贪心算法

GREEDY ALGORITHM

贪婪是原罪

天国就好像一个国王，要和他的仆人算账。在开始算的时候，有人带了一个欠1000万银子的人来。因为他没有偿还的债务很多，主人吩咐把他和他的妻子儿女，及一切所有的都卖了偿还。那仆人就俯伏在地说，主人啊，宽限我一些日子吧，将来我都要还清的。那仆人的主人就动了慈心，把他释放了，并且赦免了他的所有债务。那仆人出来，遇见他的一个同伴，欠他10两银子，便揪着他，掐住他的喉咙说，你把所欠的还我。他的同伴就俯伏央求他说，宽限我些日子吧，将来我必还清。他不肯，竟去把他下在监牢里，等他还了所欠的债。众同伴看见他所做的事，就甚忧愁，把这事都告诉了主人。于是主人叫了他来，对他说，你这恶奴才，你央求我，我就把你所欠的都免了。你不应当向我怜恤你那样怜恤你的同伴吗？主人就大怒，把他交给掌刑的，等他还清了所欠的债。

——摘自《圣经·马太福音》

贪婪无处不在

- 良禽择木而栖，良将选主而事，人类似乎永远在追求最好的东西。
- 国家之间无休止的军备竞争；
- 公司雇人要雇最能干的人；
- 买东西要买物美价廉的；
- 各种评优选优活动
- ……

将人类贪婪的本性应用到算法的设计中，就是本章所要学习的贪婪算法或称贪心算法。特点就是只看眼前效果。

学习要点

- 理解贪心算法的概念。
- 掌握贪心算法的基本要素
  - 最优子结构性质
  - 贪心选择性质
- 通过应用范例学习贪心设计策略
- 理解贪心算法的一般理论

- 背包问题
- 货箱装船问题
- 活动安排问题；
- 最短路径问题
- 哈夫曼编码问题
- 最小代价生成树
- 拓扑排序问题
- 偶图覆盖问题

找零钱问题

贪心算法总是作出在当前看来是最好的选择。也就是说贪心算法并不从整体最优 上加以考虑，它所作出的选择只是在某种意义上的局部最优选择。

假设有三种硬币，它们的面值分别为一元、五角和一角。

现在要找给某顾客四元八角钱。

- 这时，我们会不假思索地拿出4个一元的硬 币，1个五角的硬币和3个一角的硬币交给顾客。
- 这种找硬币方法与其他的找法相比，所拿出的硬币个数是最少的。这里，我们下意识地使用了这样的找硬币算法： 首先选出一个面值不超过四元八角的最大硬币，即一元；然后从四元八角中减去一元，剩下三元八角；再选出一个面值不超过三元八角的最大硬币，即又一 个一元，如此一直做下去。这个找硬币的方法实际上就是贪心算法。

0/1背包问题

- 假定你因天缘进入一个藏宝地宫，里面奇珍异宝n件(n当然大得你数不过来)。你当然想尽可能多地带走珍宝，无奈你只有一个背包，且该背包只可承重c千克。超出此重量背包就会裂开。地宫里的每件珍宝都有重量w_i和价值v_i。请问，如何挑选珍宝才能使你背出的珍宝价值最大？当然是在背包不裂开的情况下。
- 需要对珍宝排序！
- 按照价值？☹价值高的珍宝可能很重，导致背出的件数很少。
- 以v_i/w_i的比率为标准从高到底排列，然后按照这个次序进行挑选。每次捡最好的东西拿就是贪心算法的精髓！

背包问题

- 背包问题可描述成如下的最优化问题:
  - 使用0/1数组$(x_1,\ldots,x_n)$表示一个装法: x_i=1表示装物品i, 否则不装;
  - 约束条件： $\sum_{i=1}^{n} w_i x_i \leq c$;
  - 目标函数： $\sum_{i=1}^{n} v_i x_i$,等于装入物品的总效益值.
- 极大化目标函数

- 密度贪心法的伪代码:
- 将物品按密度从大到小排序
- for (i=1;i<n;i++)
- if (物品i 可装入到背包内)
  x_i=1(装入)
- else x_i=0(舍弃);
- 算法的时间复杂度为O(nlogn)

0/1背包问题

- 背包问题有若干种贪心策略：
  - 价值贪心准则：从剩余的物品中选出可以装入背包的价值最大的物品；
  - 重量贪心准则：从剩余的物品中选出可以装入背包的重量最小的物品；
  - 价值密度贪心准则：从剩余物品中选出可装入背包的价值密度值最大的物品。
- 算法的主要计算时间在于将各种物品依其单位重量的价值从大到小排序。因此，算法的计算时间上界为O(nlogn)。

0/1背包问题

- 0/1背包问题是NP完全问题，无法在多项式时间内得到最优解。
- 假定我们有珍宝1, 2, 3, 4, 5，其重量和价值如下表所示：
- 背包的总承重为50千克。
- 按照算法，我们装进背包的珍宝为1, 2和4，总价值190万元，总重量40千克。

i	1	2	3	4	5
p_i	60	100	120	30	40
w_i	10	20	30	10	20
p_i/w_i	6	5	4	3	2

- 最优方案是2和3，总价值220万元，总重量50千克。

0/1背包问题

- 问题：珍宝不可分，贪心选择无法保证最终能将背包装满，部分闲置的空间使每公斤背包空间的价值降低了！
- 贪心算法不能保证得到0/1背包问题的最优解，但是可以为我们提供一个好的近似解，是有价值的启发式算法。
- 可以利用k-优化策略改进贪心算法，修正贪心解与最优解的误差。

K-优化算法

- k-优化算法也要先对物品按密度从大到小排序;
  - 先将最多k件物品装入背包,果这些物品的重量超过c，则放弃它们。
  - 对其余物品用贪心法;
  - 考虑最多有k件物品的所有可能子集，得到的最好的解是k-优化算法的解.
- 算法的时间复杂度随k 的增大而增加:
  - 需要测试的子集数目为$O(n^k)$;
  - 每一个子集做贪心法需时间O(n);
  - 因此当k >0时总的时间开销为$O(n^{k+1})$.

0/1背包问题整数规划形式

- 问题描述：给定c>0，w_i>0 (1≤ i ≤n), 要求找出一个n元0-1向量(x_1, x_2, …., x_n) (1≤ i ≤n), 使得$\sum_{i=1}^{n} w_i x_i \le c$, 且$\sum_{i=1}^{n} v_i x_i$达到最大.
- 整数规划问题: ,

$$\begin{cases} \max \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \sum_{i=1}^{n} w_i x_i \le c \end{cases}, \quad \text{其中} x_i = \begin{cases} 0 & \text{如果不选择第i件珍宝} \\ 1 & \text{如果选择第i件珍宝} \end{cases}$$

0/1背包问题递归公式

- 递归公式：设m(I, j)是背包容量为j，可选择物品为i, i+1, …, n时0-1背包问题的最优值，有

$$m(i,j)=\begin{cases}\max\{m(i+1,j),m(i+1,j-w_i)+v_i\} & j\geq w_i\\ m(i+1,j) & 0\leq j<w_i\end{cases}$$

$$m(n,j)=\begin{cases}v_n & j\geq w_n\\ 0 & 0\leq j<w_n\end{cases}$$

```
Template <class Type>
void Knapsack(Type v, int w, int c, int n, Type * * m ) {
    int jMax = min (w[n]-1, c);
    for (int j = 0; j <= jMax; j++) m[n][j] = 0;
    for (int j = w[n]; j<= c; j++) m[n][j] = v[n];
    for (int i = n-1; i > 1; i--) {
      jMax = min(w[i] – 1, c);
      for (int j = 0; j<= jMax; j++)m[i][j] = m[i+1][j];
      for (int j = w[i]; j<=c; j++)
        m[i][j]=max(m[i+1][j],m[i+1][j-w[i]]+v[i]);
      }
    m[1][c] = m[2][c];
    if (c>=w[1]) m[1][c] = max(m[1][c],m[2][c-w[1]]+v[1]);}
```

算法复杂度：
O(nc)

当背包容量c很大时，算法需要的计算时间较多。例如，当$c>2^n$时，算法需要$\Omega(n2^n)$计算时间。

贪心策略

- 贪心策略也是一种分治策略，即将大问题化为小问题，在以最优方式解决小问题后获得大问题的解。
- 与一般分治不同的是，贪心策略每步解决的子问题数量为1个。
- 对背包问题来说，我们的子问题就是每种珍宝的选择，而在每一步我们需要决定的是对哪一种珍宝进行选择，即在所有子问题里面(所有珍宝种类的选择上)挑选一个子问题(一种珍宝)进行考虑。
- 选择的原则为是以价值重量比最高为标准，因此是贪心选择。

贪心算法

- 贪心选择属性是指一个(全局)最优的解答是经由局部最优(贪心)的选择而获得的。
- 如果经由一系列贪心选择可以获得一个问题的最优解，则该问题具有所谓的贪心选择属性。
- 贪心策略要想获得最优解，必须满足下面两个条件：
  - 每个大问题的最优解里面包括下一级小问题的最优解；
  - 每个小问题的解可由贪心选择获得。
- 如果一个问题不具备上述条件，并不说明不能采用贪心策略，只不过贪心策略将不能保证获得最优解。

最优子结构性质

贪心选择性质

贪心算法：局部最优 ⟶ 整体最优

- 贪心算法总是作出在**当前**看来最好的选择。也就是说贪心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。当然，希望贪心算法得到的最终结果也是整体最优的。(比如人生)
- 虽然贪心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，即使贪心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。贪心算法经常用于求解难解问题或NP完全问题的近似解。

货箱装载问题

- 设有n个集装箱，集装箱大小一样，第i个集装箱的重量为w_i $(1 \leqslant i \leqslant n)$，设船的载重量为c. 试设计一种装船的方法使得装入的集装箱数目最多.
- 数学描述：给定c>0，$w_i>0$ $(1 \leq i \leq n)$，要求找出一个n元0-1向量$(x_1, x_2, \ldots, x_n)$ $(1 \leq i \leq n)$，使得$\sum_{i=1}^{n} w_i x_i \leq c$，且$\sum_{i=1}^{n} x_i$达到最大.
- 整数规划问题：

$$\begin{cases} \max \sum_{i=1}^{n} x_i \\ \text{s.t.} \sum_{i=1}^{n} w_i x_i \leq c \end{cases}$$

，其中$x_i = \begin{cases} 0 & \text{如果货厢i不装船} \\ 1 & \text{如果货箱i装船} \end{cases}$

货箱装载问题

- 对策：把货箱分步装载到货船上，一步装载一个货箱。每一步决定装载哪一个货箱。
- 贪心选择准则：从剩下的货箱中，选择重量最小的货箱。
- 最优子结构性质：设$(x_1, x_2, ..., x_n)$是最优装载的满足贪心选择性质的最优解。易知，如果x_1=1，$(x_2, ..., x_n)$是轮船载重量为c-w_1，待装船集装箱为{2,3，.....n}时相应的最优装载问题的最优解；如果x_1=0，$(x_2, ..., x_n)$是轮船载重量为c，待装船集装箱为{2,3，.....n}时相应的最优装载问题的最优解。

货箱装载问题的贪心属性

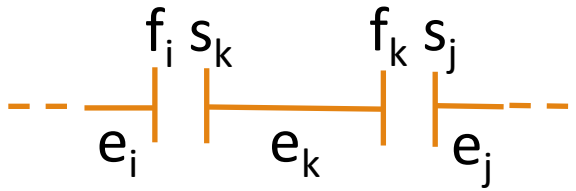
- 贪心选择性质：设集装箱已依其重量从小到大排序，$x=(x_1, x_2, \cdots, x_n)$是装载问题的贪心解，$y=(y, \cdots, y_n)$是任意一个可行解。则$\sum_{i=1}^{n} x_i \geq \sum_{i=1}^{n} y_i$。

- 证明：
  - 由算法知，存在$k=\max\{i \mid x_i=1\}\{1 \leq i \leq n\}$，使得$x_i=1$当$i \leq k$，且$x_i=0$当$i > k$。即$x=(x_1, x_2, \cdots, x_n)=(1, \cdots, 1, 0, \cdots, 0)$。由算法知，$\sum_{i=1}^{k} w_i x_i \leq c$，$\left(\sum_{i=1}^{k} w_i x_i\right)+ w_{k+1} > c$。
  - 设$p: 1 \leq p \leq k$是满足$x_i \neq y_i$的最小的下标，则$y_p = 0$。
  - 如果存在一个$q\ (k < q \leq n)$使得$y_q = 1$。则令$y_p = 1$，$y_q = 0\ (w_p \leq w_q)$，产生一个新的可行解z，满足$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} z_i$。如果不存在这样的q，则命题 成立。
  - 继续上述操作，直到找不到这样的q为止。
  - 所以$\sum_{i=1}^{n} y_i = \sum_{i=1}^{n} z_i \leq \sum_{i=1}^{n} x_i$，命题成立。

货箱装载问题

```
template<class Type>
void Loading(int x[],  Type w[], Type c, int n)
{
    int *t = new int [n+1];
    Sort(w, t, n);
    for (int i = 1; i <= n; i++) x[i] = 0;
    for (int i = 1; i <= n && w[t[i]] <= c; i++) {x[t[i]] = 1; c -= w[t[i]];}
}
```

活动安排问题



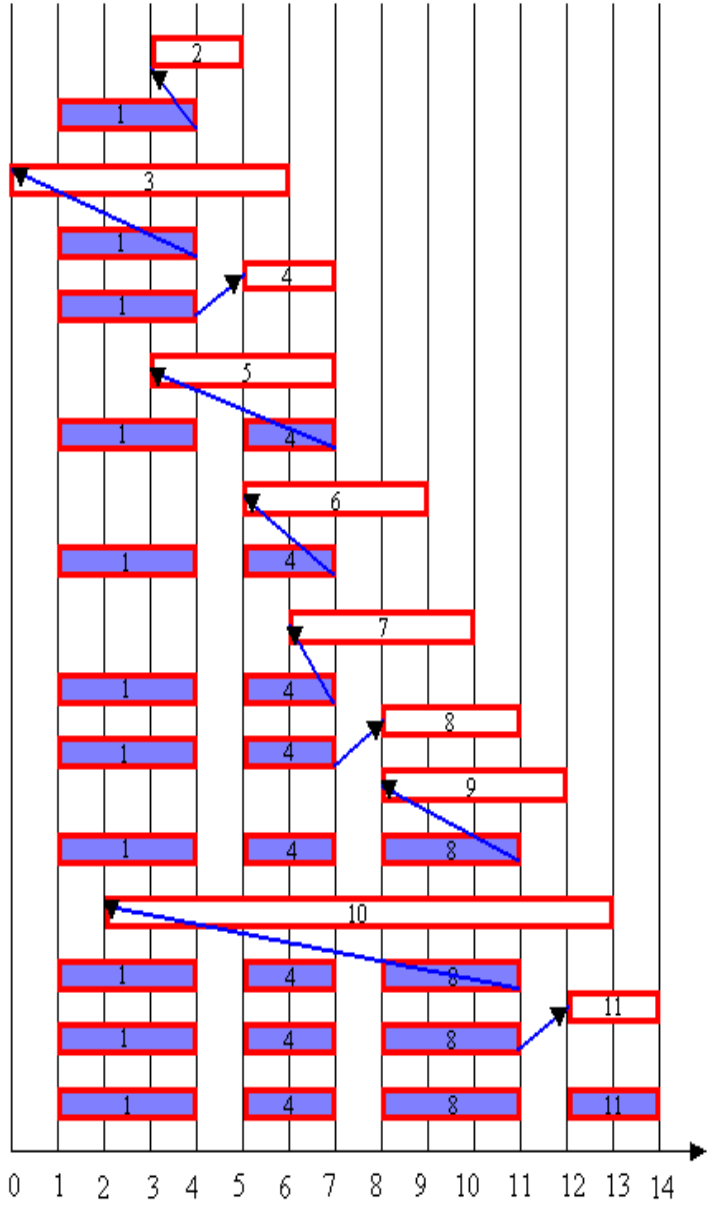
- 设有n个活动的集合E={e_1, e_2, …, e_n}，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。
- 每个活动e_i都有一个要求使用该资源的起始时间s_i和一个结束时间f_i,且$s_i < f_i$ 。如果选择了活动e_i，则它在半开时间区间[s_i, f_i)内占用资源。
- 若区间[s_i, f_i)与区间[s_j, f_j)不相交，则称活动e_i与活动e_j是相容的。也就是说，当$s_i \geq f_j$或$s_j \geq f_i$时，活动e_i与活动e_j相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

活动安排问题例

- 设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

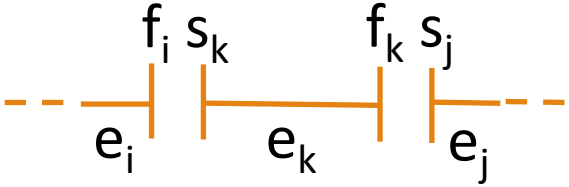
e_i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

- 右图中，阴影长条表示已选入集合A的活动，而空白长条表示当前正在检查相容性的活动。
- 看出来的吗?



活动安排问题的贪心算法

- 设$E_{ij}=\{e_k \in E: f_i \le s_k < f_k \le s_j\}$=所有在活动$e_i$结束后开始但在$e_j$开始前结束的活动。那么$E_{ij}$里面的活动与所有满足下列条件的活动兼容，即可以被同时容纳：
  - 所有在f_i或之前结束的活动；
  - 所有在s_j或之后开始的活动。



- 引入两个莫须有的活动$e_0=[-\infty, 0)$和$e_{n+1}=[\infty, \infty+1)$, 这样我们可以将所有活动的集合E表示为$E_{0,n+1}$。因此$E_{ij}$的下标范围为$0 \le i, j \le n+1$.
- 与背包问题一样，我们假定所有的活动都按照结束时间以非递减方式排列，即$f_0 < f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$, 则$i \ge j \Rightarrow E_{ij}=\phi$. 因此我们只需要考虑满足$0 \le i < j \le n+1$的$E_{ij}$即可。

活动安排问题的贪心算法

- 假如E_{ij}的某解决方案里面包含活动e_k，则我们将可以把E_{ij}分解为E_{ik}和E_{kj}两个子问题，而E_{ij}的某解决方案就是:

$$(E_{ik}\text{的解决方案})\cup\{e_k\}\cup(E_{kj}\text{的解决方案})$$

- 由于活动e_k不属于任何子问题，这两个子问题之间没有重叠，也就是说，

$$|E_{ij}\text{的解决方案}|=|E_{ik}\text{的解决方案}|+1+|E_{kj}\text{的解决方案}|$$

活动安排问题贪心算法的最优性

- 如果一个E_{ij}的最优解决方案包括活动e_k，则E_{ik}的解决方案和E_{kj}的解决方案也必须是最优的。
- 证明：设A_{ij}是E_ij的一个最优解决方案，则$A_{ij}=A_{ik}U\{e_k\}UA_{kj}$(这里假定$E_{ij}$非空且$e_k$已知)。因此，如果$A_{ik}$不是最优，我们可以将最优的方案替换$A_{ik}$而获得一个更优的方案。这与假设$A_{ij}$是$E_ij$的一个最优解决方案矛盾，因此，$A_{ik}$和$A_{kj}$是最优的。

活动安排问题的递归式

- 这样我们就获得解决活动安排问题的递归解法。令es[i, j]=Eij里面相互兼容的活动的最大数。因为$i \geq j \Rightarrow E_{ij}=\phi$，所以es[i, j]=0.
- 如果$E_{ij} \neq \phi$, 且假定我们知道某活动e_k在该子集里，则es[i, j]=es[i, k]+1+es[k, j]. 但事实上，我们并不知道哪一个e_k在该子集里，因此，我们有

$$es[i,j]=\begin{cases}0 & \text{if } E_{ij}=\emptyset \\ \max\limits_{i<k<j, e_k \in E_{ij}} \{es[i,k]+es[k,j]+1\} & \text{if } E_{ij} \neq \emptyset\end{cases}$$

- 这样定义k的取值范围是因为由$E_{ij}=\{e_k \in E: f_i \leq s_k < f_k \leq s_j\}$可以推出$e_k$不可能是$e_i$或$e_j$。同时需要确保$e_k$在$E_{ij}$里。

活动安排问题贪心算法的最优性(续)

- 定理 设$E_{ij} \neq \emptyset$，且e_m是E_{ij}里面完成时间最早的活动，即$f_m=\min\{f_k: e_k \in E_{ij}\}$, 则：
  1) e_m必定属于E_{ij}里面某个最大兼容子集；
  2) $E_{im} = \emptyset$，因此，选择e_m导致E_{mj}成为唯一非空的子集。
- 证明 1)设A_{ij}是E_{ij}的一个最大兼容集合。将该集合里面的所有活动按照结束时间进行非递减排序。设e_k是A_{ij}里最早结束的活动，则下列情况必居其一：
  - 如果$e_k=e_m$，则$e_m \in A_{ij}$，证毕；
  - 否则，构建$A'_{ij}=A_{ij}-\{e_k\} \cup \{e_m\}$, 则A'ij也是$E_{ij}$里面的一个最大兼容子集。
- 2) 假设存在某$e_k \in E_{im}$，则$f_i \leq s_k < f_k \leq s_m < f_m \Rightarrow f_k < f_m$. 所以$e_k \in E_{ij}$并且其完成时间早于fm，这与em的选择矛盾。因此，$e_k \in E_{im}$不成立，$E_{im} = \emptyset$。

活动安排问题的贪心算法

- 活动安排问题的求解优以下规律：
  - 最优解里面包括的子问题个数为1个；
  - 需要考虑的选择数量为1个；
  - 最优解里包含了子问题的最优解。

我们的贪心选择

- 满足贪心策略的要素：分解为一个子问题，且该子问题的最优解将导致全局最优解！这样，我们就可以使用贪心策略来解决活动安排问题：
  - 每次选择一个活动时，选择完成时间最早的$e_m \in E_{ij}$。
  - 然后解决问题Emj。

活动安排问题的贪心算法

```
public static int greedySelector(int [] s, int [] f, boolean a[])
{  int n=s.length-1;
   a[1]=true;
   int j=1;
   int count=1;
   for (int i=2;i<=n;i++) {
      if (s[i]>=f[j]) {
         a[i]=true;
         j=i;
         count++;  }
      else a[i]=false;}
   return count; }
```

- 各活动的起始时间和结束时间存储于数组s和f中且按结束时间的非减序排列
- **如果所给出的活动未按非减序排列，可以用O(nlogn)的时间重排。**

活动安排问题的贪心算法

- 算法greedySelector的效率极高。当输入的活动已按结束时间的非减序排列，算法只需O(n)的时间安排n个活动，使最多的活动能相容地使用公共资源。

- 如果活动未按非减序排列，可选复杂度为O(nlogn)的排序算法。

多机调度问题(I)

- 现有任务n个, 机器不限。假定任何时间一台机器只能执行一个任务.
- s_i为任务i的开始时间, f_i为完成时间($s_i<f_i$). [s_i, f_i] 为处理任务i 的时间区间.
- 两个任务i, j 重叠是指两个任务的时间区间有重叠(不含始点或终点). 例如：区间[1,4]与区间[2,5]重叠, 而与区间[4,7]不重叠.
- 一个**任务分配方案是可行**的是指没有两个处理时段重叠的任务分配给同一台机器.
- **最优分配**指占用机器数最少的可行分配.

多机调度问题(I)

- 假设有n=7个任务，标号从a到g。它们的处理时段如表(1)所示。
- 暴力分配方案：将每个任务分配个一台机器.
- 把任务a、b、d分配给同一台机器，机器数降为5台。
- 贪心选择策略：逐步分配任务。每步分配一个任务，按任务开始时间的递增次序分配(见表(2))。若有“旧”机器可用，则将任务分配给它。否则，将任务分配给一台“新”机器。

不是最优分配

(1)

任务	a	b	c	d	e	f	g
开始	0	3	4	9	7	1	6
结束	2	7	7	11	10	5	8

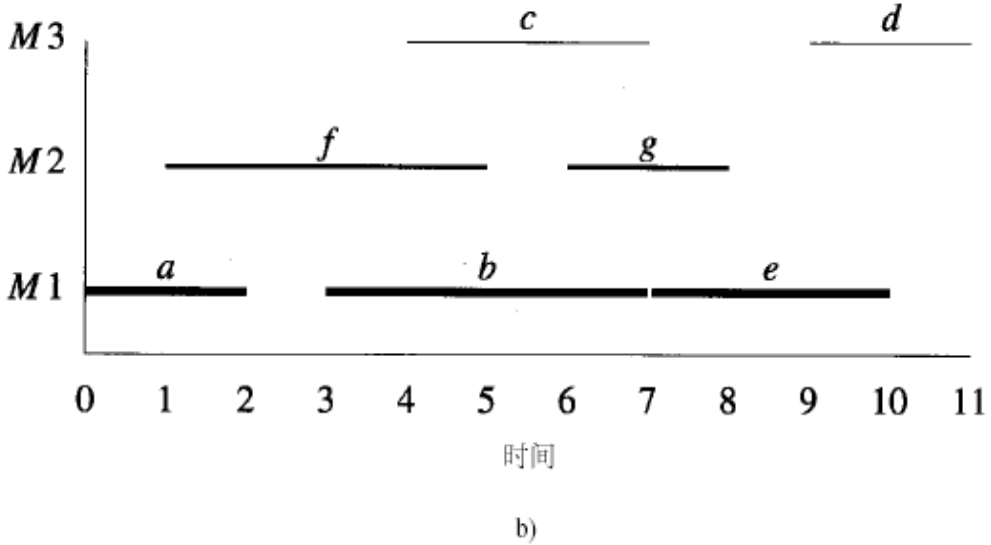
(2)

任务	a	f	b	c	g	e	d
开始	0	1	3	4	6	7	9
结束	2	5	7	7	8	10	11

多机调度问题(I)

- 证明多机调度问题的贪心算法却总能求得的整体最优解。
- 证明：
  - 任何可行解使用的机器数≥最大重迭任务数;所以优化调度使用的机器数≥最大重迭任务数。
  - 贪心解使用的机器数不超过最大重迭任务数：任何时候当算法使用一台新机器时,当前这些机器上的任务一定是彼此重叠的。

任务	a	f	b	c	g	e	d
开始	0	1	3	4	6	7	9
结束	2	5	7	7	8	10	11



b)

多机调度问题(II)

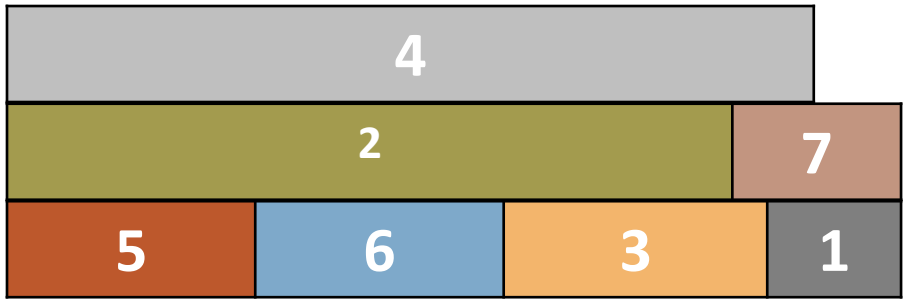
- 设有n个独立的作业{1, 2, …, n}, 由m台相同的机器进行加工处理。作业i所需的处理时间为ti。现约定，每个作业均可在任何一台机器上加工处理，但未完工前不允许终端处理。作业不能拆分成更小的子作业。
- 多机调度问题要求给出一种作业调度方案，使所给的n个作业在尽可能短的时间内由m台机器加工处理完成。
- 这个问题是NP完全问题，到目前为止还没有有效的解法。对于这一类问题，用贪心选择策略有时可以设计出较好的近似算法。

多机调度问题(II)

- 贪心选择策略：最长处理时间作业优先。
  - 当n≤m时，只要将机器i的[0, t_i]时间区间分配给作业i即可。
  - 当n>m时，首先将呢个作业依其所需的处理时间从大到小排序。然后依次顺序将作业分配给空闲的处理机。
- 当n≤m时，算法需要O(1)时间。
- 当n>m时，因为排序耗时O(nlogn), 算法需要O(nlogn)时间。

多机调度问题(II)

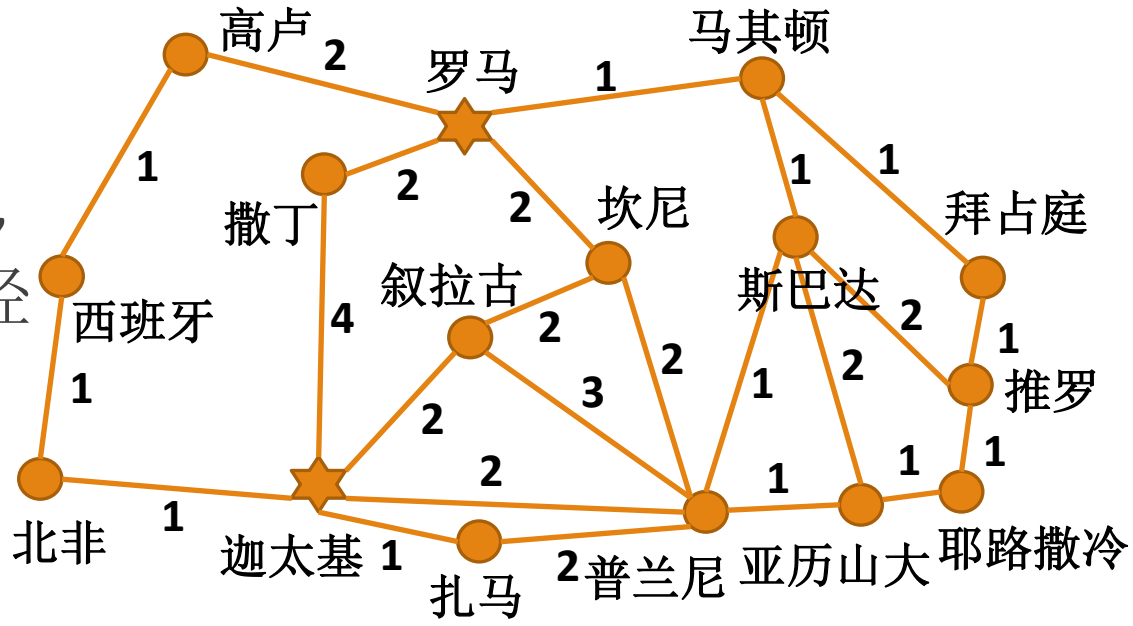
- 设7个独立作业{1,2,3,4,5,6,7}由3台机器M1，M2和M3加工处理。各作业所需的处理时间分别为{2,14,4,16,6,5,3}。按算法greedy产生的作业调度如下图所示，所需的加工时间为17。



最短路径问题

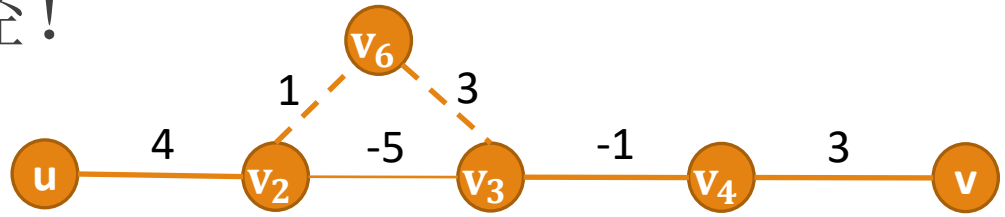
- 公元前218年，迦太基卓越的军事统帅汉尼拔决定进兵罗马，准备最后解决罗马问题。而进军罗马当然需要仔细选择行军路线。由于条条道路通罗马，选择的范围自然很大。如何选择最佳的进军路线就是摆在汉尼拔面前的首要任务。
- 要定义最短路径，先得定义路径权重。给定一个带权重的图G=(V,E), 边的权重函数为w: E→R, 则一条路径 $p=v_1 \to v_2 \to \cdots \to v_k$的权重为该路径上每条边的权重之和：

$$w(p)=\sum_{i=1}^{k-1} w(v_i, v_{i+1})$$



最短路径问题

- 从任意节点u到另一节点v的**最短路径**是所有从u到v的路径中权重最小的路径。
- **最短路径权重**定义为$\delta(u,v) = \min\{w(p): p是从u到v的路径\}$. 如果从u到v不存在路径，则$\delta(u,v) = \infty$。
- 如果一个图包含一个权重为负的循环，则有些节点之间将不存在最短路径。例如下图中的节点u和v之间就不存在最短路径。不管找出的路径有多短，我们总可以沿着负环路多绕行一次而获得一条更短的路径！



单源单点最短路径问题

- 对于没有负循环的连通图，任意两个节点之间都存在最短路径，且该最短路径必不会包括任何环路。
- Naive解法：遍历！
- 深度优先搜索：以源点为始点，在某条线索上进行最深度搜索不果后，才会回溯考察别的线索。
  - 皇帝总是将位子传给儿子，儿子穿给孙子，在没有儿子或孙子可传的情况下，传给兄弟或别的亲戚。没有亲戚的时候传给朋友等。
- 广度优先搜索：以源点为中心，一圈圈地往外搜索，直到所有节点都被访问为止。
  - 洪水爆发后，其蔓延的趋势呈现出广度优先模式，所以也被称为flood algorithm.

采用DFS的单源单点最短路径算法

- DFS=Depth First Search，类似地，BFS=Breadth First Search.

1. 从源点开始，初始化总成本初值为∞。
2. 从源点开始，按照DFS算法对图进行遍历。每经过一个新的节点时，对总成本更新如下：
   1) 如果总成本为∞，将新的成本设置为总成本。
   2) 如果总成本不等于∞，则将新的成本加到总成本上。
3. 如果抵达目标节点，则将该条路径及其长度记录下来，我们找到了一条路。
4. 如果无法往前推进，则进行回溯，回溯的时候需要将被回溯的边的权重从总成本中减去。如果总成本减到0，则将总成本设置为∞(表示从源点开始探索另一条路)。
5. 在DFS遍历结束后，比较所有找出来的路径，长度最短的即为最短路径。

单源单点最短路径问题

- 采用DFS的单源单点最短路径算法的时间复杂度与一对节点之间的路径数成正比。O(V!)属于难解问题范畴！
- 采用BFS的算法时间复杂度类似！
- **例外**：如果图的所有边的权重相等，或者是无权重的图，采用BFS的算法效率非常好！
- 观察：在搜索一对节点之间的最短路径时，我们已经考察了源节点到任意节点的所有路径。

单源多点最短路径问题

- 将问题由单源单点改变为单元多点，带来两个好处：
  - 摊销成本
  - 贪心选择属性
- Dijkstra算法(1956)：给定一个源点$s \in V$, 找出该节点到图G中其它所有节点的最短路径权重$\delta(s, v)$, 这里$v \in V$。

Edsger Wybe Dijkstra

- 狄杰斯特拉(迪科斯彻)，荷兰人，生于1930年，大学修数学与物理，3年获得学士学位。
- 1951年参加剑桥大学的短期程序员培训，成为第一批程序员之一。
- 1956年设计并实现了Dijkstra最短路径算法,
- 1959年设计了一种处理程序，成功地解决了“实时中断”（real-time interrupt)问题并获得博士学位。
- 最早指出“goto是有害的”，并首创结构化程序设计。
- 第一个Algol60 编译器的设计者和实现者，The操作系统的设计者和开发者。
- 结构化程序设计之父，与Donald Ervin Knuth(唐纳德 克努特)并称为最伟大的计算机科学家。
- 1972年获得图灵奖

最优子结构条件：每个大问题的最优解里面包括下一级小问题的最优解；
贪心选择属性：每个小问题的解可由贪心选择获得。

- Dijkstra’s 最短路算法是图论算法中应用最为广泛的算法, 主要原因是其计算复杂度低且容易实现.
- 所使用的贪心策略:按最短路长度从小到大依次求解.
- 贪心策略要想获得最优解，必须满足下面两个条件：
  - 最优子结构条件：最短路径里的任意一段路径都是相关两个节点之间的最短路径；
  - 贪心选择属性：从源节点的角度来看，路径往外延伸的下一个节点就是离源点距离最近的节点。即新节点到源节点的距离就是源点到该节点的最短路径。

Dijkstra最短路径算法

问题：为什么只更新跟x相邻点的d[y]，而不是更新所有跟集合 s 相邻点的 d 值？

因为第8步只确定了u点到初始点的最短距离，集合内其它点是**之前加入的**，也经历过第9~11步，所以与 x 没有相邻的点的 d 值是已经更新过的了，不会受到影响.

Dijkstra-shortest-path-algorithm (G, V, s)

```
1.  d[s] ← 0                                  O(V*1)   //源点到源点的距离当然是0
2.  for each v∈V-{s} do                       }O(V*1)  //所有其他节点到源点的距离初始化为无穷大
3.      d[v] ← ∞
4.  S ← ∅                                     }O(1)    //一开始，路径已知的节点集合S为空
5.  Q ← V                                              //Q是V-S节点集合组成的优先队列
6.  while Q ≠ ∅ do                                     //只要队列Q里还有节点，即循环寻找最短路径
7.      u ← EXTRACT-MIN (Q)                            //找出Q里离源点距离最短的节点u
8.      S ← S ∪ {u}                           O(V*1)   //把节点u加入集合S，u到源点的最短路径就是d(u)
9.      for each v ∈ Adj [u] do                        //对u的邻节点实施降距操作(Decrease-Distance)
10.         if d[v] > d[u]+w(u, v) then                //降距操作
11.         d[v] ← d[u]+w(u, v)
```

$$TD=\Theta(V\cdot T_{EXTRACT\text{-}MIN}+E\cdot T_{DECREASE\text{-}DISTANCE})$$

Dijkstra最短路径算法

Dijkstra-shortest-path-algorithm (G, V, s)

```
1.  d[s] ← 0                                  O(1)
2.  for each v∈V-{s} do                       }O(V*1)
3.      d[v] ← ∞                              }
4.  S ← ∅                                     }O(V)
5.  Q ← V                                     }
6.  while Q ≠ ∅ do
7.      u ← EXTRACT_MIN (Q)
8.      S ← S ∪ {u}                           O(V*1)
9.      for each v ∈ Adj [u] do
10.         if d[v] > d[u]+w(u, v) then
11.         d[v] ← d[u]+w(u, v)
```

//源点到源点的距离当然是0
//所有其他节点到源点的距离初始化为无穷大
//的节点集合S为空
//Q是V-S中点集合组成的优先队列
//短路径
//找出Q集离源点距离最短的节点u
//对u的邻节点实施降距操作(Decrease-Distance)

while循环执行的次数为V

执行时间与优先队列的实现有关, V*O(V)~V*O(lgV)

For循环的执行次数与每个节点的邻节点个数有关, 不好估计

注意到每条边只可能在一次降距操作中使用，所以在整个算法运行中for循环的执行次数为O(E)

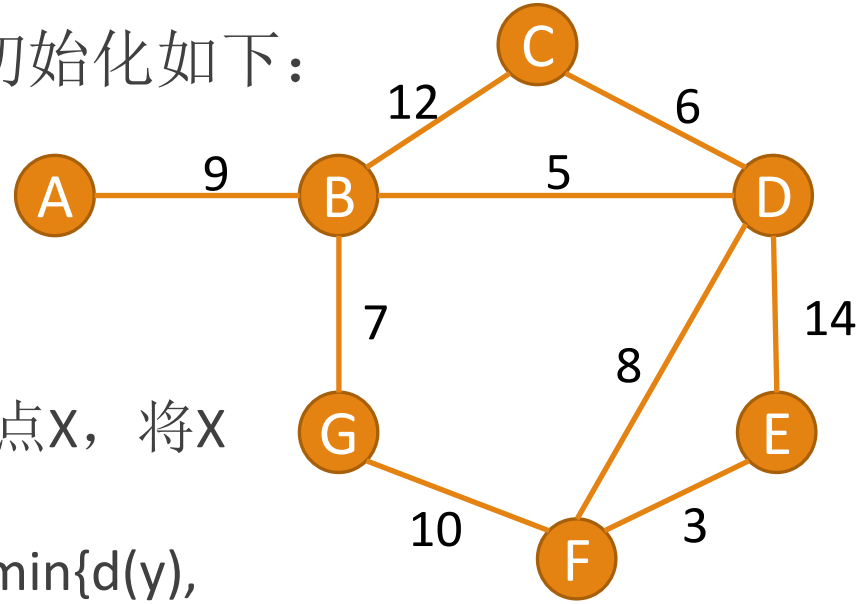
$$TD=\Theta(V\cdot T_{EXTRACT\_MIN}+E\cdot T_{DECREASE\_DISTANCE})$$

Dijkstra算法举例

用Dijkstra算法计算从节点A到所有其他节点的最短路径。

1. S初始化为 ∅ ，所有节点到源点A的距离d初始化如下：

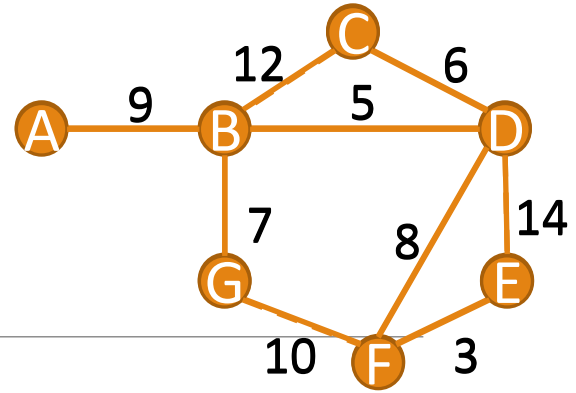
节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞



2. 循环n次：
   ① 在所有不属于S的节点里面选取d值最小的节点X，将X加入到S中。
   ② 更新从X出发的所有邻点到源点的距离d(y)= min{d(y), d(x)+w(x,y)} 。

1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞



2. S={A}, 更新各节点到源点的距离。

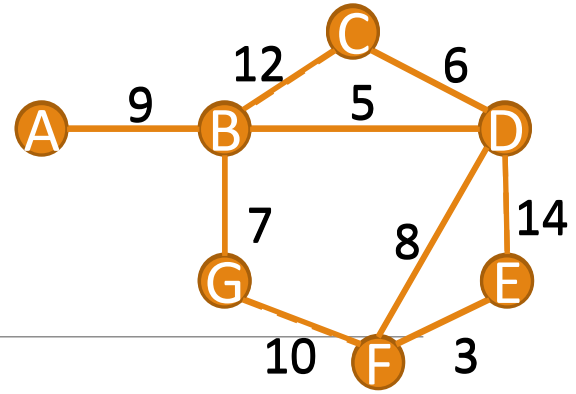
节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

问题：为什么当前的d(B)就是B点到源点的最短距离？

因为B是目前离A点最近的点，并且图中各边权重均为正数，再增加中间节点只能使节点B到源点的距离变大。

1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞



2. S={A}, 更新各节点到源点的距离。

节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

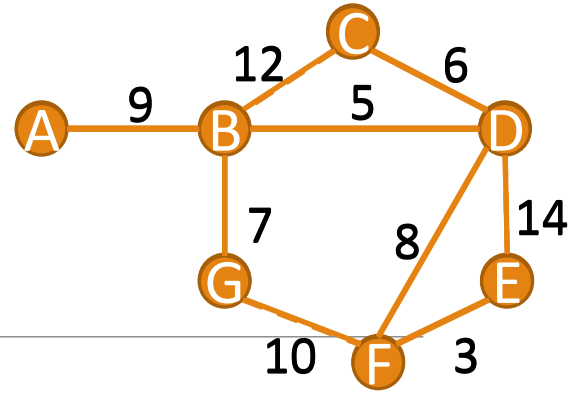
3. S={A, B},

节点	A	B	C	D	E	F	G
离源点距离	0	9	21	14	∞	∞	16

- B的邻点有C、D、G，依次比较d(v)和d(B)+w(B, v)，v依次取C、D、G。
- 当d(v)>d(B)+w(B, v) 时，要更新d(v)的值。例如d(B)+w(B, C)=9+12=21<∞(原来d(C)的值)，更新d(C)的值为21，即源点A通过B点到达C点的路径长度为21。这个过程叫做“降距”或者“松弛”。这便是Dijkstra算法的主要思想：通过“边”来松弛源点A到其余各个顶点的距离。
- 同理，通过点B，可以将d(D)的值从∞降距/松弛为14，d(G)的值降距为16.
- 降距/松弛完毕后d数组为：

1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞



2. S={A}, 更新各节点到源点的距离。

节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

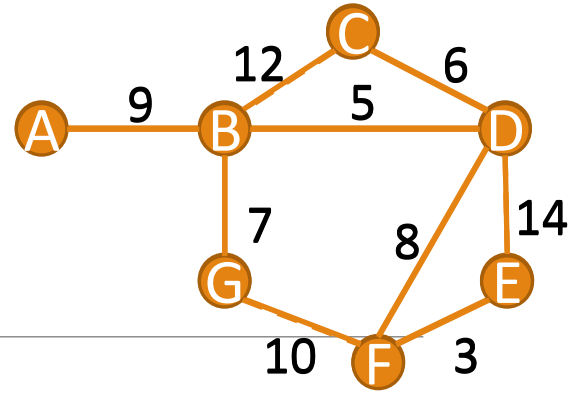
3. S={A, B},

节点	A	B	C	D	E	F	G
离源点距离	0	9	21	14	∞	∞	16

4. S={A, B, D},

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

- 更新后的d数组中，Q集合中离源点最近的顶点是D点，把D点加入集合S。
- d(D)从“估计值”变为“确定值”。
- D的邻点为C、E、F。依次比较d(v)和d(D)+w(D, v)，当d(v)>d(B)+w(B, v) 时，要更新d(v)的值。



1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞

2. S={A}, 更新各节点到源点的距离。

节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

3. S={A, B},

节点	A	B	C	D	E	F	G
离源点距离	0	9	21	14	∞	∞	16

4. S={A, B, D},

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

问题：为什么只更新跟D相邻点的d值，而不是更新所有跟集合S相邻点的 d 值？

回答：因为上一步只确定了D点到初始点的最短距离，集合S内其它点是**之前加入的**，已经经历过降距，所以与 D 没有相邻的点的 d 值是已经更新过了的，不会受到影响。比如d(G)。

1. S=∅,

节点	A	B	C	D	E	F	G
离源点距离	0	∞	∞	∞	∞	∞	∞

2. S={A}, 更新各节点到源点的距离。

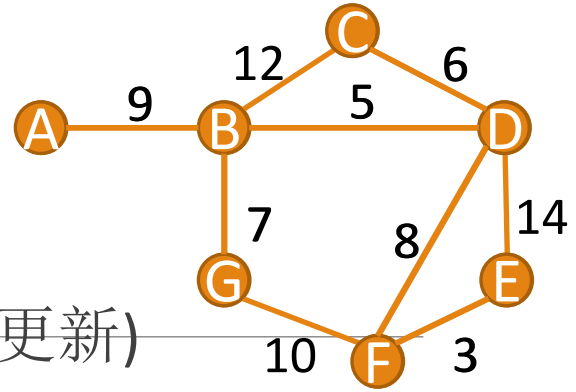
节点	A	B	C	D	E	F	G
离源点距离	0	9	∞	∞	∞	∞	∞

3. S={A, B},

节点	A	B	C	D	E	F	G
离源点距离	0	9	21	14	∞	∞	16

4. S={A, B, D},

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16



5. S={A, B, D, G}(没有更新)

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

6. S={A, B, D, G, C}(没有更新)

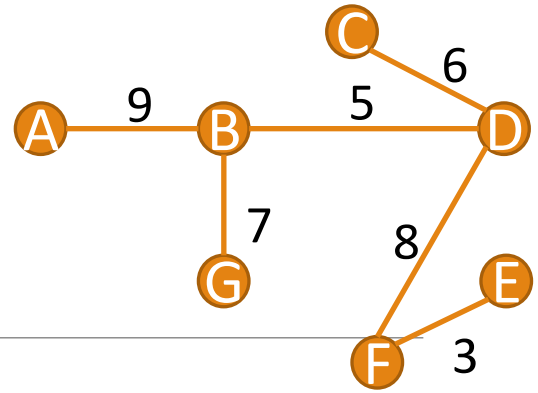
节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	28	22	16

7. S={A, B, D, G, C, F}

节点	A	B	C	D	E	F	G
离源点距离	0	9	20	14	25	22	16

8. S={A, B, D, G, C, f, E}, 算法结束。

Dijkstra算法的时间复杂度



	A	B	C	D	E	F	G
A	0	∞	∞	∞	∞	∞	∞
B	0	9	∞	∞	∞	∞	∞
D	0	9	21	14	∞	∞	16
G	0	9	20	14	28	22	16
C	0	9	20	14	28	22	16
F	0	9	20	14	28	22	16
E	0	9	20	14	25	22	16

$O(V^2)$

- Dijkstra算法的时间效率依赖于用来表示输入图本身的数据结构和用来实现优先队列的数据结构。
- 如果图用权重矩阵或者邻接表表示，优先队列用无序数组来实现，算法时间复杂度为$O(V^2)$。
- 如果图用邻接链表表示，优先队列用二叉堆或最小堆来实现，时间复杂度为$O(V+E)\log V)$。
- 如果用一种斐波那契堆的更复杂的数据结构来实现优先队列，它的最差算法时间复杂度为$O(E+V\log V)$。

Dijkstra算法的时间复杂性

- Dijkstra算法的时间复杂度为：

$$TD=\Theta(V\cdot T_{EXTRACT\text{-}MIN}+E\cdot T_{DECREASE\text{-}DISTANCE})$$

从一堆元素里面寻找最小值，即次序选择问题

从一堆元素里面寻找一个特定的元素(u的邻节点)

Q的实现方式	$T_{EXTRACT\text{-}MIN}$	T_{for}	总计
数组	O(V)	O(1)	$O(V^2)$
堆	O(lgV)	O(lgV)	O(ElgV)
斐波那契堆	O(lgV)聚类分析时间	O(1)聚类分析时间	O(E+VlgV)最坏时间

Dijkstra算法的正确性

证明对于任意节点v∈V，d[v]=δ(s,v)成立。

分三个步骤：

1. 任何处于集合S外面的节点，其距源点的距离不短于其至源点的最短路径长度；
2. 任何一个节点的最短路径在其前驱节点的最短路径确定后才能确定；
3. 任意节点在被加入到集合S的时候，其离源点的最短路径已经确定，即等式d[v]=δ(s,v)成立。

- 引理1 在Dijkstra算法初始化后，对于任意一个节点v∈V来说，不等式$d[v] \geq \delta(s,v)$必成立，且是整个算法的一个不变式，在算法执行的整个过程中都成立。

算法中的当前最短距离

最短路径权重

- 证明：1) 首先，在初始化后，除源点外每个节点的距离都被设置为无穷大，因此，不等式$d[v] \geq \delta(s,v)$成立。

2) 用反证法证明该不变式在算法的所有步骤都成立。由于节点到源点的距离只在降距操作时改变，因此，若不等式不成立，则必在某次降距操作时发生。设v是第一个在降距操作时打破不变式的节点，即$d[v]<\delta(s,v)$。又设u是导致节点v降距的节点，即$d[v]=d[u]+w(u,v)$, 则我们有

边的权重函数

$$d[v]<\delta(s,v) \leq \delta(s,u)+\delta(u,v) \leq \delta(s,u)+w(u,v) \leq d[u]+w(u,v)$$

即$d[v]<d[u]+w(u,v)$, 而这与$d[v]=d[u]+w(u,v)$矛盾! 因此引理1成立.

反证假设

三角不等式

最短路径短于任意具体路径

- 引理2 设节点u是从源点到节点v的最短路径上节点v的前驱节点(紧挨着v的节点)。如果d[u]=δ(s,u)，则在边(u,v)用于降距后，我们有d[v]=δ(s,v)。

算法中的当前最短距离

最短路径权重

- 证明：1) 由于u是v的前驱，我们有

$$\delta(s,v)=\delta(s,u)+w(u,v)$$

2)假设在降距操作考虑边(u,v)前有d[v]>δ(s,v)。

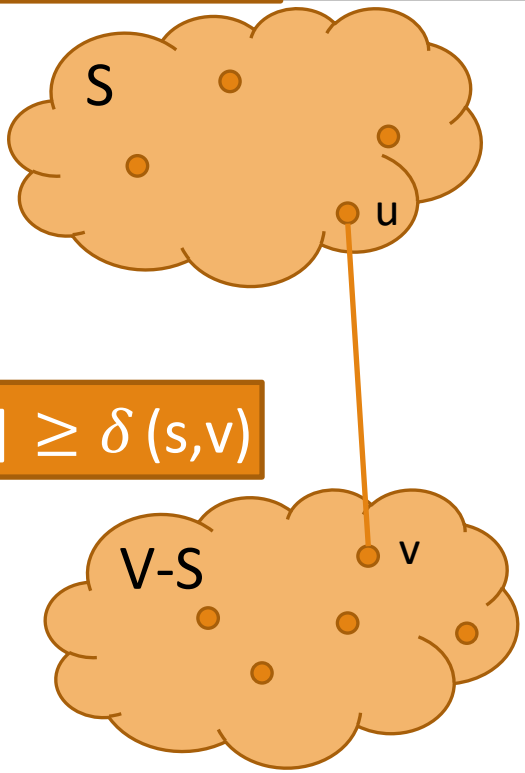
我们有

引理1已证明d[v] $\geq \delta$(s,v)

$$d[v]>\delta(s,v)=\delta(s,u)+w(u,v)=d[u]+w(u,v)$$

即d[v]>d[u]+w(u,v)。因此，在算法对u的邻节点进行降距操作时将设置

$$d[v]=d[u]+w(u,v)=\delta(s,v)。$$

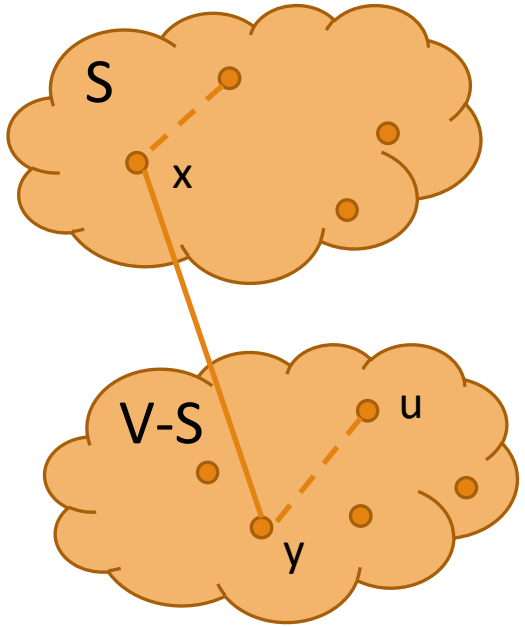


$d[v] \geq \delta(s,v)$

- 定理 Dijkstra算法终结时，$d[v]=\delta(s,v)$，这里$v \in V$。
- 证明：我们只需证明在v被加入到集合S的时候$d[v]=\delta(s, v)$即可。

使用反证法. 假设u是第一个被加入到集合S的违反上述等式的节点, 即$d[v]>\delta(s, v)$. 那么从源点到u的最短路径必经过S集合外面的某个或某些节点.

设y是该最短路径经过集合S外的第一个节点，又设该路径上节点y的前驱是节点x，则x必然属于集合S。



由于u是第一个违反定理等式$d[v]=\delta(s,v)$的节点，我们有$d[x]=\delta(s, x)$。当x被加入到S的时候，x的邻节点y的距离将被降距。这意味着

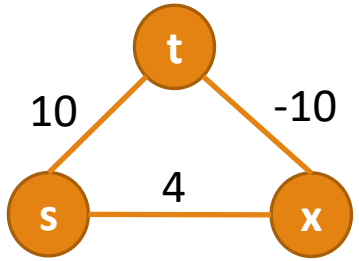
$$d[y] = \delta(s,y) \leq \delta(s, u) < d[u]$$

但是u的选择使得$d[u] \leq d[y]$，从而产生矛盾。因此，定理成立。

Dijkstra算法不能用于包含负权重的图

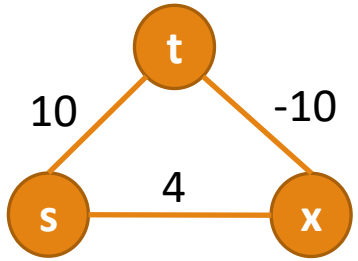
- 如右图:

节点	s	t	x
离源点距离	0	∞	∞
	0	10	4
	0	-6	4



- Dijkstra算法算出的 {s,x}之间的最短路径为4。
- 最短权重的边被考虑得太晚！

遍历点⟶遍历边



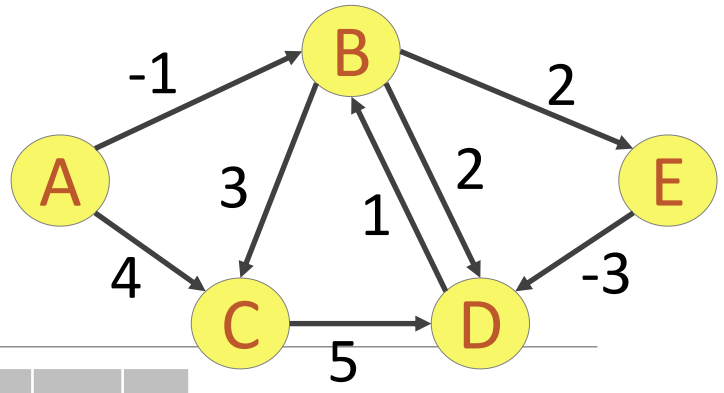
- Dijkstra算法对节点划分, 遍历节点。
- 对有负权重边的图, 节点到源点的最短路径不能确定, 不能采用对节点进行划分的方法。
- 换个想法：把针对节点进行降距改为对每条边进行降距。
- 依据: 每次降距都是针对每一条边进行, 因为降距的核心就是从一条新的边获得更短的路径。
- 结束准则：当一轮降距结束时(即每条边都考察一次后)，如果没有任何距离被降低，则说明最短路径已全部找出；如果有距离降低，则需要再来一轮降距。
- 最多需要V轮降距操作。

Bellman-Ford算法

Bellman-Ford-SHORTEST-PATHS (G, S)

1. d[s] ← 0
2. for v ∈ V - {s} do
3. d[v] ← ∞
4. for i ← 1 to V - 1 do //对所有的边进行V-1次降距操作
5. for edge (u, v) ∈ E do //对每条边进行考察，顺序任意
6. if d[v] > d[u] + w(u, v) then //降距操作
7. d[v] ← d[u] + w(u, v)

Example of Bellman-Ford



	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
	0	∞	∞	∞	∞	0	-1	2	∞	∞	0	-1	2	-2	1	0	-1	2	-2	1
BE(2)					∞					1					1					1
DB(1)		∞					-1					-1					-1			
BD(2)				∞					1					-2					-2	
AB(-1)		-1					-1					-1					-1			
AC(4)			4					2					2					2		
DC(5)			4					2					2					2		
BC(3)			2					2					2					2		
ED(-3)				∞					-2					-2					-2	
	0	-1	2	∞	∞	0	-1	2	-2	1	0	-1	2	-2	1	0	-1	2	-2	1

- ✓V=5个节点，
- ✓ V-1=4。最多需要做 4次循环。
- ✓对8条边遍历。
- ✓如果

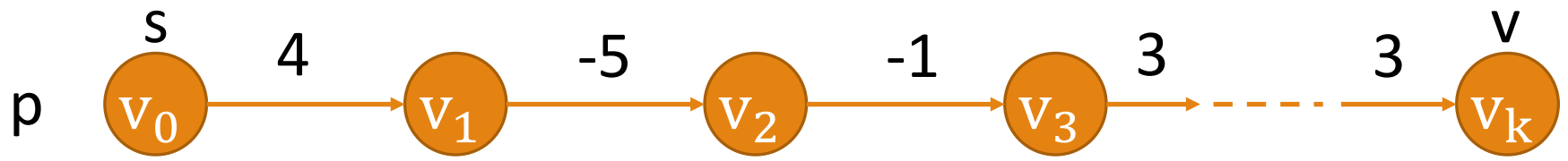
d[v]>d[u]+w(u,v)

那么

d[v]=d[u]+w(u,v)

Bellman-Ford算法的正确性

- 定理 如果图G=(V, E)不包括负循环，则在Bellman-Ford算法终止后，$\forall v \in V, d[v] = \delta(s, v)$.
- 证明 设v∈V为任意节点, 考虑从源点s到任意节点v的最短路径p.



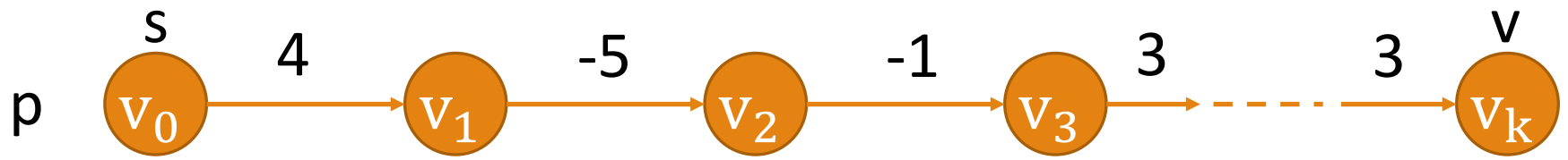
由于p是最短路径，我们有$\delta(s, v_i)=\delta(s, v_{i-1})+w(v_{i-1}, v_i)$.

在算法开始时，第1行的初始化将把源点s的距离设置为0，即$d[v_0]=0=\delta(s, v_0)$, 并且$d[v_0]$在算法的执行过程中保持不变。

在第1轮降距循环中，由于所有的边都考虑了一遍，因此，边(s, v_1)一定会被考虑到，从而导致v_1的最短距离被更新为真正的最短路径长度，即$d[v_1]=\delta(s, v_1)$.

在第2轮降距循环中，由于所有的边又轮转考虑了一遍，因此，边(v_1, v_2)一定会被考虑到，从而导致v_2的最短距离被更新为真正的最短路径长度$d[v_2]=\delta(s, v_1)+w(v_1, v_2)$，即$d[v_2]=\delta(s, v_2)$.

……

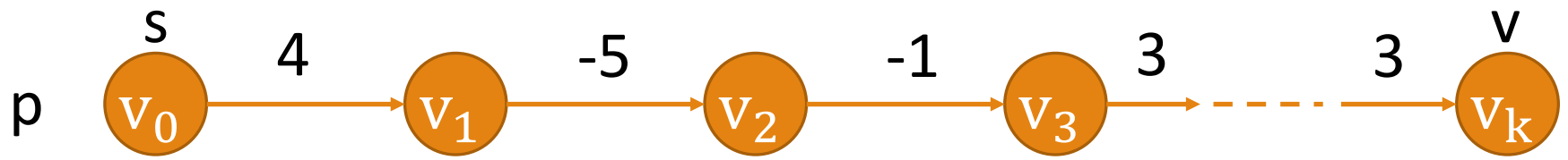


在第k轮降距循环中，由于所有的边又轮转考虑了一遍，因此，边(v_{k-1}, v_k)一定会被考虑到，从而导致v_k的最短距离被更新为真正的最短路径长度$d[v_k]=\delta(s, v_{k-1})+w(v_{k-1}, v_k)$，即

$$d[v_k]=\delta(s, v_k)=\delta(s, v).$$

在第1轮降距循环中，由于所有的边都考虑了一遍，因此，边(s, v_1)

由于图G不包括负循环，路径p必定是一条简单路径(不包括循环的路径)，如果不是简单路径，我们可以将路径上的循环去掉，结果仍然是一条最短路径。因此路径p最长只有V-1条边，而算法循环的遍数是V-1遍。因此，算法结束后，最短路径p将被发现，即对于任意节点v，Bellman-Ford算法确实能将源点到节点v的最短路径求出。



在第k轮降距循环中，由于所有的边又轮转考虑了一遍，因此，边(v_{k-1}, v_k)一定会被考虑到，从而导致v_k的最短距离被更新为真正的最短路径长度$d[v_k]=\delta(s, v_{k-1})+w(v_{k-1}, v_k)$，即

$$d[v_k]=\delta(s, v_k)=\delta(s, v).$$

负循环检查问题

- **Bellman-Ford 算法虽然能应对负权重，但却不能求解有负循环的所有最短路径。**
- 如果有负循环存在，节点之间可能不存在最短路径。
- 我们可以侦测一个图是否存在负循环。
- 观察：如果没有负环路，最短路径包含的边的条数最多只有V-1条，因此，在V-1轮降距操作后，所有最短路径都已经找出。换言之，如果存在负环路，则某些“最短路径”包含的边的条数将超过V-1条(实际上为无穷)，因此，在V-1轮降距操作后，仍然存在d[v]>d[u]+w(u, v)的情况。

负循环检查问题

- 推论 如果任意d[v]经过V-1次循环后还是不收敛(即仍可降低)，则该图存在一个负循环。
- 根据推论，对Bellman-Ford算法进行修改：

Bellman-Ford-SHORTEST-PATHS-II (G, s)

1. d[s] ← 0 $O(1)$
2. for v ∈ V - {s} do d[v] ← ∞ $O(1)$
3. for i ← 1 to V - 1 do V次
4. for edge (u, v) ∈ E do E次 $O(VE)$
5. if d[v] > d[u] + w(u, v) then d[v] ← d[u] + w(u, v)
6. for edge (u, v) ∈ E do $O(E)$
7. if if d[v] > d[u] + w(u, v) then 报告负环路存在

Richard Bellman(1920-1984)

- 理查德· 贝尔曼1920年出生于美国纽约，是美国数学家，美国国家科学院院士。
  - 1941年在布鲁克林大学获得数学学士学位
  - 1946年在普林斯顿大学获得博士学位
  - 随后在一个理论物理部门的团体工作
  - 1953年发明了动态规划。
  - 之后他的兴趣转移到生物和医药领域。
  - 1967年创办了杂志《Mathematical Biosciences》并任主编。
  - 1973年被诊断出脑瘤，手术后瘫痪。1984年离世。
  - 1979年，被授予IEEE奖章
  - 1985年，为了纪念理查德·贝尔曼，人们创立了贝尔曼数学生物科学奖。

小莱斯特·福特(1927-)

- Lester Randolph Ford Jr. (born September 23, 1927, Houston) is an American mathematician specializing in network flow problems. He is the son of mathematician Lester R. Ford Sr.
- Ford's paper with D. R. Fulkerson on the maximum flow problem and the Ford-Fulkerson algorithm for solving it, published as a technical report in 1954 and in a journal in 1956, established the max-flow min-cut theorem(最大流最小割定理). Ford also developed the Bellman-Ford algorithm for finding shortest paths in graphs that have negatively weighted edges before Bellman.

David Albert Huffman (1925–1999)

1944年毕业于俄亥俄州立大学电子工程专业，获学士学位;

1949年毕业于俄亥俄州立大学电子工程专业，获硕士学位;

1951年，哈夫曼在MIT的信息论导师Robert M. Fano给哈夫曼和他的同学们的学期报告的题目是“寻找最有效的二进制编码”。学生们可以选择是完成学期报告还是期末考试。由于无法证明哪个已有编码是最有效的，哈夫曼放弃对已有编码的研究，转向新编码的探索，最终发现了基于有效频率二叉树编码的想法，并很快证明了这个方法是最有效的。

1952年，发表了《A Method for the Construction of Minimum-Redundancy Codes》一文，它就是Huffman编码。

1953年毕业于MIT电子工程专业，获博士学位。

1953-1967在MIT任教.

1967年加入加州大学Santa Cruz分校，创办了计算机系并担任系主任(1970-1973).

哈夫曼编码

- Hufmann coding 是最古老，以及最优雅的数据压缩方法之一。其压缩率通常在20%～90%之间。
- 它是以最小冗余编码为基础的，即如果我们知道数据中的不同符号在数据中的出现频率，我们就可以对它用一种占用空间最少的编码方式进行编码，这种方法是，对于最频繁出现的符号制定最短长度的编码，而对于较少出现的符号给较长长度的编码。
- 哈夫曼编码可以对各种类型的数据进行压缩，但在本节中我们仅仅针对字符进行编码。

哈夫曼编码

- 例如一个包含100,000个字符的文件，各字符出现频率不同，如表中所示：

	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

- 定长编码需要300,000位，而按表中变长编码方案，文件的总码长为

$$(45\times1+13\times3+12\times3+16\times3+9\times4+5\times4)\times1000=224,000$$

- 变长编码方案比用定长码方案总码长较少约45%.

前缀码

- 对每一个字符规定一个0, 1串作为其代码，并要求任一字符的代码都不是其它字符代码的前缀。这种编码称为前缀码。
- 编码的前缀性质可以使译码方法非常简单。
- 表示最优前缀码的二叉树总是一棵完全二叉树，即树中任一结点都有2个儿子结点。
- 平均码长定义为：

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

- 使平均码长达到最小的前缀码编码方案称为给定编码字符集C的最优前缀码。

构造哈夫曼编码

- i=1, {a:45, b:13, c:12, d:16, e:9, f:5}

  f.freq=5最小， x1=f;

  e.freq=9最小， y1=e;

  z1.freq=5+9=14.

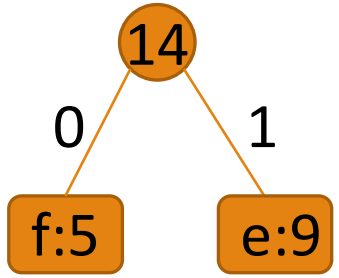
	a	b	c	d	e	f
频率	45	13	12	16	9	5
变长码	0	101	100	111	1101	1100

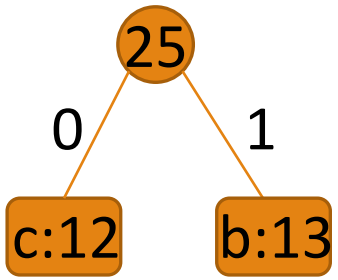
- i=2, {a:45, b:13, c:12, d:16, z1:14}

  c. freq=12最小， x2=c;

  b. freq=13最小， y2=b;

  z2.freq=12+13=25.





构造哈夫曼编码

	a	b	c	d	e	f
频率	45	13	12	16	9	5
变长码	0	101	100	111	1101	1100

- i=3, {a:45, d:16, z1:14, z2:25}

  z1. freq=14， x3=z1; d. freq=16， y3=d;

  z3.freq=14+16=30.

- i=4, {a:45, z2:25, z3:30}

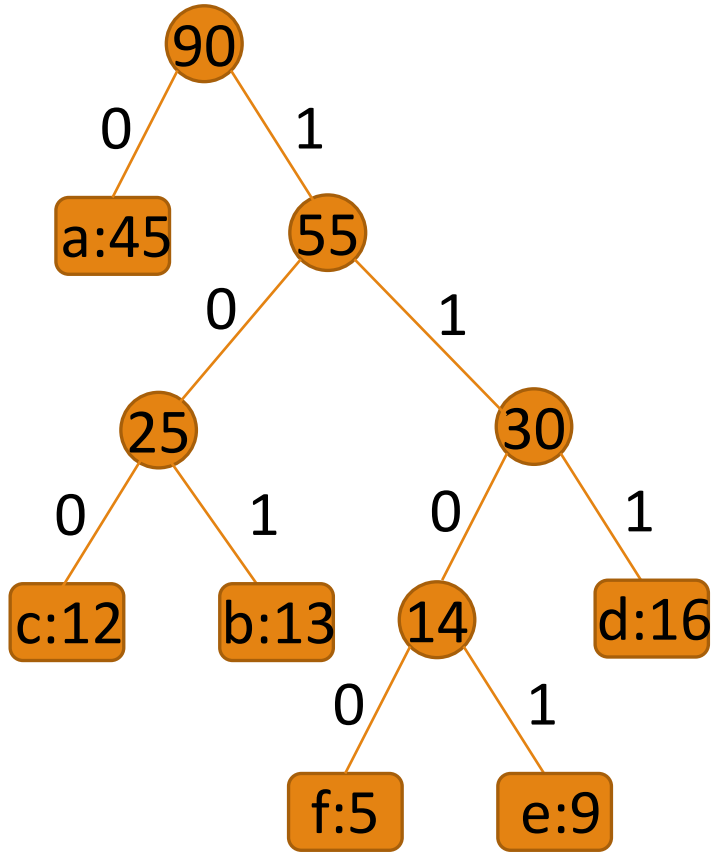
  z2. freq=25， x4=z2; z3. freq=30， y4=z3;

  z4.freq=25+30=55.

- i=5, {a:45, z4:55}

  a. freq=45， x5=a; z4. freq=55， y5=z4;

  z5.freq=45+55=90.



构造哈夫曼编码

	a	b	c	d	e	f
频率	45	13	12	16	9	5
变长码	0	101	100	111	1101	1100

- i=3, {a:45, d:16, z1:14, z2:25}

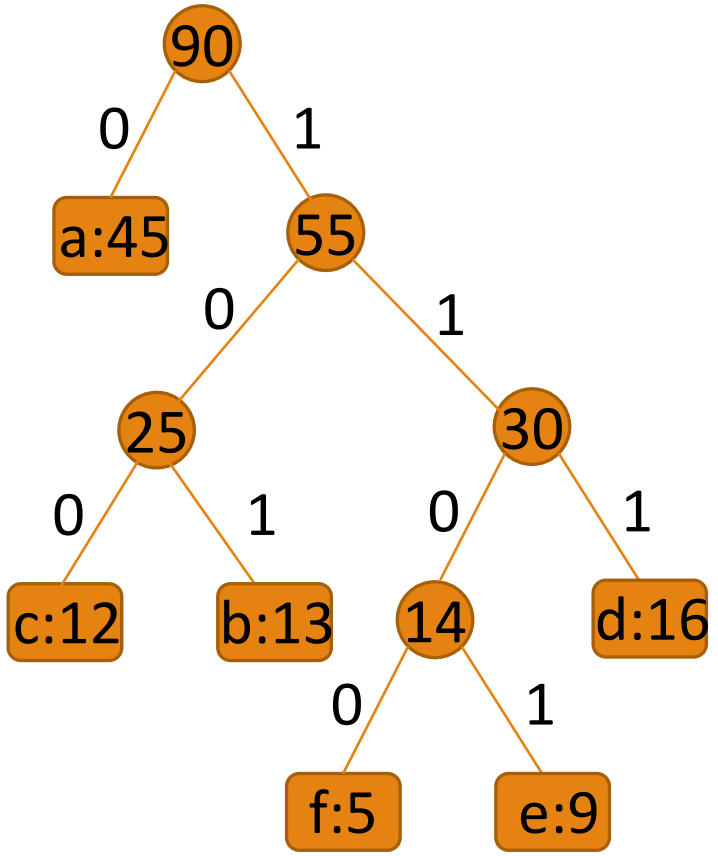
  z1. freq=14， x3=z1; d. freq=16， y3=d;

  z3.freq=14+16=30.

哈夫曼算法以自底向上的方式构造表示最优前缀码的二叉树T。

算法以|C|个叶结点开始, 执行|C| - 1次的“合并”运算后产生最终所要求的树T。

哈夫曼算法的贪心选择性质：合并出现频率最低的两个字符。



哈夫曼算法

1. 初始：根据n个字符的频率{w1，w2，…，wn}构成n个根结点的集合F={T1，T2，…，Tn}，其中每棵二叉树Ti中只有一个带权为wi的根结点，其左右子树均为空(叶子结点).
2. 在F中选取两棵根结点的权值最小的树作为左右子树来构造一棵新的二叉树，且置新的二叉树的根结点的权值为其左、右子树结点的根结点的权值之和.
3. 在F中删除这两棵树，同时将新得到的二叉树加入F中.
4. 重复（2）和（3），直到F中只含一棵树时为止。称这棵树为最优二叉树（或哈夫曼树）

如果约定将每个结点的左分支表示字符“0”，右分支表示字符“1”，则可以把从根结点到某叶子结点的路径上分支字符组成的字符串作为该叶子结点的编码.

哈夫曼算法

- 假定C是一个n个字符的集合，
- 每个字符c∈C都是一个对象，
- c.freq是字符的出现频率。
- 算法使用一个以freq为关键字的最小优先队列Q，以识别两个最低频率的对象并将其合并。
- 当合并两个对象时，得到的新对象的频率设置为原来的两个对象的频率之和。

- HUFFMAN(C)

```
1.  n=|C|                          O(1)
2.  Q=C                            O(n)
3.  for i=1 to n-1                 O(n)
4.      allocate a new node z
5.      z.left=x=EXTRACT-MIN(Q)
6.      z.right=y=EXTRACT-MIN(Q)
7.      INSERT(Q, z)               O(logn)
8.  return EXTRACT-MIn(Q)
```

哈夫曼编码过程

哈夫曼编码压缩数据由以下步骤组成：

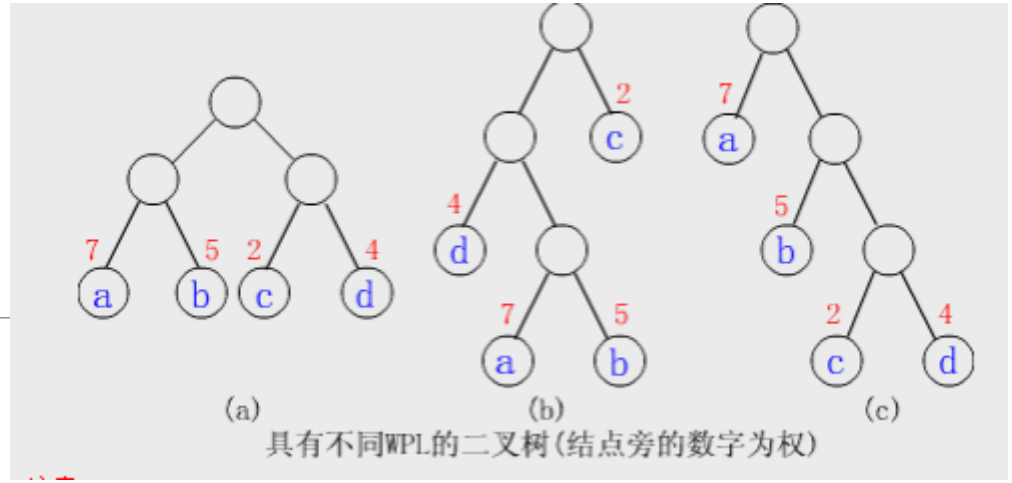
a）检查字符在数据中的出现频率。对要压缩的文本进行扫描，然后记录下各个字符出现的次数。为了降低文件头的尺寸，可以等比例缩小字符频率，但不能把在文本中出现的字符的频率缩小成0。

b）构建哈夫曼树。首先，我们把所有出现的字符作为一个单节点数，在节点上标识一个数字代表字符出现频率。每次我们选取具有最低频率的两个树，并将他们合并，把两个树的频率相加，赋给新树的根节点。重复这个步骤，直到最后只剩下一棵树，就是最终我们需要的哈夫曼树。

c）创建哈夫曼编码表。每个叶子节点代表了一个在原文中出现的字符。每个字符的编码就是从根节点到该叶子节点的路径。由于字节中的每一位由0，1两种状态，这也正是二叉树尤其重要和常用的原因。从根节点出发，如果进入左子树，则在编码上填0，如果进入右子树，则在编码上填1，直到到达叶子节点，就完成了该字符的编码。

d）生成压缩后结果，由一个文件头和压缩后的数据组成。

哈夫曼树



- **哈夫曼树或最优二叉树**

  在权为w_1，w_2，…，w_n的n个叶子所构成的所有二叉树中，带权路径长度最小(即代价最小)的二叉树称为**哈夫曼树或最优二叉树**

- 【例】给定4个叶子结点a，b，c和d，分别带权7，5，2和4。构造如下图所示的三棵二叉树(还有许多棵)，它们的带权路径长度分别为：

  (a)WPL=7*2+5*2+2*2+4*2=36
  (b)WPL=7*3+5*3+2*1+4*2=46
  (c)WPL=7*1+5*2+2*3+4*3=35

  其中(c)树的WPL最小，可以验证，它就是哈夫曼树。

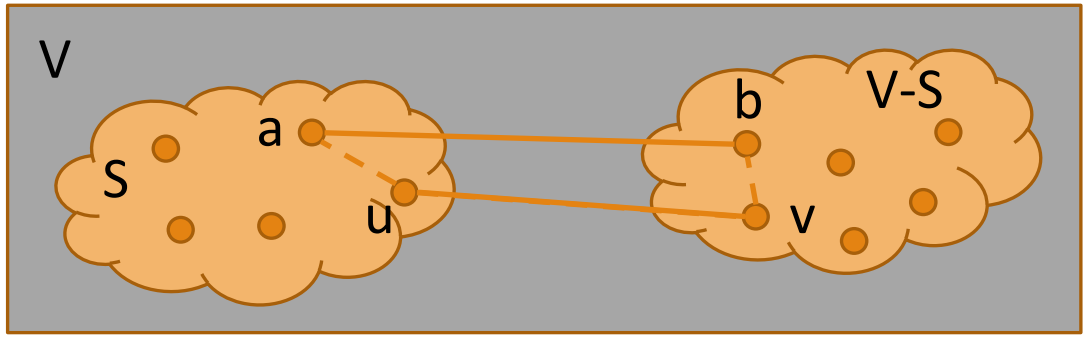
- 注意：

① 叶子上的权值均相同时，完全二叉树一定是最优二叉树，而完全二叉树不一定是最优二叉树。
② 最优二叉树中，权越大的叶子离根越近。
③ 最优二叉树的形态不唯一，WPL最小

最小生成树（Minimum Spanning Tree）

- 给定无向连通带权图G =(V,E)， E中每条边的权重为w(u, v). 如果G的子图T是一棵包含G的所有顶点的树, 则称T为图G的生成树。生成树上各边权的总和W(T)=$\sum_{(u,v)\in T} w(u,v)$, 称为该生成树的耗费。在G的所有生成树中，耗费为最小的生成树称为G的最小生成树(MST)。
- 具有n 个顶点的连通的无向图G , 每个生成树刚好具有n-1条边，所以问题是用某种方法选择n-1条边使它们形成G的最小生成树。
- 例如：要在n个城市之间铺设光缆，主要目标是要使这 n 个城市的任意两个之间都可以通信，但铺设光缆的费用很高，且各个城市之间铺设光缆的费用不同，因此另一个目标是要使铺设光缆的总费用最低。这就需要找到带权的最小生成树。

最小生成树的性质



- MST性质：设G=(V，E)是一个连通带权图，S是顶点集V的一个非空真子集．若(u，v)是G中一条一个端点在S中(即u∈S)，另一个端点不在S中(即v∈V-U)的边．且在所有这样的边中(u，v)具有最小权值，则一定存在G的一棵最小生成树包含(u，v)．

- 证明　用反证法，假设G中任何一棵MST都不含边(u，v)，否则性质已经成立。
  - 则若T为G的任意一棵MST，且(u，v)∉T。那么根据树的定义，T中必有一条从u到v的路径p，p上必有一条边连接集合S和V-S，不妨记为(a，b)，否则u和v不连通。当把边(u，v)加入树T时，p∪{(u，v)}必构成了一个环路。删去(a，b)后环路消失，由此生成另一生成树T’。
  - T’和T的差别仅在于T’用(u，v)取代了T中权重可能更大的边(a，b)。因为w(u，v)≤w(a，b)，所以w(T’)=w(T)+w(u，v)-w(a，b)≤w(T)。即T’是一棵比T更优的MST，所以T不是G的MST，这与假设矛盾。所以MST性质成立。

Prim算法

- 沃伊捷赫·亚尔尼克(Vojtěch Jarník, 1897 – 1970)被称为“第一个在世界科学领域具有广泛而长远的影响的捷克斯洛伐克数学家”。他的研究领域包括数论，数学分析和图算法。1930年发表了关于构造最小生成树算法的论文。
- 罗伯特·普里姆(Robert Clay Prim, 1921 -)是美国数学家和计算机科学家。1941年于德州大学奥斯汀分校获得电气工程学士学位。1949年获得普林斯顿大学的数学博士学位。 1958年-1961年普里姆在贝尔实验室(Bell Laboratories)做数学研究室主任时发现了Prim算法。
- 1959年，艾兹格·狄杰斯特拉(Edsger Dijkstra)再次发现了该算法(1956年Dijkstra最短路径算法)。
- 在某些场合，Prim算法又被称为DJP算法、亚尔尼克算法或普里姆-亚尔尼克算法。

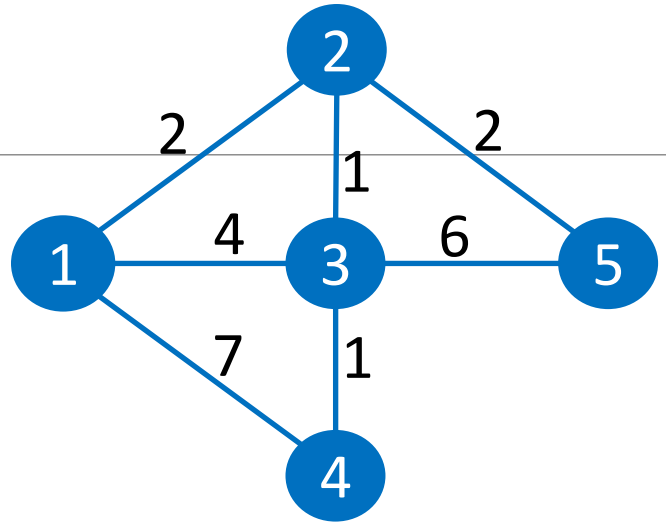
Prim算法

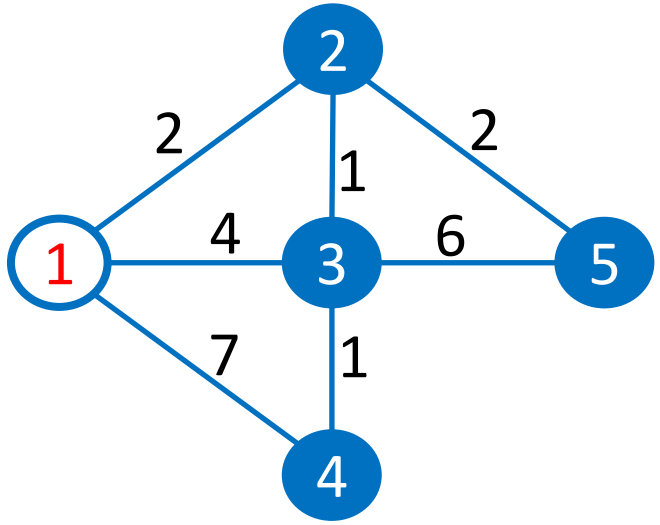
- Prim算法采用与Dijkstra、Bellman-Ford算法一样的”蓝白点” 思想。白点代表已经进入最小生成树的点，蓝点代表未进入最小生成树的点。
- 首先置S={s}, 然后, 只要S是V的真子集, 就作如下的贪心选择：
  - 选取满足条件u∈S, v∈V-S, 且w(u, v)最小的边, 将顶点v添加到S中。这个过程一直进行到S=V时为止。
- 在这个过程中选取到的所有边恰好构成G的一棵最小生成树。

Prim算法

- 初始时S=∅,权值之和MST=0. 所有的点都是蓝点, min[1]=0, min[2,3,4,5]=∞, min[1]=0最小. 找到蓝点1. 将1变为白点。

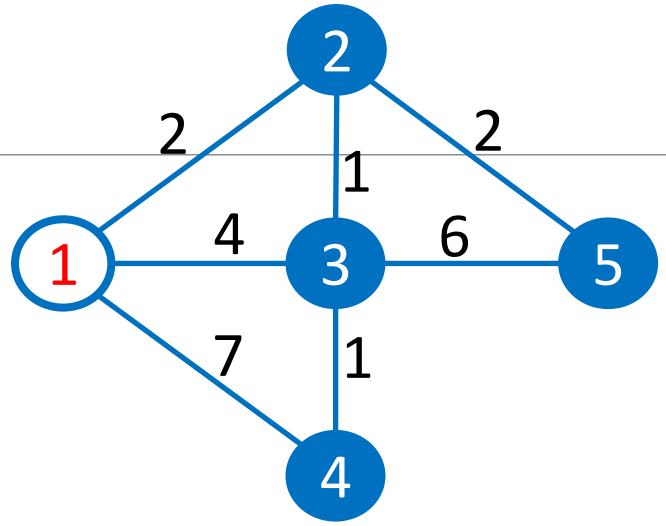
节点	1	2	3	4	5
最小边权	0	∞	∞	∞	∞





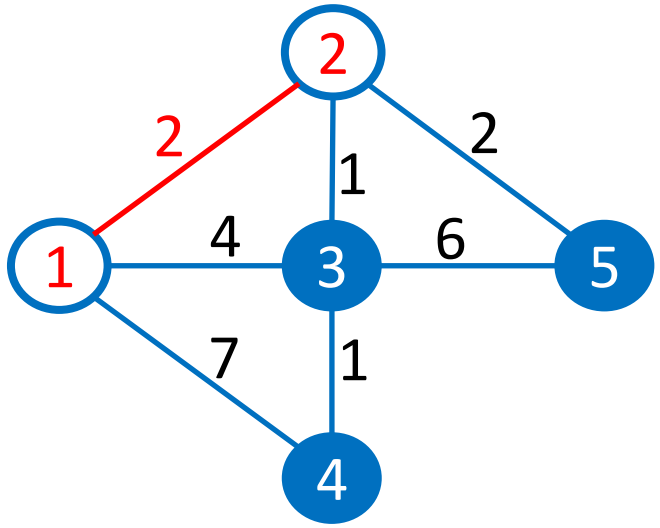
Prim算法

节点	1	2	3	4	5
最小边权	0	∞	∞	∞	∞



- 接着枚举与1相连的所有蓝点并修改它们与白点相连的最小边权。
min[2]=w(1,2)=2; min[3]=w(1,3)=4;
min[4]=w(1,4)=7。

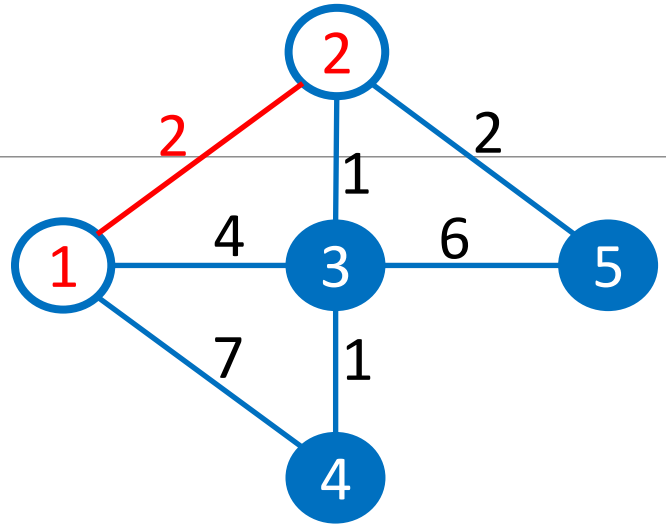
节点	1	2	3	4	5
最小边权	0	2	4	7	∞



- min[2]最小，将2变为白点，并标记(1,2)边。

Prim算法

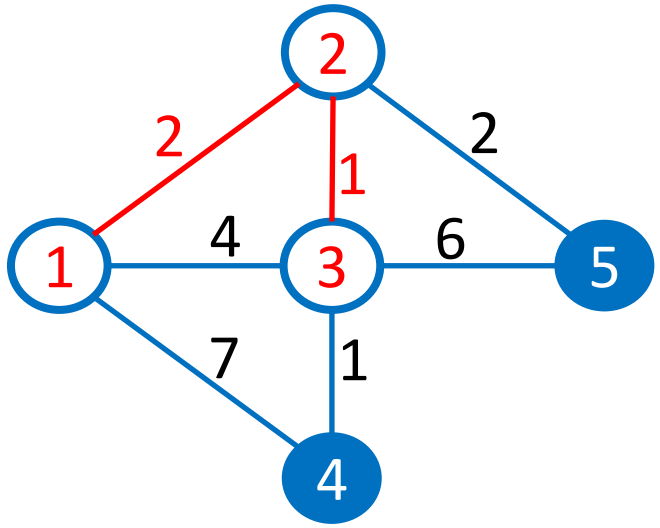
节点	1	2	3	4	5
最小边权	0	2	4	7	∞



- 接着枚举与2相连的所有蓝点并修改它们与白点相连的最小边权.

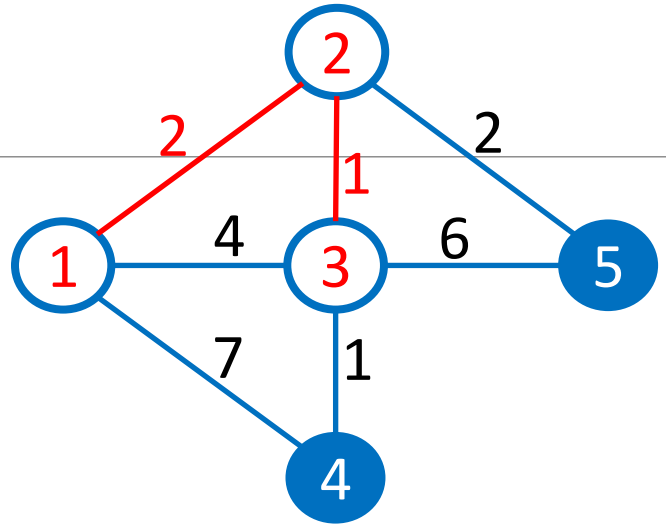
min[3]=w(2,3)=1;min[5]=w(2,5)=2. min[3]最小. 将3变为白点, 并标记(2,3)边。

节点	1	2	3	4	5
最小边权	0	2	1	7	2



Prim算法

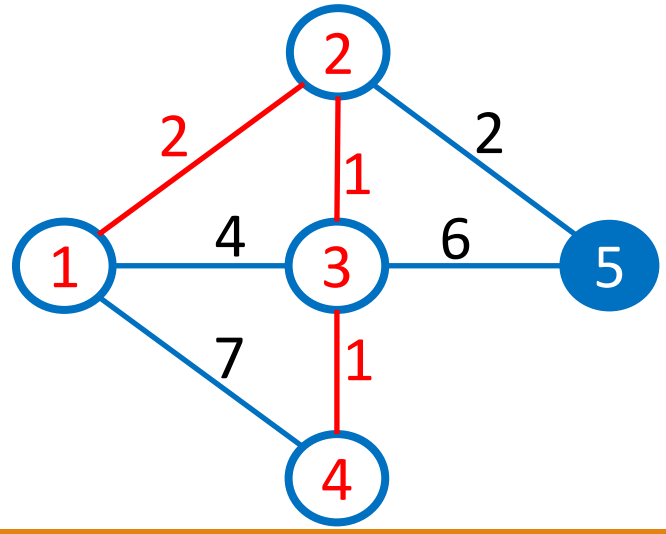
节点	1	2	3	4	5
最小边权	0	2	1	7	2



- 接着枚举与3相连的所有蓝点并修改它们与白点相连的最小边权.

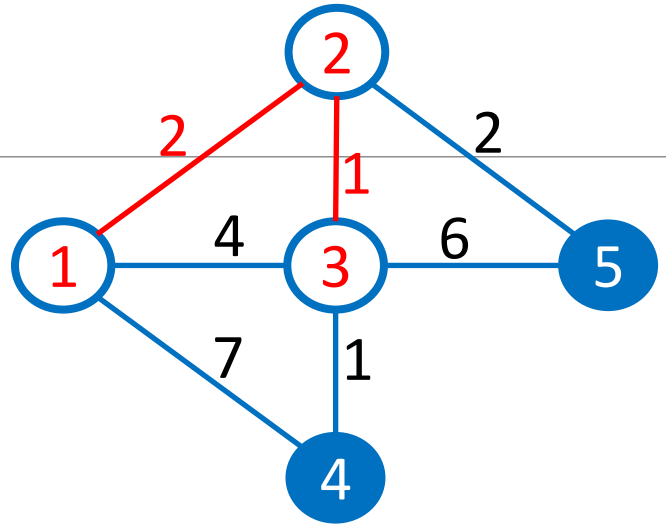
min[4]=w(3,4)=1; min[5]=w(3,5)=6>2, 所以不修改5的边权. min[4]最小. 将4变为白点, 并标记(3,4)边。

节点	1	2	3	4	5
最小边权	0	2	1	1	2



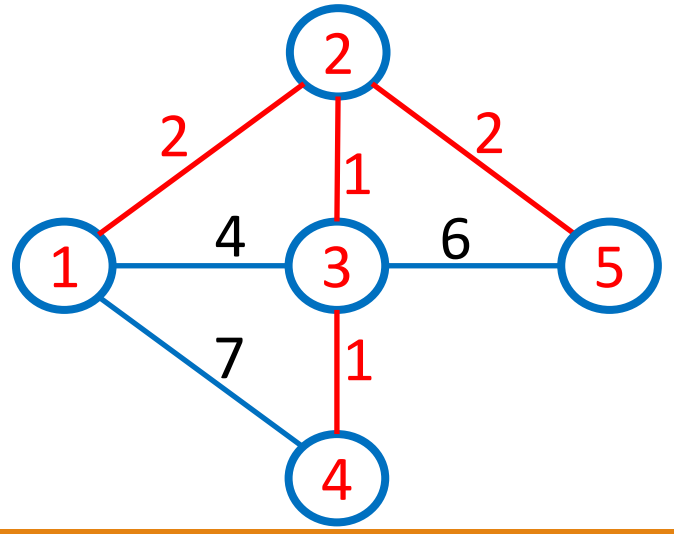
Prim算法

节点	1	2	3	4	5
最小边权	0	2	1	1	2



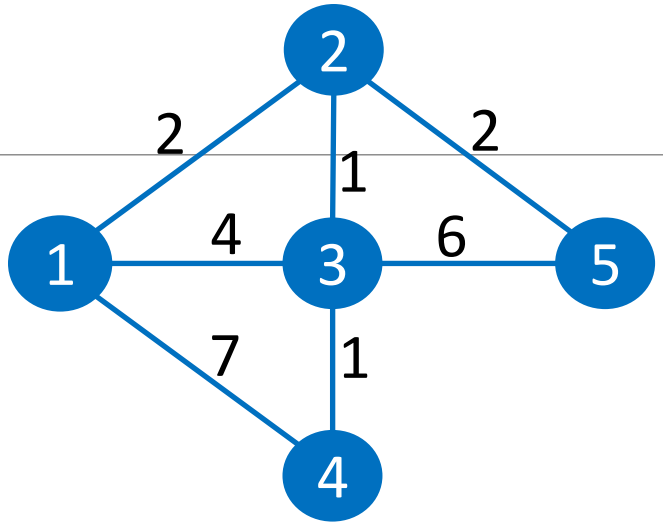
- 接着枚举与4相连的所有蓝点并修改它们与白点相连的最小边权.4没有相连的蓝点, 不需修改边权. 此时，唯一剩余的蓝点5，边权为w(2,5)=2, 将5变为白点, 并标记(2,5)边。

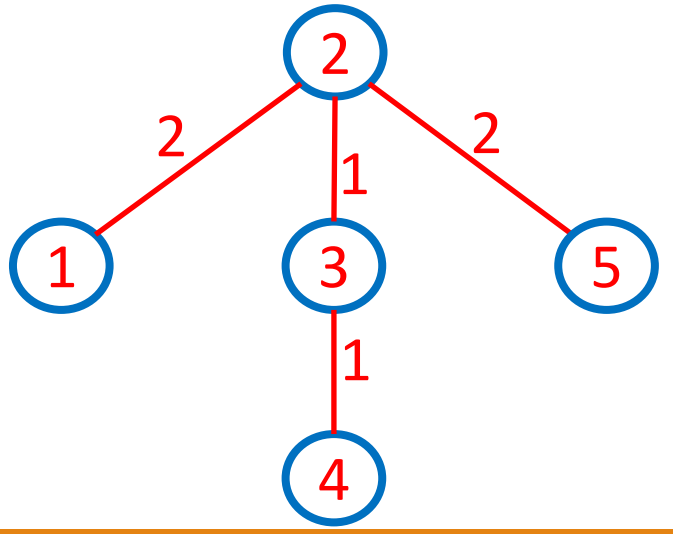
节点	1	2	3	4	5
最小边权	0	2	1	1	2



Prim算法

- 右下图为Prim算法生成的最小生成树。
- n个顶点的图的最小生成树有n-1条边。
- 权值之和MST=6。
- 从所有不在S中的点中，选一个距离u最近的点，时间复杂度为O(n)；选取点后，更新相关边权重，时间复杂度为O(n)。要对n-1个结点进行以上操作，所以总时间复杂度为$O(n^2)$。
- 使用不同的数据结构，Prim算法有和Dikjstra算法相同的表现。





Prim算法

$T=O(V)\cdot T_{EXTRACT\text{-}MIN}+O(E)\cdot T_{DECREASE\text{-}KEY}$
$=O(VlgV+ElgV)$
$=O(ElgV)$ (二叉堆)

```
MST-PRIM(G, w, r)
1.  for each u∈G.V
2.      u:key=∞
3.      u:π=NIL
4.  s:key=0
5.  Q=G.V
6.  while Q ≠ ϕ
7.      u=extract_min(Q)
8.      for each v∈G.Adj[u]
9.      if v∈Q and w(u, v)<v.key
10.         v.π=u
11.         v.key=w(u, v)
```

O(V)

O(VlgV)

O(lgV)

在while循环的每遍循环之前，我们有：

1. A={(v, v.π)|v∈V-{r}-Q}
2. 已经加入到最小生成树的结点为集合V-Q
3. 对于所有的结点v∈Q，如果v.π ≠ NIL, 则v.key<∞并且v.key是连接结点v和最小生成树中某个结点的轻边(v, v.π)的权重。

//找出连接V-Q和Q的轻边的一个端点u, 接着将u从Q中删除, 并加入V-Q中, 即(v, v.π)→A

//将每个与u邻接但却不在树中的结点更新

总执行次数为O(E)

//DECREASE-KEY操作

Prim算法的时间复杂度

- $T=O(V)\cdot T_{EXTRACT\text{-}MIN}+O(E)\cdot T_{DECREASE\text{-}KEY}$

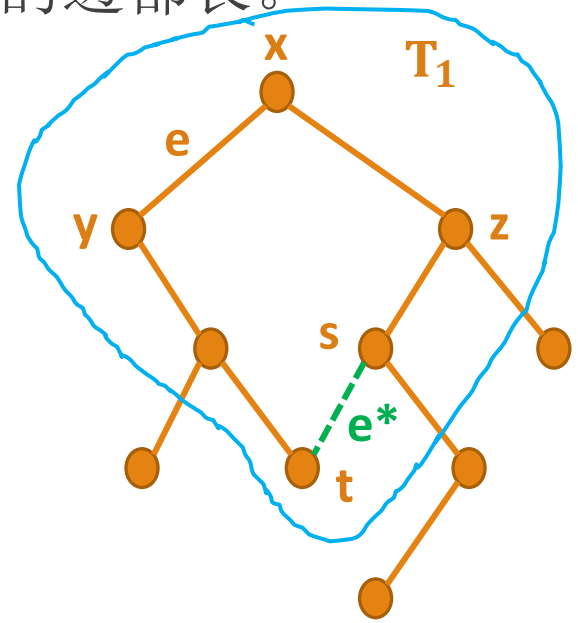
Q	$T_{EXTRACT\text{-}MIN}$	$T_{DECREASE\text{-}KEY}$	Total
array	O(V)	O(1)	$O(V^2)$
binary heap	O(lgV)	O(lgV)	O(ElgV)
Fibonacci heap	O(lgV)	O(1)	O(E+VlgV)
	amortized	amortized	worst case

Prim算法的正确性

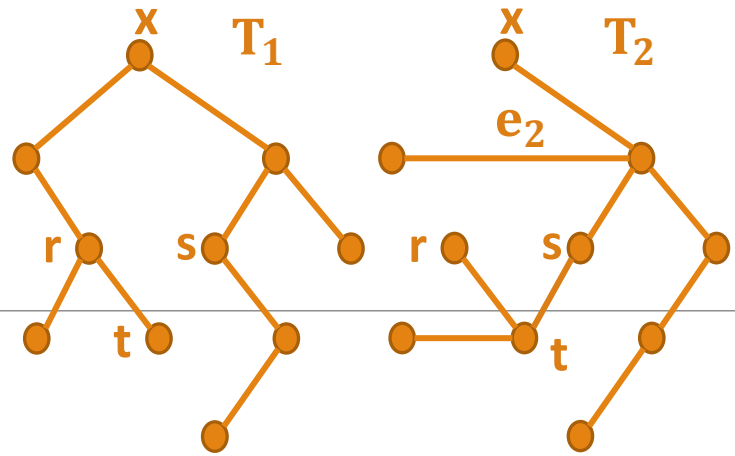
引理 对于Prim算法生成的MST记为T_1，e*是图G中任意一条不属于T_1的边。将e*添入T_1后形成回路，则e*比回路中任意一条属于T_1的边都长。

证明 反证法。

- 假设T_1中存在至少一条边e’，s.t. w(e’)>w(e*)。
- 令e=xy ∈ T_1为环路中最长的边，有w(e)≥w(e’)>w(e*)。
- 不妨设x先于y被添加到T_1中。
- 由于e是环路中最长的边，根据Prim算法，在x被加入T_1后，会优先选择环路中e以外的边加入T_1。那么路径x, z, s, t, …, y会被选进T_1。而边e不会被选入T_1。
- 这与e为T_1中的边矛盾，则假设错误，原命题成立。



Prim算法的正确性

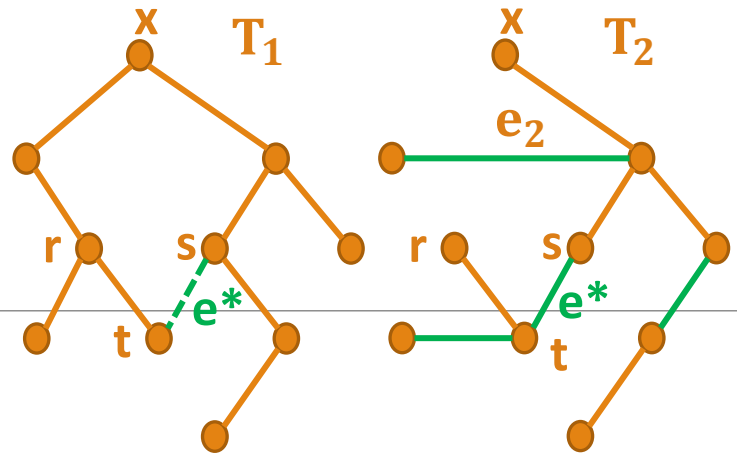


定理 Prim算法生成的MST，记为T_1，是最小生成树。

证明 反证法。假设T_1不是最小生成树。

- 假设存在T_2，为与T_1具有相同边数最多的实际MST，则$w(T_2) < w(T_1)$。
- 假定e*=st为在T_2中且不在T_1中的权值最小的边。
- 将e*加入T_1中，形成环路。
- 根据引理，则环路中任意一条T_1边的权值都比e*小。

Prim算法的正确性

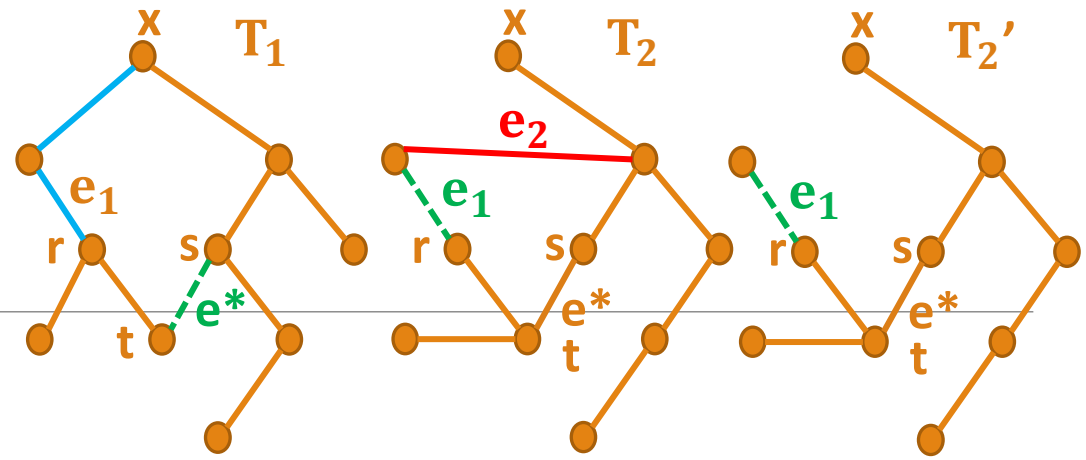


定理 Prim算法生成的MST，记为T_1，是最小生成树。

证明 反证法。假设T_1不是最小生成树。

- 假设存在T_2，为与T_1具有相同边数最多的实际MST，则$w(T_2) < w(T_1)$。
- 假定e*=st为在T_2中且不在T_1中的权值最小的边。
- 将e*加入T_1中，形成环路。
- 根据引理，则环路中任意一条T_1边的权值都比e*小。

Prim算法的正确性



定理 Prim算法生成的MST，记为T_1，是最小生成树。

证明 反证法。

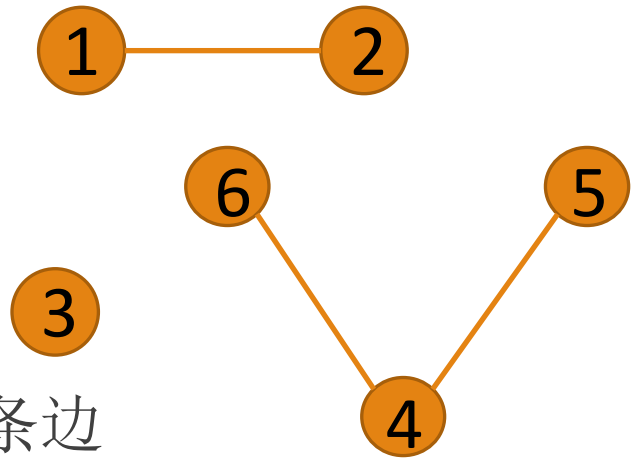
- 假设存在T_2，为与T_1具有相同边数最多的实际MST，则 $w(T_2) < w(T_1)$。
- 假定e*=st为在T_2中且不在T_1中的权值最小的边。将e*加入T_1中，形成环路。
- 根据引理，则环路中任意一条T_1边的权值都比e*小。
- 设e_1是在环路上且不在T_2中，权值最小的边。将e_1加入T_2中构成回路。在回路中找一条在T_2中但不在T_1中的边e_2并删除之，得T_2'。
- 若:
  ① $w(e_1)<w(e_2)$,则$w(T_2')<w(T_2)$,与T_2是一棵最小生成树矛盾;
  ② $w(e_1)\geq w(e_2)$, 因为$w(e*)>w(e_1)$, 与e*是在T_2中不在T_1中的最小权值的边矛盾。
- 所以，Prim算法执行过程是正确的。

Joseph Bernard Kruskal, Jr.

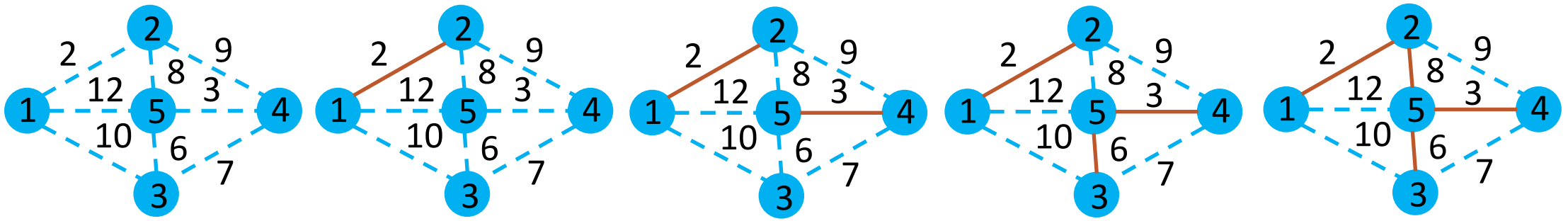
- Joseph Bernard Kruskal, Jr. (1928-2010), 美国数学家，统计学家，计算机科学家，心理测量专家。
- 1948年于芝加哥大学获得工程数学学士学位。
- 1949年于芝加哥大学获得工程数学硕士学位。
- 1954年于普林斯顿大学获得博士学位。
- 1956年发表Kruskal算法。

Kruskal算法

- 首先我们把无向图中相互连通的一些点称为处于同一个连通块中。如右图，由3个连通块，1, 2处于一个连通块中，4, 5, 6处于一个连通块中，孤立点3叶称为一个连通块。



- Kruskal算法将一个连通块当作一个集合。
- 首先将所有的边按权重从小到大顺序排列。
- 按顺序枚举每一条边：
  - 如果这条边连接着两个不同的集合，那么就把这条边加入最小生成树，这两个不同的集合就合并成了一个集合；
  - 如果这条边连接的两个点属于同一集合，就跳过。
- 直到选取了n-1条边为止。



- 算法开始时，认为每一个点都是孤立的，分属于n个独立的集合。
- 每次选择一条最小的边。而且这条边的两个顶点分属于两个不同的集合。将选取的这条边加入最小生成树，并且合并集合。

1. 选择边(1, 2)，将这条边加入到生成树中，并且将它的两个顶点1, 2合并成一个集合。现在有4个集合{(1, 2), {3}, {4}, {5}}, 生成树中有一条边(1, 2).
2. 选择边(4, 5)，将其加入到生成树中，并且将它的两个顶点4, 5合并成一个集合。现在有3个集合{{1, 2}, {3}, {4, 5}}, 生成树中有两条边(1,2), (4,5).
3. 选择边(3, 5}, 将这条边加入到生成树中，并且将它的两个顶点3, 5所在的两个集合合并成一个集合。现有2个集合{{1, 2},{3, 4, 5}}, 生成树中有3条边{(1, 2), (4, 5), (3, 5)}.
4. 选择边(2,5), 将这条边加入到生成树中，并且将它的两个顶点2, 5所在的两个集合合并成一个集合。现有1个集合{{1, 2, 3, 4, 5}}, 生成树中有4条边{(1, 2), (4, 5), (3, 5), (2,5)}.
5. 算法结束，最小生成树权值为19.

Kruskal算法

MST-KRUSKAL (G, w)

1. A=ϕ //将集合A初始化为一个空集合
2. **for** each vertex v∈G.V
3. MAKE-SET(v) //创建|V|棵树, 每棵树仅包含一个结点
4. sort the edges of G. E into nondecreasing order by weight w
5. **for** each edge (u, v)∈G. E, taken in nondecreasing order by weight
6. **if** FIND-SET(u)≠FIND-SET(v) //检查(u,v)的端点是否属于同一棵树
7. A=A∪{(u, v)} //将边(u,v)加入集合A中
8. UNION(u, v) //将两棵树的结点进行合并
9. **return** A

Kruskal算法

```
MST-KRUSKAL (G, w)
1.  A=ϕ                          //将
2.  for each vertex v∈G.
3.      MAKE-SET(v)  //创
4.  sort the edges of G.
5.  for each edge (u, v)∈
6.      if FIND-SET(u)≠FIN
7.          A=AU{(u, v)}
8.          UNION(u, v)
9.  return A
```

如果给出的边是无序的，则Kruskal的算法需要O(|E|log|V|)的时间来对边进行排序，而查找的时间复杂性也是O(|E|log|V|)。由于排序的时间复杂性已经处于最低状态，对查找的任何改善都将无济于事。所以，Kruskal算法的时间复杂性就是O(|E|log|V|)。

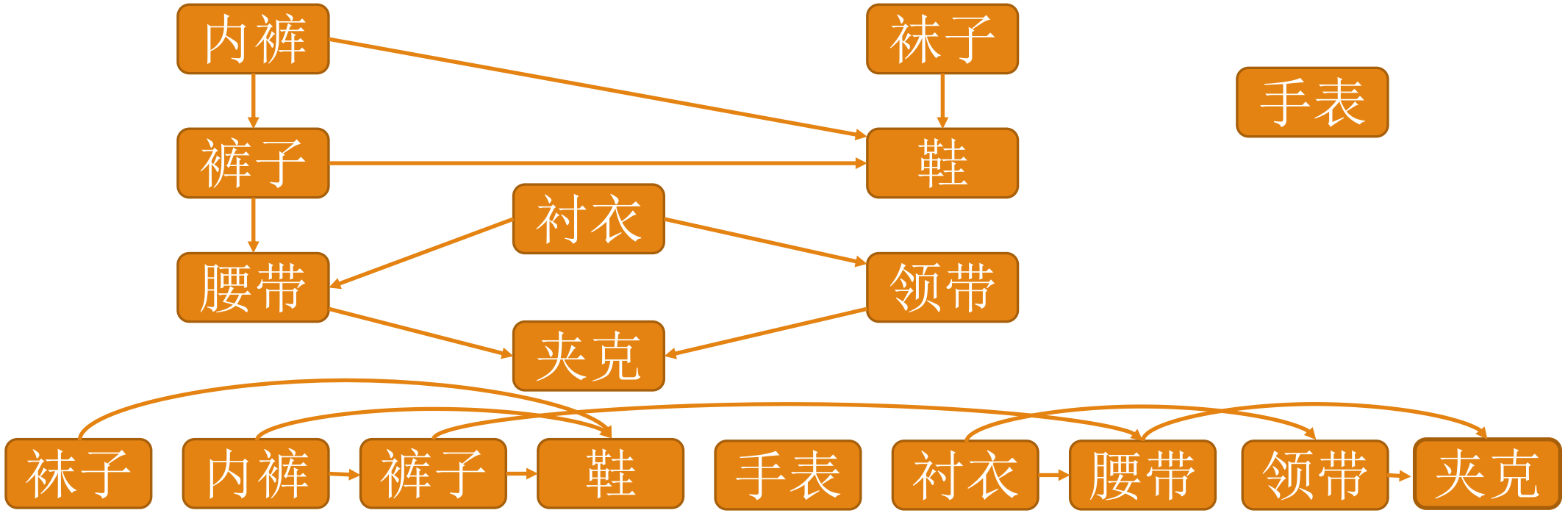
如果给出的边是按权重顺序排列的或者权重的值都很小，则线性时间内可以完成排序。这样，查找的时间复杂性O(|E|log|V|)将超过排序的时间复杂性O(|E|)，查找和合并成为算法的瓶颈。

Kruskal算法的正确性

- MST属性：设T为图G=(V, E)的最小生成树，并且A⊆V是图G中的任意大小的结点集合。假设(u, v)∈E是连接A和V-A的所有边里面权重最小的边，则(u, v)∈E(已证)。
- 在任意时刻，已经挑选出来的边形成一个部分的解决方案，即一组连通的组件，每个组件本身具有树的结构。下一条加进来的边e将这些组件中的两个连接起来，这里称T_1和T_2。由于e是不产生环路的权重最小的边，毫无疑问，它是连接T_1和V- T_1之间的权重最小的边，因此满足MST属性。
- 因此，这样加上去的边都属于某棵最小生成树的边。因此，当加入的边数达到n-1时，形成的必然是一棵最小生成树。

拓扑排序

- 生活中很多事情发生是有先后次序的。下图是Kruskal教授每天早上起床穿衣所发生的事件的次序图。



拓扑排序——AOV网

- 在日常生活中，一项大的工程可以看作是由若干个子工程(这些子工程称为“活动”)组成的集合，这些子工程(活动)之间必定存在一些先后关系，即某些子工程(活动)必须在其他一些子工程(活动)完成之后才能开始，我们可以用有向图来形象地表示这些子工程(活动)之间的先后关系，子工程(活动)为顶点，子工程(活动)之间的先后关系为有向边，这种有向图称为“**顶点活动网络**”，又称“**AOV**”网。
- 在AOV网中，把一条有向边起点的活动称为终点活动的**前驱活动**，同理终点的活动称为起点活动的**后继活动**。
- 只有当一个活动全部的前驱全部都完成之后，这个活动才能进行。

拓扑排序

- 拓扑排序只适用于有向无环图。如果图G包含环路，则不可能排出一个线性次序。
- 对于一个有向无环图G=(V, E)来说，其拓扑排序是G中所有结点的一种线性次序，该次序满足如下条件：
  - 如果图G包含边(u, v)，则结点u在拓扑排序中处于结点v的前面。
- 可以将图的拓扑排序看做是将图的所有结点在一条水平线上排开，图的所有有向边都从左指向右。

拓扑排序的贪心算法

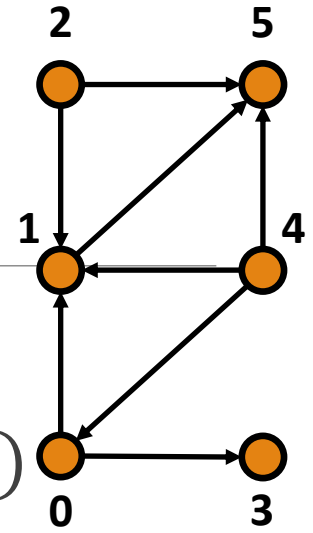
- 从空集开始，每步产生拓扑排序序列中的一个顶点v。顶点v需要满足贪心策略。
- 贪心策略：从当前尚不在拓扑排序序列的顶点中选择一顶点v,其所有前驱节点u都在已产生的拓扑序列中(或无前驱顶点)，并将v加入到拓扑序列中.
- 用减节点入度的方法确定v:入度变成0的顶点为要加到拓扑序列中的顶点。

Kahn算法

1. 计算每个顶点的入度
2. 将入度为0的顶点入栈
3. While(栈不空) {
   1) 任取一入度为0的顶点放入拓扑序列中;
   2) 将与其相邻的顶点的入度减1;
   3) 如有新的入度为0的顶点出现,将其放入栈中; }
4. 如有剩余的顶点未被删除，说明该图有环路。

- Kahn算法的时间复杂度是O(n+e), 每个顶点要入栈一次，每条边要扫描一次。

拓扑排序

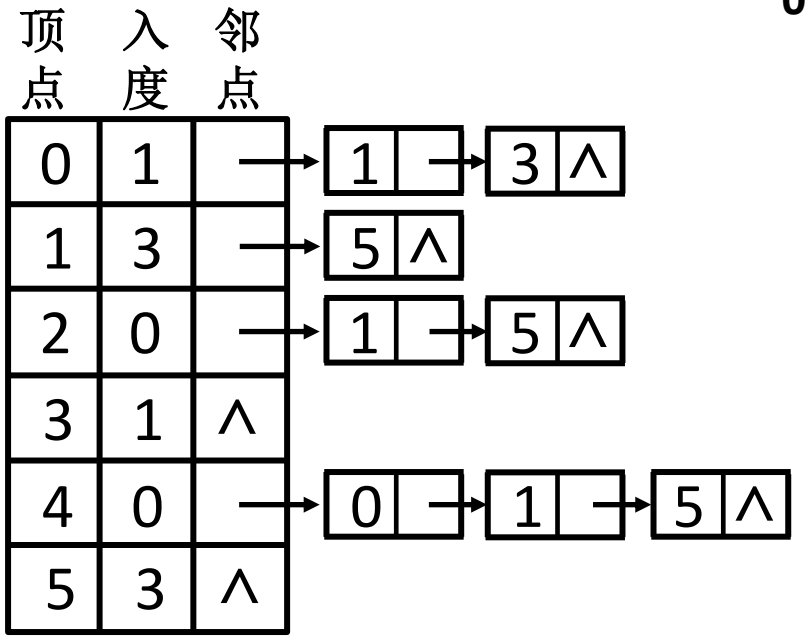


对于图G(V, E)，V集合包含n个顶点，E集合包含e条边。

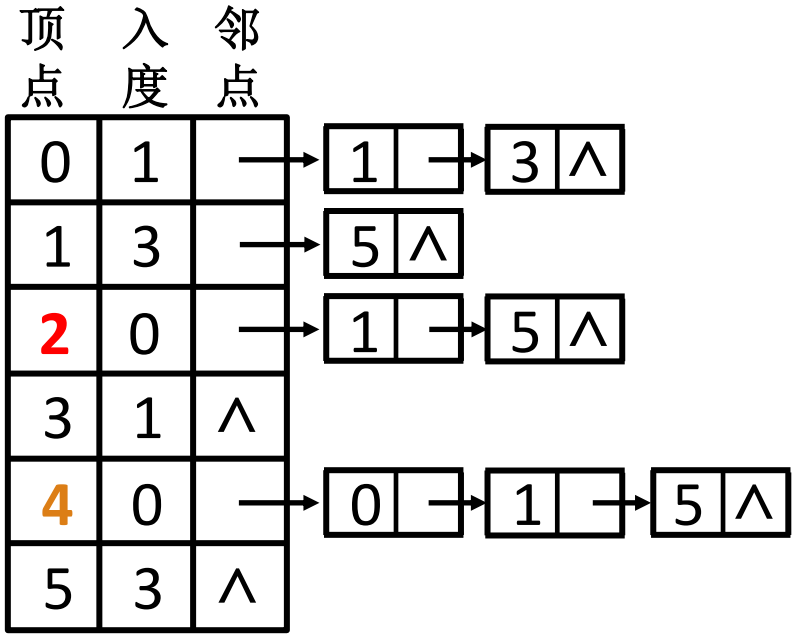
- 邻接矩阵存储——$O(n+n^2)$

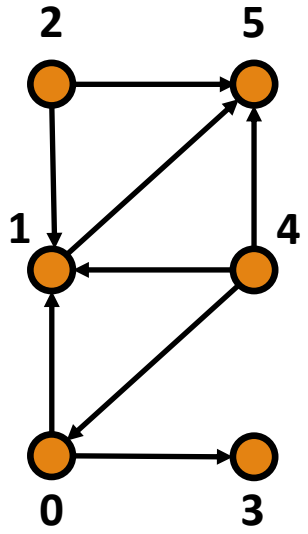
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

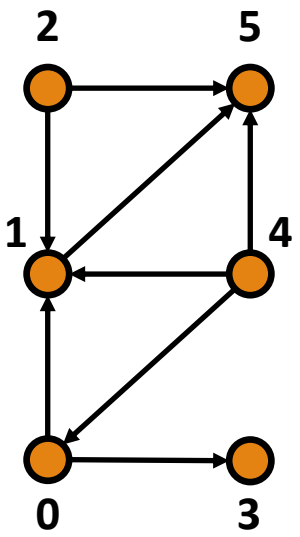
- 邻接表存储——$O(n+e)$



拓扑排序：2

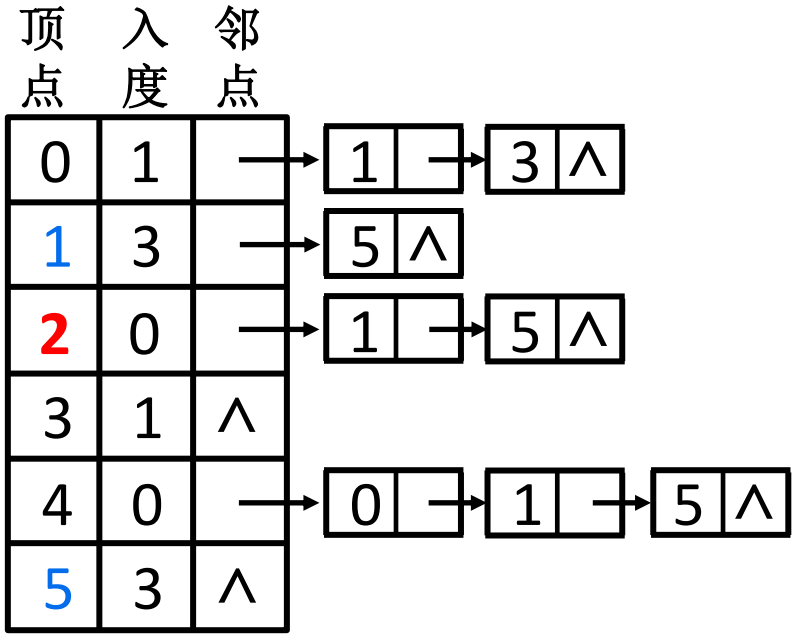


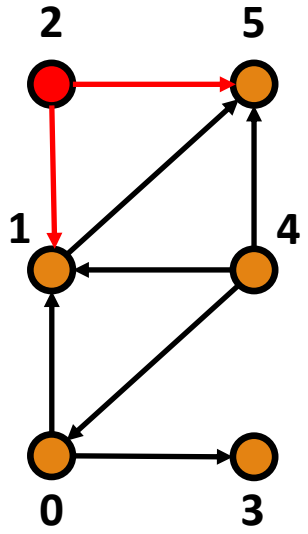


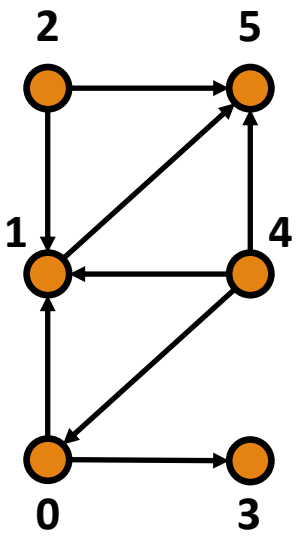


- 有2个入度为零的点，不妨按编号顺序选择其中一个。选定2 进入拓扑序列。(也可以按出边多少顺序)

拓扑排序:2

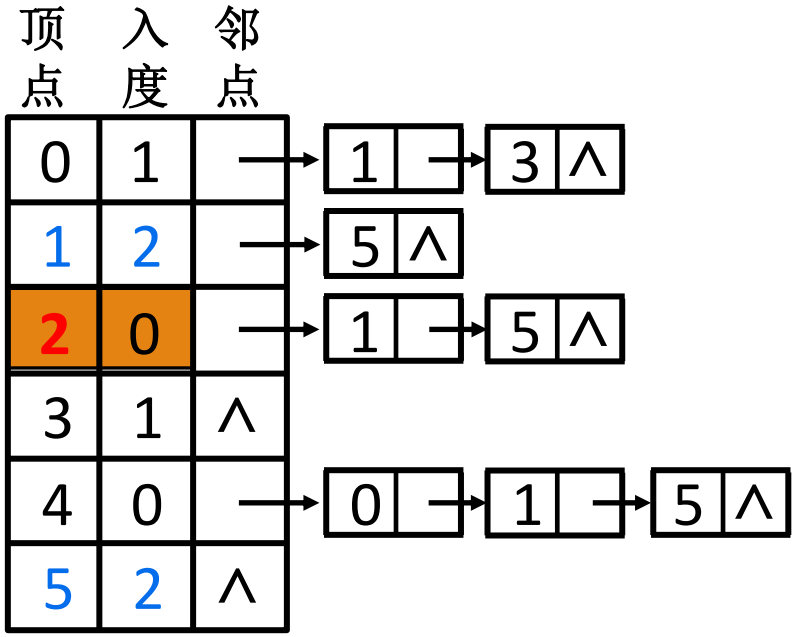


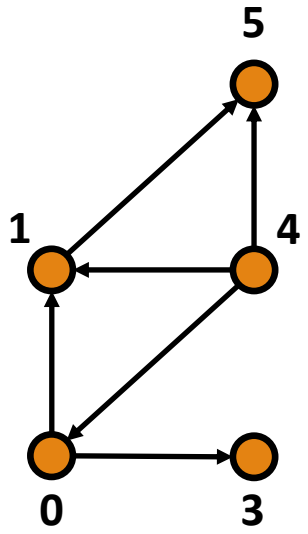


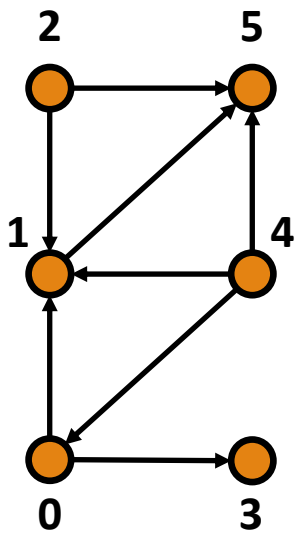


- 将顶点2的邻点的入度减1，并删除点2及其邻边。

拓扑排序：2

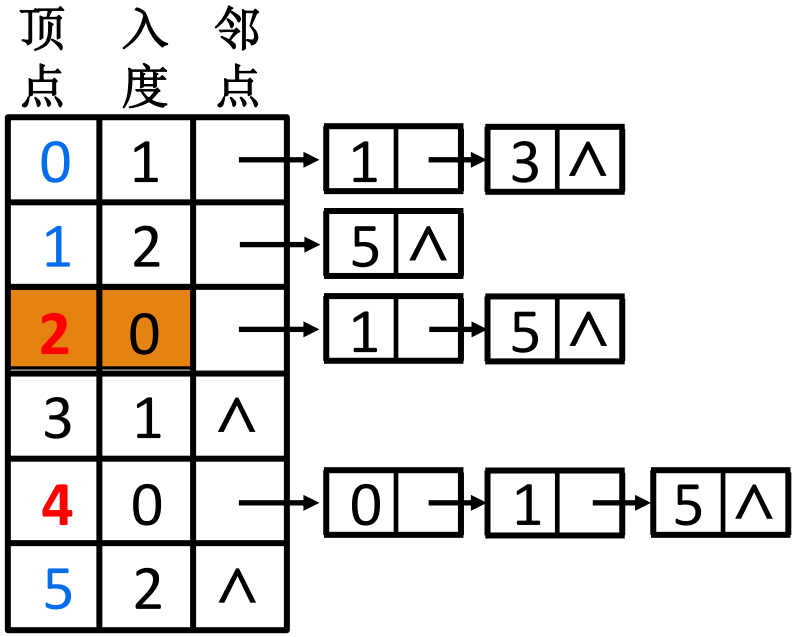


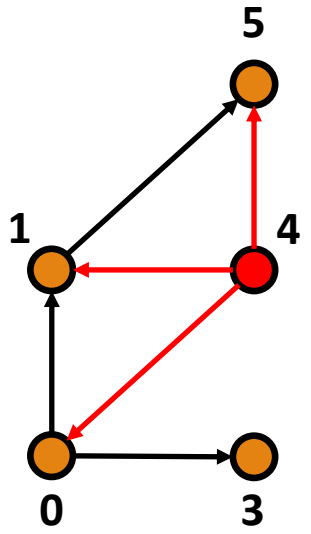


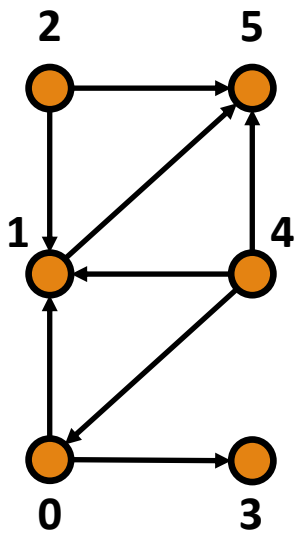


- 将顶点2的邻点的入度减1，并删除点2及其邻边。

拓扑排序：2、4

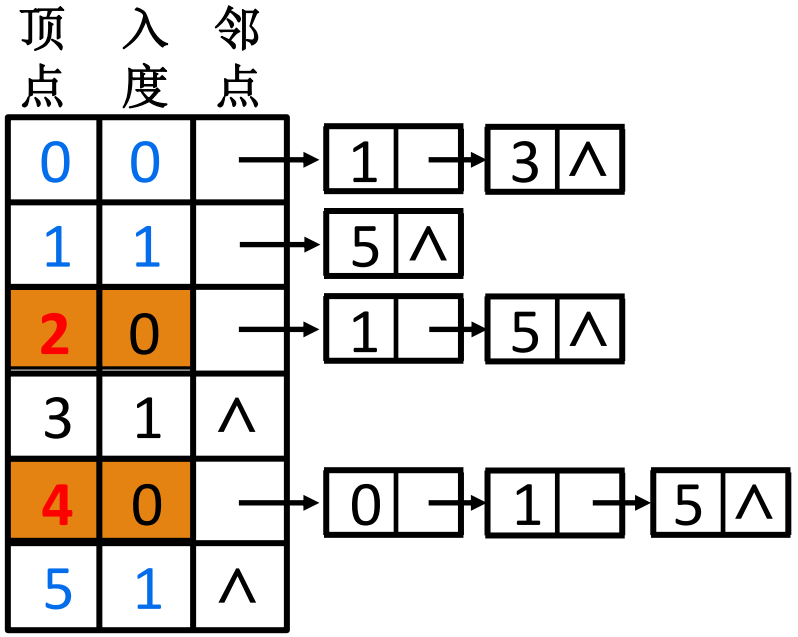


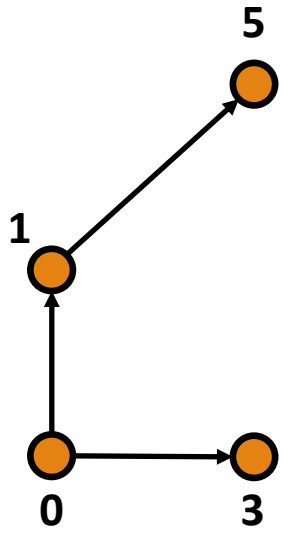


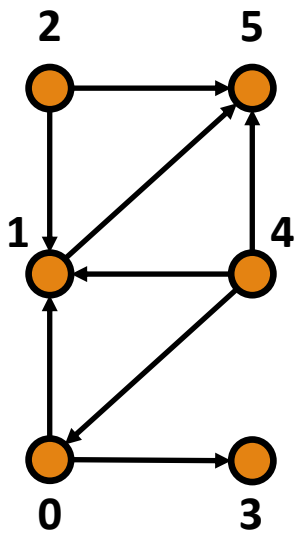


- 选择入度为0的顶点4进入拓扑序列。将顶点4的邻点的入度减1，并删除点4及其邻边。

拓扑排序：2、4

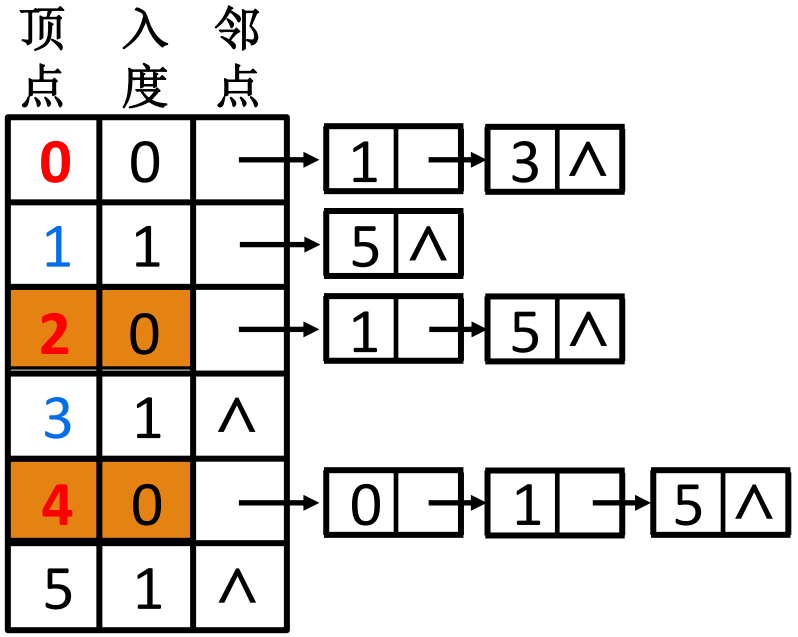


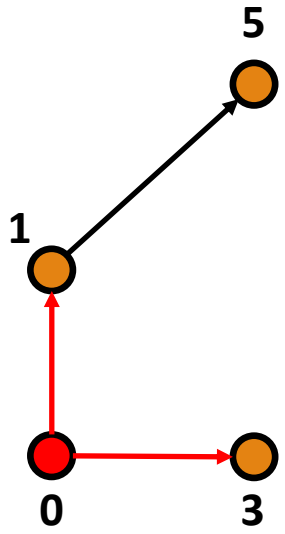


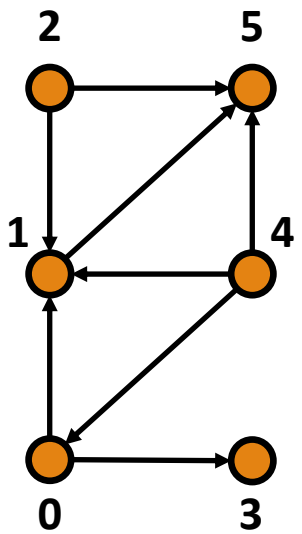


- 选择入度为0的顶点4进入拓扑序列。将顶点4的邻点的入度减1，并删除点4及其邻边。

拓扑排序：2、4、0

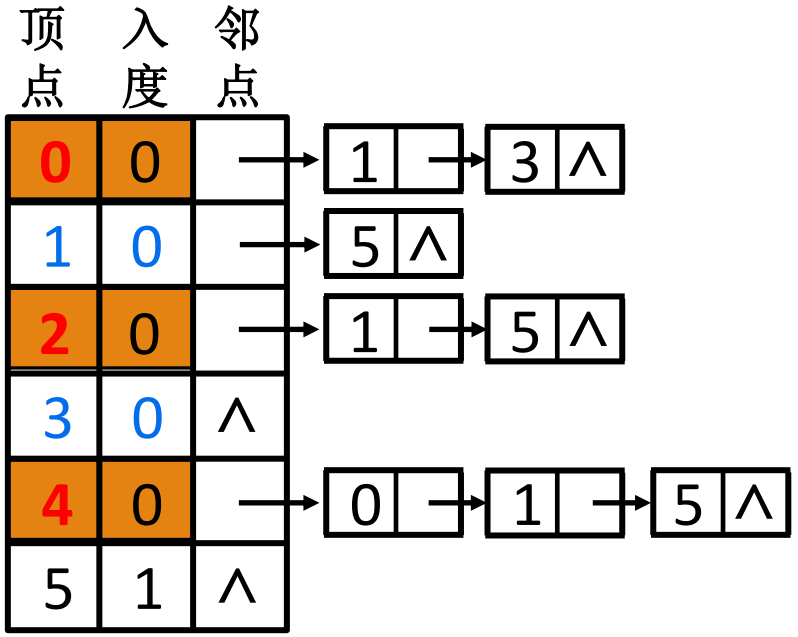


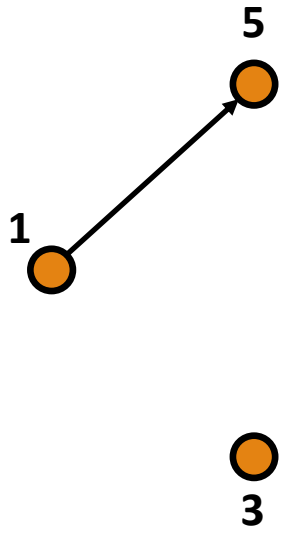


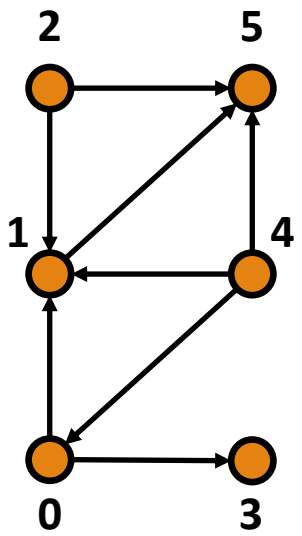


- 选择入度为0的顶点0进入拓扑序列。将顶点0的邻点的入度减1，并删除点0及其的邻边。

拓扑排序：2、4、0



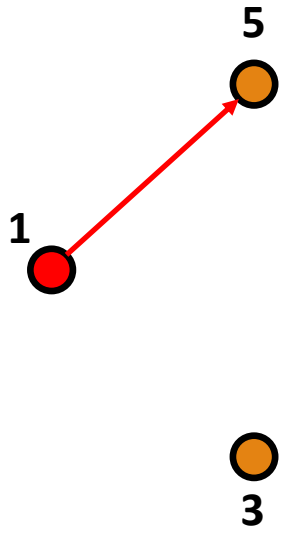


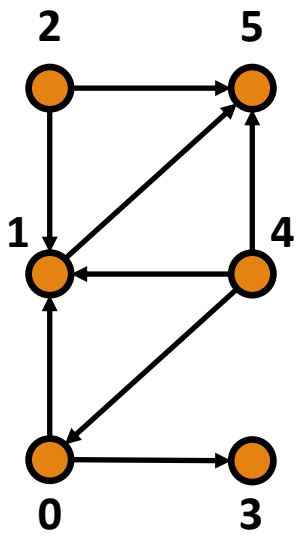


- 选择入度为0的顶点0进入拓扑序列。将顶点0的邻点的入度减1，并删除点0及其的邻边。

拓扑排序：2、4、0、1

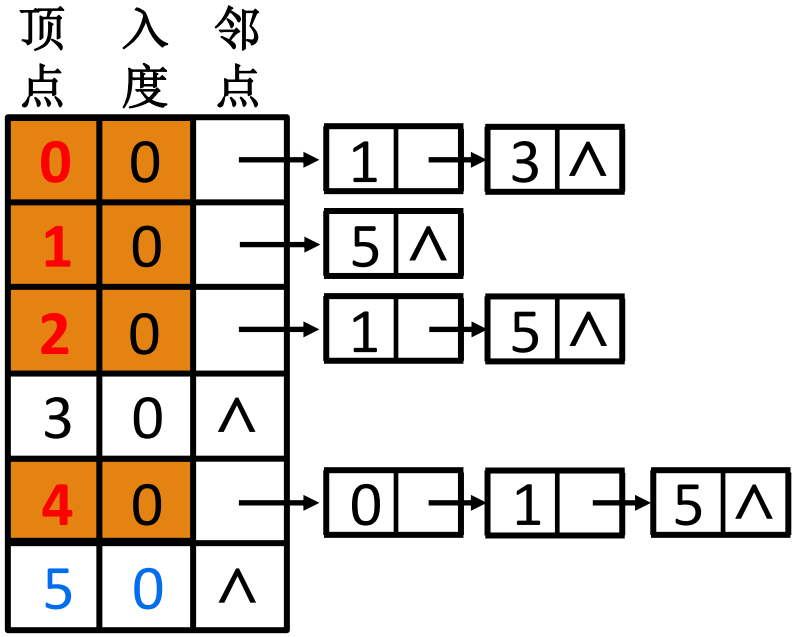
顶点	入度	邻点
0	0	1 → 3 ∧
1	0	5 ∧
2	0	1 → 5 ∧
3	0	∧
4	0	0 → 1 → 5 ∧
5	1	∧

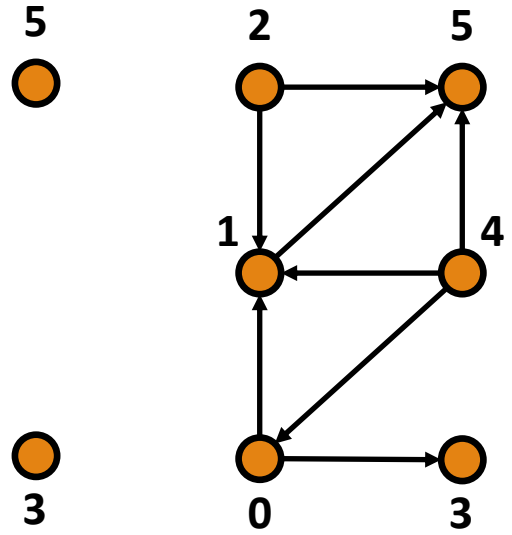




- 有2个入度为零的点，不妨按编号顺序选择其中一个。选定1进入拓扑序列。将顶点1的邻点的入度减1，并删除点1及其的邻边。

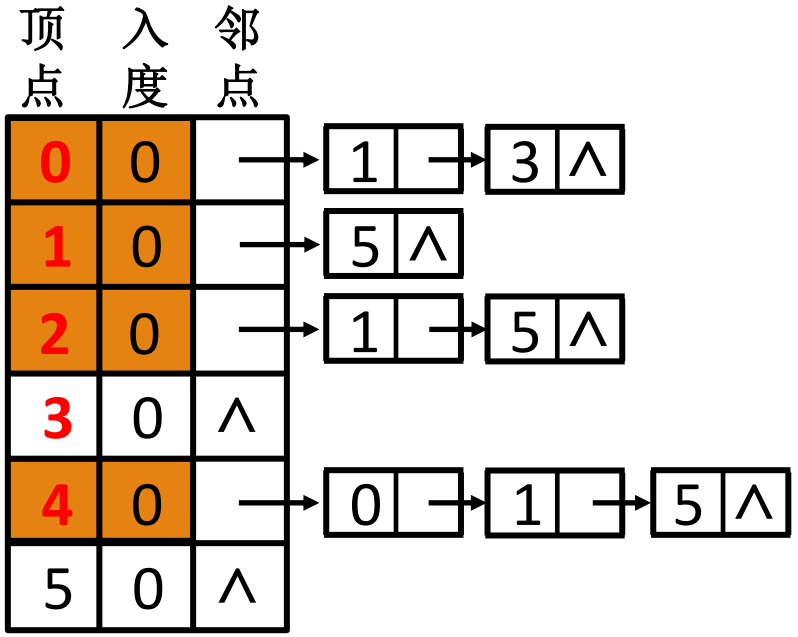
拓扑排序：2、4、0、1

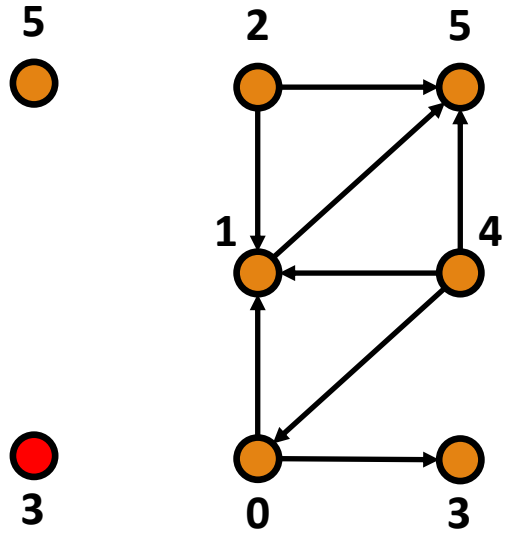




- 有2个入度为零的点，不妨按编号顺序选择其中一个。选定1进入拓扑序列。将顶点1的邻点的入度减1，并删除点1及其的邻边。

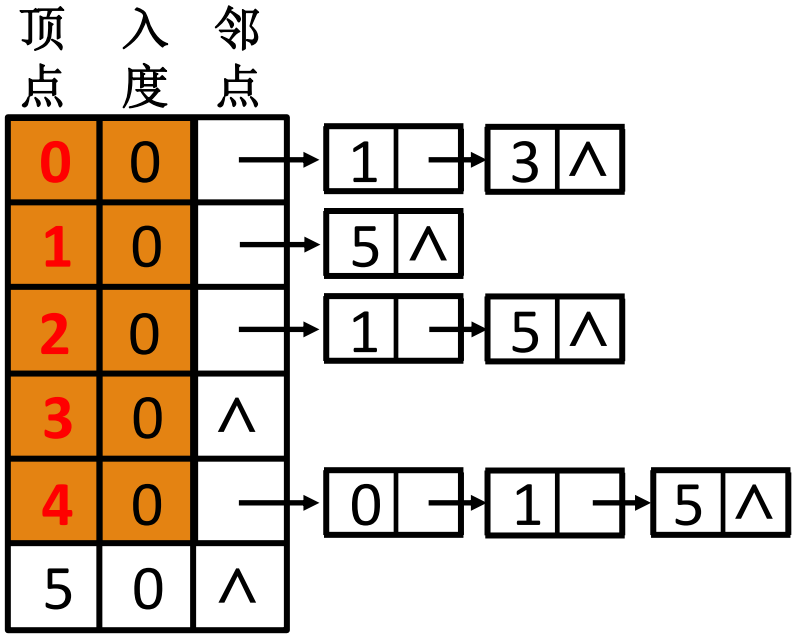
拓扑排序：2、4、0、1、3

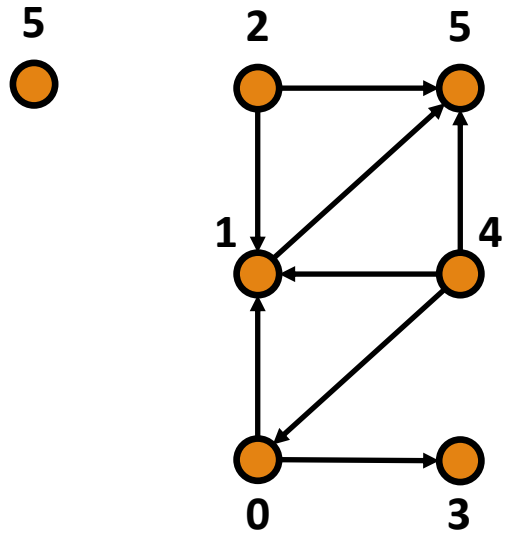




- 有2个入度为零的点，不妨按编号顺序选择其中一个。选定3进入拓扑序列。顶点3没有邻点和邻边，删除点3。

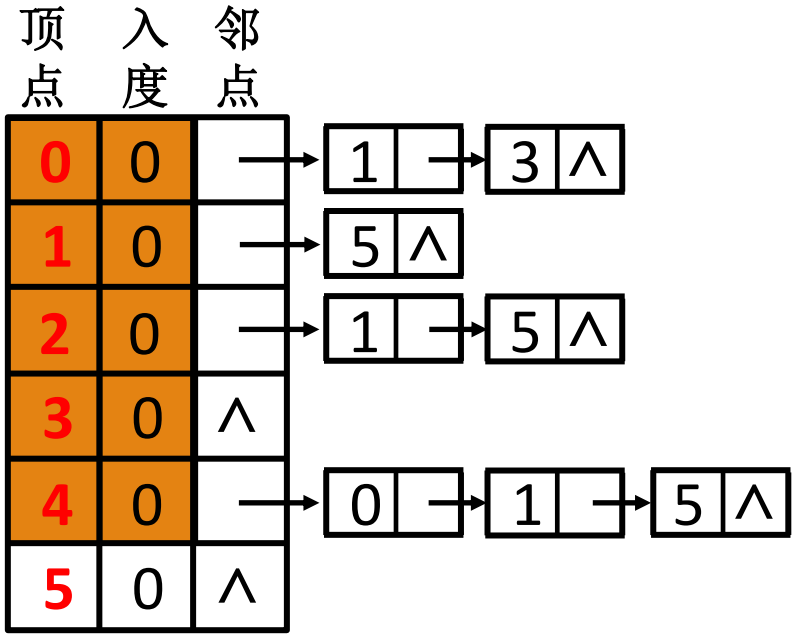
拓扑排序：2、4、0、1、3

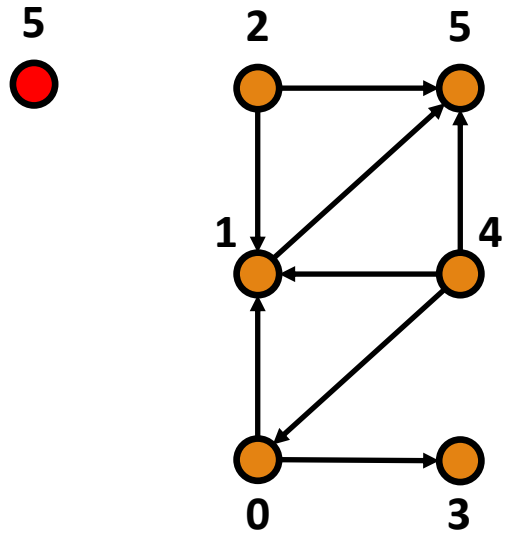




- 有2个入度为零的点，不妨按编号顺序选择其中一个。选定3进入拓扑序列。顶点3没有邻点和邻边，删除点3。

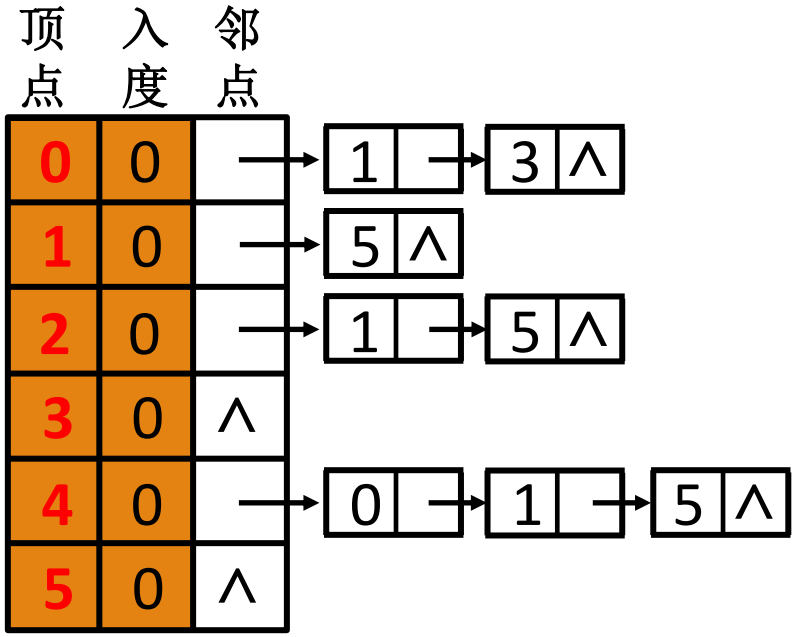
拓扑排序：2、4、0、1、3、5

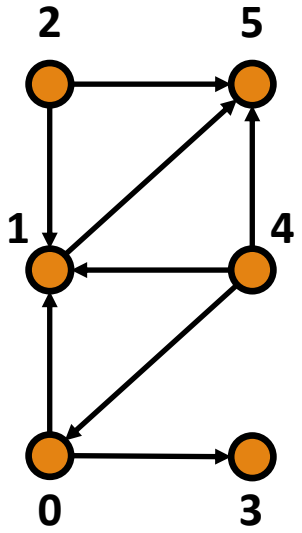




- 余下一个入度为0的顶点5。选定5进入拓扑序列。顶点5没有邻点和邻边，删除点5。

拓扑排序：2、4、0、1、3、5



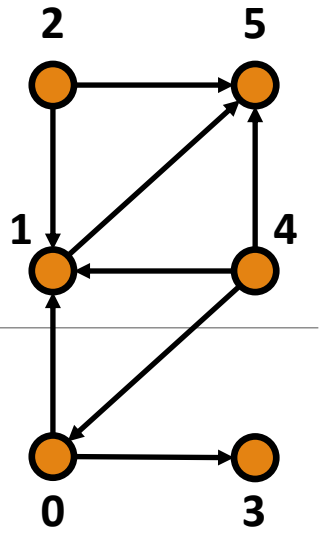


- 余下一个入度为0的顶点5。选定5进入拓扑序列。顶点5没有邻点和邻边，删除点5。
- 至此全部顶点都已进入拓扑序列，排序完毕。

如果算法失败, 则有向图含有环路

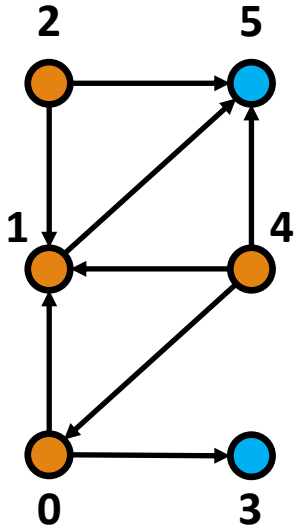
- 当算法失败时|V|<n, 且剩下的顶点不能加入已排好的序列V中.
- 若节点q_1不在V中，则q_1至少有一条入边，设为 (q_2,q_1)。q_2必不在V中, 否则q_1可加入V中.
- 同理, 有边(q_3,q_2)且q_3不在V中. 若$q_3=q_1$ 则$q_1q_2q_3$ 是有向图中的一个环路; 若$q_3\neq q_1$，则必存在q_4, (q_4,q_3)是有向图的边且q_4不在V中. 否则q_3应在V中. 若q_4为q_1,q_2,q_3 中的任何一个, 则该有向图含有环……
- 因为有向图有有限个节点,重复上述步骤,一定能找到一个环路.

拓扑排序的DFS解法

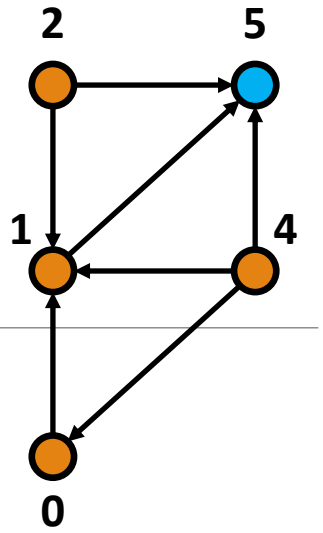


待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。

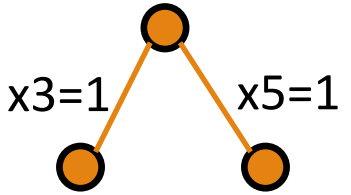


拓扑排序的DFS解法

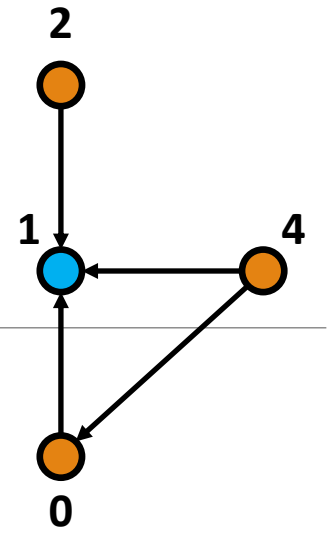


待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。
- Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。

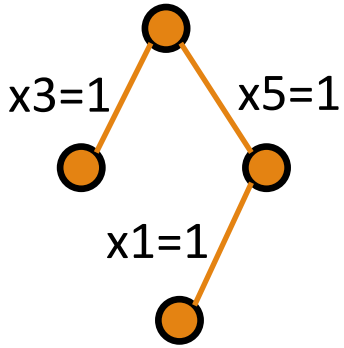


拓扑排序的DFS解法

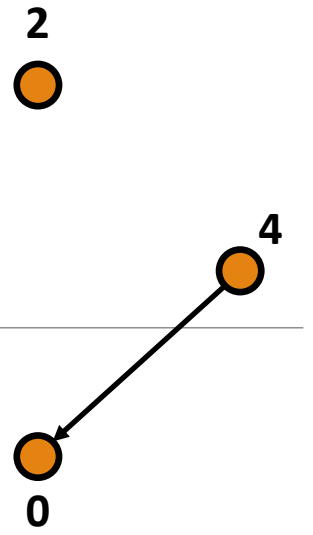


待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。
- Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。从Q中删除5，并删除5的入边。
- 5在Q中的邻点有3个，从1开始下一步查找。

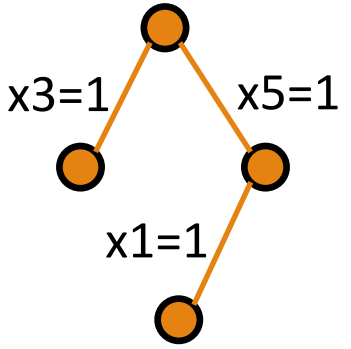


拓扑排序的DFS解法



待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

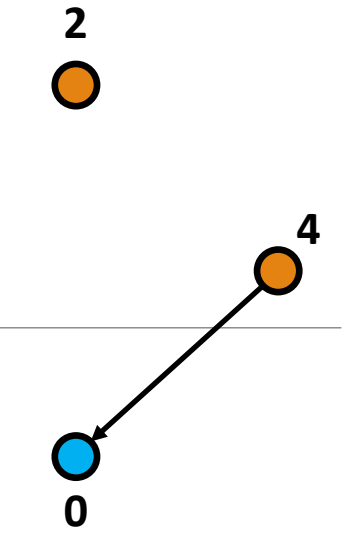
- Q中出度为零的点为3和5。不妨先从3开始。
- Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。从Q中删除5，并删除5的入边。
- 5在Q中的邻点有3个，从1开始下一步查找。
- 从Q中删除1，并删除1的入边。1的邻点有3个，0、2、4。

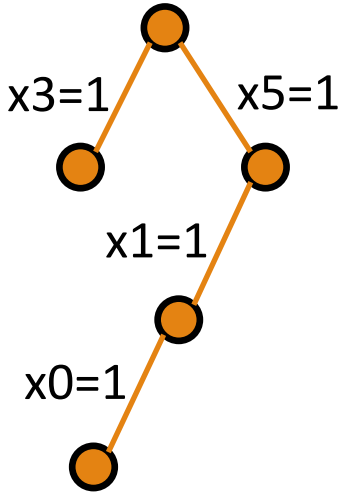


拓扑排序的DFS解法

待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。
- Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。从Q中删除5，并删除5的入边。
- 5在Q中的邻点有3个，从1开始下一步查找。
- 从Q中删除1，并删除1的入边。1的邻点有3个，0、2、4。
- 0的出度为0，从Q中删除0，并删除0的入边。

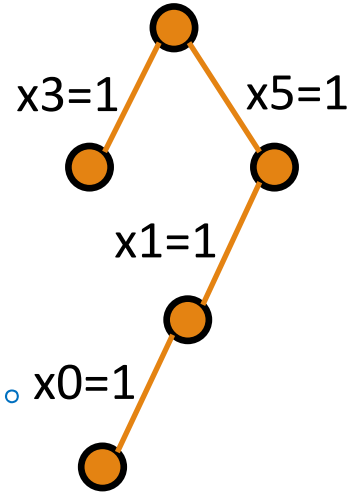




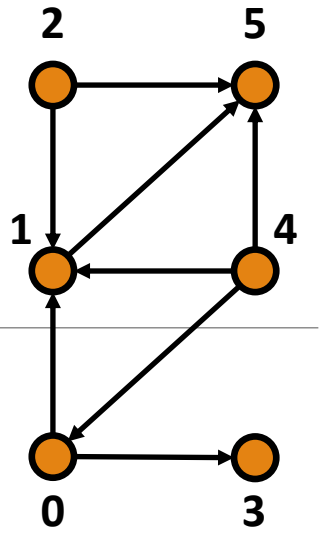
拓扑排序的DFS解法

待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。
- Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。从Q中删除5，并删除5的入边。
- 5在Q中的邻点有3个，从1开始下一步查找。
- 从Q中删除1，并删除1的入边。1的邻点有3个，0、2、4。
- 0的出度为0，从Q中删除0，并删除0的入边。
- 0的邻点4没有出边，从Q中删除4。4没有邻边和邻点，回溯。

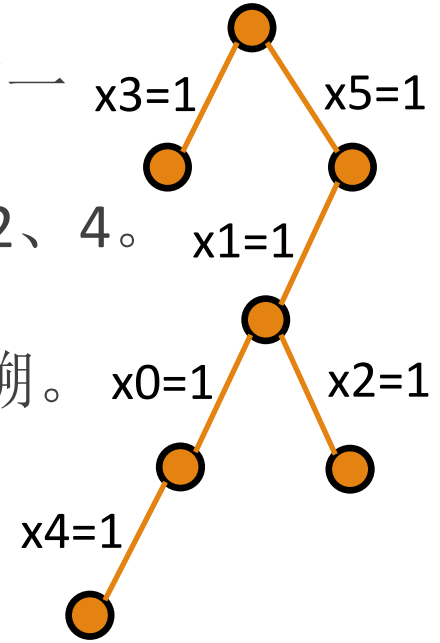


拓扑排序的DFS解法



待查点集Q={0,1,2,3,4,5}。从出度为零的点开始查找：

- Q中出度为零的点为3和5。不妨先从3开始。
- 从Q中删除3，并删除3的入边。3的邻点0的出度不为零，回溯。
- Q中出度为零的点为5。从Q中删除5，并删除5的入边。
- 5在Q中的邻点有3个，2、1、4。1的出度为零，从1开始下一步查找。
- 从Q中删除1，并删除1的入边。1在Q中的邻点有3个，0、2、4。
- 0的出度为0，从Q中删除0，并删除0的入边。
- 0的邻点4没有出边，从Q中删除4。4没有邻边和邻点，回溯。
- 1的邻点2没有出边，从Q中删除2。
- Q=∅，算法结束。

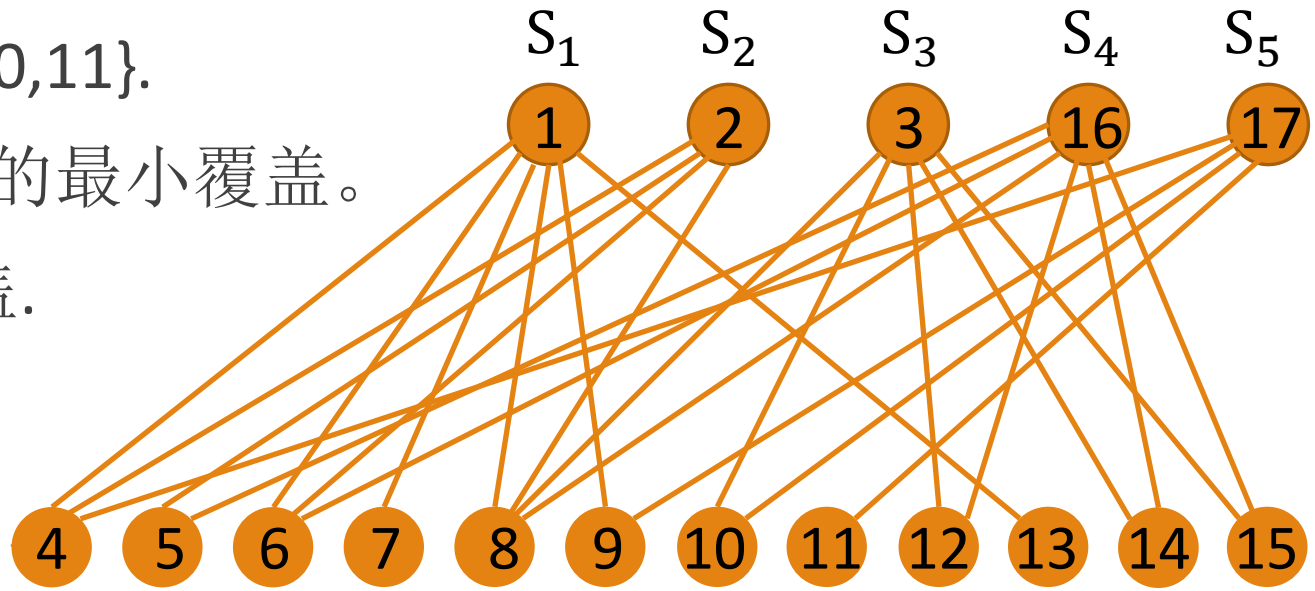


偶图覆盖问题(二分覆盖)

- 偶图是一个无向图，它的顶点分为集合A和集合B，且同一集合中的任意两个顶点无边相连.
- A的一个子集 A’ 覆盖集合B iff B中每一顶点至少和 A’ 中一顶点相连。覆盖 A’ 的大小指 A’中的顶点数目.
- 在偶图中寻找最小覆盖的问题称为偶图覆盖(bipartite-cover)问题. 偶图覆盖等价于集合覆盖问题, 是NP难问题.
- 集合覆盖问题: k个集合的族$F=\{S_1, \ldots, S_k\}$满足$U_{i=1}^{k} S_i = U$, S’为F的子族. 如果$U_{j \in S'} S_j = U$, 则称S’为U的一个覆盖，S’中集合的数目就是覆盖的大小. 目标是寻找包含的集合数目最小的覆盖.

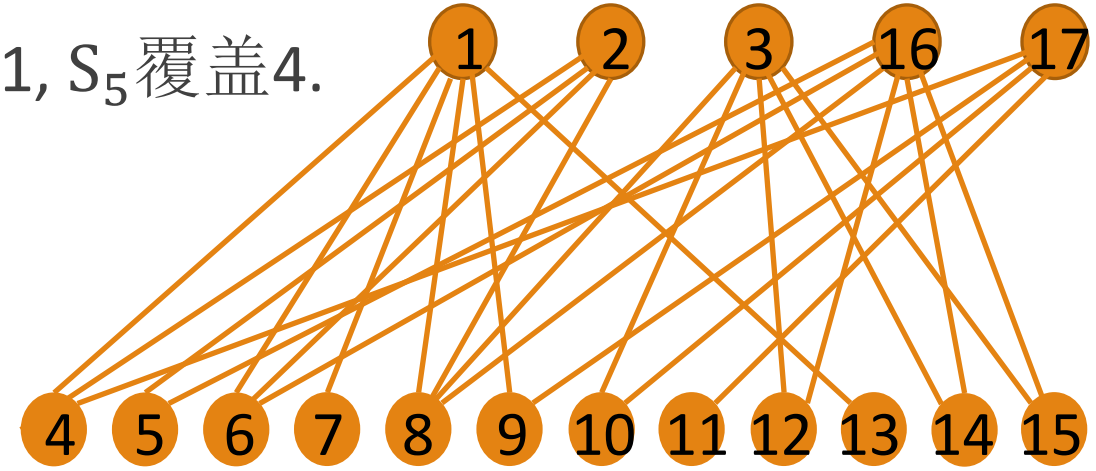
二分图最小覆盖例

- A={1,2,3,16,17}, B={4,5,6,7,8,9,10,11,12,13,14,15}=U.
- $F=\{S_1, \ldots, S_5\}$满足$U_{i=1}^{5} S_i = U$
- S_1={4,6,7,8,9,13}, S_2={4,5,6,8}, S_3={8,10,12,14,15},
  S_4={5,6,8,12,14,15}, S_5={4,9,10,11}.
- S'={S_1, S_4, S_5}是一个大小为3的最小覆盖。
- 即{1,16,17}是偶图的最小覆盖.
- NP难问题！



偶图覆盖问题的贪心解

- 贪心策略: 选择覆盖B中那些尚未被覆盖的顶点数最多的A的节点.
- S_1={4,6,7,8,9,13}, S_2={4,5,6,8}, S_3={8,10,12,14,15},
  S_4={5,6,8,12,14,15}, S_5={4,9,10,11}.
- step1: 选择S_4, 余节点{4,7,9,10,11,13};
- step2: S_1覆盖4, S_2覆盖1, S_3覆盖1, S_5覆盖4.
- step3: 选择S_1, 余节点{10,11}.
- step4: 选择S_5.
- 贪心解：{S_1, S_4, S_5}.



贪心算法

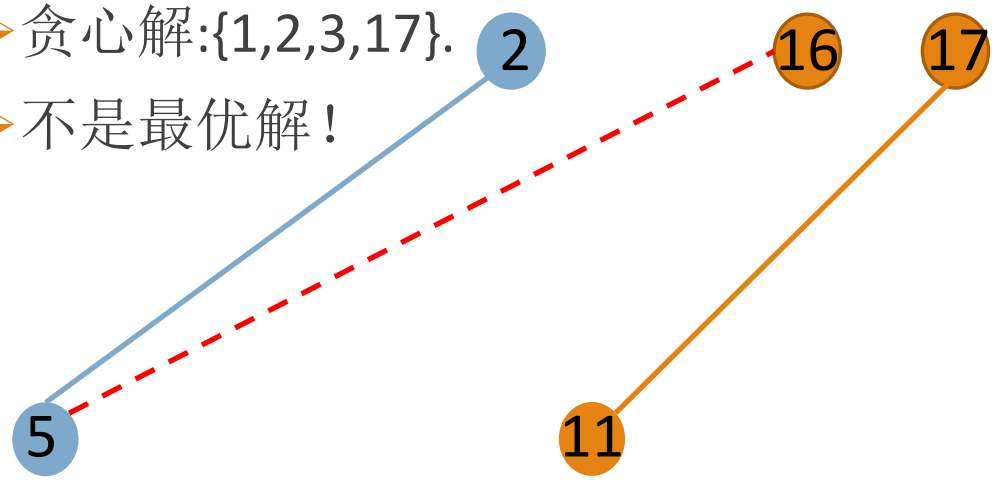
```
for all i∈A, New[i]=degree[i];
for all i∈B, covered[i]=false;
A'=∅;
while (for some i∈A, New[i]>0) {
    选取v为A-A'中New[i]值最大的节点
    A'=A'+{v};
    for 所有被v覆盖的B中结点j {
        covered[j]=true
        for 所有覆盖结点j的A中的顶点k
            New[k]=New[k]-1} }
if 有B中顶点未被覆盖  return "失败"
else 找到一个覆盖
```

- 更新New的时间为O(e)，其中e为二分图中边的数目。
- 若使用邻接矩阵，则需要花$O(n^2)$的时间来寻找图中的边
- 若用邻接链表，则需要O(n+e)的时间
- 逐步选择顶点所需时间为O(|A|) 。
- A的所有顶点都有可能被选择, 因此, 所需步骤数为$O(|A|^2)$
- 覆盖算法总的算法复杂性为$O(|A|^2+n^2)$或$O(|A|^2+n+e)$.

贪心算法

```
for all i∈A, New[i]=degree[i];
for all i∈B, covered[i]=false;
A'=∅;
while (for some i∈A, New[i]>0) {
    选取v为A-A'中New[i]值最大的节点
    A'=A'+{v};
    for 所有被v覆盖的B中结点j {
        covered[j]=true
        for 所有覆盖结点j的A中的顶点k
            New[k]=New[k]-1} }
if 有B中顶点未被覆盖  return "失败"
else 找到一个覆盖
```

- S_1={4,6,7,8,9,13}, S_2={4,5,6,8}, S_3={8,10,12,14,15}, S_4={5,6,8,12,14,15}, S_5={4,9,10,11}.
- New=[6,4,5,6,4], 选v=1(或v=16).
- A'={1}. 1覆盖B中{4,6,7,8,9,13}
- 更新: New=[0, 1, 4, 4, 2]
- New中有非0元素, 选v=3(或v=16).
- A'={1,3}. 3覆盖B中{10,12,14,15}
- 更新: New=[0,1,0,1,1], 选v=2(或16, 17)…
- 贪心解:{1,2,3,17}.
- 不是最优解！



使用最大堆

- 使用有序数组、最大堆等可将每步选择顶点的复杂性将为O(1).
- 如果利用最大堆，则每一步都需要重建堆来记录New值的变化，可以在每次New值减1时进行重建。这种减法操作可引起被减的New值最多在堆中向下移一层，因此这种重建对于每次New值减1需O(1)的时间，总共的减操作数目为O(e)。因此在算法的所有步骤中，维持最大堆仅需O(e)的时间，
- 因而利用最大堆时覆盖算法的总复杂性为$O(n^2)$或O(n+e)。

贪心算法的理论基础

借助于**拟阵**工具，可建立关于贪心算法的较一般的理论。这个理论对**确定何时使用贪心算法**可以得到问题的整体最优解十分有用。

1、拟阵

拟阵M定义为满足下面3个条件的有序对(S,I)：

(1) S是非空有限集。

(2) I是S的一类具有遗传性质的独立子集族, 即若$B \in I$, 则B是S的独立子集, 且B的任意子集也都是S的独立子集。空集$\varnothing$必为I的成员。

(3) I满足交换性质，即若$A \in I, B \in I$且$|A| < |B|$，则存在某一元素$x \in B-A$，使得$A \cup \{x\} \in I$。

贪心算法的理论基础

例如，设S是一给定矩阵中行向量的集合，I是S的线性独立子集族，则由线性空间理论容易证明(S,I)是一拟阵。拟阵的另一个例子是无向图G=(V,E)的图拟阵$M_G=(S_G, I_G)$.

给定拟阵M=(S,I)，对于I中的独立子集$A \in I$，若S有一元素$x \notin A$，使得将x加入A后仍保持独立性，即$A \cup \{x\} \in I$,则称x为A的**可扩展元素**。

当拟阵M中的独立子集A没有可扩展元素时，称A为**极大独立子集**。

贪心算法的理论基础

下面的关于**极大独立子集**的性质是很有用的。

定理4.1：拟阵M中所有极大独立子集大小相同。

这个定理可以用反证法证明。

若对拟阵M=(S,I)中的S指定权函数W，使得对于任意x∈ S，有W(x)>0，则称拟阵M为**带权拟阵**。依此权函数，S的任一子集A的权定义为$W(A)=\sum_{x\in A} W(x)$。

2、关于带权拟阵的贪心算法

许多可以用贪心算法求解的问题可以表示为求带权拟阵的**最大权独立子集.**

贪心算法的理论基础

给定带权拟阵M=(S,I)，确定S的独立子集A∈I使得W(A)达到最大。这种使W(A)最大的独立子集A称为拟阵M的**最优子集**。由于S中任一元素x的权W(x)是正的，因此，**最优子集也一定是极大独立子集**。

例如，在最小生成树问题可以表示为确定带权拟阵M_G的最优子集问题。求带权拟阵的最优子集A的算法可用于解最小生成树问题。

下面给出求**带权拟阵最优子集**的贪心算法。该算法以具有正权函数W的带权拟阵M=(S,I)作为输入，经计算后输出M的最优子集A。

贪心算法的理论基础

Set **greedy** (M,W)

{A=∅;

将S中元素依权值W（大者优先）组成优先队列;

while (S!=∅) {

S.removeMax(x);

if (A∪{x}∈I) A=A∪{x};

}

return A

}

贪心算法的理论基础

算法**greedy**的计算时间复杂性为O(nlogn+nf(n)).

引理4.2(拟阵的贪心选择性质)

设M=(S,I)是具有权函数W的带权拟阵，且S中元素依权值从大到小排列。又设x∈ S是S中第一个使得{x}是独立子集的元素，则存在S的最优子集A使得x∈ A。

算法**greedy**在以贪心选择构造最优子集A时，首次选入集合A中的元素x是单元素独立集中具有最大权的元素。此时可能已经舍弃了S中部分元素。可以证明这些被舍弃的元素不可能用于构造最优子集。

贪心算法的理论基础

引理4.3： 设M=(S,I)是拟阵。若S中元素x不是空集　的可扩展元素，则x也不可能是S中任一独立子集A的可扩展元素。

引理4.4(拟阵的最优子结构性质)

设x是求带权拟阵M = (S，I)的最优子集的贪心算法**greedy**所选择的S中的第一个元素。那么，原问题可简化为求带权拟阵M'=(S',I')的**最优子集**问题，其中：

$$S'=\{y \mid y \in S \text{且} \{x,y\} \in I\}$$

$$I'=\{B \mid B \subseteq S-\{x\} \text{且} B \cup \{x\} \in I\}$$

M'的权函数是M的权函数在S'上的限制(称M'为M关于元素x的**收缩**)。

贪心算法的理论基础

定理4.5(带权拟阵贪心算法的正确性)

设M = (S,I)是具有权函数W的带权拟阵，算法greedy返回M的最优子集。

3、任务时间表问题

给定一个**单位时间任务**的有限集S。关于S的一个**时间表**用于描述S中单位时间任务的执行次序。时间表中第1个任务从时间0开始执行直至时间1结束，第2个任务从时间1开始执行至时间2结束，…，第n个任务从时间n-1开始执行直至时间n结束。

贪心算法的理论基础

具有**截止时间**和**误时惩罚**的单位时间任务时间表问题可描述如下。

(1) n个单位时间任务的集合S={1,2,…,n}；

(2) 任务i的截止时间d_i,$1 \le i \le n$,$1 \le d_i \le n$，即要求任务i在时间d_i之前结束；

(3) 任务i的误时惩罚w_i,$1 \le i \le n$,即任务i未在时间d_i之前结束将招致的w_i惩罚; 若按时完成则无惩罚。

任务时间表问题要求确定S的一个时间表（最优时间表）使得总误时惩罚达到最小。

贪心算法的理论基础

这个问题看上去很复杂，然而借助于**拟阵**，可以用**带权拟阵的贪心算法**有效求解。

对于一个给定的S的时间表，在截止时间之前完成的任务称为**及时任务**，在截止时间之后完成的任务称为**误时任务**。

S的任一时间表可以调整成**及时优先形式**，即其中所有及时任务先于误时任务，而不影响原时间表中各任务的及时或误时性质。

类似地，还可将S的任一时间表调整成为**规范形式**，其中及时任务先于误时任务，且及时任务依其截止时间的非减序排列。

贪心算法的理论基础

首先可将时间表调整为及时优先形式，然后再进一步调整及时任务的次序。

任务时间表问题**等价于**确定最优时间表中**及时任务子集A**的问题。一旦确定了及时任务子集A，将A中各任务依其截止时间的非减序列出，然后再以任意次序列出误时任务，即S-A中各任务，由此产生S的一个规范的最优时间表。

对时间t=1,2,…,n, N_t, **设** (A)是任务子集A中所有截止时间是t或更早的任务数。考察任务子集A的独立性。

贪心算法的理论基础

引理4.6：对于S的任一任务子集A，下面的各命题是等价的。

(1) 任务子集A是独立子集。

(2) 对于t=1,2,…,n, N_t (A)≤t。

(3) 若A中任务依其截止时间非减序排列，则A中所有任务都是及时的。

任务时间表问题要求使总误时惩罚达到最小，这等价于使任务时间表中的及时任务的惩罚值之和达到最大。下面的**定理**表明可用带权拟阵的贪心算法解任务时间表问题。

贪心算法的理论基础

定理4.7：设S是带有截止时间的单位时间任务集，I是S的所有独立任务子集构成的集合。则有序对(S,I)是拟阵。

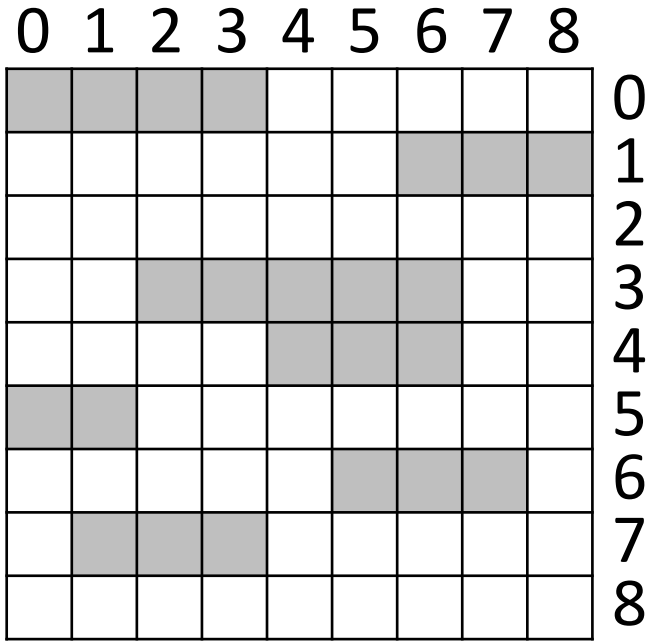
由**定理4.5**可知，用带权拟阵的贪心算法可以求得最大权(惩罚)独立任务子集A，以A作为最优时间表中的及时任务子集，容易构造最优时间表。

任务时间表问题的贪心算法的**计算时间复杂性**是O(nlogn+nf(n))。其中f(n)是用于检测任务子集A的独立性所需的时间。用引理4.6中性质(2)容易设计一个O(n)时间算法来检测任务子集的独立性。因此，整个算法的**计算时间**为$O(n^2)$。具体算法**greedyJob**可描述如P130。

用抽象数据类型并查集UnionFind可对上述算法作进一步改进。如果不计预处理的时间，改进后的算法fasterJob所需的**计算时间**为O(nlog*n)

Pass-Muraille

- 问题描述：在现代的魔术表演中，穿越墙壁是非常受欢迎的，魔术师在一个预先设计好的舞台上表演穿越几面墙壁。在每次穿越墙壁的表演中，穿墙魔术师有一个有限的穿墙能量，通过至多k面墙。墙壁被放置在一个网格状的区域中。右图给出 俯视图。所有的墙的厚度是一个单元，但长度不同。观众选择一列方格，穿墙者从图的上方沿着一列方格向下走，穿过每一面在他路上遇到的墙，到达图的下方。如果他试图走的那一列要穿过的墙超过k面，他将无法完成这个节目。例如右图中，一个穿墙者在k=3的情况下，从上到下可以选择除了第6列以外的任何一列。



- 给出一个穿墙者的能量以及一个表演舞台，要求在舞台上拆除最少数量的墙，使得表演者可以沿任意观众选择的列穿过所有的墙。

Pass-Muraille

- 输入：输入的第一行给出一个整数t(1≤t≤10)，表示测试用例的个数，然后给出每个测试用例的数据。每个测试用例的第一行给出两个整数n(1≤n≤100，表示墙的面数)和k(1≤k≤100，表示穿墙者可以通过的墙的最大面数)。在这一行后，给出n行，每行包含两个(x, y)对，表示一面墙的两个端点坐标。坐标是小于等于100的非负整数。左上角的方格的坐标为(0, 0)。
- 输出：每个测试用例一行，给出一个整数，表示最少拆除墙的面数，使得穿墙者能从上方任何一列开始穿越。

样例输入

```
2
3 1
2 0 4 0
0 1 1 1
1 2 2 2
7 3
0 0 3 0
6 1 8 1
2 3 6 3
4 4 6 4
0 5 1 5
5 6 7 6
1 7 3 7
```