高级数据结构及其应用

目录

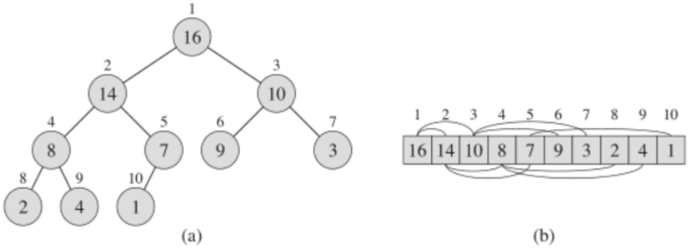
堆的定义(1/2)

定义

一棵二叉树满足“**堆结构特性**”和“**堆偏序特性**”，则称它为一个**大根堆**:

(1) **堆结构特性**是指，一棵二叉树要么它是满二叉树，要么它仅比一个满二叉树在最底层(深度最大的层)少若干结点，并且最底层的结点从左向右紧挨着依次排列.

(2) **堆偏序特性**是指，堆结点中存储的元素满足双亲结点的值 大于所有子结点的值.



堆的定义(2/2)

表示堆的数组A包含两个属性:

(1) $A.length$:数组元素的个数.

(2) $A.heapsize$: 有多少堆元素存储在该数组中.

结点的高度: 结点距离树叶的距离;

结点的深度: 结点距树根的距离.

堆的性质

命题

(1) 含n个元素的堆的高度为$O(\log n)$.

(2) 在最大堆的任一子树中，该子树所包含的最大元素在该子树的根结点上.

(3) 当用数组表示存储n个元素的堆时，叶结点下标分别是:

$$\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n.$$

问题3.1

(1) 高度为h的堆中，最多有多少元素，最少有多少元素？

(2) 假设一个最大堆得所有元素都不相同，那么该堆的最小元素应该位于哪里？

(3) Is an array that is in sorted order a min-heap?

维护大根堆的性质

算法3.1 **Max-Heapify**(A, i)

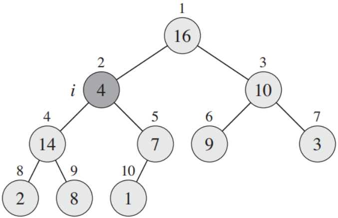
输入： 堆结构A, A的结点i

输出： 从i向下满足堆存储要求的堆结构

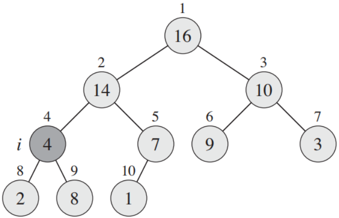
$l \leftarrow 2i$, $r \leftarrow 2i + 1$
$largest \leftarrow i$
if $l \leq A.heapsize$ and $A[l] > A[i]$
    $largest \leftarrow l$
if $r \leq A.heapsize$ and $A[r] > A[largest]$
    $largest \leftarrow r$
if $largest \neq i$
    $A[i] \leftrightarrow A[largest]$
    Max-Heapify(A, $largest$)

算法的时间复杂度: $O(\log n)$.

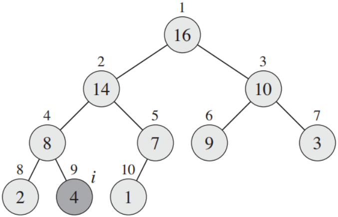
维护大根堆的实例



(a)



(b)



(c)

问题3.2

(1) 当元素$A[i]$比其他孩子的值都大时，调用**Max-Heapify**(A, i)会有什么结果？

(2) 当$i > A.heapsize/2$时，调用**Max-Heapify**(A, i) 会有什么结果？

(3) 使用循环控制结构取代递归，重写调用**Max-Heapify**代码.

(4) 维护小根堆**Min-Heapify**(A, i) 的算法如何写？

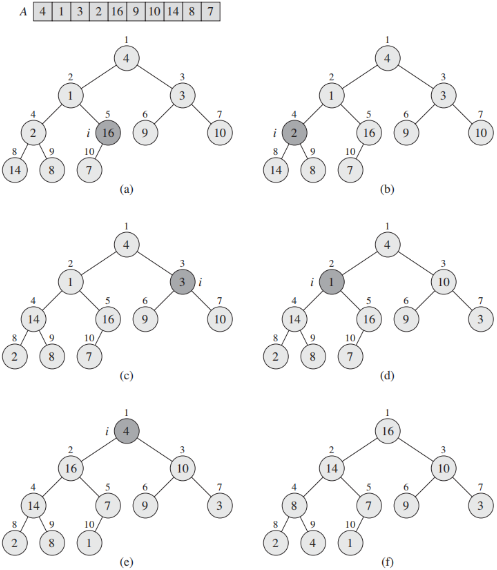
建大根堆

目录
堆
堆的定义和基本性质
维护大根堆的性质
优先队列
不相交集数据结构
应用

算法3.2 **Build-Max-Heap**(A)

$A.heapsize \leftarrow A.length$
for $i \leftarrow \lfloor A.length/2 \rfloor$ **downto** 1
    Max-Heapify(A, i)

建堆的实例



问题3.3

(1) 问题: 算法3.2伪代码第2行能否改成:

2 **for** $i \leftarrow 1$ **to** $\lfloor A.length/2 \rfloor$

(2) 建立小根堆**Build-Min-Heap**(A, i) 的算法如何写？

建堆的工作量:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \text{高为}h\text{的结点数} \times O(h)$$

定理

n个结点的堆算法***Build-Max-Heap***的时间复杂度是$O(n)$.

证明 令k表示堆的高度. 调用**Max-Heapify**的总次数的上界是:

$$\sum_{h=0}^{k} h2^{k-h} = 2^k \sum_{h=1}^{k} \frac{h}{2^h} < 2n$$

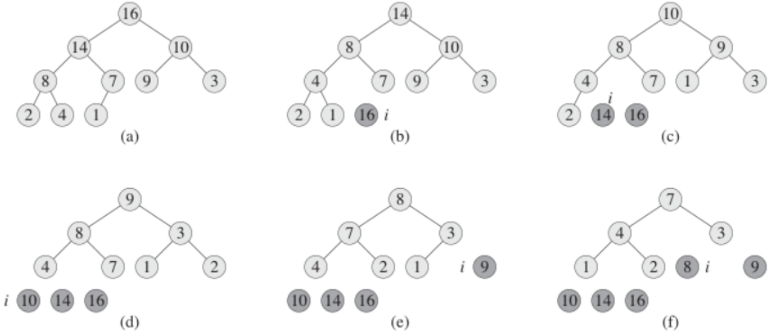
堆排序算法

算法3.3 **HeapSort**(A)

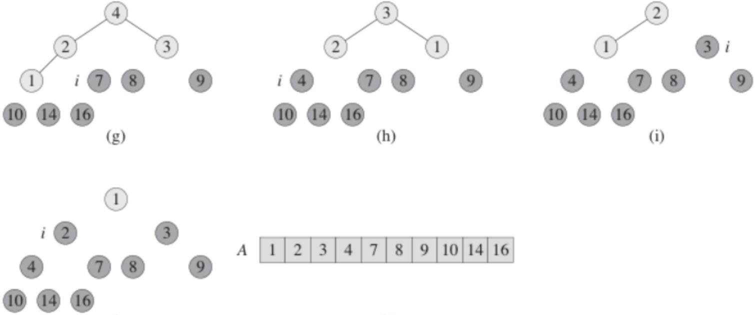
Build-Max-Heap(A)
for $i \leftarrow A.length$ **downto** 2
    $A[1] \leftrightarrow A[i]$
    $A.heapsize \leftarrow A.heapsize - 1$
    Max-Heapify(A, 1)

堆排序算法在最坏情形下的时间复杂度: $O(n \log n)$.

堆排序算法的实例(1/2)


16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
2
i
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

堆排序算法的实例(2/2)



(g) (h) (i) (j) (k)

优先队列

优先队列(priority queue)是一种用来维护由一组元素构成的集合S的数据结构, 其中的每一个元素都有一个相关的值, 称为优先级或关键字(key). 一个最大优先队列支持以下操作:

(1) **Get-Max**(S): 返回优先队列中优先级最高的元素.

(2) **Extract-Max**(S): 返回优先队列中优先级最高的元素, 并将该元素删除, 并修复删除后的优先队列.

(3) **Increase-Key**(S, x, k): 将优先队列中的元素x的优先级提高到k, 这里假设k的值不小于x的原来关键字.

(4) **Insert**(S, x): 向优先队列中添加一个新的元素x.

优先队列的应用

共享计算机系统的作业调度:

(1) (最大)优先队列记录将要执行的各种作业以及它们之间的相对优先级.

(2) 当一个作业完成或者被中断后，调度器调用**Extract-Max**从所有的等待作业中，选出具有最高优先级的作业来执行.

(3) 调度器可以调用**Insert**把一个新作业加入到队列中来.

基于大根堆实现优先队列(1/5)

算法3.4 **Get-Max**(A)

1 **return** $A[1]$

时间复杂度: $O(1)$.

基于大根堆实现优先队列(2/5)

算法3.5 **Extract-Max**(A)

$max \leftarrow A[1]$
$A[1] \leftarrow A[A.heapsize]$
$A.heapsize \leftarrow A.heapsize - 1$
Max-Heapify(A, 1)
return max

算法3.5的时间复杂度: $O(\log n)$.

基于大根堆实现优先队列(3/5)

算法3.6 **Increase-Key**(A, i, k)

1 $A[i] \leftarrow k$

2 **while** $i>1$ and $A[i]>A[\lfloor i/2 \rfloor]$ **do**

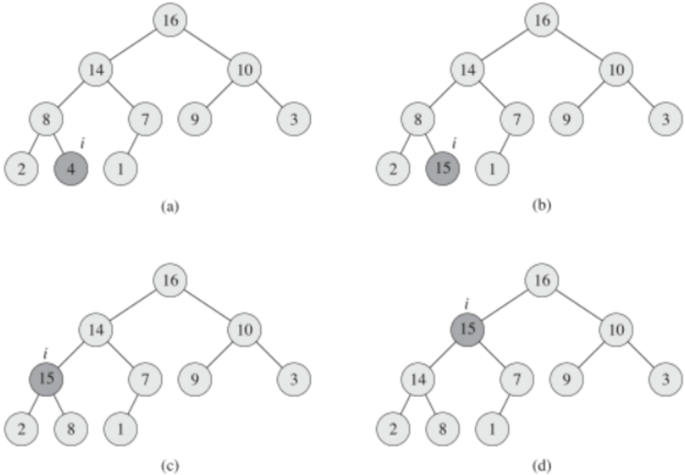
3 $\quad A[i] \leftrightarrow A[\lfloor i/2 \rfloor]$

4 $\quad i \leftarrow \lfloor i/2 \rfloor$

时间复杂度: $O(\log n)$.

基于大根堆实现优先队列(4/5)

算法**Increase-Key**的实例:



基于大根堆实现优先队列(5/5)

算法3.7 **Insert**(A, key)

1 $A.heapsize \leftarrow A.heapsize + 1$
2 $A[A.heapsize] \leftarrow -\infty$
3 **Increase-Key**(A, $A.heapsize$, key)

时间复杂度: $O(\log n)$.

问题: 为什么先把关键字设为$-\infty$, 然后又将其增加到所需的值呢?

问题3.4

(1) 设计算法基于**链表**实现优先队列的基本操作, 且分析算法的时间复杂性.

(2) 设计算法基于**有序数组**实现优先队列的基本操作, 且分析算法的时间复杂性.

(3) 设计算法将结点i从最大堆A中删掉.

例

对集合$S=\{1, 2, ..., 8\}$定义如下的等价关系：

$$R=\{(x, y) : x\in S, y\in S, (x-y)\%3=0\}$$

求S关于R的等价类.

解:

(1) 初始化: $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}$.

(2) $(1, 4) \in R$: $\{1, 4\}, \{2\}, \{3\}, \{5\}, \{6\}, \{7\}, \{8\}$.

(3) $(4, 7) \in R$: $\{1, 4, 7\}, \{2\}, \{3\}, \{5\}, \{6\}, \{8\}$.

(4) $(2, 5) \in R$: $\{1, 4, 7\}, \{2, 5\}, \{3\}, \{6\}, \{8\}$.

(5) $(5, 8) \in R$: $\{1, 4, 7\}, \{2, 5, 8\}, \{3\}, \{6\}$.

(6) $(3, 6) \in R$: $\{1, 4, 7\}, \{2, 5, 8\}, \{3, 6\}$.

不相交集数据结构的三种操作

(1) **Make-set**(x)
建立一个包含元素x的新的单点集, 并将此集合命名为"x".

(2) **Find**(x)
寻找并返回包含元素x的集合的名字.

(3) **Union**(x, y)
将包含元素x和y的两个集合用它们的并集替换. 并集的名字, 或为包含元素x的那个集合的名字, 或为包含元素y的那个集合的名字.

设计这三种运算的有效算法,需要一种数据结构:

(1) 简单

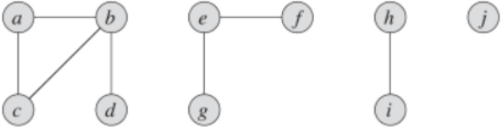
(2) 能有效实现合并和寻找这二种运算

确定无向图的连通分量

算法3.8 Connected-Components(G)

```
for each vertex v ∈ V do
    Make-set(v)
for each edge (u, v) ∈ E do
    if (Find(u)≠Find(v)) then
        Union(u, v)
```

实例



(a)

Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

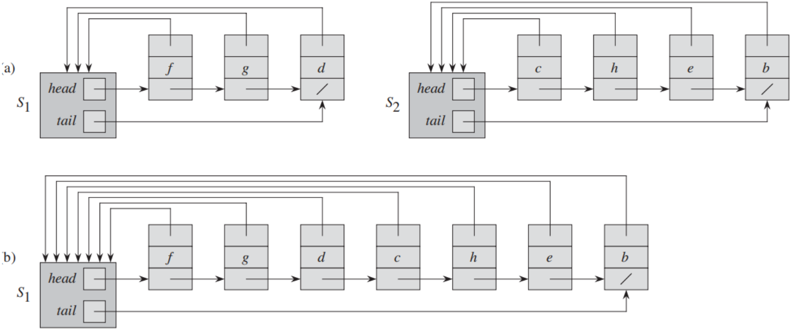
(b)

问题3.5

(1) Connected-Components算法处理完所有的边后，两个结点在相同的连通分量中当且仅当它们在同一个集合中.

(2) 在Connected-Components算法作用于一个有k个连通分量的无向图G的过程中，**Find**需要调用多少次？**Union**需要调用多少次？

基于链表的实现



合并的简单实现

Make-Set操作和**Find**操作的时间复杂性: $O(1)$.

(1) **Make-set**(1), ...,**Make-set**(n).

(2) **Union**(1, 2),...,**Union**($n-1$, n).

$n-1$次**Union**的时间复杂度: $O(n^2)$.

加权合并的启发式策略

将较短的链表拼接到较长的表中.

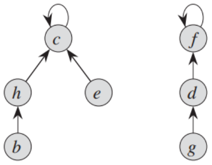
定理

使用不相交集合的链表表示, 一个具有m个*Make-Set*、*Find*和*Union*操作的序列(其中有n个是*Make-Set* 操作)需要的时间为$O(m + n\log n)$.

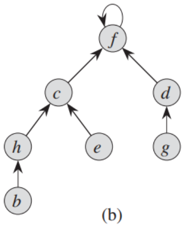
证明 由于每个**Union**操作合并两个不相交集合, 因此至多执行$n-1$个**Union**操作.

- 最大集合最多包含n 个元素.
- 每个对象的指针在所有的**Union**操作中最多被更新$O(\log n)$次.

基于森林的实现



(a)



(b)

直接进行合并运算

(1) **Make-set**(1), ...,**Make-set**(n).

(2) **Union**(1, 2),...,**Union**($n-1$, n).

(3) **Find**(1),...,**Find**(n).

n次寻找运算的时间复杂度为: $O(n^2)$.

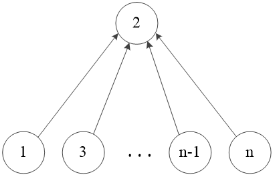
按秩合并的启发式策略

为限制每棵树的高度，采用按秩合并措施:

- 给每个结点存储一个非负数作为该结点的秩，记为rank，结点的秩基本上就是它的高度.

设x和y是当前森林中两棵不同的树的根，初始状态时，每个结点的秩是0.
执行运算**Union**(x, y)时，比较x.rank和y.rank.

- x.rank<y.rank: 使y为x的双亲结点.
- x.rank>y.rank: 使x为y的双亲结点.
- x.rank=y.rank: 使y为x的双亲结点，并将y.rank加1.



Make-set操作的算法

算法3.9 **Make-set**(x)

输入: 元素x

输出: 包含x的树

1 $x.p \leftarrow x$

2 $x.\text{rank} \leftarrow 0$

Union操作的算法

算法3.10 **Union**(x, y)

输入: 两个元素x, y

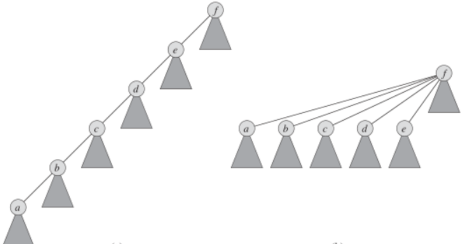
输出: 包含x和y的两个树的合并, 原来的树被破坏

$u \leftarrow$ **Find**(x), $v \leftarrow$ **Find**(y)
if u.rank$\leq v$.rank **then**
    $u.p \leftarrow v$
    if u.rank$=v$.rank **then** v.rank $\leftarrow v$.rank$+1$
else $v.p \leftarrow u$

路径压缩的启发式策略

(1) 在**Find**(a)中，找到根结点f之后；

(2) 再一次遍历从a到f 的路径，并沿着路径改变所有结点指向双亲结点的指针，使它们直接指向f.



(a) (b)

Find操作的算法

算法3.11 Find(x)

输入: 结点x

输出: root(x), 包含x的树的根

```
y ← x
while y.p≠y            // 寻找包含x的树的根
    y ← y.p
root ← y, y ← x
while y.p≠y            // 执行路径压缩
    w ← y.p
    y.p ← root
    y ← w
return root
```

按秩合并措施的特点

引理

包括根结点x在内的树中结点的个数至少是$2^{x.rank}$.

证明 对**Union**的次数应用归纳法.
最初x自身形成一棵树，它的秩为0. 设x和y为两个根，考虑运算**Union**(x, y). 假设引理在这项运算之前成立.

- x.rank<y.rank: 使y为x的双亲结点. 以y 为根形成的树比老的以y为根形成的树的结点多，并且它的秩未改变.
- x.rank>y.rank: 使x为y的双亲结点. 以x 为根形成的树比老的以x为根形成的树的结点多，并且它的秩未改变.
- x.rank=y.rank: 使y为x的双亲结点，并将y.rank加1. 根据归纳法，在这种情况下，以y为根形成的树至少有

$$2^{x.\mathrm{rank}} + 2^{y.\mathrm{rank}} = 2^{y.\mathrm{rank}+1}$$

个结点. 由于y.rank每次加1，所以运算之后引理成立.

时间复杂度

定理

使用不相交集合的树表示和按秩合并措施, 一个具有m 个***Make-Set***、***Find***和***Union***操作的序列(其中有n个是***Make-Set***操作)需要的时间为$O(n + m\log n)$.

问题的导入

问题

假设有城市集合$V = \{v_1, v_2, ..., v_n\}$, 我们想在它们的顶部建立一个通信网络, 网络应该是连通的: 在每对城市之间应该有一条路径. 在满足这个需求的同时, 我们希望尽可能便宜地建立它.

建模:对于确定的边(v_i, v_j), 可能以某个费用$w(v_i, v_i)$建立v_i与v_j之间的直接连接. 用一个无向图$G = (V, E)$来表示可能被建立的连接的集合.与每条边(v_i, v_j)相关的有一个正的费用$w(v_i, v_i)$.
问题: 找一个边的子集$T \subseteq E$使得图(V, T) 是连通的, 且总费用 $\sum_{(u,v)\in T} w(u, v)$最小.

命题

设T是上述定义的网络设计问题的最小费用解. 则(V, T) 是一棵树.

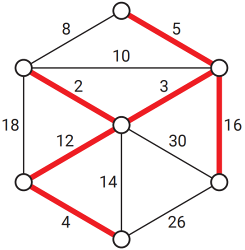
最小生成树的定义

给定带权无向图$G=(V, E)$, 每条边$e=(v_i, v_j)$的权$w(e)$为实数, 表示从结点v_i到v_j的距离.

定义

(1) 图G的生成树T是其子图, 满足: T包含图G中的所有结点, 且T是连通无环路. 生成树的权为其所有边权的和.

(2) 如果T是无向图G的生成树, 且图中不存在任何其他比T的权小的生成树, 则称T为图G的最小生成树(MST).



最小生成树的形成策略

Generic-MST(G, w)

$T \leftarrow \emptyset$
while T does not form a spanning tree **do**
    find an edge (u, v) that is safe for T
    $T \leftarrow T \cup \{(u, v)\}$
return T

注

(1) 在每遍循环之前, T 是某棵最小生成树的一个子集.

(2) 在每一步, 选择一条安全边(u, v), 将其加入到集合T, 使$T \cup \{(u, v)\}$也是某棵最小生成树的子集.

辨认安全边的规则(1/3)

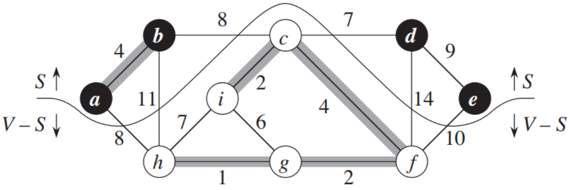
定义

(1) 无向图G的一个切割$(S,\ V-S)$是集合V 的一个划分.

(2) 如果边$(u,\ v)\in E$ 的一个端点位于集合S, 另一个端点位于集合$V-S$, 则称该条边横跨切割$(S,\ V-S)$.

(3) 设$A\subseteq E$.若A中不存在横跨该切割的边,则称该切割尊重A.

(4) 横跨一个切割的所有边中,权重最小的边称为轻量级边.



辨认安全边的规则(2/3)

定理

设G是一个在边E上定义了实数值权重函数w的连通无向图.

(1) 设$A \subseteq E$, 且A包括在图G的某棵MST中.

(2) 设$(S, V-S)$是图G中尊重集合A的任意一个切割.

(3) 设(u, v)是横跨切割$(S, V-S)$的一条轻量级边.

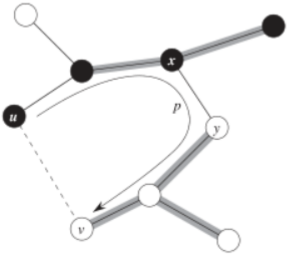
那么边(u, v)对于集合A是安全的.

辨认安全边的规则(3/3)

证明 设T是一个包含A的MST, 且假设$(u, v)\notin T$. 由于结点u和v分别处在切割$(S, V-S)$的两端, T中至少有一条边属于简单路径p并且横跨该切割. 设(x, y) 为这样的边, $w(u, v)\leq w(x, y)$. 令

$$T'=T-\{(x, y)\}\cup\{(u, v)\}.$$

$$w(T')=w(T)-w(x, y)+w(u, v)\leq w(T)$$



问题3.7

(1) 设(u, v)是连通图G中的一条权重最小的边, 则边(u, v)为图G的某棵最小生成树中的一条边.

(2) 如果图G的一条边(u, v)包含在图G的某棵最小生成树中, 则该条边是横跨图G的某个切割的一条轻量级边.

Kruskal算法的思想

Kruskal算法找到安全边的方法是:

- 在所有连接森林中两棵不同树的边里面, 找到权重最小的边(u, v).

具体思想:

(1) 对G的边以非降序权重排列;

(2) 对于排序表中的每条边, 如果现在把它放入T不会形成回路的话, 则把它加入到生成树T 中; 否则将它丢弃.

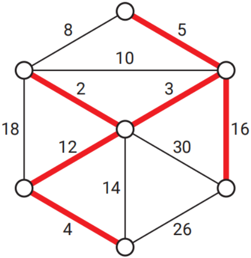
Kruskal算法的伪代码

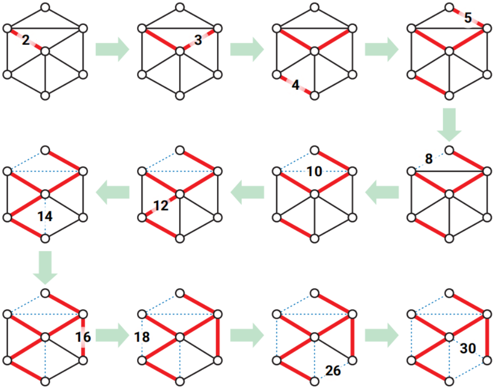
算法3.12 **MST-Kruskal**(G, w)

$T \leftarrow \emptyset$
for each vertex $v \in V$ **do**
    Make-Set(v)
sort the edges into nondecreasing order by weight w
for each edge $(u, v) \in E$ taken in nondecreasing order by weight
    if Find(u)$\neq$**Find**(v) **then**
        $T \leftarrow T \cup \{(u, v)\}$
        Union(u, v)
return T

时间复杂性: $O(|E| \log |E|)$

实例





Prim算法的思想

$T \leftarrow \emptyset$, $X \leftarrow \{s\}$, $Y \leftarrow V - \{s\}$
while $Y \neq \emptyset$ **do**
    Let (u, v) be of minimum weight such that $u \in X$ and $v \in Y$.
    $T \leftarrow T \cup \{(u, v)\}$
    $X \leftarrow X \cup \{v\}$
    $Y \leftarrow Y - \{v\}$

时间复杂性: $O(|V|^2)$

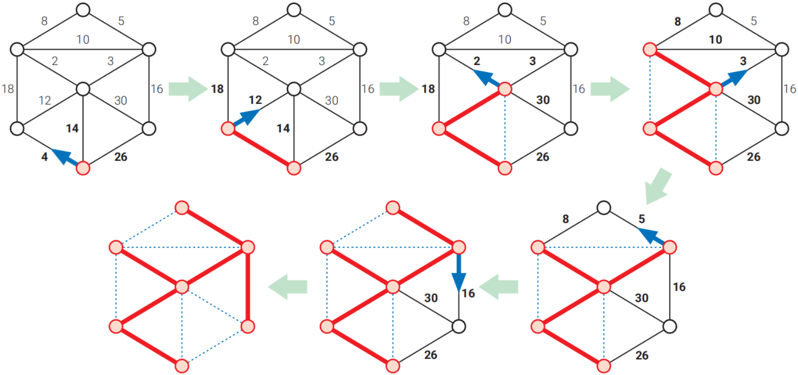
Prim算法的伪代码

算法3.13 **MST-Prim**(G, w, s)

```
for each u∈V do
    u.key ← +∞
    u.p ← NULL
s.key ← 0
Q ← V
while Q≠∅ do
        u ← Extract-Min(Q)
        for each (u, v)∈E do
            if v∈Q and w(u, v)<v.key then
                v.p ← u
                Decrease-Key(Q, v, w(u, v))
```

时间复杂性: $O(|E| \log |V|)$

实例



作业

编程实现:

(1) 用最小堆实现最小优先队列的以下操作:

- **Get-Min**(S): 返回优先队列中优先级最小的元素.
- **Extract-Min**(S): 返回优先队列中优先级最小的元素, 并将该元素删除, 并修复删除后的优先队列.
- **Decrease-Key**(S, x, k): 将优先队列中的元素x的优先级减少到k, 这里假设k的值不大于x的原来关键字.
- **Insert**(S, x): 向优先队列中添加一个新的元素x.

(2) 使用链表表示和加权合并的启发式策略, 写出**Make-Set**操作、**Find**操作和**Union**操作的程序.

(3) 基于最小优先队列实现Prim算法.

Thank you!