 1 到 n 最小的燃料消耗。

其中 $0 < n < 5000, 0 < m < 30000$。

每个点拆成黑白两个

每个点拆成黑白两个

由于每秒虫洞变色，所以黑点之间边权为 v + delta

每个点拆成黑白两个

由于每秒虫洞变色，所以黑点之间边权为 v + delta

其余连边同理

每个点拆成黑白两个

由于每秒虫洞变色，所以黑点之间边权为 v + delta

其余连边同理

求最短路

例题 2 BZOJ2143

给出两个 $n*m$ 的矩阵 A，B，以及 3 个人的坐标

在 (i,j) 支付 $A_{i,j}$ 的费用可以弹射到曼哈顿距离不超过 $B_{i,j}$ 的位置

问三个人汇合所需要的最小总费用

其中 $0 < n, m < 150, 0 < A < 1000, 0 < B < 10^9$。

这道题点很少，但是边可能很多，直接建图做最短路显然不可行

这道题点很少，但是边可能很多，直接建图做最短路显然不可行

把弹射看成获得了可以走 $a_{i,j}$ 的能量

这道题点很少，但是边可能很多，直接建图做最短路显然不可行

把弹射看成获得了可以走 $a_{i,j}$ 的能量

每走一格消耗 1 的能量，$f(i,j,k)$ 表示在 (i,j) 且有 k 的能量的最少费用

这道题点很少，但是边可能很多，直接建图做最短路显然不可行

把弹射看成获得了可以走 $a_{i,j}$ 的能量

每走一格消耗 1 的能量，$f(i,j,k)$ 表示在 (i,j) 且有 k 的能量的最少费用

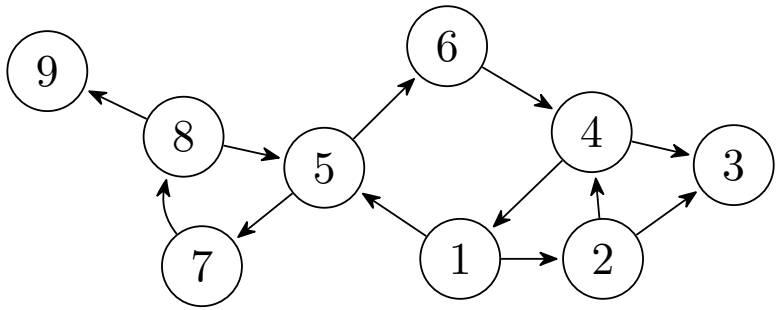
求三次最短路，枚举汇合点

有向图的强连通分量

在图论中，一个有向图被成为是**强连通的**当且仅当每一对不相同结点 u 和 v 间既存在从 u 到 v 的路径也存在从 v 到 u 的路径。有向图的极大强连通子图（这里指点数极大）被称为**强连通分量**。

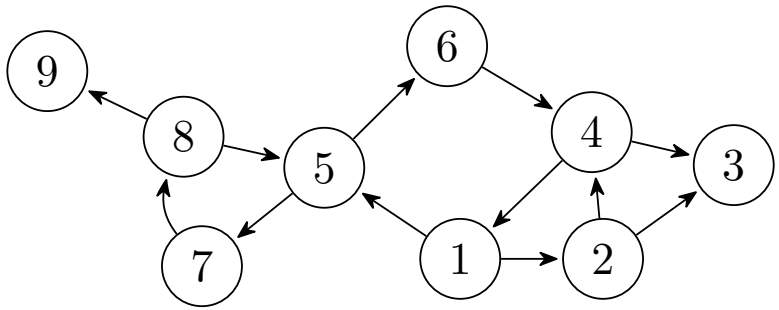
有向图的强连通分量

在图论中，一个有向图被成为是**强连通的**当且仅当每一对不相同结点 u 和 v 间既存在从 u 到 v 的路径也存在从 v 到 u 的路径。有向图的极大强连通子图（这里指点数极大）被称为**强连通分量**。



有向图的强连通分量

在图论中，一个有向图被成为是**强连通的**当且仅当每一对不相同结点 u 和 v 间既存在从 u 到 v 的路径也存在从 v 到 u 的路径。有向图的极大强连通子图（这里指点数极大）被称为**强连通分量**。



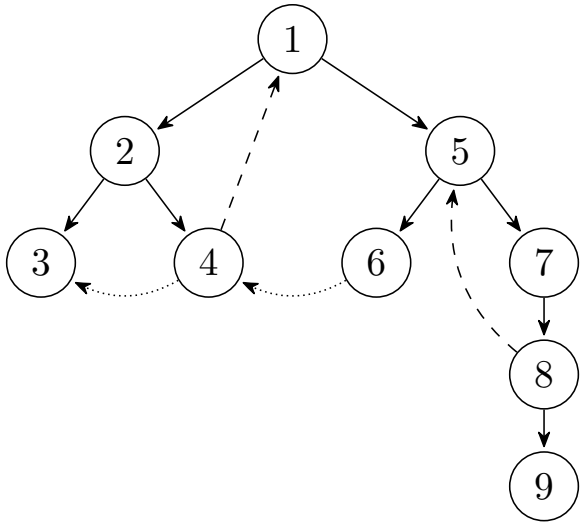
比如说这个有向图中，点 $1, 2, 4, 5, 6, 7, 8$ 和相应边组成的子图就是一个强连通分量，另外点 $3, 9$ 单独构成强连通分量。

搜索树

在介绍求解强连通分量的算法之前先来介绍一下搜索树。

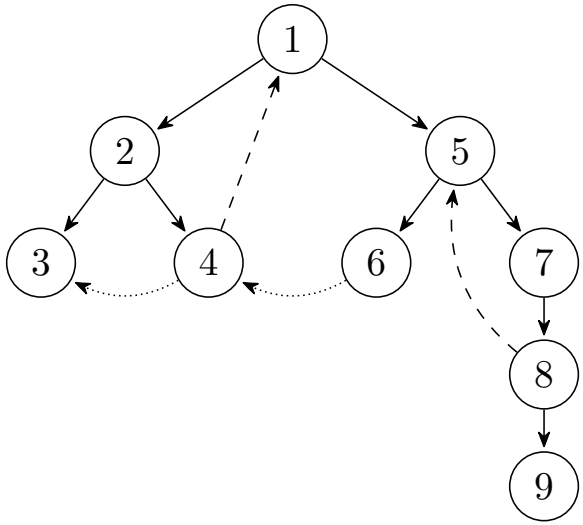
搜索树

在介绍求解强连通分量的算法之前先来介绍一下搜索树。



搜索树

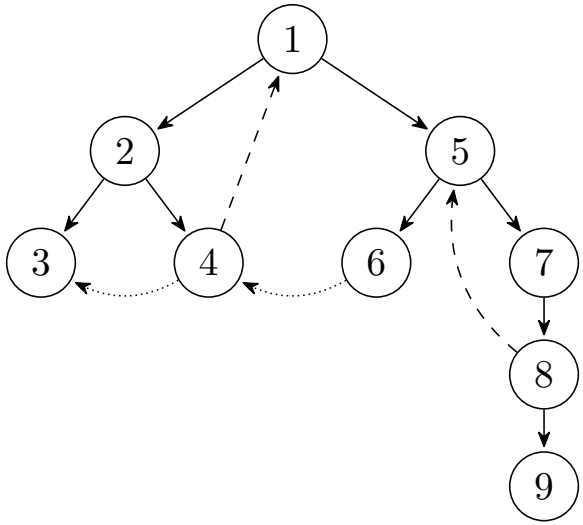
在介绍求解强连通分量的算法之前先来介绍一下搜索树。



有向图的搜索树主要有 4 种边（这张图只有三种），其中用实线画出来的是**树边**（tree edge），每次搜索找到一个还没有访问过的结点的时候这就形成了一条树边。

搜索树

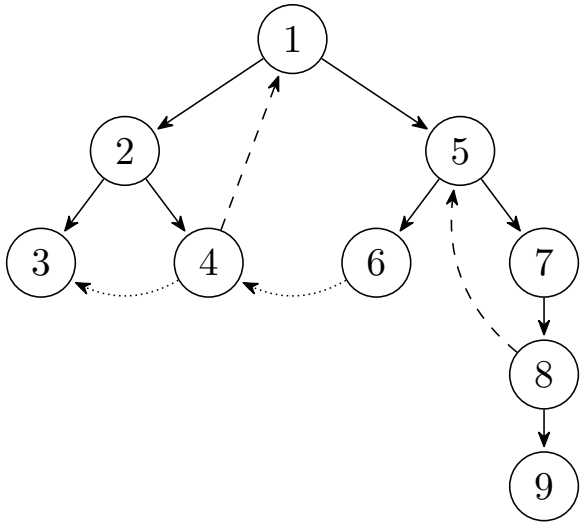
在介绍求解强连通分量的算法之前先来介绍一下搜索树。



有向图的搜索树主要有 4 种边（这张图只有三种），其中用实线画出来的是**树边**（tree edge），每次搜索找到一个还没有访问过的结点的时候这就形成了一条树边。用长虚线画出来的是**反祖边**（back edge），也被叫做**回边**。

搜索树

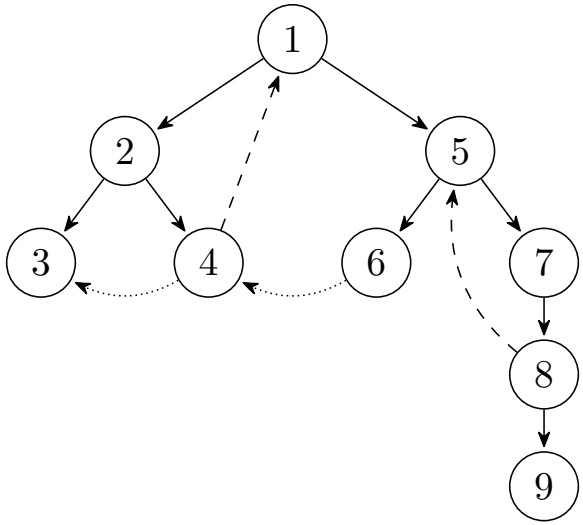
在介绍求解强连通分量的算法之前先来介绍一下搜索树。



有向图的搜索树主要有 4 种边（这张图只有三种），其中用实线画出来的是**树边**（tree edge），每次搜索找到一个还没有访问过的结点的时候这就形成了一条树边。用长虚线画出来的是**反祖边**（back edge），也被叫做**回边**。用短虚线画出来的是**横叉边**（cross edge），它主要是在搜索的时候遇到了一个已经访问过的结点，但是这个结点并不是当前节点的祖先时形成

搜索树

在介绍求解强连通分量的算法之前先来介绍一下搜索树。



有向图的搜索树主要有 4 种边（这张图只有三种），其中用实线画出来的是**树边**（tree edge），每次搜索找到一个还没有访问过的结点的时候这就形成了一条树边。用长虚线画出来的是**反祖边**（back edge），也被叫做**回边**。用短虚线画出来的是**横叉边**（cross edge），它主要是在搜索的时候遇到了一个已经访问过的结点，但是这个结点并不是当前节点的祖先时形成

Tarjan 算法

Tarjan 算法是由 Robert Tarjan 提出的用于寻找有向图的强连通分量的算法。它可以在 $\mathcal{O}(|\mathsf{V}| + |\mathsf{E}|)$ 的时间内得出结果。

Tarjan 算法主要是利用 DFS 来寻找强连通分量的。现在我们来看看在 DFS 的过程中强连通分量有什么性质。

Tarjan 算法

Tarjan 算法是由 Robert Tarjan 提出的用于寻找有向图的强连通分量的算法。它可以在 $\mathcal{O}(|\mathsf{V}| + |\mathsf{E}|)$ 的时间内得出结果。

Tarjan 算法主要是利用 DFS 来寻找强连通分量的。现在我们来看看在 DFS 的过程中强连通分量有什么性质。

很重要的一点是如果结点 u 是某个强连通分量在搜索树中遇到的第一个结点（这通常被称为这个强连通分量的**根**）。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

- 栈里的元素表示的是当前已经访问过但是没有被归类到任一强连通分量的结点。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

- 栈里的元素表示的是当前已经访问过但是没有被归类到任一强连通分量的结点。
- dfn[u] 表示结点 u 在 DFS 中第一次搜索到的次序，通常被叫做时间戳。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

- 栈里的元素表示的是当前已经访问过但是没有被归类到任一强连通分量的结点。
- dfn[u] 表示结点 u 在 DFS 中第一次搜索到的次序，通常被叫做时间戳。
- low[u] 稍微有些复杂，它表示从 u 或者以 u 为根的子树中的结点，再通过一条反祖边或者横叉边可以到达的时间戳最小的结点 v 的时间戳，并且要求 v 有一些额外的性质：v 还要能够到达 u。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

- 栈里的元素表示的是当前已经访问过但是没有被归类到任一强连通分量的结点。
- dfn[u] 表示结点 u 在 DFS 中第一次搜索到的次序，通常被叫做时间戳。
- low[u] 稍微有些复杂，它表示从 u 或者以 u 为根的子树中的结点，再通过一条反祖边或者横叉边可以到达的时间戳最小的结点 v 的时间戳，并且要求 v 有一些额外的性质：v 还要能够到达 u。
  显然通过反祖边到达的结点 v 满足 low 的性质，但是通过横叉边到达的却不一定。

Tarjan 算法

Tarjan 算法主要是在 DFS 的过程中维护了一些信息：dfn、low 和一个栈。

- 栈里的元素表示的是当前已经访问过但是没有被归类到任一强连通分量的结点。
- dfn[u] 表示结点 u 在 DFS 中第一次搜索到的次序，通常被叫做时间戳。
- low[u] 稍微有些复杂，它表示从 u 或者以 u 为根的子树中的结点，再通过一条反祖边或者横叉边可以到达的时间戳最小的结点 v 的时间戳，并且要求 v 有一些额外的性质：v 还要能够到达 u。
  显然通过反祖边到达的结点 v 满足 low 的性质，但是通过横叉边到达的却不一定。

可以证明，结点 u 是某个强连通分量的根等价于 dfn[u] 和 low[u] 相等。

Tarjan 算法

当通过 u 搜索到一个新的节点 v 的时候可以有多种情况：

Tarjan 算法

当通过 u 搜索到一个新的节点 v 的时候可以有多种情况：

1° 结点 u 通过树边到达结点 v

$$\mathrm{low}[u] = \min(\mathrm{low}[u], \mathrm{low}[v])$$

Tarjan 算法

当通过 u 搜索到一个新的节点 v 的时候可以有多种情况：

1° 结点 u 通过树边到达结点 v

$$\mathrm{low}[u] = \min(\mathrm{low}[u], \mathrm{low}[v])$$

2° 结点 u 通过反祖边到达结点 v，或者通过横叉边到达结点 v 并且满足 low 定义中 v 的性质

$$\mathrm{low}[u] = \min(\mathrm{low}[u], \mathrm{dfn}[v])$$

Tarjan 算法

在 tarjan 算法进行 DFS 的过程中，每离开一个结点，我们就判断一下 low 是否小于 dfn，如果是，那么着个结点可以到达它先前的结点再通过那个结点回到它，肯定不是强连通分量的根。如果 dfn 和 low 相等，那么就不断退栈直到当前结点为止，这些结点就属于一个强连通分量。

Tarjan 算法

在 tarjan 算法进行 DFS 的过程中，每离开一个结点，我们就判断一下 low 是否小于 dfn，如果是，那么着个结点可以到达它先前的结点再通过那个结点回到它，肯定不是强连通分量的根。如果 dfn 和 low 相等，那么就不断退栈直到当前结点为止，这些结点就属于一个强连通分量。

如何更新 low？

Tarjan 算法

在 tarjan 算法进行 DFS 的过程中，每离开一个结点，我们就判断一下 low 是否小于 dfn，如果是，那么着个结点可以到达它先前的结点再通过那个结点回到它，肯定不是强连通分量的根。如果 dfn 和 low 相等，那么就不断退栈直到当前结点为止，这些结点就属于一个强连通分量。

如何更新 low？

关键就在于第二种情况，当通过反祖边或者横叉边走到一个结点的时候，只需要判断这个结点是否在栈中，如果在就用它的 low 值更新当前节点的 low 值，否则就不更新。因为如果不在栈中这个结点就已经确定在某个强连通分量中了，不可能回到 u。

Tarjan 算法

Tarjan算法寻找有向图的强连通分量 - Miskcoo's Space - Mozilla Firefox (Private Browsing)

Tarjan算法寻找有向图的...

blog.miskcoo.com/2016/07/tarjan-alg

Search

```
/* 寻找有向图强连通分量的tarjan算法
 * dfn_index 表示的就是时间戳
 * scc_index 表示强连通分量的个数
 * scc[u] 表示结点u属于那一个强连通分量
 * on_stack[u] 表示结点u是否仍然在栈中
 * stack 和 st 分辨表示栈和栈顶位置 */
void tarjan(int u)
{
    dfn[u] = low[u] = ++dfn_index;
    on_stack[u] = 1; stack[st++] = u;
    for(int k = head[u]; k; k = next[k])
    {
        int v = point[k];
        if(!dfn[v])
        {
            // 树边的情况
            tarjan(v);
            if(low[v] < low[u])
                low[u] = low[v];
        } else if(on_stack[v] && dfn[v] < low[u]) {
            // 横叉边或者反祖边的情况
            low[u] = dfn[v];
        }
    }

    if(low[u] == dfn[u])
    {
        ++scc_index, tmp = 0;
        while(tmp != u)
        {
            tmp = stack[--st];
            scc[tmp] = scc_index;
            on_stack[tmp] = 0;
        }
    }
}
```

无向图的双连通分量、割点与桥

在图论中，如果在一个无向连通图中删除一个结点及与它相关的边之后，该图被分成两个或者更多的连通分量，那么这个点就被称为**割点**（cut vertex），也被称为**关节点**（articulation vertex）。如果一个无向连通图没有割点，那么就称该图为**点双连通图**。无向图的极大点双连通子图被称为**点双连通分量**。

无向图的双连通分量、割点与桥

在图论中，如果在一个无向连通图中删除一个结点及与它相关的边之后，该图被分成两个或者更多的连通分量，那么这个点就被称为**割点**（cut vertex），也被称为**关节点**（articulation vertex）。如果一个无向连通图没有割点，那么就称该图为**点双连通图**。无向图的极大点双连通子图被称为**点双连通分量**。

如果在一个无向连通图中删除一条边后，该图被分成两个或者更多的连通分量，那么这条边就被称为**割边**（cut edge），也被称为**桥**（bridge）。如果一个无向连通图没有割边，那么就称该图为**边双连通图**。无向图的极大边双连通子图被称为**边双连通分量**。

无向图的双连通分量、割点与桥

求解割点的方法和有向图的 tarjan 算法类似。我们保留 dfn 的定义不变，由于无向图在 DFS 的过程中不会出现横叉边，low 的定义改变为从子树中经过反祖边能够到达的时间戳最小的结点。

无向图的双连通分量、割点与桥

求解割点的方法和有向图的 tarjan 算法类似。我们保留 dfn 的定义不变，由于无向图在 DFS 的过程中不会出现横叉边，low 的定义改变为从子树中经过反祖边能够到达的时间戳最小的结点。

如果结点 u 是整棵搜索树的根，那么它是割点当且仅当 u 有两个及以上的儿子。

无向图的双连通分量、割点与桥

求解割点的方法和有向图的 tarjan 算法类似。我们保留 dfn 的定义不变，由于无向图在 DFS 的过程中不会出现横叉边，low 的定义改变为从子树中经过反祖边能够到达的时间戳最小的结点。

如果结点 u 是整棵搜索树的根，那么它是割点当且仅当 u 有两个及以上的儿子。

如果结点 u 不是搜索树的根，那么当存在 v 是 u 的儿子并且并且满足 $dfn[u] \leq low[v]$，那么结点 u 也是割点。

无向图的双连通分量、割点与桥

求解割点的方法和有向图的 tarjan 算法类似。我们保留 dfn 的定义不变，由于无向图在 DFS 的过程中不会出现横叉边，low 的定义改变为从子树中经过反祖边能够到达的时间戳最小的结点。

如果结点 u 是整棵搜索树的根，那么它是割点当且仅当 u 有两个及以上的儿子。

如果结点 u 不是搜索树的根，那么当存在 v 是 u 的儿子并且并且满足 $\mathrm{dfn}[u] \leq \mathrm{low}[v]$，那么结点 u 也是割点。

至于割边，上面不等式改成小于号即可。

例题 1 UOJ27

给出一个有 n 个结点，m 条边的无向图，问有哪些结点满足：删去该点后原图变为一棵树。

其中 $0 < \mathrm{n}, \mathrm{m} < 10^5$。

例题 1 UOJ27

给出一个有 n 个结点，m 条边的无向图，问有哪些结点满足：删去该点后原图变为一棵树。

其中 $0 < n, m < 10^5$。

树是 n 个点 n-1 条边的无向连通图

例题 1 UOJ27

给出一个有 n 个结点，m 条边的无向图，问有哪些结点满足：删去该点后原图变为一棵树。

其中 $0 < n, m < 10^5$。

树是 n 个点 n-1 条边的无向连通图

删去该点原图连通且边变为 n-2 条

例题 1 UOJ27

给出一个有 n 个结点，m 条边的无向图，问有哪些结点满足：删去该点后原图变为一棵树。

其中 $0 < n, m < 10^5$。

树是 n 个点 n-1 条边的无向连通图

删去该点原图连通且边变为 n-2 条

选择度数为 m-(n-2) 的非割点结点

例题 1 UOJ27

给出一个有 n 个结点，m 条边的无向图，问有哪些结点满足：删去该点后原图变为一棵树。

其中 $0 < n, m < 10^5$。

树是 n 个点 n-1 条边的无向连通图

删去该点原图连通且边变为 n-2 条

选择度数为 m-(n-2) 的非割点结点

混合图

你有一个混合图，它有 n 个顶点，m 条边，其中 a 条是有向边，b 条是无向边。现在要求为这 b 条无向边确定一个方向，使得最后的有向图上不存在环。

其中 $1 \leq \mathsf{n}, \mathsf{a}, \mathsf{b} \leq 10^5, \mathsf{m} = \mathsf{a} + \mathsf{b}$。不用考虑无解的情况。

混合图

既然要求最后的有向图没有环，也就是要求最后是个 DAG！

混合图

既然要求最后的有向图没有环，也就是要求最后是个 DAG！

DAG 进行拓扑排序后边的方向都是拓扑序小的连向拓扑序大的。

混合图

既然要求最后的有向图没有环，也就是要求最后是个 DAG！

DAG 进行拓扑排序后边的方向都是拓扑序小的连向拓扑序大的。

先把无向边忽略，对图进行拓扑排序，最后无向边的方向就是拓扑序小的连向拓扑序大的。

灾后重建

B 地区在地震过后，所有村庄都造成了一定的损毁，而这场地震却没对公路造成什么影响。但是在村庄重建好之前，所有与未重建完成的村庄的公路均无法通车。换句话说，只有连接着两个重建完成的村庄的公路才能通车，只能到达重建完成的村庄。

给出 B 地区的村庄数 n，村庄编号从 1 到 n，和所有 m 条公路的长度，公路是双向的。并给出第 i 个村庄重建完成的时间 t[i]，你可以认为是同时开始重建并在第 t[i] 天重建完成，并且在当天即可通车。若 t[i] 为 0 则说明地震未对此地区造成损坏，一开始就可以通车。之后有 Q 个询问 (x, y, t)，对于每个询问你要回答在第 t 天，从村庄 x 到村庄 y 的最短路径长度为多少。如果无法找到从 x 村庄到 y 村庄的路径，经过若干个已重建完成的村庄，或者村庄 x 或村庄 y 在第 t 天仍未重建完成，则需要返回-1。

数据保证 t[1]≤t[2]≤ $\cdots$ ≤t[n]。并且
$0 < \mathrm{n} \leq 200, 0 < \mathrm{Q} \leq 50000$。

Source：福建 NOIP 夏令营

灾后重建

在第 i 个村庄重建完成时，前面村庄都重建完成了，后面的村庄都没有重建完成。

灾后重建

在第 i 个村庄重建完成时，前面村庄都重建完成了，后面的村庄都没有重建完成。

最短路只能经过前 i 个村庄。

灾后重建

在第 i 个村庄重建完成时，前面村庄都重建完成了，后面的村庄都没有重建完成。

最短路只能经过前 i 个村庄。

Floyd！

肮脏的道路

有一个 n 个结点 m 条边的无向图，边有权值。

从一个结点 s 到另一个结点 t 的一条路径的肮脏值定义为路径上权值最大的那条边的权值。

要求计算出从 s 到 t 的所有路径中肮脏值最小的那条路径的肮脏值。

其中 $1 \leq \mathrm{n} \leq 10000, 1 \leq \mathrm{m} \leq 20000$。

肮脏的道路

二分？

肮脏的道路

二分？

类似 Kruskal 算法直接判断连通。

肮脏的道路

二分？

类似 Kruskal 算法直接判断连通。

事实上，这样的路径叫做**最小瓶颈路**。

Candies

有 n 个人分糖果，现在有 m 个限制，比如说某一个人得到的糖果不能比另一个人少 k 个以上。

请你求出小 A 比小 B 最多能多得到多少个糖果。

其中 $1 \leq \mathrm{n} \leq 30000, 1 \leq \mathrm{m} \leq 150000$。

Source：POJ 3159. Candies

Candies

比如小 B 得到的糖果不能比小 A 少 k 个以上，那么

$$A - B \leq k$$

Candies

比如小 B 得到的糖果不能比小 A 少 k 个以上，那么

$$A - B \leq k$$

将每个人看成一个点，比如上面这个限制就从 B 向 A 连接一条长度为 k 的边。

Candies

比如小 B 得到的糖果不能比小 A 少 k 个以上，那么

$$A - B \leq k$$

将每个人看成一个点，比如上面这个限制就从 B 向 A 连接一条长度为 k 的边。

最短路

和平委员会

某国有 n 个党派，每个党派在议会中恰有 2 个代表。有 m 对代表之间会有矛盾存在。

现在要成立和平委员会，该会满足：

- 每个党派在和平委员会中有且只有一个代表。
- 如果某两个代表存在矛盾，则他们不能都属于委员会。

判断是否能够成立这个委员会。如果能，给出方案。

其中 $1 \leq n \leq 8000, 1 \leq m \leq 20000$。

Source：Poi 0106 Peaceful Commission

和平委员会

将每个代表看成一个点。

和平委员会

将每个代表看成一个点。

如果代表 A 在委员会，代表 B 必须在委员会则连一条有向边从 A 到 B。

和平委员会

将每个代表看成一个点。

如果代表 A 在委员会，代表 B 必须在委员会则连一条有向边从 A 到 B。

若一个党派的人在一个环内无解！

和平委员会

将每个代表看成一个点。

如果代表 A 在委员会，代表 B 必须在委员会则连一条有向边从 A 到 B。

若一个党派的人在一个环内无解！

缩环！

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

如果一个点 A 在，它对应的另一个点 A' 就不能在。

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

如果一个点 A 在，它对应的另一个点 A' 就不能在。

所有 A' 的祖先也都不能在。

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

如果一个点 A 在，它对应的另一个点 A' 就不能在。

所有 A' 的祖先也都不能在。

将边反向

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

如果一个点 A 在，它对应的另一个点 A' 就不能在。

所有 A' 的祖先也都不能在。

将边反向

按照拓扑序来找点。

和平委员会

剩下的情况就是有解吗？是的话如何给出方案？

图有什么性质？

边是对称的

如果一个点 A 在，它对应的另一个点 A' 就不能在。

所有 A' 的祖先也都不能在。

将边反向

按照拓扑序来找点。

这个问题实际上被称为 2-SAT 问题。