第四章　贪心算法

算法基本思想

调度问题

最小生成树问题

单点源最小路径问题

Huffman编码

贪心算法的基本思想

- 找零钱：给孩子找回87分硬币，现有硬币规格50分、10分、5分、2分、1分。

$$50+3*10+5+2=87$$

一般方法：尽量找面值大的硬币。

- 装箱问题：有物品n件，重量分别是 $w_1,\ldots,w_n$；有箱子m个：$B_1,\ldots$，B_m，每个的容量都是C($C\geq w_i$)。设计装箱方法，使得所用箱子最少。

1. NF(Next Fit)方法：当前箱子装不下，就开启装下一个箱子。
2. FF(First Fit)方法：每个物品都选择装进第一个可装的箱子

- 贪心算法的基本思想：

在每一步决策中总是作出在当前看来是最好的选择

- 贪心准则，局部最优。

背包问题

- 背包容量为M ， 物品件数 n。重量w_i , 价值p_i
- 变量 x_i ， $0 \le x_i \le 1$
- 数学模型
- $\max \sum p_i x_i$
- s.t. $\sum w_i x_i \le M$
- 贪心准则：

  1. 价值大的物品优先装包；
  2. 重量轻的物品优先装包；
  3. 单位价值大的物品优先装包。
- **例子** n=3, M=20, p=(25, 24, 15), w=(18,15,10)
- 结论：以“单位价值最大的物品优先装包”为贪心准则的贪心算法获得的效益值最大。

背包问题的贪心算法

GreedyKnapsack (p, w, M, x, n) //价值数组p[1..n]、重量数
//组w[1..n]，它们元素的排列顺序满足p[i]/w[i]≥p[i+1]/w[i+1]
//M是背包容量，x[1..n]是解向量
float p[1..n], w[1..n], x[1..n], M, rc;
integer i, n;
x:= 0; // 将解向量初始化为零
rc:= M; // 背包的剩余容量初始化为M

- **for** i to n **do**
  if w[i] ≤ rc **then**
  x[i]:=1; rc:=rc-w[i];
  else break; **end{if}**
- **end{for}**
  if i≤n **then**
  x[i]:=rc/w[i];
  end{if}

end{GreedyKnapsack}

Greedy Knapsack获得最优解

反证法：设**Greedy Knapsack**给出的解 $x=(x_1,x_2,\cdots,x_n)$ 不是最优解，根据算法可设 $x_1=\cdots=x_{j-1}=1, 0\le x_j<1, x_{j+1}=\cdots=x_n=0$， $\sum w_i x_i = M$ 设 $y=(y_1,y_2,\cdots,y_n)$ 是最优解，且 $y_1=x_1,\cdots,y_{k-1}=x_{k-1}, y_k\neq x_k$ 则必然 $y_k<x_k$。因为，当 $k\ge j$ 时，若 $x_k<y_k$ 则 y 不是可行解：

$$\sum_{i=1}^{k-1} w_i y_i + w_k y_k + \sum_{i=k+1}^{n} w_i y_i > \sum_{i=1}^{k-1} w_i x_i + w_k x_k = \sum_{i=1}^{n} w_i x_i = M$$

由 $y_1=x_1,\cdots,y_{k-1}=x_{k-1}, y_k<x_k$ 及 $\sum_{i=1}^{n} w_i y_i = M$ 得 $\sum_{i=k+1}^{n} w_i y_i \ge w_k(x_k-y_k)>0$

取新向量 $z=(z_1,z_2,\cdots,z_n)$ 满足如下要求： $z_1=y_1,\cdots,z_{k-1}=y_{k-1}, z_k=x_k$

$$0\le z_{k+1}\le y_{k+1},\cdots,0\le z_n\le y_n \qquad \sum_{k+1\le i\le n} w_i(y_i-z_i)=w_k(z_k-y_k)$$

这样取的向量是可行解，而且，总价值为：与 x相同分量的个数增1

$$\begin{aligned}\sum_{1\le i\le n} p_i z_i &= \sum_{1\le i\le n} p_i y_i + (z_k-y_k)w_k p_k/w_k - \sum_{k+1\le i\le n}(y_i-z_i)w_i p_i/w_i \\ &\ge \sum_{1\le i\le n} p_i y_i + \left((z_k-y_k)w_k - \sum_{k+1\le i\le n}(y_i-z_i)w_i\right)p_k/w_k \\ &= \sum_{1\le i\le n} p_i y_i\end{aligned}$$

贪心算法抽象控制流程

Greedy(A, n) // A[1:n]代表输入

- solution={}; //解初始化为空集
- **for** i **to** n **do**
- x:=Select(A);
- **if** Feasible(solution, x) **then**
- solution:=Union(solution, x);
- **end{if}**
- **end{for}**
- return(solution);

end{Greedy}

- Select(A)是按照贪心准则选取A中的输入项；
- Feasible(solution, x)是判断已知的解的部分solution与新选取的x的结合是否是可行解。
- Union(solution, x)x与已经选到的部分solution结合成解的更大部分

调度问题

- **活动安排问题**
- 已知n个活动 E={1, 2, … ,n}，要求使用同一资源，第k个活动要求的开始和结束时间为s_k, f_k, 其中s_k <f_k,k=1, 2, … , n .
- 活动k与活动j称为相容的如果s_k >f_j或者s_j >f_k。活动安排问题就是要在所给的活动集合中选出最大(活动个数最多)的相容活动子集。

```
GreedyAction(s, f，n)
  //s[1..n]、f[1..n]分别代表n项活动
  //的起始时间和结束时间, 并且满
  //足f[1]≤ f[2]≤…≤ f[n]
 j:=1; solution:={1}; //解向量初始化
 for i from 2 to n do
   if si≥fj then
    // 将i加入解中
     solution:=solution ∪ {i};
     j:=i;
   end{if}
 end{for}
 return(solution);
end{GreedyAction}
```

带期限的单机作业安排问题

已知n项作业 E={1, 2, … ,n} 要求使用同台机器完成（该台机器在同一时刻至多进行一个作业），而且每项作业需要的时间都是1。第k项作业要求在时刻d_k之前完成, 而且完成这项作业将获得效益p_k，k=1, 2, … , n 。

作业集E的子集称为相容的如果其中的作业可以被安排由一台机器完成。带限期单机作业安排问题就是要在所给的作业集合中选出总效益值最大的相容子集。

GreedyJob(d, p, n)

- //d[1..n]和p[1..n]分别代表各项
- //作业的限期和效益值，而且n
- //项作业的排序满足：
- //$p_1 \geq p_2 \geq \ldots \geq p_n$
- **local** J;
- J:={1}; //初始化解集
- **for** i **from** 2 **to** n **do**
- **if** J ∪ {i} 中的作业是相容的
- **then** //此步验证需要认真设计
- J:= J ∪ {i}; // 将i加入解中
- **end{if}**
- **end{for}**
- **end{GreedyJob}**

GreedyJob获得最优解

- 假设贪心算法所选择的作业集J不是最优解，则一定有相容作业集I，其产生更大的效益值。假定I是具有最大效益值的相容作业集中使得|I∩J|最大者，往证 I=J。
- 反证法：若I=J 不成立，则这两个作业集合之间没有包含关系。这是因为算法GreedyJob的特性和假定I 产生的效益值比J的效益值更大。假设a是J\I中具有最大效益的作业，即**J中比a具有更大效益的作业(如果有的话)都应该在 I 中**。如果作业 b∈I\J 且 $p_b>p_a$，那么由算法中对J中作业的选取办法（相容性要求），J中至少有 f_b个效益值≥p_b的作业，其期限值≤ f_b。这些作业一定在 I 中，因而 I 中至少有f_b+1个作业，其期限值≤ f_b ，这与 I 的相容性矛盾。所以，I\J**中作业的效益值均不超过p_a**。
- 称区间[k-1,k]为时间片k，相容作业集 I 的一个调度表就是指定 I 中各个作业的加工时间片。如果 I 有一个调度表 S 将时刻 f_a前的时间片均安排给I∩J中的作业，且I\J中最早被安排的是作业b，在时间片k上，则$k>f_a$。

- 前k-1个时间片上安排的都是I∩J中的作业，这些作业的集A_1再添上作业 a 得到 J 中k个作业。由作业集的相容性， A_1中至少有一个作业其期限值≥k。将这个作业与作业b交换安排的时间片，得到新的调度表S_1。如果在调度表S_1中作业 b 安排在时间片 k_1，且$k_1>f_a$, 则同理，可以将作业 b 再向前移，得到新的调度表S_2。如此做下去，必得到 I 的调度表S′，其在时刻 f_a 前的时间片安排有 I\J 中作业 b 。令 I′=(I\{b})∪{a}，则 I′ 也是相容的作业集。而且，由于$p_b≤p_a$， I′ 的效益总值不小于 I 的效益总值，因而 I′ 具有最大的效益总值，但| I′ ∩J|>|I∩J|，与 I 的选取相悖。因而，I=J，J 是最优解。　　证毕

时间复杂度

- 算法GreedyJob的关键操作是比较和相应移动作业的位置，都是围绕着判断作业集J ∪ {i}的相容性。考察J 外的作业 i 该加入 J时，作业 i 至多和 J 中的每个作业都比较一次。因而，J ∪ {i}的相容性判断最坏情况下需要|J|次比较。上述算法在最坏情况下的时间复杂度为
- **例子** 设n=7, $(p_1, p_2, \ldots, p_n)=(35,30,25,20,15,10,5)$,
  $(d_1, d_2, \ldots, d_n)=(4,2,4,3,4,8,3)$,

- 算法GreedyJob的执行过程。

d_1； d_2, d_1； d_2, d_1, d_3； d_2, d_4, d_1, d_3； d_2, d_4, d_1, d_3, d_6；
4； 2， 4； 2，4， 4； 2， 3， 4， 4； 2， 3， 4， 4， 8；

单机作业调度问题的贪心算法

GreedyJob(D,J,n,k)

- //D(1),…,D(n)是期限值，作业已按$p_1 \geq p_2 \geq \ldots \geq p_n$排序。J(i)是最
- //优解中的第i个作业。终止时，D(J(i))≤ D(J(i+1))，1≤i≤k
- integer D[0..n],J[0..n],i,k,n,r
- D(0):=0; J(0):=0; //初始化
- k:=1; J(1):=1; //计入作业1，k 表示当前选择的作业个数
- **for** i **from** 2 **to** n **do**
- //按p的非增次序考虑作业，找i的位置，并检查插入的可能性
- r:= k;
- **while** D(J(r))>D(i) and D(J(r))< > r **do**
- r:= r-1;
- **end{while}**; //期限不晚于D(i)的作业个数小于D(i)时
- **if** D(J(r)) ≤ D(i) and D(i) > r **then**
- **for** j **from** k **by** –1 **to** r+1 **do** //给作业i腾出位置
- J(j+1):=J(j);
- **end{for};**
- J(r+1):=i; k:=k+1;
- **end{if}**;
- **end{for}**;

end{GreedyJob}

例子

- 为避免调度表调整，将算法GreedyJob稍做改进：在每次选择作业时，在调度表中尽量地向后分配时间片，这样好为后面的作业安排留下更多的空间。

- 1，[3,4]; 1,2,[3,4],[1,2]; 1,2,3,[3,4],[1,2],[2,3];
- 1,2,3,4,[3,4],[1,2],[2,3],[0,1];
- 1,2,3,4,6,[3,4], [1,2],[2,3],[0,1],[7,8].

- 根据该思路可构造时间复杂度近为O(n·β(2n,n))的算法，这里，β(2n,n)是Ackermann函数的逆函数 。

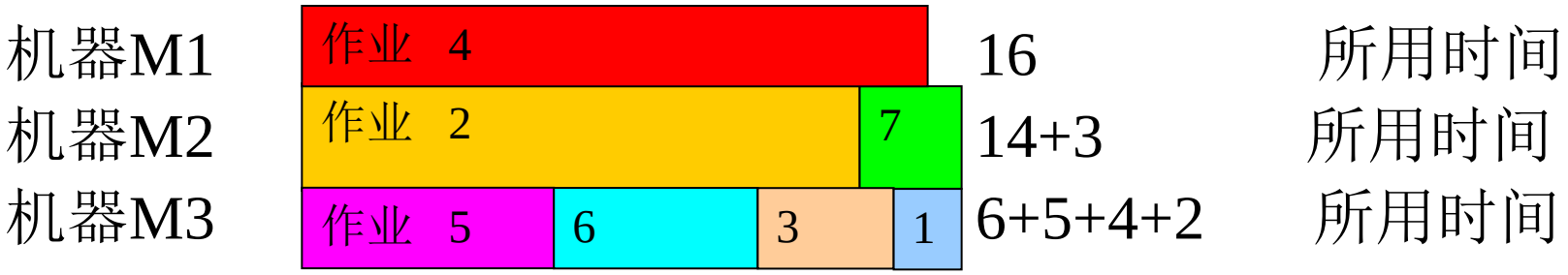
多机作业调度问题

有n项独立的作业{1,2,…, n},由m台相同的机器加工处理。作业 i 所需要的处理时间为t_i。约定：任何一项作业可在任何一台机器上处理，但未完工前不准中断处理；任何作业不能拆分更小的子作业分段处理。多机调度问题要求给出一种调度方案，使所给的n个作业在尽可能短的时间内由m台机器处理完。

贪心准则：将加工时间长的作业优先安排给空闲早的机器。

例子：三台机器，七项作业，所需加工时间分别 2,14,4,16,6,5,3

将作业按照所需加工时间由长到短排序：4,2,5,6,3,7,1

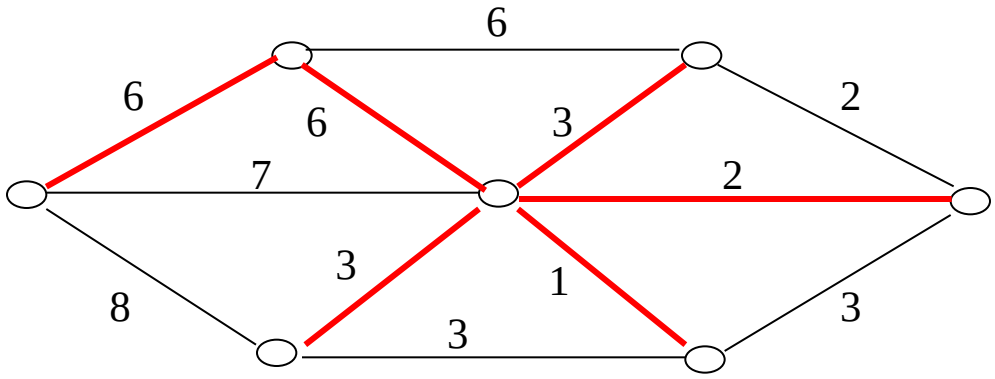


最优生成树问题

- 无向图G=(V, E)，不妨假定该图代表城镇间的交通情况，顶点代表城镇，边代表连接两个城镇的可能的交通线路。将每条边赋予一个权值，这些权值可代表建造该条线路的成本、交通线的长度、经济效益或其它信息。在道路规划建设中，一个最基本的需求是选择修建一组交通线路，它们满足

  1. 连通所有的城镇；2. 具有最小的建造成本。

- 这个问题归结为求连通赋权图的具有最小权值的生成树。称为最优生成树问题。



Prim算法与Kruskal算法

- **Prim算法的基本思想**

1). 选择图G的一条权值最小的边 e_1，形成一棵两点一边的子树；

2). 假设G的一棵子树T已经确定；

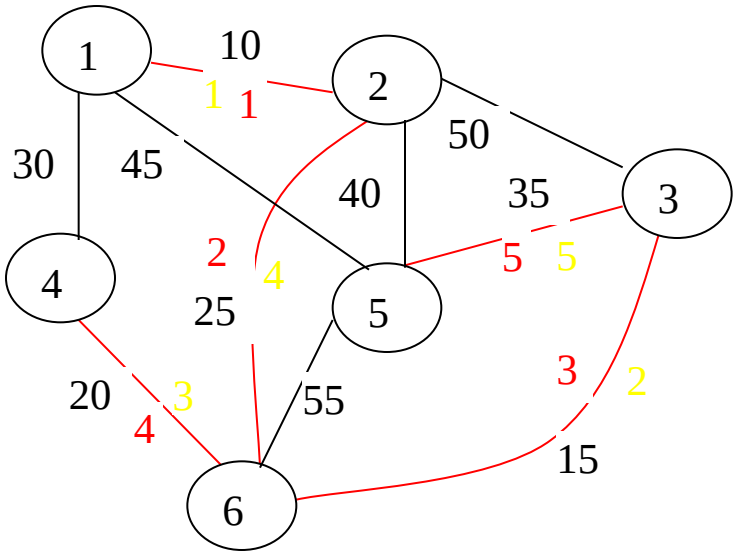
3). 选择G的不在T中的具有最小权值的边e，使得T∪{e}仍是G的一棵子树。

- **Kruskal算法的基本思想**

1). 选择图G的一条权值最小的边 e_1；

2). 假设已经选好G的一组边 $L=\{e_1,e_2,\ldots,e_k\}$；

3). 选择G的不在L中的具有最小权值的边e_{k+1}，使得$L\cup\{e_{k+1}\}$诱导出的G的子图不含G的圈。



Prim算法：

(1,2), (2,6), (6,3), (6,4), (3,5)

Kruskal算法：

(1,2), (3,6), (4,6), (2,6), (3,5)

Prim算法伪代码

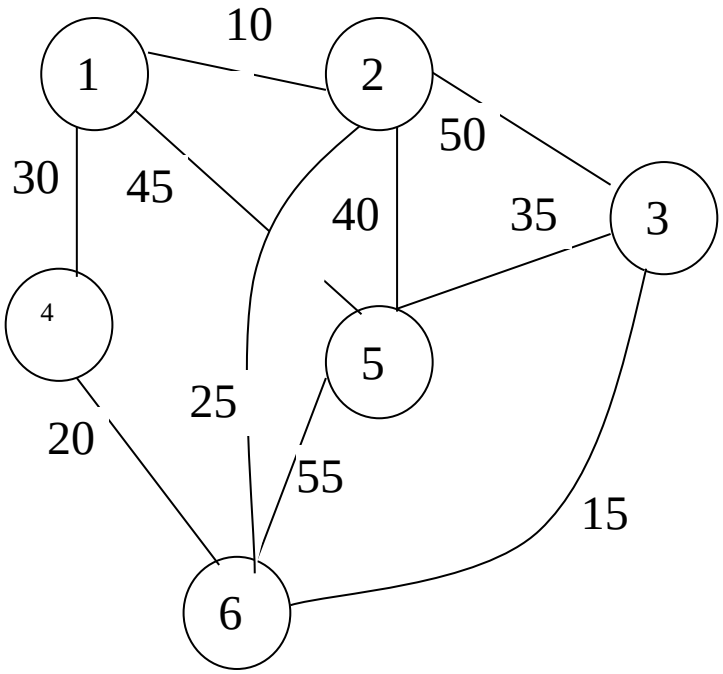
```
PrimTree(E,COST,n,T,mincost) //E是
  //图G的边集，COST是G的带权邻
  //接矩阵。计算一棵最小生成树T并
  //把它作为一个集合存放到数组
  //T[1..n-1,1..2]中，这棵树的权值赋
  //给mincost
1   real COST[1..n,1..n],mincost;
2   integer  NEAR[1..n], n, i, j, k, s,
      T[1..n-1,1..2];
3   (k,s):= 权值最小的边;
4   mincost:＝COST[k,s];
5   (T[1,1],T[1,2]):=(k,s); //初始子树
6   for i to n do //将NEAR赋初值
      //（关于初始子树T）
7     if COST[i,s]<COST[i,k] then
8       NEAR(i)=s;
9     else NEAR(i)=k;
10    end{if}
11  end{for}
12   NEAR(k):=0,NEAR(s):=0;
13  for i from 2 to n-1 do  //寻找T的其余
      //n-2条边
14    choose an index j such that
      NEAR(j)≠0 and COST[j,NEAR(j)]
      is of minimum value;
15   (T[i,1],T[i,2]):=(j,NEAR(j));
      //加入边（j，NEAR(j)）
16    mincost:=mincost
                  +COST[j,NEAR(j)];
17    NEAR(j):=0;
18    for t to n do  //更新NEAR
19      if NEAR(t)≠0 and
    COST[t,NEAR(t)]>COST[t,j] then
20        NEAR(t):=j;
21      end{if}
22    end{for}
23  end{for}
24  if mincost ≥ ∞ then
25    print(‘no spanning tree’);
26  end{if}
27 end{PrimTree}
```

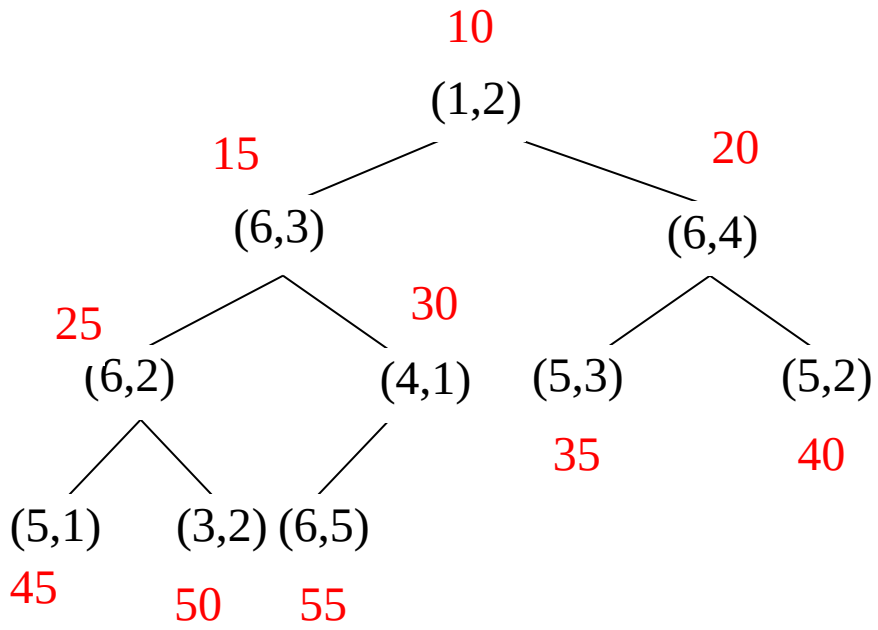
Kruskal算法伪代码

KruskalTree(E,COST,n,T,mincost)//说明同算法PrimTree

```
1   real mincost, COST[1..n,1..n];
2   integer Parent[1..n],T[1..n-1],n;
3   以带权的边为元素构造一个min-堆;
4   Parent:=-1; //每个顶点都在不同的集合中;
5   i:= 0; mincost:= 0;
6   while i<n-1 and min-堆非空 do
7     从堆中删去最小权边（u，v）并重新构造min-堆
8     j:= Find(u); k:= Find(v);
9     if j≠k then  //保证不出现圈
10      i=:i+1; T[i,1]:=u; T[i,2]:=v;
11      mincost:=mincost+COST[u,v];
12      Union(j,k); //把两个子树联合起来
13    end{if}
14  end{while}
15  if i≠n-1 then
16    print(‘no spanning tree’);
17  end{if}
18  return;
    end{KruskalTree}
```

一个赋权图和它的边的最小堆





Kruskal算法产生最优生成树

- 设T是用Kruskal算法产生的G的一棵生成树，而T′是G的一棵最优生成树且使得|E(T′)∩E(T)|最大。用E(T)表示树T的边集，w(e)表示边e的权值，而边集E的权值之和用w(E)表示。以下证明E(T)=E(T′).
- 反假设E(T)≠E(T′),因为|E(T)|=|E(T′)|,所以E(T)与E(T′)没有包含关系。设e是E(T)\E(T′)中权值最小的边。将e添加到T′中即得到T′的一个圈,不妨记为： e，e_1 ,e_2 ,…,e_k。因为T是树，诸e_i中至少有一个不属于E(T)。不妨设e_i不属于E(T),则必然有$w(e)\le w(e_i)$。否则，由$w(e)>w(e_i)$以及E(T)中比e权值小的边都在T′中，e_i同这些边一起不含有圈，因而，按Kruskal算法，e_i将被选到E(T)中，矛盾。在T′中去掉边e_i，换上边e，得到G的一棵新的生成树T″,这棵树有两个特点：

  a). T″的权值不大于T′的权值，因而与T′有相等的权值；

  b). |E(T″)∩E(T)| > |E(T′)∩E(T)|.

a)说明T″也是一棵最优树，而b)与T′取法相悖。因此E(T) =E(T′)，T是最优生成树。

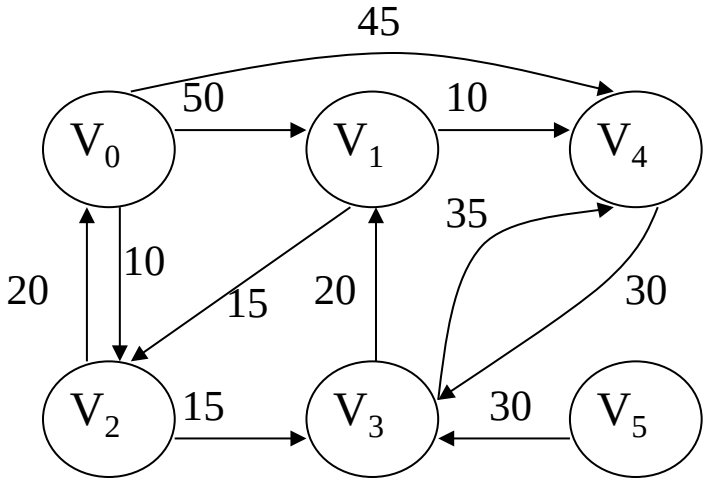
单点源最短路径问题

- 赋权有向图G=(V,E,w), 指定的顶点v_0，求由v_0 出发到G中其它各个顶点的最短路径。
- 贪心准则：迄今已生成的所有路径长度之和为最小
- S-当前已构造出最短路径终点集，$v_0 \in S$, Dist(v)=|v_0-v|

P= $v_0 v_1 \ldots v_{s-1} v_s w$ —短路延续

$$D(S) = \min_{S(v)=1, S(w)=0} \{Dist(v) + COST(v,w)\}$$

- $w \in V \backslash S$, Dist(w)=D(S)
- S:=S∪{w}, 即S(w):=1;



从v_0到各顶点的最短路径

路径	长度
(1) v_0v_2	10
(2) $v_0v_2v_3$	25
(3) $v_0v_2v_3v_1$	45
(4) v_0v_4	45

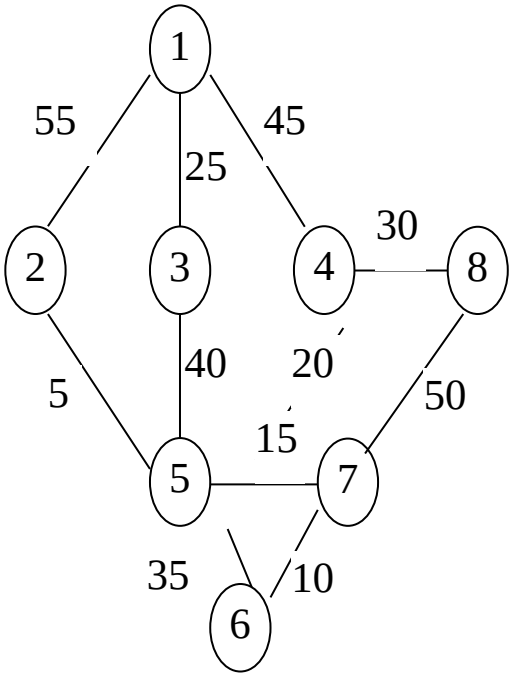
Dijkstra算法

DijkstraPaths(v,COST,n,
Dist,Parent)
//G是具有n个顶点{1,2,…,n}的
//有向图， v是G中取定的顶点
//COST是G的邻接矩阵,Dist表
//示v到各点的最短路径之长度
//Parent表示各顶点在最短路径
//上的前继。
bool S[1..n]; **float**
COST[1..n,1..n], Dist[1..n];
integer u,v,n,num,i,w;
for i **to** n **do**
//将集合S初始化为空
S[i]:=0; Parent[i]:=v;
Dist[i]:=COST[v,i];
end{for}

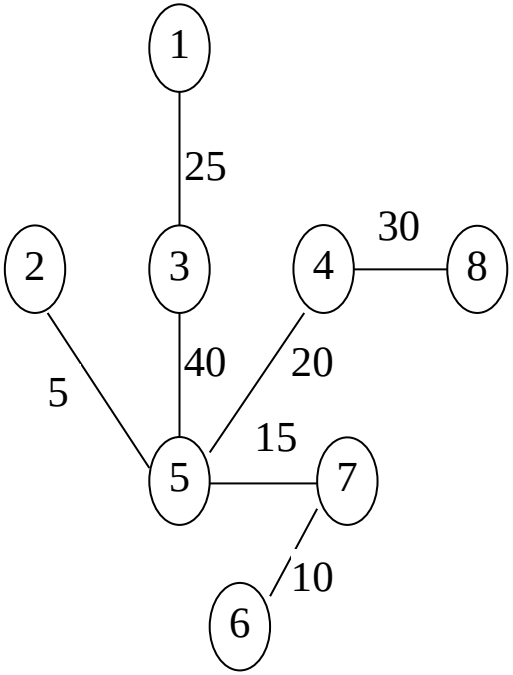
- S[v]:=1; Dist[v]:=0;
- Parent[v]:=-1;
- //首先将节点v记入 S
- **for** i **to** n-1 **do**
  //确定由节点v出发的n-1条最短路
- 选取顶点w,使得
- $Dist(w) = \min_{S(u)=0}\{Dist(u)\}$
- S[w]:=1；
- **while** S[u]=0 **do**
- //修改v通过S到达S以外的结
- //点w的最小距离值
- **if** Dist[u]> Dist[w]+COST[w,u] **then**
- Dist[u]:=Dist[w]+COST[w,u];
- Parent[u]:=w;
- **end{if}**
- **end{while} ;**
- **end{for}**

end{DijkstraPath}

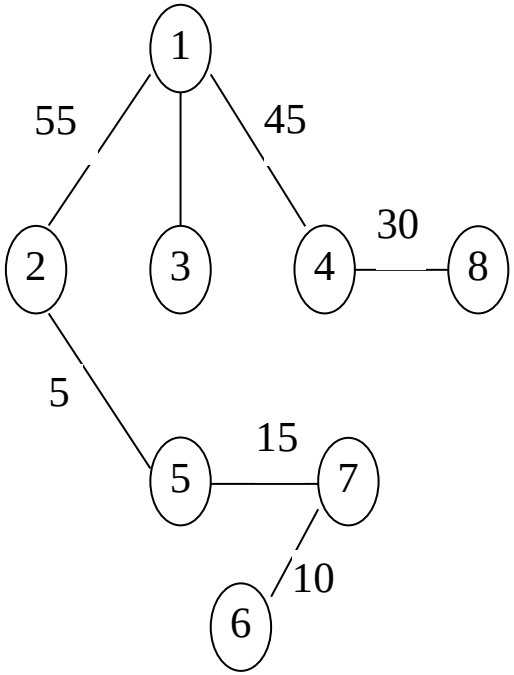
单点源最短路径树



赋权连通图G



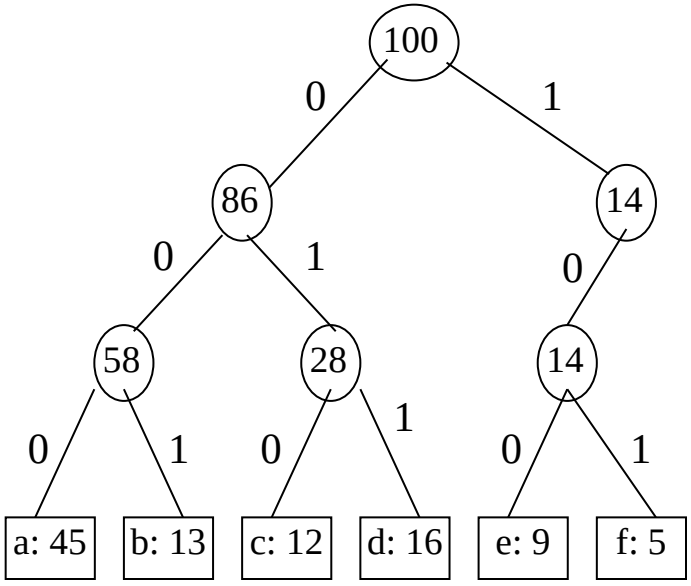
G的最优生成树

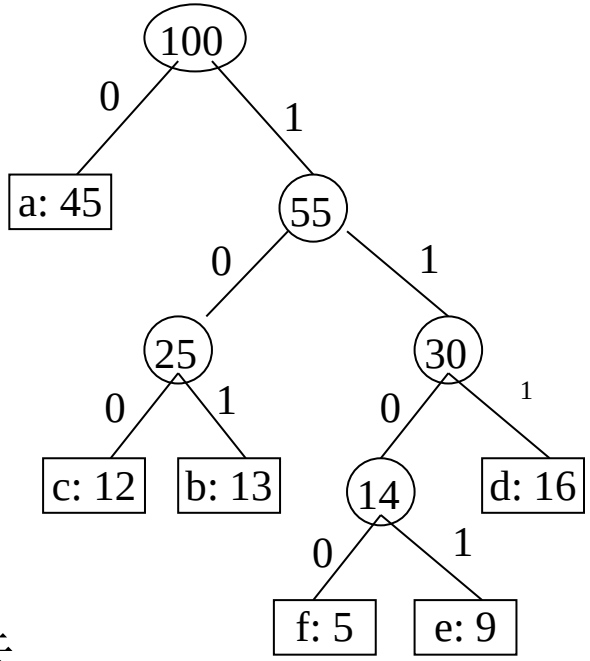


G的一棵单点源
最短路径生成树

哈夫曼(Huffman)编码

不同的字符	a	b	c	d	e	f
频率（千次）	45	13	12	16	9	5
定长码	000	001	010	011	100	101
变长码	0	101	100	111	1101	1100





前缀码的二叉树表示

长为100000的文件出现六种字符。频率分布如左表。

定长码：300000 bit

变长码：224000 bit

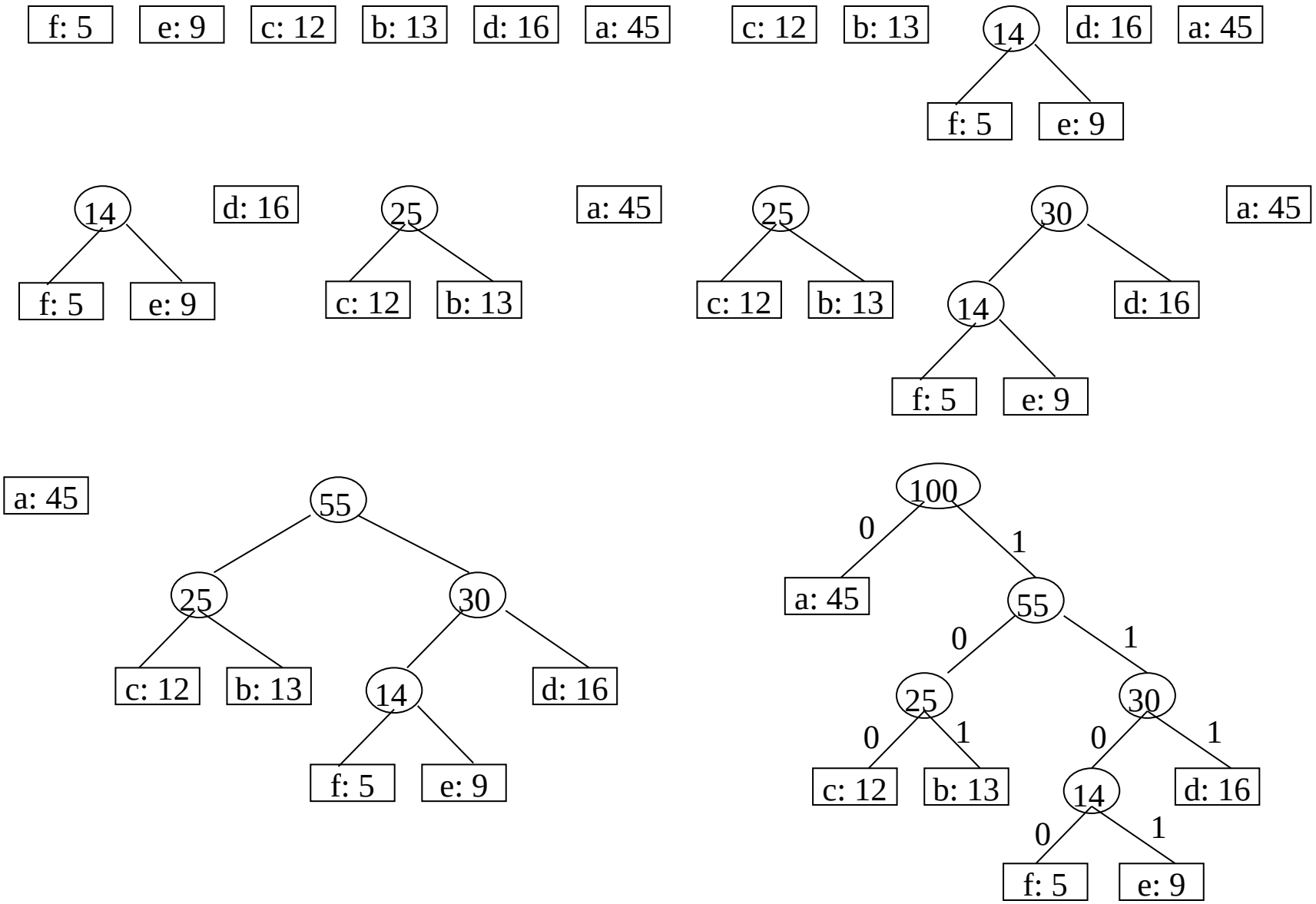
节省25%

前缀码：表示一个字符的0,1字串不能是表示另一个字符的0,1字串的前部。

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

平均码长达到最小的前缀编码方案称为C的一个最优编码。

Huffman编码树的构造



Huffman编码是最优编码

- 设T是一棵表示最优前缀编码的二叉树，编码的字符集为C。我们先证明：对于频率最小的两个字符x,y，可以将它们调换到最深叶顶点的位置而使新树T′的平均码长不增加。
- 事实上，假设b,c是两个最深的叶顶点，它们是兄弟，具有同样的深度。不妨设f(b)≤f(c),f(x)≤f(y)。在树T中将节点b与节点x互换，得到一个新树T′。此时

$$B(T)- B(T')= f(b)d_T(b)+ f(x)d_T(x)- f(b)d_T(x)- f(x)d_T(b)$$
$$= (f(b)-f(x))(d_T(b)- d_T(x))\geq 0$$

如果在树T′中将节点y与节点c互换，得到一棵新树T″，同理可证B(T′)≥ B(T″)但T是最优编码树，所以B(T″)＝B(T)，说明T″也是最优编码树。

- 其次，如果令T″中节点x,y的父节点代表一个新的字符z，出现频率为

$$f(z)=f(x)+f(y),$$

则T″去掉节点x,y后得到一个以{C\{x,y})∪{z}为字符集的最优前缀编码树。对于这棵树叶可以象对待树T那样进行调整，使得新树将具有最小频率的两个字符放在最深的叶子节点位置。如此继续下去，最后得到一棵只有一个节点的树，它是只有一个字符，出现率为100%的最优编码树。以这棵树为根逐步将上述过程中摘掉的叶节点恢复出来，得到的就是Huffman树。所以Huffman树是最优树，即平均码长最短的树。