第五章　贪婪法

5.1 贪婪法引言

5.2 背包问题

5.3 单源最短路径问题

5.4 最小花费生成树问题

5.5 霍夫曼编码问题

5.1 贪婪法引言

例:日常生活中经常会碰到找硬币的例子。

现有四种硬币各10枚，它们的面值分别是1元，5角，1角和1分。现在要给顾客2元1角3分钱。如何找使得所拿出的硬币个数最少?

要求：用最少的货币张数支付现金。

$P = \{p_1, p_2, \ldots, p_n\}$ 集合表示 n 张面值为p_i的货币，$1 \le i \le n$。出纳员需支付的现金为 A。

从 P 中选取一个最小的子集 S，使得：

$$p_i \in S \quad \text{并且} \quad \sum_{i=1}^{n} p_i = A$$

用向量$X = (x_1, x_2, \ldots, x_n)$表示 S 中所选取的货币，使得：

$$x_i = \begin{cases} 1 & p_i \in S \\ 0 & p_i \notin S \end{cases}$$

出纳员支付的现金必须满足$\sum_{i=1}^{n} x_i p_i = A$，使得：

$$d = min \sum_{i=1}^{n} x_i$$

解向量：问题中n个元素的具体取值所构成的向量；

解空间：问题中n个元素的各种不同取值组合所构成的向量全体；

约束方程：问题中的限制条件所列出的方程；

目标函数：问题求解所要达到的目标；

可行解：满足约束方程的向量；

最优解：使目标函数达极值的向量。

贪婪法的基本思想

在求最优解问题的过程中，依据某种贪婪标准（Greedy Criterion），从问题的初始状态出发，直接去求每一步的最优解，通过若干次的贪婪选择，最终得出整个问题的最优解。

- 贪婪算法总是作出在当前看来最好的选择；
- 贪婪算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。

- 贪婪算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解，如单源最短路径问题，最小生成树问题等；
- 在一些情况下，即使贪婪算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪婪法的设计方法描述

```
greedy(A, n)
{
    solution = ∅;   /*解向量为空
    for (i=1; i<n; i++) {
        x = select(A);
        if (feasible(solution,x))
            solution = union(solution,x);
    }
    return solution;
}
```

贪婪算法的设计要素

- 可以用贪婪算法求解的问题一般具有2个重要的性质：
  - 贪婪选择性质
  - 最优子结构性质

□ 贪婪选择性质

- 贪婪选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪婪选择来达到；
- 贪婪算法通常以自顶向下的方式进行，以迭代的方式作出相继的贪婪选择，每作一次贪婪选择就将所求问题简化为规模更小的子问题；
- 对于一个具体问题，要确定它是否具有贪婪选择性质，必须证明每一步所作的贪婪选择最终导致问题的整体最优解。

□ 最优子结构性质

- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪婪算法求解的关键特征。

✓ 付给客户的货币集合的最优解是：

$$S_n = \{p_1, p_2, p_{21}, p_{31}, p_{32}, p_{33}\}$$

✓ 第一步所简化了的子问题的最优解是：

$$S_n = \{p_2, p_{21}, p_{31}, p_{32}, p_{33}\}$$

$$S_{n-1} \subset S_n, \qquad S_{n-1} \cup \{p_1\} = S_n$$

贪婪法的例子--货郎担问题

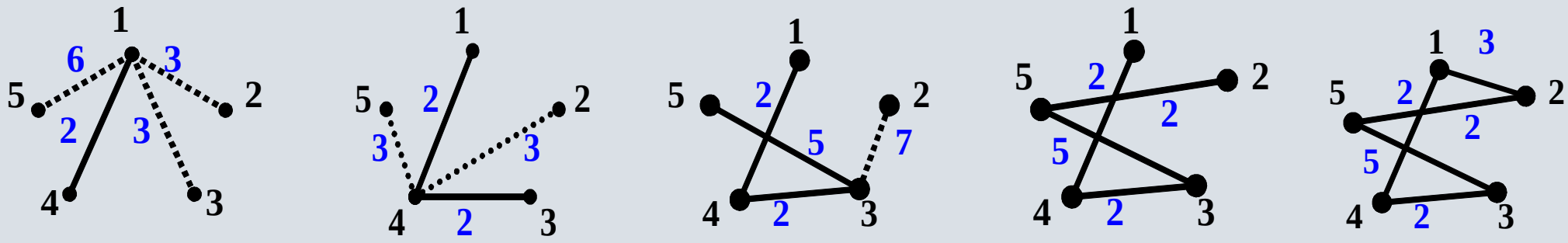
5 个城市，费用矩阵如下图的费用矩阵所示:

＼	1	2	3	4	5
1	∞	3	3	2	6
2	3	∞	7	3	2
3	3	7	∞	2	5
4	2	3	2	∞	3
5	6	2	5	3	∞

总是选择费用最小的路线前进，

选择的路线是:

1→4→3→5→2→1，总费用是14。



- 只选择一个城市作为出发城市，所需时间是$O(n^2)$；
- n 个城市都可以作为出发城市，所需时间是$O(n^3)$；
- 从城市 1 出发的最优的路线是1→2→5→4→3→1，总费用只有13。

	1	2	3	4	5
1	∞	3	3	2	6
2	3	∞	7	3	2
3	3	7	∞	2	5
4	2	3	2	∞	3
5	6	2	5	3	∞

5.2 背包问题

一 背包问题

二 思想方法

三 数据结构

四 算法描述

五 算法分析

一　背包问题

- ✓ 载重量为 M 的背包，n个重量为w_i、价值为p_i 的物体，$1 \le i \le n$，把物体装满背包，使背包内的物体价值最大；
- ✓ 物体可以分割的背包问题及物体不可分割的背包问题，把后者称为 0/1 背包问题。

二　思想方法

- ✓ x_i是物体被装入背包的部分，$0 \le x_i \le 1$

  x_i=0：物体没被装入背包

  x_i=1：物体被全部装入背包

- ✓ 约束方程：$\sum_{i=1}^{n} w_i x_i = M$
- ✓ 目标函数：$d = max \sum_{i=1}^{n} p_i x_i$
- ✓ 满足约束方程且使目标函数达到最大的解向量：$X = (x_1, x_2, \dots, x_n)$

- 用贪婪法求解背包问题的关键是如何选定贪婪策略，使得按照一定的顺序选择每个物品，并尽可能的装入背包，直至背包装满。至少有三种看似合理的贪婪策略：

- 1）按物品价值v降序装包，因为这可以尽可能快的增加背包的总价值。但是，虽然每一步选择获得了背包价值的极大增长，但背包容量却可能消耗太快，使得装入背包的物品个数减少，从而不能保证目标函数达到最大；
- 2）按物品重量w升序装包，因为这可以装入尽可能多的物品，从而增加背包总价值。但是，虽然每一步选择使背包的容量消耗得慢了，但背包价值却没能保证迅速增长，从而不能保证目标函数达到最大。
- 3）按物品价值与重量比值v/w的降序装入背包。

- 用贪婪算法解背包问题的基本步骤：
  - ✓ 首先计算每种物品价值重量比（即单位重量的价值），然后，依照贪婪选择策略，将尽可能多的价值重量比最高的物品装入背包；
  - ✓ 若将这种物品全部装入背包后，背包内的物品总重量未超过背包载重量M，则选择单位重量价值次高的物品并尽可能多地装入背包；
  - ✓ 依此策略一直进行下去，直到背包装满为止。

三　数据结构

```
typedef struct {
    float p;                    // n个物体的价值
    float w;                    // n个物体的重量
    float v;                    // n个物体的价值重量比
} OBJECT;
OBJECT   instance[ n ]; // n个物体的信息
float  x[ n ];                  // n个物体装入背包的分量
```

四. 算法描述

算法5.1 贪婪法求解背包问题

```
1. float knapsack_greedy(float M, OBJECT instance[ ],
                         float x[ ], int n)
2. {   int i;
4.     float m, p = 0;
5.     for (i=0; i<n; i++) {   /*计算物体的价值重量比*/
6.         instance[ i ].v = instance[ i ].p / instance[ i ].w;
7.         x [i ] = 0;         /*解向量赋初值*/
8.     }
```

```
 9.    merge_sort(instance,n); /*按关键值v的递减顺序排序物体*/
10.    m = M;    /*背包的剩余载重量*/
11.    for (i=0; i<n; i++) {
12.        if (instance[ i ].w <= m) /*优先装入价值重量比大的物体*/
13.            x[ i ] = 1;     m -= instance[ i ].w;
14.            p += instance[ i ].p;
15.        }
16.        else {                          /*最后一个物体的装入分量*/
17.            x[ i ] = m / instance[ i ].w;
18.            p += x[ i ] * instance[ i ].p;
19.            break;
20.        }
22.    return p;
23. }
```

五 算法分析

- 背包问题贪婪算法的复杂度：
  - 计算n个问题的价值重量比并为解向量赋初值：$\Theta(n)$
  - n个物体的价值重量比排序：$\Theta(nlogn)$
  - 判断每个物体装入背包的分量：$\Theta(n)$

✓ 本算法的运行时间：$\Theta(nlogn)$

✓ 本算法的工作空间：$\Theta(n)$

- 例：给定3个物体，背包载重量M=10，物体的价值分别为P={10,4,8}，重量为w={5,1,8}。使用贪婪法将物品装满背包，且使背包内的物体价值最大。
  - ✓ 按照物品价值重量比最大的贪婪选择策略：

物品i	1	2	3
P_i	10	4	8
w_i	5	1	8
P_i/w_i	2	4	1

物品i	2	1	3
P_i/w_i	4	2	1
放入背包的数量	1	1	0.5

- ✓ **即：物品1,2全部放入背包，而物品3只放入1/2部分；**
- ✓ **故，问题的解为X=（1,1,0.5）**

0/1背包问题

物品i	1	2	3
P_i	10	4	8
w_i	5	1	8

- ✓ 可行解为：

  (1,1,0): 14

  (0,1,1): 12

- ✓ **最优解为：选择物品1和2装入，最大价值为14.**

- 对于0/1背包问题，贪婪选择之所以不能得到最优解，是因为在这种情况下，它无法保证最终能将背包装满，部分闲置的背包空间使每单位重量背包空间的价值降低了。
- 在考虑0/1背包问题时，应比较选择该物品和不选择该物品所导致的最终方案，再做出最好选择。
- 由此就导出许多互相重叠的子问题。这正是该问题可以用动态规划算法求解的一个重要特征。

贪心选择的最优性证明

- 定理5.1 当物体的价值重量比按递减顺序排序后，算法knapsack_greedy可求得背包问题的最优解。
- 证明基本思想：
  - ✓ 把贪婪解与任一最优解相比较，如果这两个解不同，就去找开始不同的第一个x_i；
  - ✓ 然后设法用贪婪解的x_i去代换最优解的x_i，并证明最优解在分量代换之后其总价值保持不变；
  - ✓ 反复进行下去，直到最新产生的最优解与贪婪解完全一样，从而证明了贪婪解是最优解。

■ 证明：设解向量为$X=(x_1,x_2,\dots,x_n)$，分两种情况：

1. 解向量 X 中，所有 $x_i=1,i=1\sim n$，物体已全部装入，是最优解；

2. 解向量 $X=(1,1,\dots,1,x_j,0,\dots,0),x_j\neq 1,0\leq j\leq n$，由算法的实现，有：$\sum_{i=1}^{n}w_ix_i=M_1=M$ (1)

假定，算法的最优解是$Y=(y_1,y_2,\dots,y_n)$，并且满足：

$\sum_{i=1}^{n}w_iy_i=M_2=M$ (2)

若 $X \neq Y$，必存在 $k, 1 \leq k < n$，对 $1 \leq i < k$，有$x_i = y_i$ ，对 k有 $x_k \neq y_k$，这时，有两种情况：

1）若$x_k < y_k$ ，因为$y_k \leq 1$，必有$x_k < 1$。所以，根据算法的执行，有$x_{k+1} = \cdots = x_n = 0$ 。所以，$M_1 < M_2$，与(1), (2) 式矛盾，所以 $X = Y$;

2）若$x_k > y_k$，有：

$$M = \sum_{i=1}^{n} w_i x_i \geq \sum_{i=1}^{k} w_i x_i > \sum_{i=1}^{k} w_i y_i$$

所以，$y_{k+1}, \cdots, y_n$不会全为0。

令 $y_k + \Delta y_k = x_k$ 并使 $y_{k+1}, \cdots, y_n$ 相应减小，从而得到一个新的解 $Z = (z_1, z_2, \cdots, z_n)$，使得：

当 $1 \le i < k$ 时有：$z_i = y_i = x_i$

当 $i = k$ 时有：$y_k < z_k = x_k$

当 $k < i \le n$ 时有：$z_i < y_i$

并且满足：

$$(z_k - y_k)w_k - \sum_{i=k+1}^{n} (y_i - z_i)w_i = 0$$

令 $\delta = \frac{p_k}{w_k}(z_k - y_k)w_k - \frac{p_{k+1}}{w_{k+1}}(y_{k+1} - z_{k+1})w_{k+1} \quad \cdots \quad \frac{p_n}{w_n}(y_n - z_n)w_n$

$\geqslant \frac{p_k}{w_k}((z_k - y_k)w_k - (y_{k+1} - z_{k+1})w_{k+1} \quad \cdots \quad (y_n - z_n)w_n) = 0$

若$\delta > 0$，则 Z 是一个新的最优解；

若$\delta = 0$，则 Z 与 Y 同为最优解；

在此两种情况下，都用 Z 取代 Y，并且对所有的 $1 \leq i \leq k$，都有：$z_i = x_i$，而对 $k+1 \leq i \leq n$，有：$z_i \neq x_i$

对向量 Z 重复上述步骤，最终必有：对所有的 $1 \leq i \leq n$，都有：$z_i = x_i$

因此，X 是最优解。

5.3 单源最短路径问题

一 解最短路径的狄斯奎诺（Dijkstra）算法

二 Dijkstra算法的描述

三 Dijkstra算法的分析

一、解最短路径的Dijkstra算法

1. 单源最短路径问题

- 给定有向赋权图 G = (V, E)，其中每条边的权是非负实数；
- 顶点u称为源顶点；
- 求源顶点 u 到其它所有顶点的最短距离（指路上各边权之和）。

Dijkstra算法是代表性的解单源最短路径问题的贪心算法。

2. 狄斯奎诺算法的实现步骤

1）记号约定：

(u, v)：E中的边

$c_{u,v}$：边 (u, v) 的长度

V：顶点集合，划分为集合 S 和 T：

- S 中的顶点到 u 的距离已定
- T 中的顶点到 u 的距离未定

$d_{u,x}$：顶点 u 到T中顶点 x 的当前搜索状态下的距离

$p(x)$：从顶点 u 到顶点 x 的最短路径中x的前一顶点

2） Dijkstra算法思想

- 初始化：先找出从源点u到各终点x的直达路径$d_{u,x}$，即通过一条边到达的路径；
- 选择：从这些路径中找出一条长度最短的路径$d_{u,t}$；
- 更新：然后对其余各条路径进行适当调整：

  若在图中存在一条边$c_{t,x}$，且$d_{u,x} > d_{u,t} + c_{t,x}$，则以路径$(u, t, x)$代替$(u, x)$。
- 调整以后的各条路径中，再找长度最短的路径，依此类推。

3）实现步骤：

① S = { u }, T = V – { u }

② ∀ x ∈ T，若 $(u, x) \in E$，则$d_{u,x} = c_{u,x}$， $p(x) = u$ ，

否则，$d_{u,x} = \infty$ ， $p(x) = -1$ ；

③ 寻找 t ∈ T，使得 $d_{u,t} = min\{d_{u,x} | x \in T\}$

④ S = S ∪ { t }， T = T – { t }

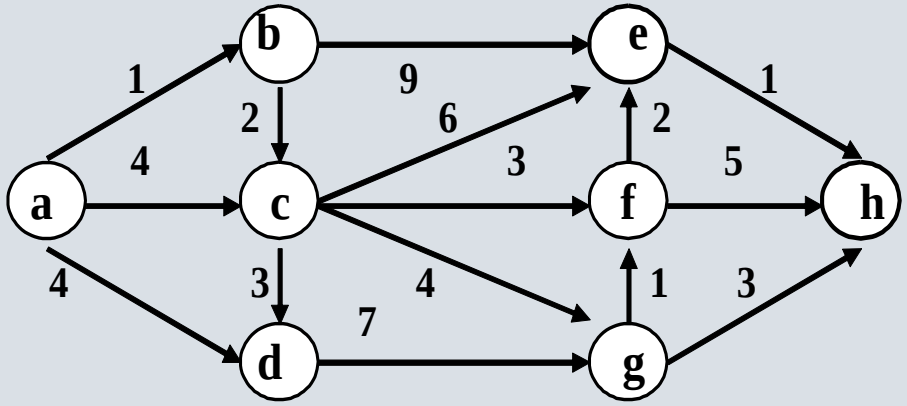
⑤ 若 T = Ø，算法结束；否则，转 ⑥

⑥ 对与 t 相邻接的所有顶点 x ，如果$d_{u,x} > d_{u,t} + c_{t,x}$

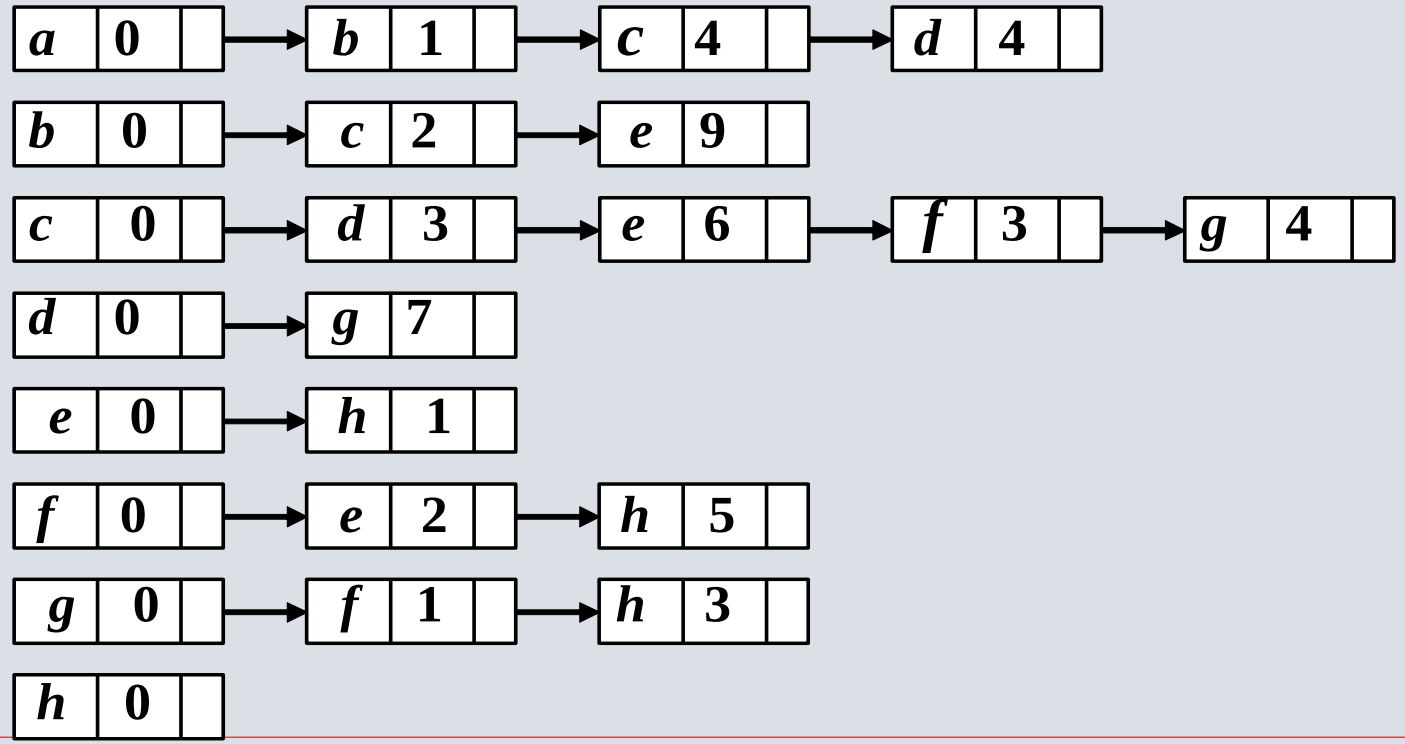
则令$d_{u,x} = d_{u,t} + c_{t,x}$， $p(x) = t$ ，转 ③

3. 例子

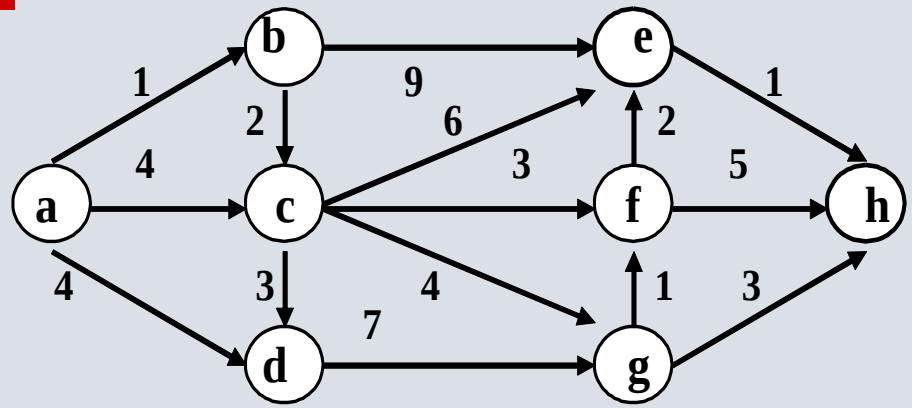
求图中顶点 a 到其它所有顶点的距离。



邻接表如下：

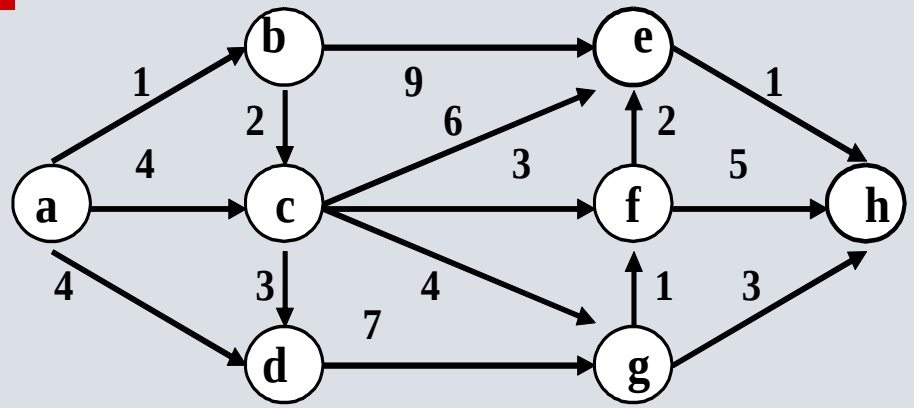


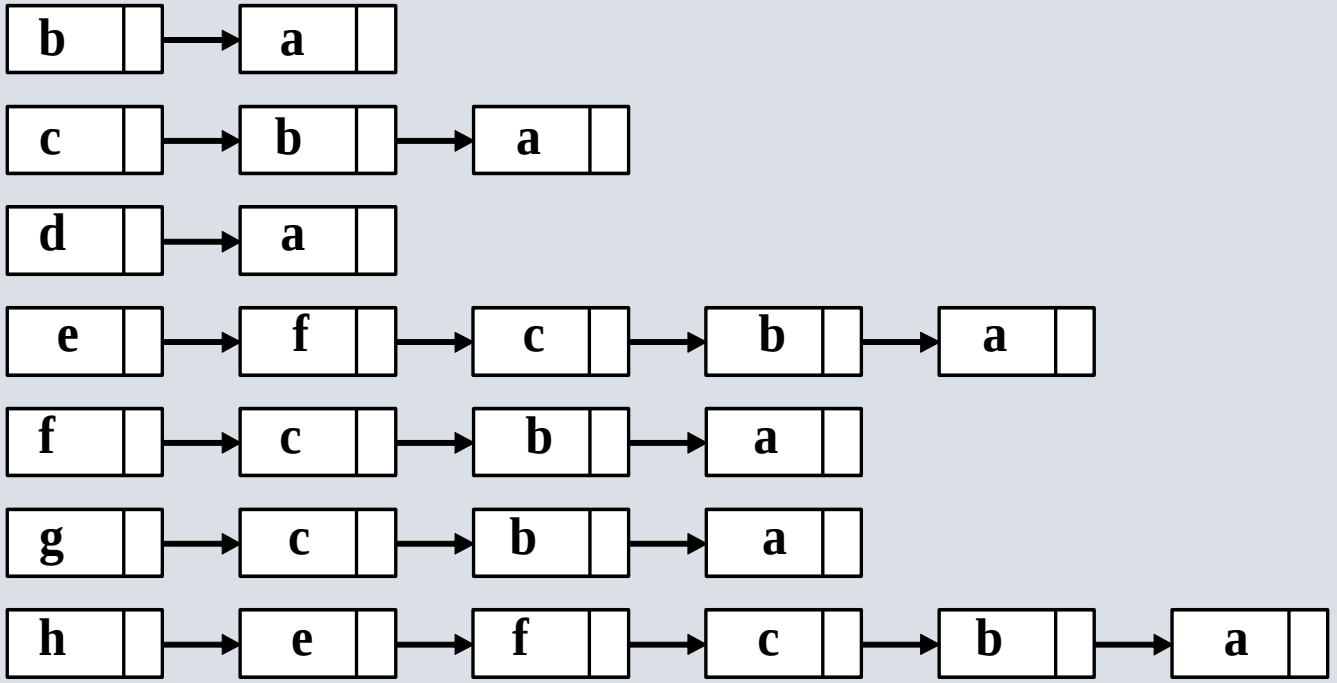
求图中顶点 a 到其它所有顶点的距离的求解过程：



	S	dab	dac	dad	dae	daf	dag	dah	dat	t
1	a	1	4	4	∞	∞	∞	∞	1	b
2	ab		3,b	4,a	10,b	∞	∞	∞	3	c
3	abc			4,a	9,c	6,c	7,c	∞	4	d
4	abcd				9,c	6,c	7,c	∞	6	f
5	abcdf				8,f		7,c	11,f	7	g
6	abcdfg				8,f			10,g	8	e
7	abcdfge							9,e	9	h
8	abcdfgeh									

顶点 a 到其它所有顶点的路径：





二　狄斯奎诺算法的描述

1. 数据结构
2. 算法描述

1. 数据结构

```
typedef  struct adj_list {      // 邻接表结点的数据结构
   int  v_num;                  // 邻接顶点的编号
   float  len;                  // 邻接顶点与该顶点的距离
   struct adj_list *next;       // 下一个邻接顶点
} NODE;


NODE  node[n];    // 邻接表头结点
float     d[n];       // 顶点到源顶点的距离
int       p[n];       // 顶点到源顶点的最短路径上前方顶点的编号
BOOL  s[n];       // 为真，顶点在 S 中，否则，不在S中
```

Dijkstra算法的设计思想

输入：有向图$G=(V,E)$, $V=\{0,1,\ldots,n-1\}$, $S=0$

输出：从S到每个顶点的最短路径

1. 初始$S=\{0\}$

2. 对于$i \in V-S$，计算0到i的相对S的最短路，长度$d_{0,i}$

3. 选择$V-S$中d值最小的j，将j加入S，修改$V-S$中顶点的d值。

4. 继续上述过程，直到$S=V$为止。

2. 算法描述（初始化，步骤①）-初始化

```
1. #define MAX_FLOT_NUM ∞                    // 最大的浮点数
2. void dijkstra(NODE node[ ], int n, int u, float d[ ], int p[ ])
3. {
4.     float   temp;
5.     int  i, j, t;
6.     BOOL  *s = new  BOOL[ n ];
7.     NODE  *pnode;
8.     for (i=0; i<n; i++) {       // 初始化
9.         d[ i ] = MAX_FLOAT_NUM;
           s[ i ] = FALSE;
           p[ i ] = - 1;
10.    }
```

算法描述（步骤①②）

```
11.    if (!(pnode = node[u].next)) // 源顶点与其它顶点不相邻接
12.        return;
13.    while (pnode) {    // 预置与源顶点相邻接的顶点距离
14.        d[pnode->v_num] = pnode->len;
15.        p[pnode->v_num] = u;
16.        pnode = pnode->next;
17.    }
18.    d[u] = 0;   s[u] = TRUE;
// 开始时,集合S仅包含顶点u
```

算法描述（步骤③④）-选择具有最短距离的顶点

```
19.    for (i=1; i<n; i++) {
20.        temp = MAX_FLOAT_NUM;   t = u;
21.        for (j=0; j<n; j++)  // 在 T 中寻找距离 u最近的顶点 t
22.            if (!s[ j ] && d[ j ] < temp) { /
23.                t = j;    temp = d[ j ];
24.            }
25.        if (t==u) break;                    // 找不到, 跳出循环
26.        s[ t ] = TRUE;                      // 否则, 把 t 并入集合s
```

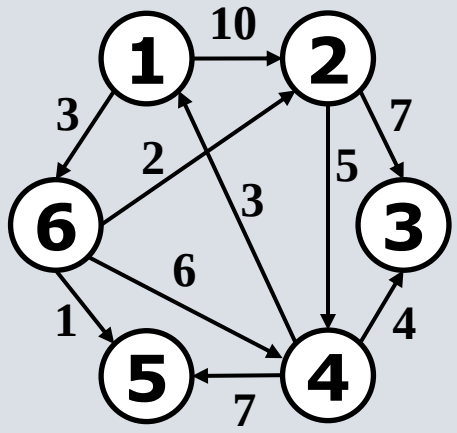
更新与t邻接的顶点到源点u的距离，进入新的一轮循环。

```
27.         pnode = node[ t ].next; // 更新与t相邻接的顶点到u的距离
28.         while (pnode) {
29.             if (!s[pnode->v_num] &&
                        d[pnode->v_num] > d[ t ]+pnode->len) {
30.                 d[pnode->v_num] = d[ t ] + pnode->len;
31.                 p[pnode->v_num] = t;
32.             }
33.             pnode = pnode->next;
34.         }
35.     }
36.     delete s;
37. }
```

实例：

输入：$G=<V,E,W>$，源点1，

$V=\{1,2,3,4,5,6\}$



$S=\{1\}$	$S=\{1,6\}$	$S=\{1,6,5\}$	$S=\{1,6,5,2\}$	$S=\{1,6,5,2,4\}$	$S=\{1,6,5,2,4,3\}$
$d_{1,2}=10$	$d_{1,2}=5$	$d_{1,2}=5$	$d_{1,2}=5$	$d_{1,2}=5$	$d_{1,2}=5$
$d_{1,3}=\infty$	$d_{1,3}=\infty$	$d_{1,3}=\infty$	$d_{1,3}=12$	$d_{1,3}=12$	$d_{1,3}=12$
$d_{1,4}=\infty$	$d_{1,4}=9$	$d_{1,4}=9$	$d_{1,4}=9$	$d_{1,4}=9$	$d_{1,4}=9$
$d_{1,5}=\infty$	$d_{1,5}=4$	$d_{1,5}=4$	$d_{1,5}=4$	$d_{1,5}=4$	$d_{1,5}=4$
$d_{1,6}=3$	$d_{1,6}=3$	$d_{1,6}=3$	$d_{1,6}=3$	$d_{1,6}=3$	$d_{1,6}=3$

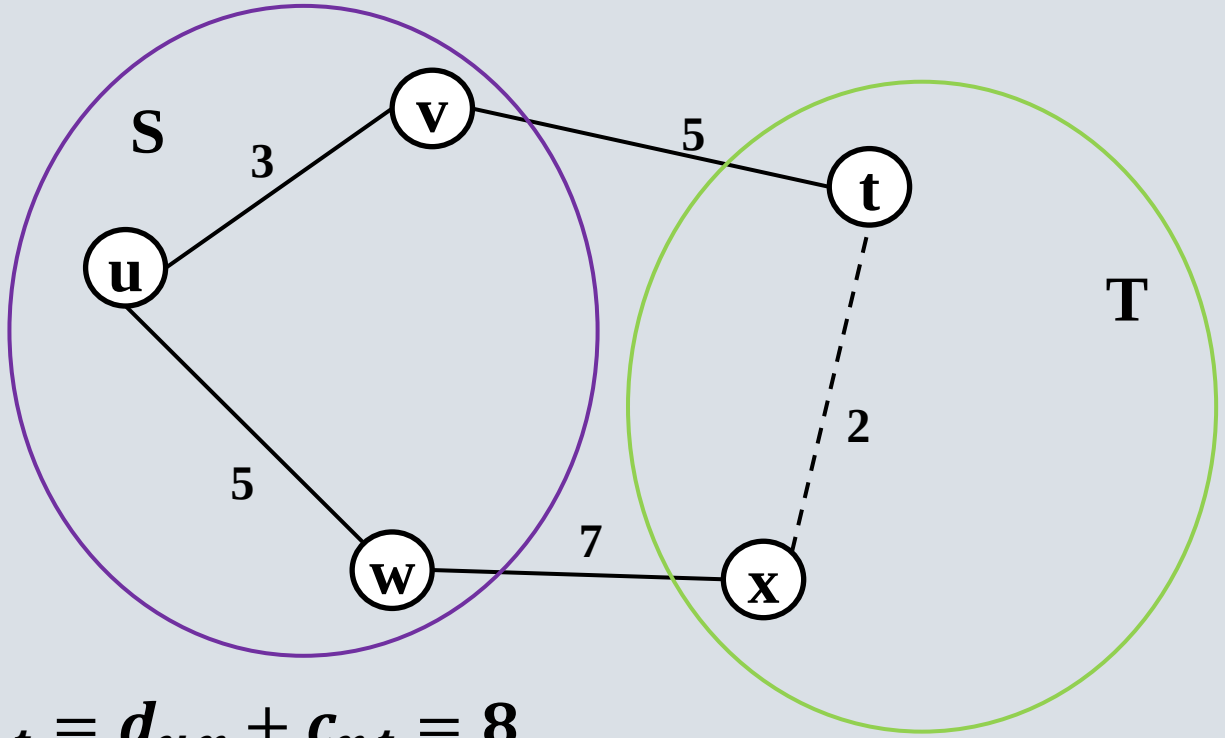
3. 狄斯奎诺算法的分析

- 时间复杂性：

  算法进行n-1步，每步挑选1个具有最小路径的结点进入S，需要$O(n)$时间。

  所以，算法的时间复杂性为$O(n^2)$

- 选用基于堆实现的优先队列的数据结构，可以将算法的时间复杂度降到$O(n\log n)$



$$d_{u,t} = d_{u,v} + c_{v,t} = 8$$

$$d_{u,x} = d_{u,w} + c_{w,x} = 12$$

当把T集合中的t点并入集合S后，$d_{u,x}$就不是u到x的最短路径长度了，因此需更新

$$d_{u,x} = min\{d_{u,x}, d_{u,t} + c_{t,x}\}$$

定理5.2 设$G=(V,E)$是有向赋权图，$S\subseteq V$， $u\in S$， $T=V-S$，若$t\in T$，$d_{u,t}=\min\{d_{u,x}\mid x\in T\}$，则$d_{u,t}$就是顶点$t$的最短路径长度。

定理5.3 设$G=(V,E)$是有向赋权图，$S\subseteq V$, $u\in S$，$T=V-S$， $t\in T$，$d_{u,t}=\min\{d_{u,x}\mid x\in T\}$；令$\overline{S}=S\cup\{t\}$， $\overline{T}=T-\{t\}$，则对任意的，有$d_{u,x}=min\{d_{u,x},d_{u,t}+c_{t,x}\}$

Dijkstra算法是正确的。

5.4 最小花费生成树问题

5.4.1 克鲁斯卡尔（Kruskal）算法

5.4.2 普里姆（Prim）算法

定义5.1：设图 $G = <V, E>$和图 $G' = < V', E' >$，若 $E' \subseteq E$且 $V' = V$，则称 G' 是 G 的生成子图。

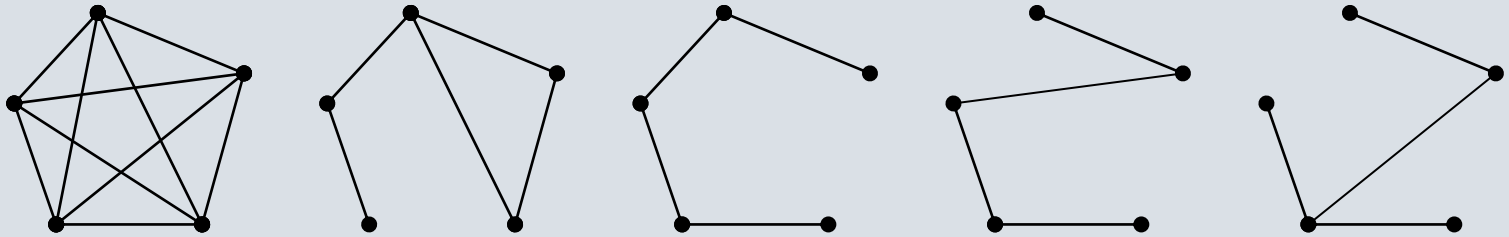
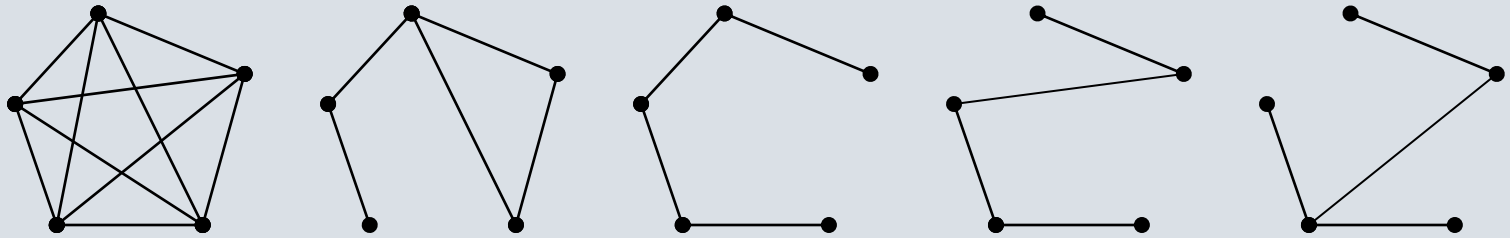


图 (a) 是无向完全图，图 (b), (c), (d), (e)是它的生成子图

定义5.2： 若无向图 G 的生成子图 T 是树，则称 T 是 G 的生成树或支撑树。生成树中的边称为树枝。

- 生成树：所有顶点均由边连接在一起，但不存在回路的图。
  - 一个图可以有许多棵不同的生成树：

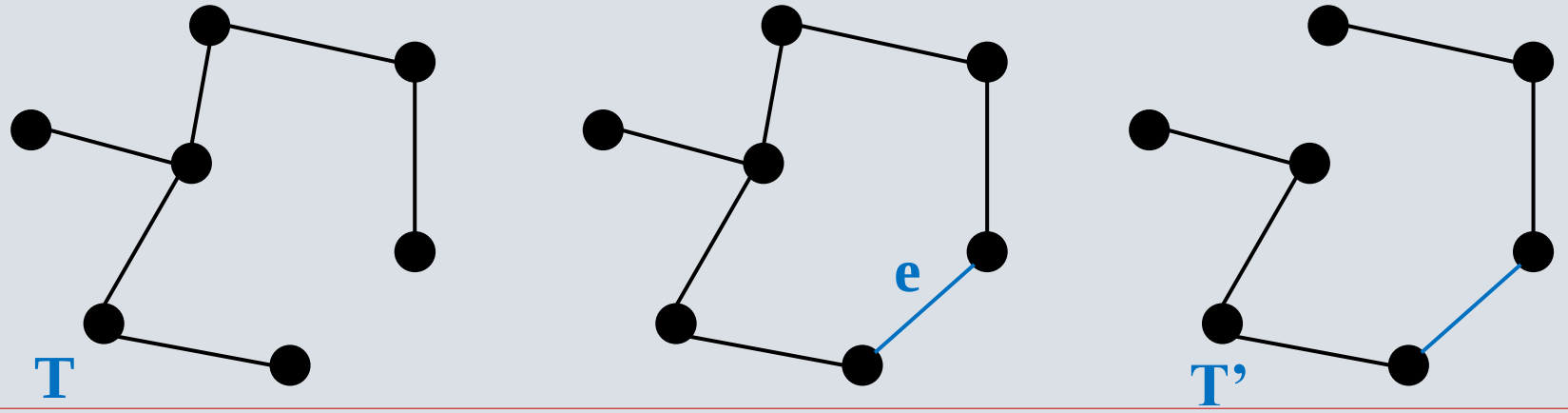


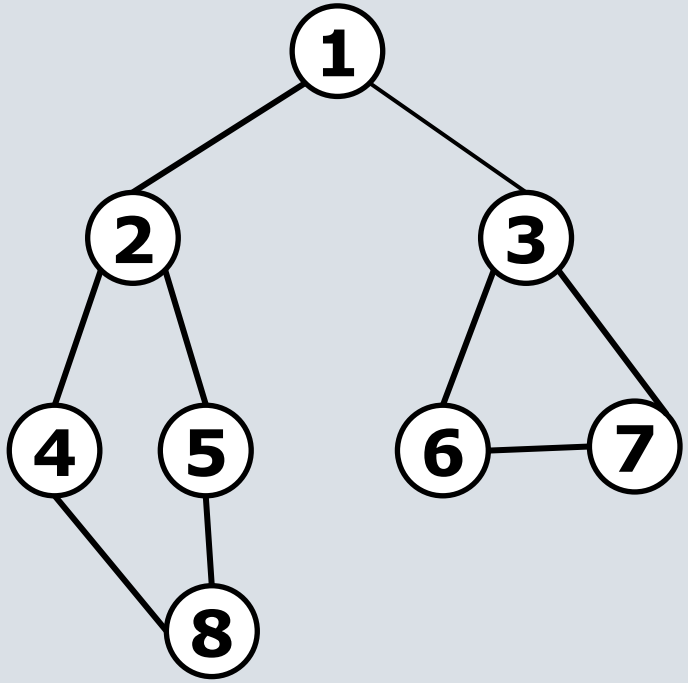
(a) (b) (c) (d) (e)

图（a）是无向图，图 (b) 不是图 (a) 的生成树，而图 (c), (d), (e)都是 (a) 的生成树。

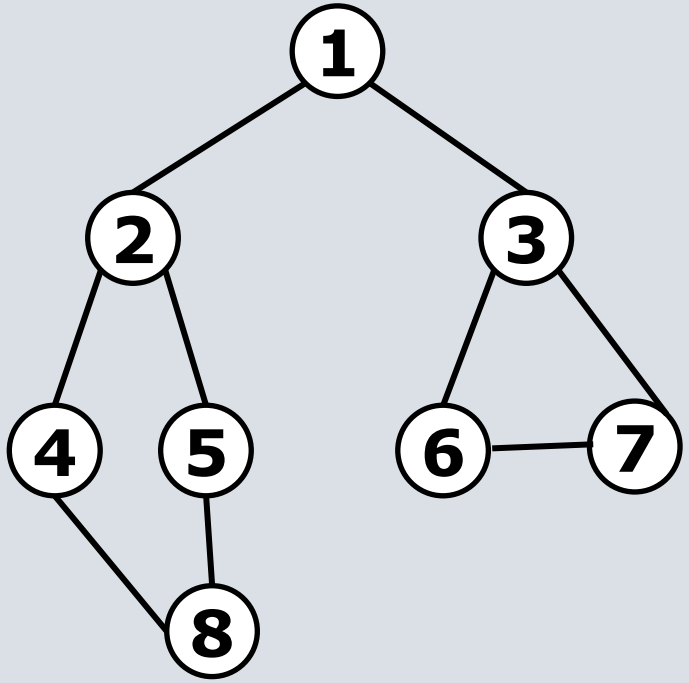
- 所有生成树具有以下共同特点：
  - 生成树的顶点个数与图的顶点个数相同；
  - 生成树是图的极小连通子图，去掉一条边则非连通；
  - 一个有n个顶点的连通图的生成树有$n-1$条边；
  - 在生成树中再加一条边必然形成回路。

- 若连通图 $G = <V, E>$ ， T 是 G 的生成树。则生成树 T 有如下性质：
  - 性质 1： T 是不含简单回路的连通图；
  - 性质 2： T 中的每一对顶点 u 和 v，恰好有一条从 u 到 v 的基本通路；
  - 性质 3：若$T = <V, E'>$， $|V| = n$，$|E'| = m$，则 $m = n - 1$；
  - 性质 4：在 T 中的任何两个不相邻接的顶点之间增加一条边，则得到 T 中唯一的一条基本回路。



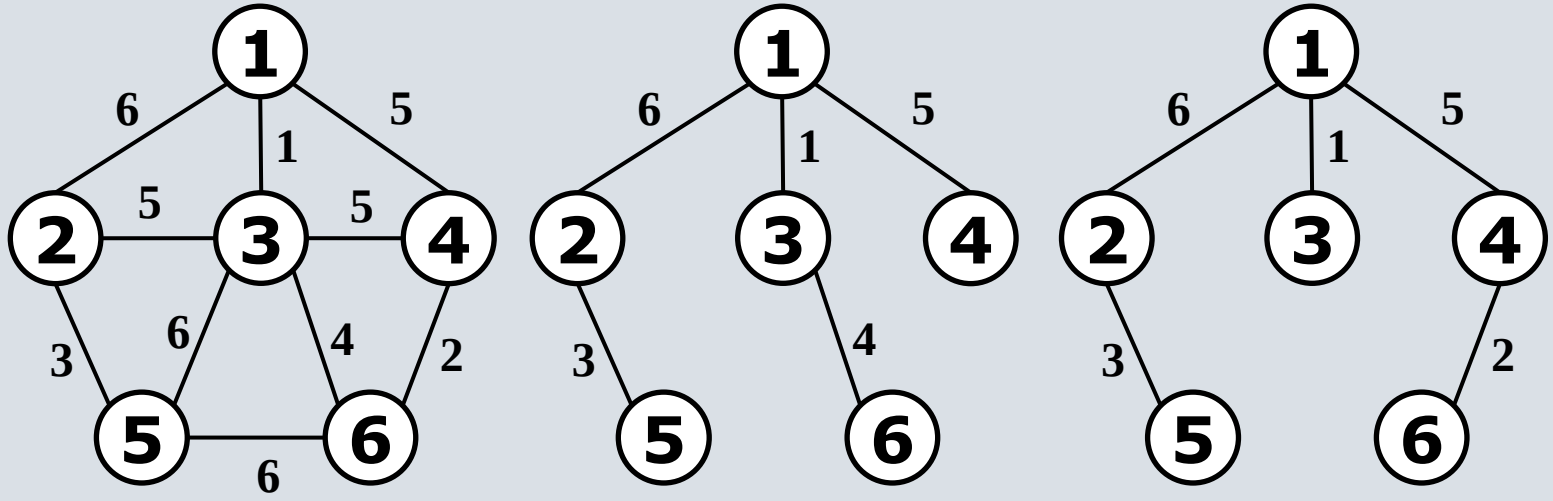


深度优先生成树



广度优先生成树

最小生成树



各边权值和=19　　各边权值和=17

最小生成树：给定一个无向网络，在该网的所有生成树中，使得各边权值之和最小的那棵生成树称为该网的最小生成树，也叫最小花费生成树。

定义5.3： 若图 $G = <V, E, W>$是赋权图，称 T 是 G 的生成树，则T的每个树枝上的权之和称为T的权； G中权最小的生成树，称为G的最小花费生成树或最小生成树。

最小生成树的典型用途

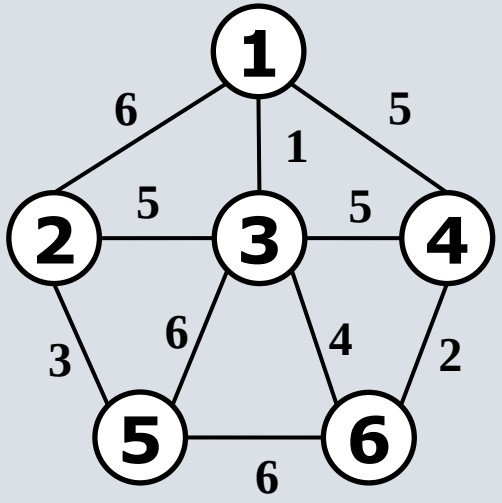
- 欲在*n*个城市间建立通信网，则*n*个城市应铺*n*-1条路线；
- 但因为每条路线都会有相应的经济成本，而*n*个城市最多有*n*(*n*-1)/2条路线，那么，如何选择*n*-1条路线，使总费用最少？

  - 数学模型：
    - ✓ 顶点——表示城市，有*n*个；
    - ✓ 边——表示线路，有*n*-1条；
    - ✓ 边的权值——表示线路的经济代价；
    - ✓ 连通网——表示*n*个城市间通信网。

此连通网就是一个生成树

构造最小生成树 Minimum Spanning Tree

- 构造最小生成树的算法很多，其中多数算法都利用了MST性质。
- MST性质：设G=(V,E)是一个连通图，U是顶点集V的一个非空子集。若边(u,v)是一条具有最小权值的边，其中u∈U，v∈V-U，则存在一棵包含边(u,v)的最小生成树。



$G=(V,E,W)$

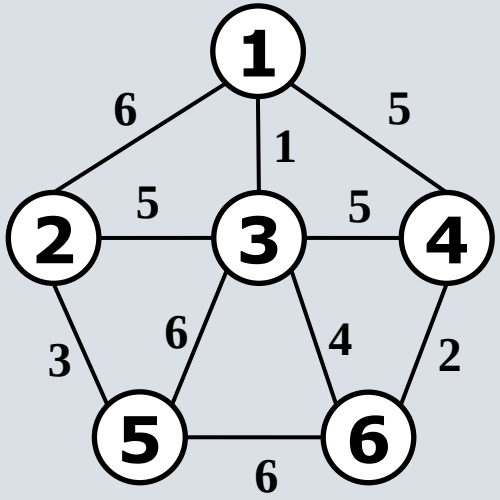
V={1,2,3,4,5,6}

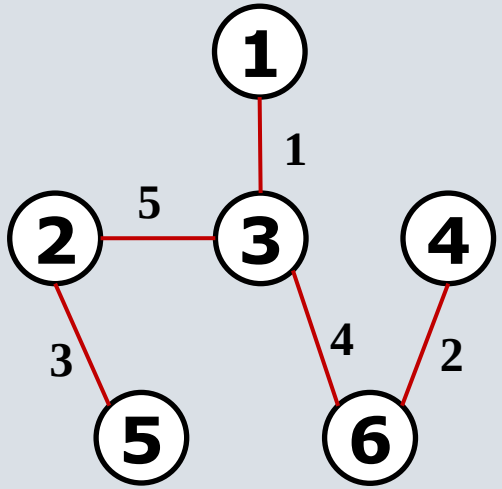
E = {{1,2},{1,3},{1,4},{2,3},{2,5},
{3,4},{3,5},{3,6},{4,6},{5,6}}

U={1}

$G = (V, E, W)$, V={1,2,3,4,5,6}

E = {{1,2},{1,3},{1,4},{2,3},{2,5},{3,4},{3,5},{3,6},{4,6},{5,6}}





MST性质解释

- 生成树的构造过程中，图中*n*个顶点分属两个集合：
  - 已经落在生成树上的顶点集：U
  - 尚未落在生成树上的顶点集：V-U

✓ 接下来则应在所有连通U中顶点和V-U中顶点的边中选取权值最小的边。

求最小生成树

- 问题：

  给定连通带权图$G=(V,E,W)$，$w(e)\in W$是边e的权，求G的一棵最小生成树。

- 贪婪法

  Kruskal算法

  Prim算法

- 生成树在网络中有着重要的应用

5.4.1 克鲁斯卡尔（Kruskal）算法

1. 克鲁斯卡尔算法的设计思想
2. 克鲁斯卡尔算法的实现
3. 克鲁斯卡尔算法的分析

1. 克鲁斯卡尔算法的设计思想

输入：图 $G=(V,E,W)$，　V={1,2,…,n}

输出：G的最小生成树

设计思想：

（1）按照长度从小到大对边排序；

（2）依次考察当前最短边e，如果e与T的边不构成回路，则把e加入树T，否则删除e；直到选择了n-1条边为止。

伪码

输入：连通图 G //顶点数n，边数m

输出：G的最小生成树

1. 权从小到大排序E的边，$E=\{e_1,e_2,\ldots,e_m\}$
2. $T\leftarrow\varnothing$
3. repeat
4. e←E中最短边
5. if e的两端点不在同一连通分支
6. then $T \leftarrow T \cup \{e\}$
7. $E \leftarrow E - \{e\}$
8. until T包含了n-1条边

实现步骤

① 按权的非降顺序排序 E 中的边；

② 令最小花费生成树的边集为 T，T 初始化为：$T = \varnothing$；

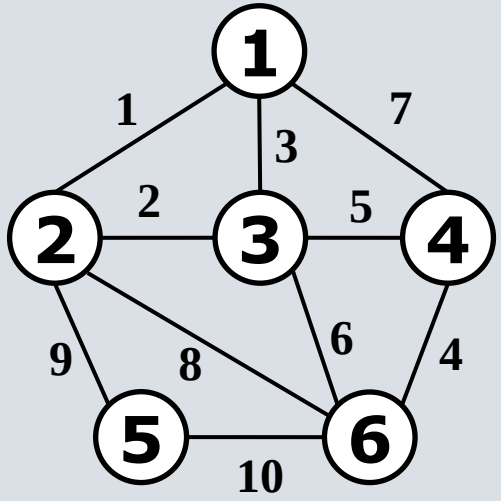
③ 把每个顶点都初始化为树的根结点；

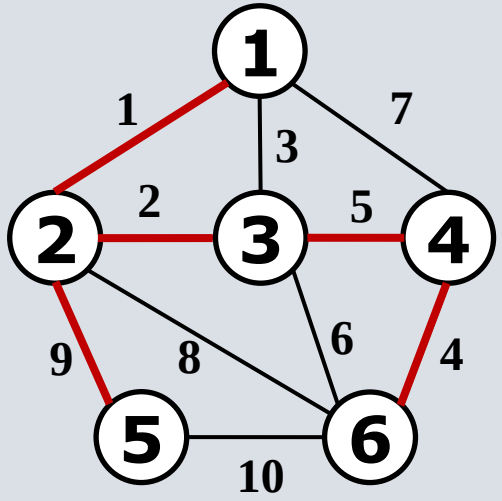
④ 令 e = (u, v) 是 E 中权最小的边，E = E – { e }；

⑤ 如果 find(u) ≠ find(v)，则执行 union(u, v) 操作，T = T ∪ { e }；

⑥ 如果 | T | < n – 1，转 ④ ；否则，算法结束。

克鲁斯卡尔算法的工作过程





2. 克鲁斯卡尔算法的实现

- 数据结构：

```
typedef struct {                    // 边的数据结构
    float     key;                  // 边的权
    int         u;                  // 与边关联的顶点编号
    int         v;                  // 与边关联的顶点编号
} EDGE;
typedef struct node {               // 顶点的数据结构
    struct node   *p;               // 指向父亲结点
    int         rank;               // 结点的秩
    int         u;                  // 顶点编号
};
typedef struct node NODE;
EDGE      E[ m + 1 ], T[ n ];  // 图的边集及生成树
NODE      V[ n ];                          // 图的顶点集合
```

算法描述

```
1. void kruskal(NODE V[ ], EDGE E[ ],
                EDGE T[ ], int n, int m)
2. {
3.     int i, j, k;
4.     EDGE  e;
5.     NODE  *u, *v;
6.     make_heap(E,m);        // 用边集构成最小堆
7.     for (i=0; i<n; i++) {      // 用顶点作为树的根结点构成森林
8.         V[ i ].rank = 0;   V[ i ].p = NULL;
9.     }
```

算法描述

```
10.     i = j = 0;
11.     while ((i<n-1) && (m>0) {
12.         e = delete_min(E, m);      // 从堆中取出权最小的边
13.         u = find(&V[ e.u ]);       // 检索该边两顶点所在树根
14.         v = find(&V[ e.v ]);
15.         if (u != v) {              // 两顶点不在同一棵树
16.             union(u, v);           // 合并成一棵树
17.             T[ j++ ] = e;          // 该边置于生成树中
18.             i++;
19.         }
20.     }
21. }
```

3. 克鲁斯卡尔算法的分析

1）时间和空间复杂性：

- 第 6 行用 m 条边构成最小堆，花费 $O(m\log m)$；
- 第 12 行的 delete_min 操作执行 m 次，花费 $O(m\log m)$；
- find 操作执行 $2m$ 次， union 操作最多执行 m 次,最多花费 $O(m\log n)$ 时间。

✓ 算法总的运行时间为$O(m\log m)$

- 对完全图，$m = n(n-1)/2$，花费的时间为$O(n^2\log n)$；
- 对平面图，$m = O(n)$，花费的时间为 $O(n\log n)$

- 工作单元：
  - 最小生成树边集所需空间为 $\Theta(n)$；
  - 其余工作单元为 $\Theta(1)$

2）算法的正确性证明

- 设G 是无向连通图，T^* 是最小花费生成树边集；
- T 是由克鲁斯卡尔算法所产生的生成树边集；
- G 中的顶点，既是 T 中的顶点，也是 T^* 中的顶点；
- 若 G 的顶点数为 n，则 $|T| = |T^*| = n - 1$；

用归纳法证明 $T = T^*$

正确性证明思路

- 命题：对于任意n，算法对n阶图找到一棵最小生成树。
- 证明思路：

归纳基础 证明：$n=2$，算法正确。G只有一条边，最小生成树就是G。

归纳步骤 证明：假设算法对于n阶图是正确的，其中$n>1$，则对于任何$n+1$阶图算法，也得到一棵最小生成树。

① 设 e_1是 G 中权最小的边，根据克鲁斯卡尔算法，有 $e_1 \in T$

若$e_1 \notin T^*$，而 T^*是 G 的最小花费生成树

和 e_1关联的顶点必是 T^*中的两个不相邻接的顶点

把 e_1加入 T^*，将使 T^* 构成唯一的一条回路

假定回路是 $e_1, e_{a1}, \cdots, e_{ak}$ ，e_1 是这条回路中权最小的边

令 $T^{**} = T^* \cup \{e_1\} - \{e_{ai}\}$，$e_{ai}$ 是 $e_{a1}, \cdots, e_{ak}$ 的任意一条边

边集 T^{**}仍然是G的生成树，

且 T^{**}的权小于或等于 T^*的权

- 若T^{**}的权小于T^*的权，与T^* 是 G 的最小花费生成树的边集矛盾，所以，$e_1 \in T^*$
- 若T^{**}的权等于T^*的权，则T^{**}也是G的最小花费生成树的边集，且$e_1 \in T^{**}$，用新的 T^* 来标记。

在这两种情况下，都有 $e_1 \in T^*$

② 设 $e_1,\cdots,e_k$ 是 T 和 T^*中前面 k 条权最小的边

令 e_{k+1}是 T 中第 $k+1$ 条权最小的边，$e_{k+1}\in T$ ，但$e_{k+1}\notin T^*$

和 e_{k+1}关联的顶点也是T^* 中的两个不相邻接的顶点

把 e_{k+1} 加入 T^*，将使 T^*构成唯一的一条回路

假定回路是 $e_{k+1},e_{a1},\cdots,e_{am}$ ，在$e_{a1},\cdots,e_{am}$中，

至少有一条边 $e_{ai}\in T^*$； 但 $e_{ai}\notin T$，否则，T 将存在回路

$e_1,\cdots,e_{k+1}$是 T 中前面$k+1$ 条权最小的边，

据克鲁斯卡尔算法，在 T 和T^*中，除$e_1,\cdots,e_{k+1}$外，

不存在权大于 e_1 且小于e_{k+1}的其他边，

所以，e_{ai} 的权大于 e_{k+1} 的权，

令$T^{**}=T^*\cup\{e_{k+1}\}-\{e_{ai}\}$，则$T^{**}$仍然是 G 的生成树，且T^{**}

的权小于或等于T^* 的权。同 ①，有 $e_{k+1}\in T^*$

综上所述，有$T=T^*$

5.4.2 普里姆（Prim）算法

1. 普里姆算法的思想方法
2. 普里姆算法的实现
3. 普里姆算法的分析

1. Prim算法的算法思想

输入：图 $G=(V,E,W)$，　V={0,1,…,n-1}

输出：最小生成树T

算法思想：

（1）初始 $S=\{0\}$；

（2）选择连接S与V-S集合的最短边 $e=\{i,j\}$，其中$i\in S, j\in V$-S，将e加入树T，j加入S；

（3）继续执行上述过程，直到 $S=V$ 为止。

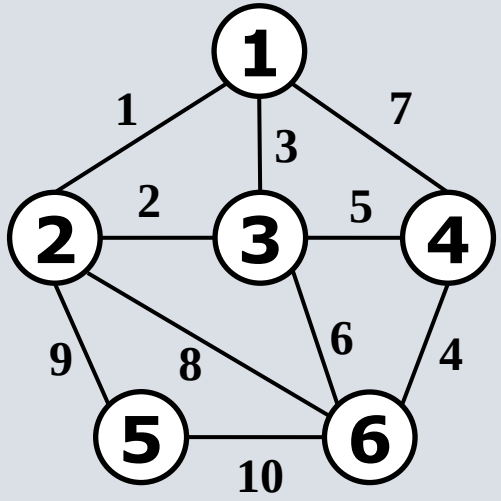
伪码

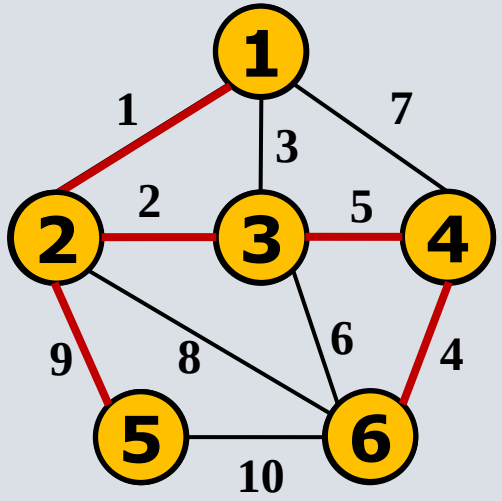
输入：图 $G=(V,E,W)$，　V={0,1,…,n-1}

输出：最小生成树T

1. $T=\varnothing$，$S\leftarrow${0};
2. while $N=V-S\neq\varnothing$　do;
3. 从 N 中选择 j，使得 j 到S中顶点的边权最小;
4. $S\leftarrow S\cup\{j\}$, N = N – { j }, T=T $\cup\{e_{i,j}\}$

Prim算法的工作过程





2. 普里姆算法的实现

1）寻找 $i \in S$，$j \in N$，c[i][j] 最小的 i 和 j 的方法：

- 边界点：若存在边 e_{ij} , $i \in S, j \in N$， 称 j 为边界点；边界点是由集合 N 转移到集合 S 的候选者；
- 近邻：若 j 是边界点， S 中至少有一个顶点 i 和 j 相邻接。把 S 中与 j 相邻接、且权 c[i][j] 最小的顶点 i ，称为顶点 j 的近邻。

- ✓ 近邻信息表：
  - 数组 neig[*n*]：顶点 *j* 的近邻
  - 数组w[*n*]：顶点 *j* 与近邻相关联的边权
- ✓ 初始化时，置neig[*j*] = 0，w[*j*] = c[0][*j*]；
- ✓ 求 w[*j*] 最小的 *j*，再从 neig[*j*] 取得 *j* 的近邻*i* ；
- ✓ 则边e_{ij} 就是 T 中的边；
- ✓ 把 *j* 加入 S，更新 N 中顶点 *j* 的neig[*j*] 和 w[*j*]。

2）数据结构

```
float    c[n][n];   // 图的邻接矩阵
BOOL     s[n];      // 集合 S中的点集
EDGE     T[n];      // 最小花费生成树的边集
int      neig[n];   // 顶点 j 的近邻（集合S中的顶点）
float    w[n];      // 顶点 j 与近邻相关联的边的权
```

3）算法描述（初始化）

```
2. void prim(float c[ ][ ], int n, EDGE T[ ], int &k)
3. {   int i, j, k, u;
5.     BOOL  *s = new  BOOL[ n ];
6.     int  *neig = new int[ n ];
7.     float min, *w = new  float[ n ];
8.     s[ 0 ] = TRUE; // s = {0}
9.     for (i=1; i<n; i++) {
10.        w[ i ] = c[ 0 ][ i ];
11.        neig[ i ] = 0;
12.        s[ i ] = FALSE;
13.    }
```

算法描述（挑选和更新）

```
14.    k = 0;
15.    for (i=1; i<n; i++) {
16.        u = 0;
17.        min = MAX_FLOAT_NUM;
18.        for (j=1; j<n; j++)
19.            if (!s[ j ] && w[ j ] < min) {
20.                u = j;   min = w[ j ];
21.            }
22.        if (u==0) break;
23.        T[ k ].u = neig[ u ];    T[ k ].v = u;
25.        T[ k++ ].key = w[u];  s[ u ] = TRUE;
```

算法描述（挑选和更新）

```
27.        for (j=1; j<n; j++) { // 更新N中顶点的近邻信息
28.            if (!s[ j ] && c[ u ][ j ] < w[ j ]) {
29.                w[ j ] = c[ u ][ j ];
30.                neig[ j ] = u;
31.            }
32.        }
33.    }
34.    delete s;   delete w;   delete neig;
35. }
```

3. 普里姆算法的分析

- 时间复杂性：
  - 算法步骤执行$O(n)$次
  - 每次执行$\Theta(n)$时间：

    找连接S与V-S的最短边
  - 算法时间：$T(n)= \Theta(n^2)$
- 空间复杂度：$\Theta(n)$

- 算法的正确性证明：归纳法

命题：对于任意$k<n$，存在一棵最小生成树，包含算法前k步选择的边。

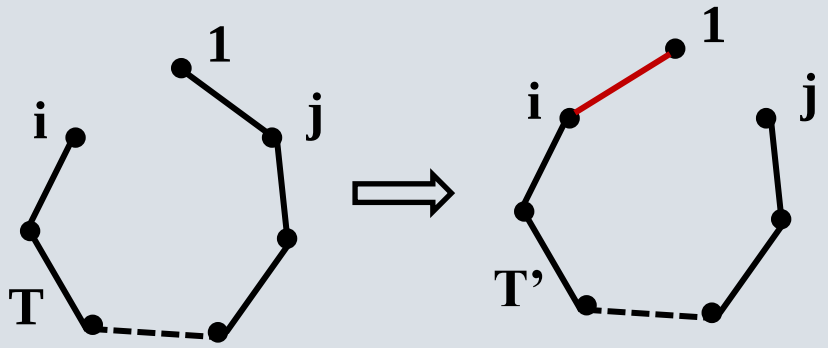
归纳基础：$k=1$，存在一棵最小生成树T包含边$e = \{1, i\}$，其中$\{1, i\}$是所有关联1的边中权最小的。

归纳步骤：假设算法前k步选择的边构成一棵最小生成树的边，则算法前$k+1$步选择的边也构成一棵最小生成树的边。

➢ 归纳基础

证明：存在一棵最小生成树T包含关联结点1的最小权的边$e = \{1, i\}$。

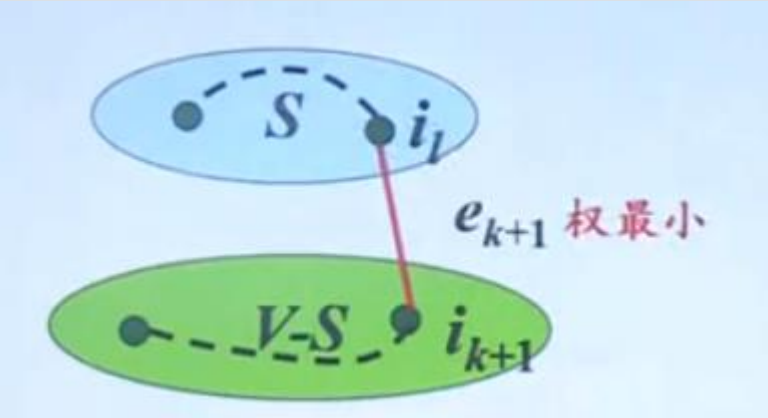
证：设T为一棵最小生成树，假设T不包含$\{1, i\}$，则$T \cup \{\{1, i\}\}$含有一条回路，回路中关联1的另一条边$\{1, j\}$。用$\{1, i\}$替换$\{1, j\}$得到树T'，则T'也是生成树，且$W(T') \leq W(T)$。



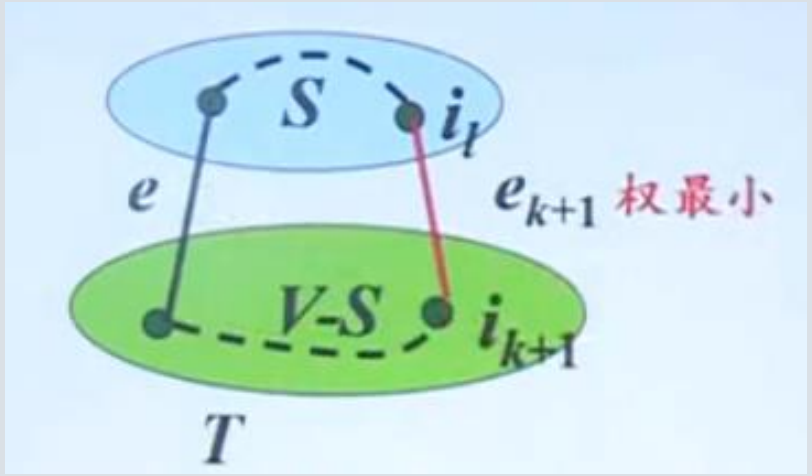
➢ 归纳步骤

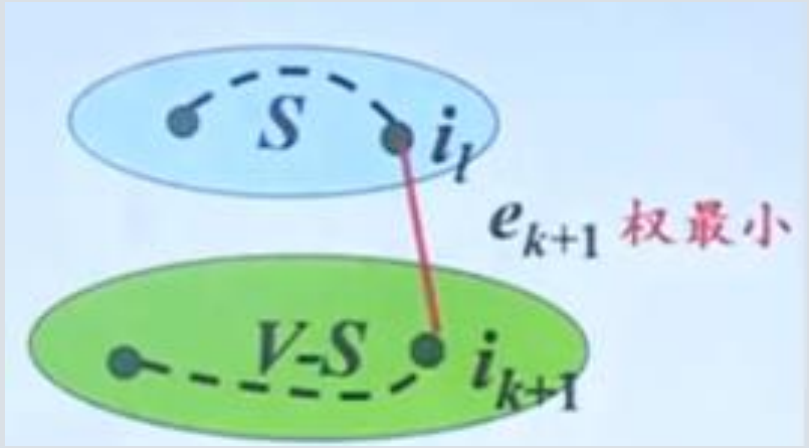
假设算法进行了k步，生成树的边为$e_1, e_2, \ldots, e_k$，这些边的端点构成集合S。由归纳假设存在G的一棵最小生成树T包含这些边。

算法第k+1步选择顶点i_{k+1}，则i_{k+1}到S中顶点边权最小，设此边$e_{k+1} = \{i_{k+1}, i_l\}$。若$e_{k+1} \in T$，算法$k$+1步显然正确。



- 假设T不含有e_{k+1}，则将e_{k+1}加到T中形成一条回路。这条回路有另外一条连接S与V-S中顶点的边e，令$T^* = (T - \{e\}) \cup \{e_{k+1}\}$，则$T^*$是$G$的一棵生成树，包含$e_1, e_2, \ldots, e_{k+1}$，且$W(T^*) \leq W(T)$。算法到$k$+1步仍得到最小生成树。





T*

两种算法比较

算法名	Kruskal	Prim
算法思想	选择边	选择点
时间复杂度	O(*e*log*e*)	O(n^2)
	（*e*为边数）	（*n*为顶点数）
适应范围	稀疏图	稠密图

5.5 霍夫曼编码问题

5.5.1 前缀码和最优二叉树

5.5.2 霍夫曼编码的实现

- 数据压缩问题
- **Huffman**编码是一种有效且被广泛应用的数据压缩技术；一般可压缩掉**20%**到**90%**，这取决于被压缩数据的特征。
- 假定数据为一列字符，**Huffman**贪婪算法使用了一张字符频度表，根据它来构造一种将每个字符表示成二进串的最优方式。

字符编码问题

假设我们有一个包含100,000(*a*-*f*)个字符的数据文件要压缩存储。各字符的出现频度见下表：

	a	*b*	*c*	*d*	*e*	*f*
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

我们可以编码多少位呢？

固定长度编码：$100,000 \times 3 = 300,000$ bits

可变长度编码：$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1,000 = 224,000$ bits

- **前缀编码**：没有一个编码是另一个编码的前缀。

可以证明，由字符编码获得的最小数据压缩总可用某种前缀编码来获得。

- **编码**：将文件中每个字符的编码并置起来即可

$$abc \rightarrow 0 \cdot 101 \cdot 100 \rightarrow 0101100$$

- 前缀编码的好处在于它简化了编码与解码

$$0101100 \rightarrow 0 \cdot 101 \cdot 100 \rightarrow abc$$

5.5.1 前缀码和最优二叉树

1. 霍夫曼编码思想：

利用文件中字符出现的不同频率，设计不同长度的字符代码，以达到压缩文件长度的目的。

2. 前缀码：

1）假设 $a_1a_2a_3\cdots a_n$ 是长度为 n 的字符串，称 $a_1a_2a_3\cdots a_k$

(k=1,2,…,n-1)为字符串 $a_1a_2a_3\cdots a_n$ 的长度为 k 的前缀。

2）二元前缀码：

假设字符串集合 $A=\{s_1,s_2,\cdots,s_m\}$ 。若对任意的 $s_i \in A$ ，$s_j \in A$ ，$i \neq j$ ，s_i和s_j不互为前缀，则称 A 为前缀码。

特别的，若 $(i=1,2,\ldots,m)$中只出现 0 和 1 两种符号，则称 A 为二元前缀码 。

- 二元前缀码

用0-1字符串作为代码表示字符，要求任何字符的代码都不能作为其他字符代码的前缀。

- 非前缀码的例子：

  例：字符 *a, b, c, d* 编码为：

a	*b*	*c*	*d*	*e*
01	011	110	111	11

字符串 *abcd* 的码文 01 011 110 111 将作为

01 01 111 01 11 被译成 *aadae*

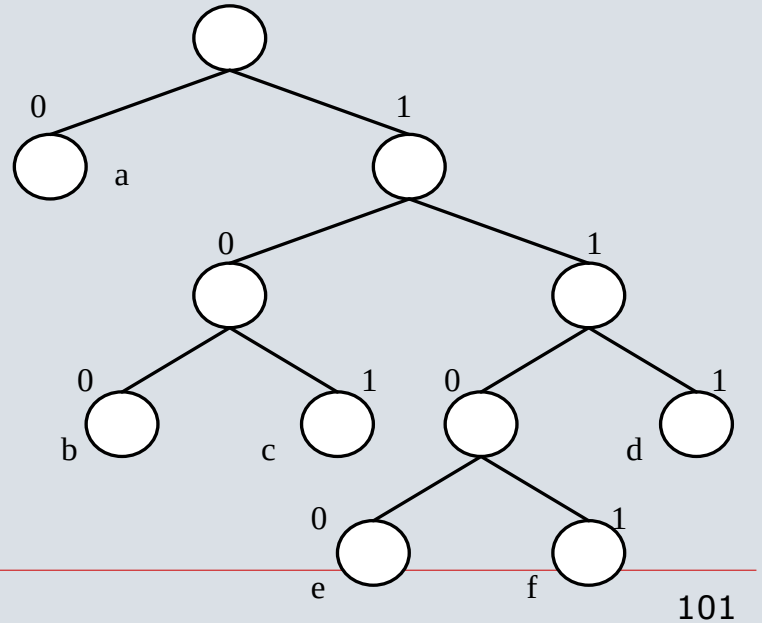
✓ 一个字符的编码是另一个字符编码的前缀，在译码过程中可能产生二义性。

3. 二元前缀码和二叉树的关系：

一个具有 ***n*** 个字符编码的二元前缀码，可以表示成一棵有 ***n*** 片叶子的二叉树，每片叶子代表一个字符，而把字符的编码看成是从根沿着表示该字符的叶子的路径，**0 表示左子树方向路径**、**1 表示右子树方向路径**。

例：字符 a,b,c,d,e 编码为：0　100　101　111　1100　1101

译码时，从根结点开始，沿码文所标示的路径向下搜索，直到叶子结点，可得到所编码的字符，再从根结点开始，继续搜索下一个字符。



4. 霍夫曼编码

对字符集$c_1, c_2, \cdots, c_n$进行编码时，令p_i为文件中字符c_i出现的频率，$l(c_i)$为对应二叉树中相应于c_i的叶子结点到根结点的路径长度，使

$$W(T) = \sum_{i=1}^{n} p_i \cdot l(c_i)$$

最小的二叉树T，其相应的前缀码便是一个最佳编码，称为霍夫曼编码，相应的树称为霍夫曼树。

5. 霍夫曼树的定义：

定义5.6 若二叉树 T 的每个分支都有 2 个儿子，称 T 是二叉正则树

定义5.7 若二叉正则树 T 有 n 片叶子 c_i，分别带权 p_i，$i=1,2,\cdots,n$，则称

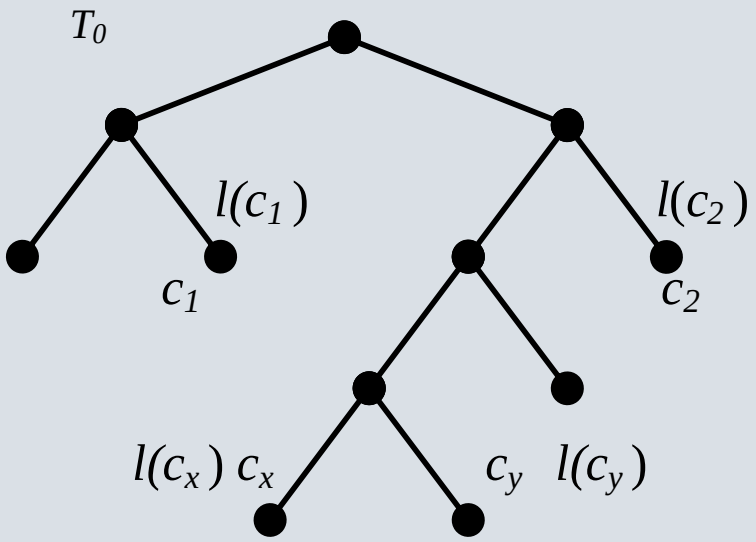
$$W(T)=\sum_{i=1}^{n} p_i \cdot l(c_i) \qquad (5.5.1)$$

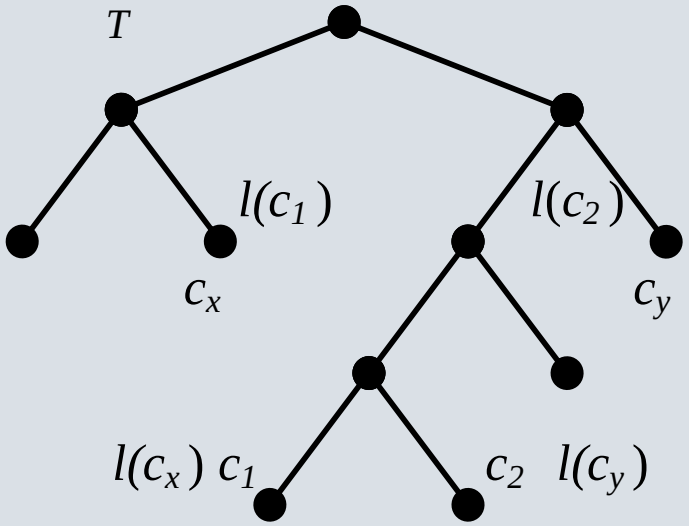
为树 T 的权。在带权 p_i 的 n 片叶子 c_i 的所有二叉正则树中，使式 (5.5.1) 最小的树，称为带权 p_i 的最优二叉树。最优二叉树也称霍夫曼树。

6. 霍夫曼树的构造：

定理 5.6 带权 $p_1 \le p_2 \le \cdots \le p_n$的最优二叉树中，必有二叉树 T，使得带权 p_1和 p_2的两片叶子是兄弟。

证明：设 T_0 是带权 $p_1 \le p_2 \le \cdots \le p_n$的最优二叉树，$c_x$ 和 c_y 是 T_0 中路径最长的两片叶子，它们分别带权 p_x和 p_y





$$p_x \geq p_i \quad p_y \geq p_i \quad l(c_x) \geq l(c_i) \quad l(c_y) \geq l(c_i) \quad i = 1, 2$$

把带权的叶子c_1、c_2分别与带权的c_x和c_y 及权互换，

得到新树 T，有：

$$W(T)-W(T_0)=p_1l(c_x)+p_2l(c_y)+p_xl(c_1)+p_yl(c_2)-p_xl(c_x)-p_yl(c_y)-p_1l(c_1)-p_2l(c_2)$$

$$=l(c_x)(p_1-p_x)+l(c_y)(p_2-p_y)+l(c_1)(p_x-p_1)+l(c_2)(p_y-p_2)$$

$$=l(c_x)(p_1-p_x)+l(c_y)(p_2-p_y)-l(c_1)(p_1-p_x)-l(c_2)(p_2-p_y)$$

$$=(p_1-p_x)(l(c_x)-l(c_1))+(p_2-p_y)(l(c_y)-l(c_2))$$

$$\leq 0$$

T_0 是最优二叉树，所以，T 也是最优二叉树，且c_1和c_2是兄弟

定理 5.7 T_1 是带权 $p_1+p_2,\ p_3,\ \cdots,\ p_n$ 的最优二叉树，$p_1 \le p_2 \le p_3 \le \cdots \le p_n$，在 T_1 中，使带权 p_1+p_2 的叶子产生两片分别带权为 p_1 和 p_2 的叶子，则得到带权为 $p_1, p_2, \cdots, p_n$ 的最优二叉树 T。

证明 因为 T_1 是带权 $p_1+p_2,\ p_3,\ \cdots,\ p_n$ 的最优二叉树，

而 T 是 T_1 中带权 p_1+p_2 的叶子产生两片分别带权为 p_1 和 p_2 的叶子得到的带权为 $p_1, p_2, \cdots, p_n$ 的二叉树，

令 $l(p_1+p_2)$ 为带权 p_1+p_2 的叶子到根结点路径的长度，

$l(p_1)$ 和 $l(p_2)$ 分别为带权为 p_1 和 p_2 的叶子到根结点路径的长度。则有：

$$l(p_1)=l(p_2)=l(p_1+p_2)+1$$

所以有：$W(T)=W(T_1)+p_1+p_2$

设T'是带权$p_1,p_2,\cdots,p_n$的最优二叉树，据定理**5.6**，可得：

带权为p_1和p_2的叶子c_1和c_2是兄弟。

在T'中删去c_1和c_2，使其父结点的权为p_1+p_2，得到树T_1'。

同样有：$W(T')=W(T_1')+p_1+p_2$

因为T_1是最优二叉树，所以：

$$W(T_1')\geq W(T_1)$$

而

$$\begin{aligned}W(T)-W(T')&=W(T_1)+p_1+p_2-(W(T_1')+p_1+p_2)\\&=W(T_1)-W(T_1')\\&\leq 0\end{aligned}$$

因为T'是最优二叉树，所以，T是最优二叉树。

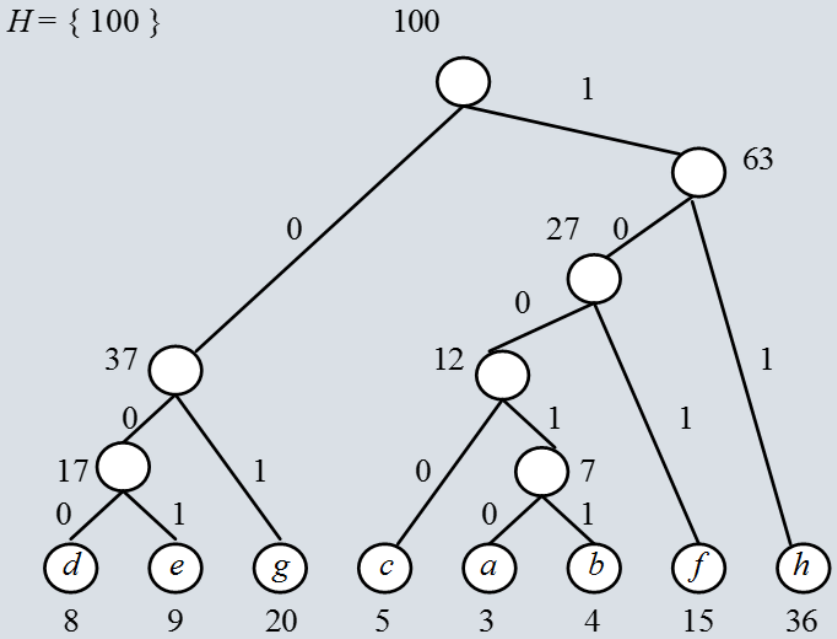
例：有 8 个字符的字符集 {a,b,c,d,e,f,g,h}，它们在文件中出现的频率的百分比分别为3,4,5,8,9,15,20,36：则构造的最优二叉树如下所示，对应的霍夫曼编码分别为：

10010，10011，

1000，000，

001，101，

01，11。



二　霍夫曼编码的实现

1．思想方法：假定，n 个符号的字符集 $C=\{c_1,c_2,\cdots,c_n\}$，
每一个符号在文件中出现的频率分别为 $p_1,p_2,\cdots,p_n$。
把代表符号的 n 个叶子结点作为带权的 n 棵树，
它们构成一个森林。
在森林中找出权最小的两棵树，
把它合并成一棵树，原来的两棵树成为新树的儿子结点，
新树的权为原来两棵树的权之和。
这种做法共进行 $n-1$ 遍，就把原来有 n 棵树的森林合并成一棵树，它就是所希望的最优二叉树。

二　霍夫曼编码的实现

算法思想：

- 每次选择两个频率最小的作为兄弟，把它们合并成一个父结点，这个父结点的频率为这两个最小频率之和；
- 通过合并后，在队列中产生一个父结点，它的两个儿子结点就是两个最小频率的字符；
- 在这个队列中，不断找两个最小的结点合并，合并以后把它从队列删除，把父结点插入队列中；
- 直到最后队列中只剩下一个结点时，整个霍夫曼树就生成了。

2. 实现步骤：

（1）把字符集C中n个符号的c_i及 p_i分别复制到数组$T[0]\sim T[n-1]$，作为二叉树T的n个叶子结点，叶子结点的左、右儿子指针及父亲指针初始化为-1。

（2）把字符集C中n个符号的c_i 及 p_i复制到数组$H[1]\sim H[n]$，H 中元素的 *index* 变量指向该元素在 T 中的下标，以 H 中元素的成员变量p 作为关键字，把 H 构造为最小堆。

（3）令$i=0$。

（4）从堆 H 中取出两个权最小的元素赋值于x和y。

（5）构造新的结点 u，令 $u.p=x.p$，$u.index=n+i$。

（6）把结点 u 插入堆 H 中。

（7）用结点 u 生成 T 中的新结点 $T[n+i]$。$T[n+i]$ 的左、右儿子指针分别指向 *x.index* 和 *y.index* 所指向的T中的结点，父亲指针置为 -1。

（8）T中相应于x和y的结点的父亲指针指向 $n+i$，成员变量 *index* 分别置为 0 和 1。

（9）$i=i+1$；若 $i=n$，算法结束，否则转步骤（4）

Huffman算法伪码

输入：$C = \{x_1, x_2, \ldots, x_n\}, f(x_i), i = 1, 2, \ldots, n$

输出：Q //队列

1. $n \leftarrow |C|$
2. Q $\leftarrow C$ // 频率递增队列Q
3. for $i \leftarrow 1$ to $n - 1$ do
4. $z \leftarrow$Allocate-Node() // 生成结点z
5. z.left$\leftarrow Q$ 中最小元 // 最小作为z左儿子
6. z.right$\leftarrow Q$ 中最小元 // 最小作为z右儿子
7. $f(z) \leftarrow f(x) + f(y)$
8. Insert (Q,z) //将z插入Q
9. return Q

3. 数据结构：

```
struct node {
    Type      c;          /* 叶结点所代表的符号 */
    float     p;          /* 结点的权 */
    int       lchild;     /* 内部结点的左儿子结点指针 */
    int       rchild;     /* 内部结点的右儿子结点指针 */
    int       parent;     /* 内部结点的父亲结点指针 */
    int       index;      /* 该结点对应的霍夫曼码的码值 */
};
struct node T[2*n];       /* 存放二叉树的结点 */
struct node H[n+1];       /* 存放二叉树中相关结点的堆 */
```

4. 算法描述：

```
2. void Huffman(Type C[ ],float p[ ],int n,struct node T[ ])
3. {
4.     struct node H[n+1], x, y, u;
5.     int i,m = n;                    /* m为堆H的初始元素个数 */
6.     for (i=0;i<n;i++) {             /* 初始化树的叶结点和数组H */
7.         T[i].c = H[i+1].c = C[i];
8.         T[i].p = H[i+1].p = p[i];
9.         T[i].lchild = T[i].rchild = T[i].parent = -1;
10.        H[i+1].index = i;           /* 堆中元素对应于树中的结点号 */
11.    }
12.    make_heap(H,n);                 /* 构造堆H */
```

```
13.    for (i=0;i<n-1;i++) {
14.        x = delete_min(H,m);       /* 取堆中最小的两个元素,并从堆中删去 */
15.        y = delete_min(H,m);
16.        u.p = x.p + y.p;   /* 构造一个新的结点 */
17.        u.index = n + i;
18.        insert(H,m,u);                          /* 新结点插入堆中 */
19.        T[n+i].lchild = x.index;    /* 新结点的左右儿子指针 */
20.        T[n+i].rchild = y.index;
21.        T[n+i].parent = -1;
22.        T[x.index].index = 0;/* 设置左右儿子结点的霍夫曼码的码值 */
23.        T[y.index].index = 1;
24.        T[x.index].parent = T[y.index].parent = n+i;
25.    }                                       /* 左、右结点的父亲指针指向新结点 */
26. }
```

小结

- 贪婪法适用于组合优化问题
- 求解过程是多步判断过程，最终的判断序列对应于问题的最优解
- 判断依据某种“短视的”贪婪选择性质，性质的好坏决定了算法的正确性。
- 贪婪性质的选择往往依赖于直觉或者经验。

小结（续）

- 贪婪法正确性证明方法：

（1）直接计算优化函数，贪婪法的解恰好取得最优值

（2）数学归纳法（对算法步数或者问题规模归纳）

（3）交换论证

- 证明贪婪法策略不对：举反例

小结（续）

- 对于某些不能保证对所有的实例都得到最优解的贪婪算法（近似算法），可做参数化分析或者误差分析
- 贪婪法的优势：算法简单，时间和空间复杂性低。