第4章 贪心算法

第4章 贪心算法

- 顾名思义，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。
- 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。

第4章 贪心算法

本章主要知识点：

4.1 活动安排问题

4.2 贪心算法的基本要素

4.3 最优装载

4.4 哈夫曼编码

4.5 单源最短路径

4.6 最小生成树

4.7 多机调度问题

4.8 分治、动态规划和贪心对比

4.1 活动安排问题

- 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，是可以用贪心算法有效求解的很好例子。
- 该问题要求高效地安排一系列争用某一公共资源的活动。贪心算法提供了一个简单、漂亮的方法，使得尽可能多的活动能兼容地使用公共资源。

4.1 活动安排问题

- 设有n个活动的集合E={1,2,…,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i都有一个要求使用该资源的起始时间si和一个结束时间fi，且si <fi 。如果选择了活动i，则它在半开时间区间[si, fi)内占用资源。
- 若区间[si, fi)与区间[sj, fj)不相交，则称活动i与活动j是相容的。也就是说，当si≥fj或sj≥fi时，活动i与活动j相容。

4.1 活动安排问题

将输入的活动以其完成时间进行非减序排列，每次选择具有最早完成时间的相容活动加入集合A中。

```
public static int greedySelector(int []s, int []f, Boolean a[]){
      int n=s.length;
      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

4.1 活动安排问题

- 直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。
- 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。如果所给出的活动未按非减序排列，可以用O(nlogn)的时间重排。

4.1 活动安排问题举例

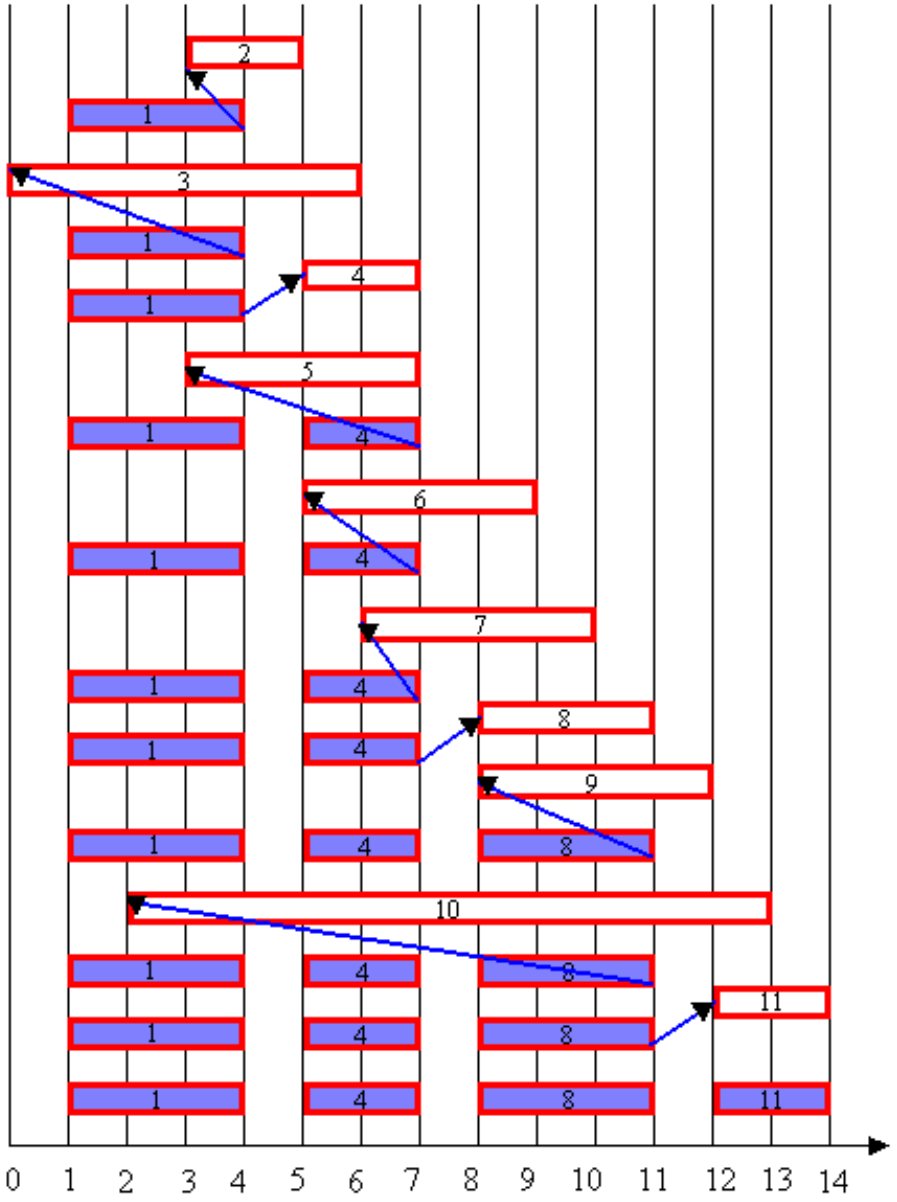
例：设待安排的11个活动的开始时间和结束时间按结束时间的非减序排列如下：

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

活动安排问题举例

图中每行相应于算法的一次迭代。阴影长条表示的活动是已选入集合A的活动，而空白长条表示的活动是当前正在检查相容性的活动。

若被检查的活动i的开始时间si小于最近选择的活动j的结束时间fj，则不选择活动i，否则选择活动i加入集合A中。



4.1 活动安排问题

贪心算法并不总能求得问题的整体最优解。但对于活动安排问题，贪心算法greedySelector却总能求得整体最优解，即它最终所确定的相容活动集合A的规模最大。这个结论可以用数学归纳法证明。

设E={1,2,…,n}为所给定的活动集合，且已按活动的结束时间排序，故活动1具有最早完成时间。

4.1 活动安排问题

1）活动安排问题有一个最优解以贪心算法选择开始，即该最优解中包含活动1。

证明：设A⊆E是所给活动安排问题的一个最优解，且A中活动也按结束时间非减序排列，A中的第一个活动是k。

- 若k=1，则A就是一个以贪心选择开始的最优解。
- 若k>1，则设B=A-{k}∪{1}。由于f1≤fk且A中活动是相容的，故B中的活动也是相容的。又由于|B|=|A|，且A是最优的，故B也是最优的。也就是说，B是以贪心选择活动1开始的最优活动安排。

由此可见，总存在以贪心选择开始的最优活动安排方案。

4.1 活动安排问题

2）若A是原问题的最优解，则A'=A-{1}是活动安排问题E'={i∈E: si⩾f1}的最优解。

证明：在做了贪心选择，即选择了活动1后，原问题就简化为对E中所有与活动1相容的活动进行活动安排的子问题。

如果能找到E'的一个解B'，包含比A'更多的活动，则将活动1加入到B'中将产生E的一个解B，它包含比A更多的活动。这与A的最优性矛盾。

4.2 贪心算法的基本要素

本节着重讨论可以用贪心算法求解的问题的一般特征。

对于一个具体的问题，怎么知道是否可用贪心算法解此问题，以及能否得到问题的最优解呢？这个问题很难给予肯定的回答。

但是，从许多可以用贪心算法求解的问题中看到这类问题一般具有2个重要的性质：**贪心选择性质**和**最优子结构性质**。

4.2 贪心算法的基本要素

1. 贪心选择性质

所谓**贪心选择性质**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，即贪心选择来达到。

这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。

对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解。

2. 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心算法求解的关键特征。

4.2 贪心算法与动态规划算法的差异

动态规划算法通常以**自底向上**的方式解各子问题，而贪心算法则通常以**自顶向下**的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题。

贪心算法和动态规划算法都要求问题具有最优子结构性质，这是两类算法的一个共同点。但是，对于具有**最优子结构**的问题应该选用贪心算法还是动态规划算法求解?是否能用动态规划算法求解的问题也能用贪心算法求解?下面研究2个经典的**组合优化问题**，并以此说明贪心算法与动态规划算法的主要差别。

背包问题与0-1背包问题

给定n种物品和一个背包。物品 i 的重量是Wi，其价值为Vi，背包的容量为C。应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

- 0-1背包问题：在选择装入背包的物品时，对每种物品 i 只有2种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品i。
- 背包问题：与0-1背包问题类似，所不同的是在选择物品 i 装入背包时，**可以选择物品 i 的一部分**，而不一定要全部装入背包。

这2类问题都具有**最优子结构**性质，极为相似，但背包问题可以用贪心算法求解，而0-1背包问题却不能用贪心算法求解。

背包问题与0-1背包问题

用贪心算法解背包问题：首先计算每种物品单位重量的价值Vi/Wi，然后，依贪心选择策略，将尽可能多的**单位重量价值最高**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C，则选择单位重量价值次高的物品并尽可能多地装入背包。依此策略一直地进行下去，直到背包装满为止。

- 对于0-1背包问题，贪心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公斤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，然后再作出最好选择。由此就导出许多互相重叠的子问题。这正是该问题可用动态规划算法求解的另一重要特征。

用贪心算法解背包问题

```
public static float knapsack(float c, float[] w, float[] v,
                             float[] x,  int n){
      Element [] d = new Element [n]; float opt=0;
      for(int i = 0; i < n; i++)
           d[i] = new Element(w[i],v[i],i);
           x[i] = 0;
      MergeSort.mergeSort(d);
      for(int i=0;i<n;i++)
           if(d[i].w>c)
                break;
           x[d[i].i]=1;
           opt+=d[i].v;
           c-=d[i].w;
      if(i<n)
           x[d[i].i]=c/d[i].w;
           opt+=x[d[i].i]*d[i].v;
      return opt;
}
```

算法knapsack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) 。

贪心算法的基本步骤

为了形式化地描述贪心算法的一般处理过程，我们对所要求解的问题做如下假设。

1）有一个以优化方式来求解的问题。

为了构造问题的解决方案，有一个候选对象的集合C，如活动安排问题中的活动集合E ={1,2,3，…，n}。

2）随着问题求解过程的进行，这个集合将逐步被划分为两个集合：一个包含已经被考虑过并被选择的候选对象集合S；另一个包含已经被考虑过但被丢弃的候选对象。

3）有一个函数solution(S)来检查一个所选择的对象集合是否提供了问题的解答，也即是否可能往该集合上添加更多的候选对象以获得一个解。该函数不考虑此时的解决方法是否最优。如活动安排问题中所有活动是否已经被考察。

贪心算法的基本步骤

4）还有一个函数feasible(S)检查是否一个所选对象的集合是可行的。和上一个函数一样，此时不考虑解决方法的最优性。在活动安排问题中，这个函数用来判断所选出的活动之间是否相容。

5）选择函数select (C)用来从剩余候选对象集合中，根据当前状态选择出最有希望得到问题最优解的对象。在活动安排问题中，选择函数是从剩余活动中选择最早结束的活动。

6）最后，有一个目标函数给出问题解的值，如在活动安排问题中被选择安排的活动数。

其中，选择函数是贪心算法设计的关键，通常与目标函数有关。

贪心算法的一般过程

```
function greedy(C) {   /*C是候选对象集合*/
        S=Φ;   /*集合S是构造解*/
        while (C!= Φ && !solution(S) )
                x=select(C);
                C=C-{x};
                if(feasible(S∪{x}))
                        S = S∪{x};
        if(solution(S))
                return S;
        else
                return “No solutions”;
}
```

4.3 最优装载

有一批集装箱要装上一艘载重量为c的轮船。其中集装箱 i 的重量为Wi。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

该问题可形式化描述为：

$$\begin{cases} \max \sum_{i=1}^{n} x_i \\ \sum_{i=1}^{n} w_i x_i \le c \end{cases} \quad x_i \in (0,1)$$

4. 3 最优装载

1. 算法描述

最优装载问题可用贪心算法求解。采用重量最轻者先装的贪心选择策略，可产生最优装载问题的最优解。具体算法描述如下。

2. 贪心选择性质

证明：设集装箱已依其重量从小到大排列，(x1, x2,...,xn)是最优装载问题的一个最优解，又设k= $\min_{1\le i\le n}\{i|xi=1\}$。已知，如果给定的最优装载问题有解，则$1 \le k \le n$。

1）当k=1时， (x1, x2,...,xn)是满足贪心选择性质的最优解。

4.3 最优装载

2）当k>1时，取y1=1，yk=0，yi=xi，1 <i≤n，i≠k。则

$$\sum_{i=1}^{n} w_i y_i = \sum_{i=1}^{n} w_i x_i - w_k + w_1 \leq \sum_{i=1}^{n} w_i x_i \leq C$$

因此，(y1,y2,…,yn)是所给最优装载问题的可行解。

由$\sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} x_i$知，(y1,y2,…,yn)是满足贪心选择性质的最优解。

4.3 最优装载

3. 最优子结构性质

设(x1, x2,...,xn)是最优装载问题的满足贪心选择性质的最优解，则容易知道，x1=1，且(x2, x3, ..., xn)是轮船载重量为c-w1，待装船集装箱为(2,3,...,n)时相应最优装载问题的最优解。

4.3 最优装载

```
public static float loading(float c, float[] w, int[] x, int n){
        Element [] d = new Element [n];
        for(int i = 0; i < n; i++)
            d[i] = new Element(w[i],i);
        MergeSort.mergeSort(d);
        float opt=0;
        for(int i = 0; i < n; i++) x[i] = 0;
        for(int i = 0; i < n && d[i].w <= c; i++)
            x[d[i].i] = 1;
            opt+=d[i].w;
            c -= d[i].w;
        return opt;
    }
```

算法**loading**的主要计算量在于将集装箱依其重量从小到大排序，故算法所需的计算时间为 **O(nlogn)**。

由最优装载问题的贪心选择性质和最优子结构性质，容易证明算法**loading**的正确性。

4.4 哈夫曼编码

哈夫曼编码是广泛地用于数据文件压缩的十分有效的编码方法。其压缩率通常在20%～90%之间。哈夫曼编码算法用字符在文件中出现的频率表来建立一个用0，1串表示各字符的最优表示方式。

给出现频率高的字符较短的编码，出现频率较低的字符以较长的编码，可以大大缩短总码长。

4.4 哈夫曼编码

1. 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他字符代码的前缀。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。

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**。

表示最优前缀码的二叉树中任一结点都有2个儿子结点。

4.4 哈夫曼编码

2. 构造哈夫曼编码

哈夫曼提出构造最优前缀码的贪心算法，由此产生的编码方案称为

哈夫曼编码。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。

编码字符集中每一字符c的频率是f(c)。

- **以f为键值的优先队列Q**用在**贪心选择**时有效地确定算法当前要合并的2棵具有最小频率的树。
- 一旦2棵具有最小频率的树合并，将产生一棵新的树，其频率为合并的2棵树的频率之和，并将新树插入优先队列Q。经过n - 1次的合并队列中只剩下一棵树，即所要求的树T。

4.4 哈夫曼编码

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。

算法huffmanTree用最小堆实现优先队列Q。初始化优先队列需要O(n)计算时间，由于最小堆的removeMin和put运算均需O(logn)时间，n - 1次的合并总共需要O(nlogn)计算时间。因此，关于n个字符的哈夫曼算法的计算时间为O(nlogn) 。

4.4 哈夫曼编码

3. 哈夫曼算法的正确性

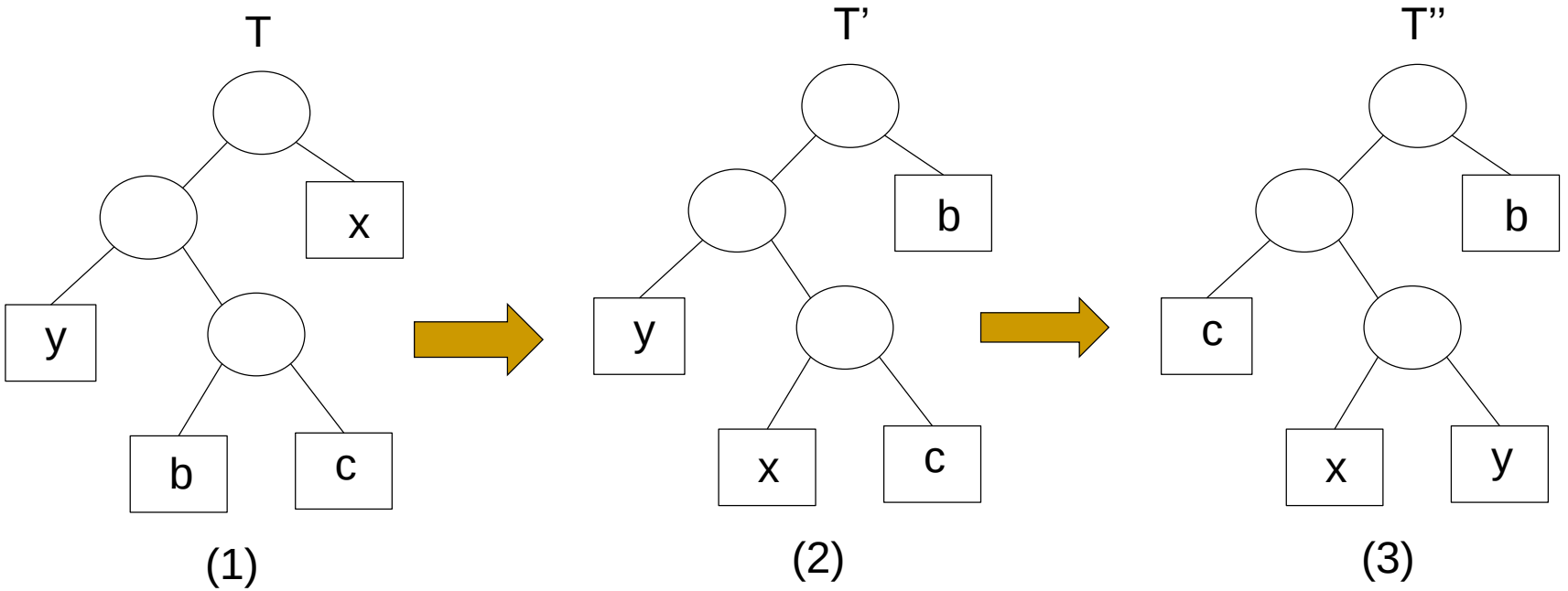
要证明哈夫曼算法的正确性，只要证明最优前缀码问题具有**贪心选择性质**和**最优子结构性质**。

设C是编码字符集，C中字符c的频率为f(c)。又设x和y是C中具有最小频率的两个字符，则存在C的最优前缀码使x和y具有相同码长且最后一位编码不同。设f(x)≤f(y)。

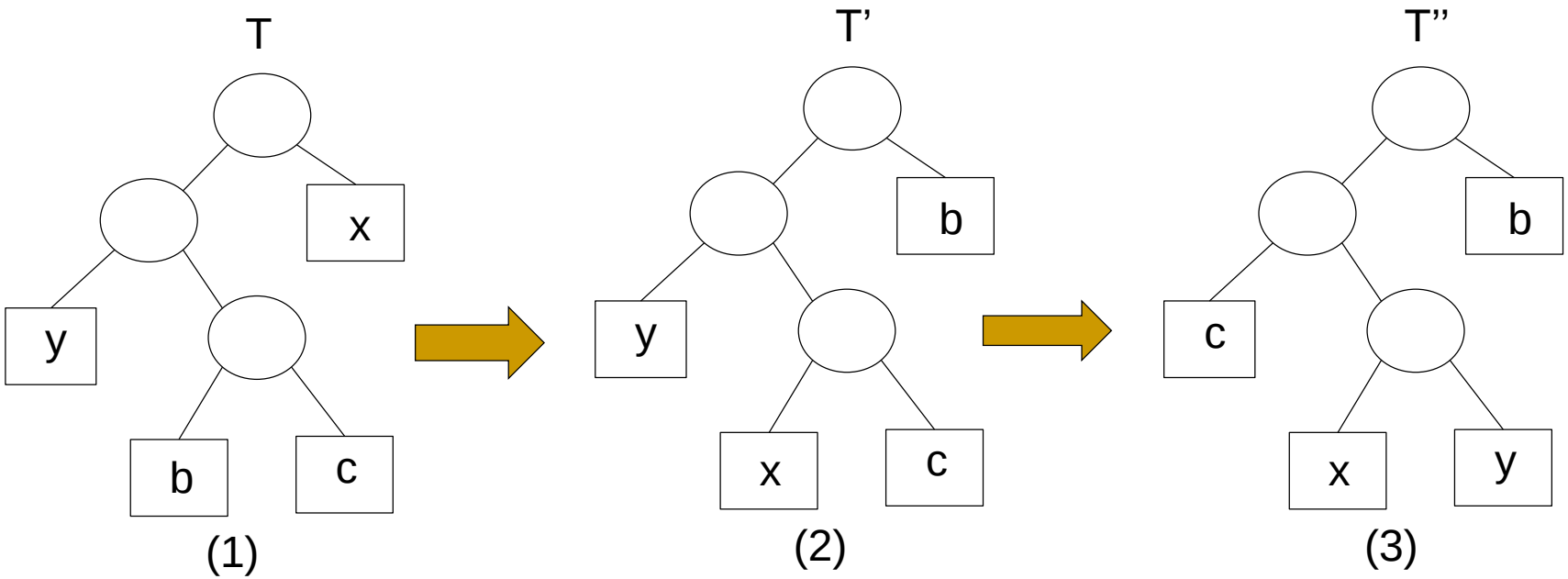
4.4 哈夫曼编码

1)贪心选择性质

证明：设二叉树T(图1)表示C的任意一个最优前缀码。b和c是T的最深叶子且为兄弟，设f(b)≤f(c)。首先在T中交换b和x得到T’(图2)，再在T’中交换c和y得到T’’(图3)。



4.4 哈夫曼编码



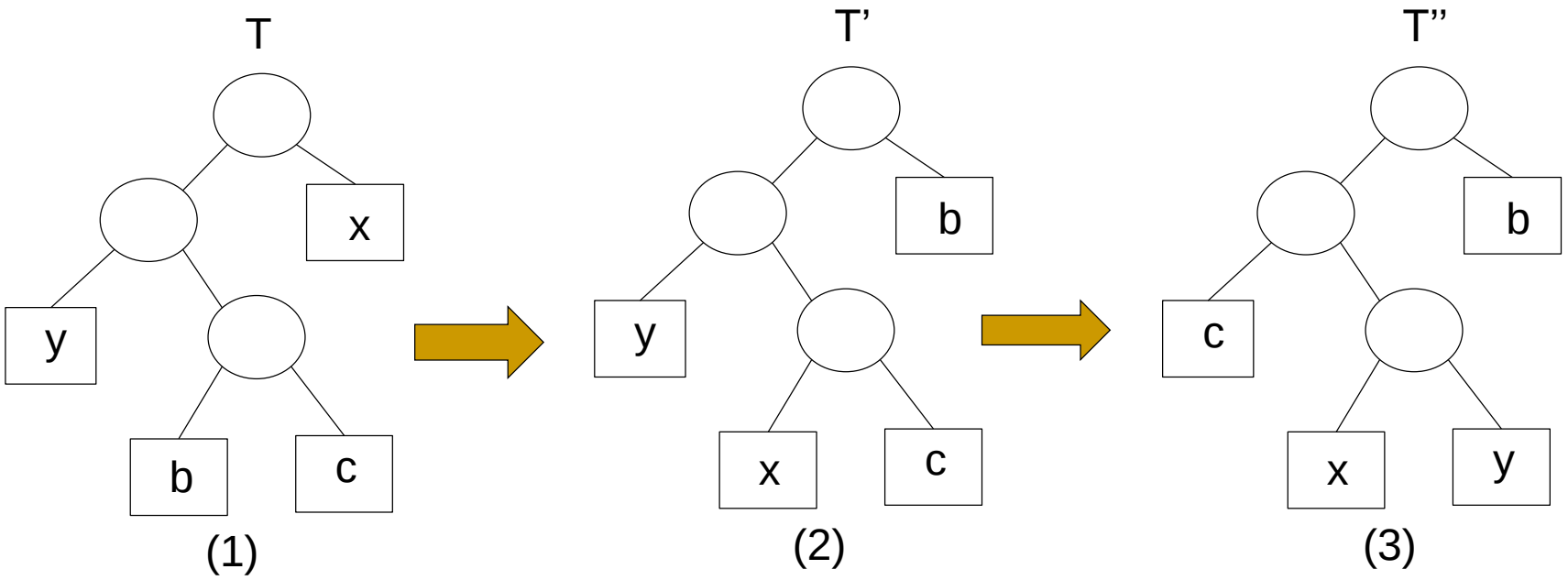
T和T'的平均码长之差为：

$$B(T)-B(T') = \sum_{c\in C} f(c)d_T(c) - \sum_{c\in C} f(c)d_{T'}(c)$$

$$= f(x)\,d_T(x) + f(b)\,d_T(b) - f(x)\,d_T(b) - f(b)\,d_T(x)$$

$$= (f(b)-f(x))(d_T(b)-d_T(x)) \geq 0$$

4.4 哈夫曼编码



同理，可以证明在T’中交换y与c的位置也不增加平均码长，即B(T’)-B(T’’)也是非负的。因此，$B(T'') \leq B(T') \leq B(T)$。另外T表示的前缀码是最优的，故$B(T) \leq B(T'')$。T’’表示的前缀码也是最优的，且x和y具有最长的码长，仅最后一位编码不同。

4.4 哈夫曼编码

1) 最优子结构性质

设T是表示字符集C的一个最优前缀码完全二叉树，C中字符c的频率为f(c)。又设x和y是T中的两个叶子且为兄弟，z是他们的父亲。若将z看做f(z)=f(x)+f(y)的字符，则树T'=T-{x,y}表示字符集C'=C-{x,y}∪{z}的一个最优前缀码。

证明：i) T的平均码长B(T)可用T'的平均码长B(T')来表示。

对任意c∈C-{x,y}有$d_T(c) = d_{T'}(c)$，故$f(c)d_T(c) = f(c)d_{T'}(c)$。另外，$d_T(x) = d_T(y) = d_{T'}(z) + 1$，故

$$f(x)d_T(x) + f(y)d_T(y) = \big(f(x) + f(y)\big)(d_{T'}(z) + 1)$$

$$= f(x) + f(y) + f(z)d_{T'}(z)$$

由此可知，B(T) = B(T') + f(x) + f(y)

4.4 哈夫曼编码

1)最优子结构性质

证明(续)：ii) $B(T) = B(T') + f(x) + f(y)$

若T’表示的字符集C’的前缀码不是最优的，则有T’’表示的C’的前缀码使得$B(T'')<B(T')$。由于z被看作C’中的一个字符，故z在T’’中是一树叶。若将x和y加入树T’’中作为z的儿子，则得到表示字符集C的前缀码的二叉树T’’’，且有

$$B(T''') = B(T'') + f(x) + f(y)$$

$$< B(T') + f(x) + f(y)$$

$$= B(T)$$

这与T的最优性矛盾。故T’表示的C’的前缀码是最优的。

4.5 单源最短路径

给定带权有向图G =(V,E)，其中每条边的权是非负实数。另外，还给定V中的一个顶点，称为源。现在要计算从源到所有其他各顶点的最短路长度。这里路的长度是指路上各边权之和。这个问题通常称为单源最短路径问题。

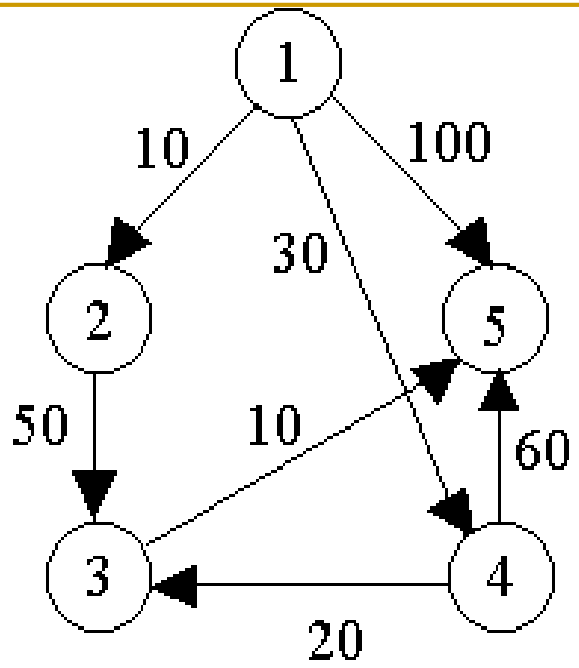
1. 算法基本思想

Dijkstra算法是解单源最短路径问题的贪心算法。其基本思想是，设置顶点集合S并不断地作贪心选择来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

4.5 单源最短路径

- 初始时，S中仅含有源。
- 设u是G的某一个顶点，把从源到u且中间只经过S中顶点的路称为从源到u的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度。
- Dijkstra算法每次从V-S中取出具有最短特殊路长度的顶点u，将u添加到S中，同时对数组dist作必要的修改。
- 一旦S包含了所有V中顶点，dist就记录了从源到所有其他顶点之间的最短路径长度。

4.5 单源最短路径



例如，对右图中的有向图，应用Dijkstra算法计算从源顶点1到其他顶点间最短路径的迭代过程列在下表中。

迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

2.算法的正确性

1) 贪心选择性质

Dijkstra算法所做的贪心选择是从V-S中选择具有最短特殊路径的顶点u，从而确定从源到u的最短路径长度dist[u]。

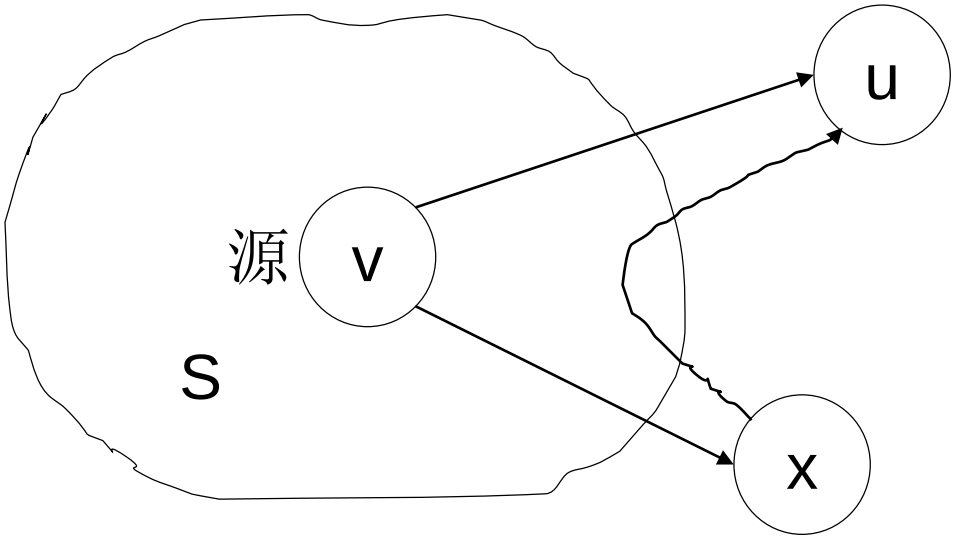
这种贪心选择为什么能导致最优解呢？换句话说，为什么从源到u没有更短的其他路径呢？

假设存在一条从源到u且长度比dist[u]更短的路，设这条路初次走出S之外到达的顶点$x \in V-S$，然后徘徊于S内外若干次，最后离开S到达u。

4.5 单源最短路径

2.算法的正确性

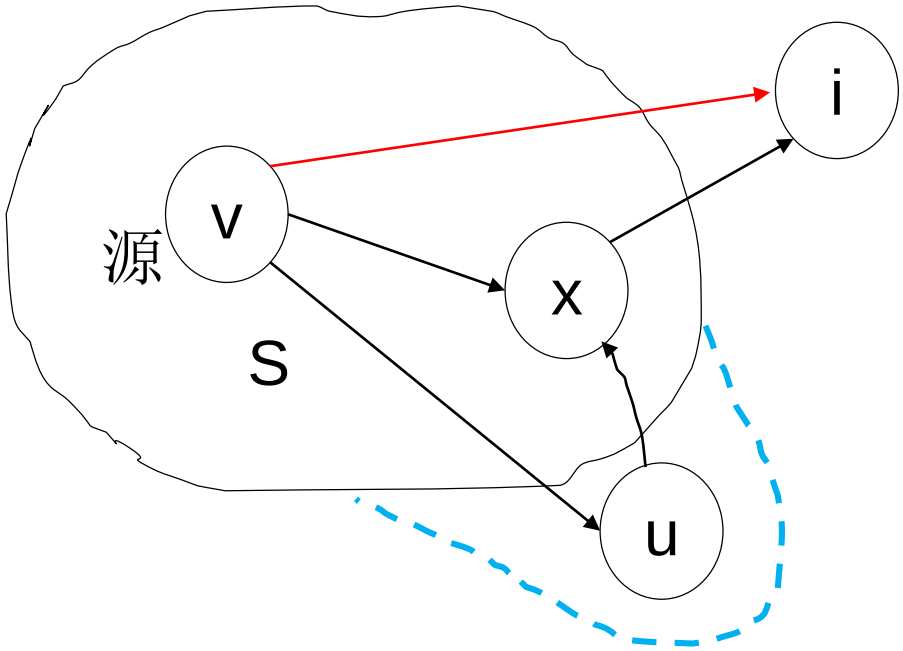
1)贪心选择性质



4.5 单源最短路径

2. 算法的正确性和计算复杂性

2) 最优子结构性质



4.5 单源最短路径

2. 算法的正确性和计算复杂性

3) 计算复杂性

对于具有n个顶点和e条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要O(n) 时间。这个循环需要执行n-1次，所以完成循环需要$O(n^2)$时间。算法的其余部分所需要时间不超过$O(n^2)$ 。

4.6 最小生成树

设G =(V,E)是无向连通带权图，即一个网络。E中每条边(v,w)的权为c[v][w]。如果G的子图G'是一棵包含G的所有顶点的树，则称G'为G的生成树。生成树上各边权的总和称为该生成树的耗费。在G的所有生成树中，耗费最小的生成树称为G的最小生成树。

网络的最小生成树在实际中有广泛应用。例如，在设计通信网络时，用图的顶点表示城市，用边(v,w)的权c[v][w]表示建立城市v和城市w之间的通信线路所需的费用，则最小生成树就给出了建立通信网络的最经济的方案。

4. 6 最小生成树

1. 最小生成树性质

用贪心算法设计策略可以设计出构造最小生成树的有效算法。本节介绍的构造最小生成树的**Prim算法**和**Kruskal算法**都可以看作是应用贪心算法设计策略的例子。尽管这2个算法做贪心选择的方式不同，它们都利用了下面的**最小生成树性质**：

设G=(V,E)是连通带权图，U是V的真子集。如果(u,v)∈E，且u∈U，v∈V-U，且在所有这样的边中，(u,v)的权c[u][v]最小，那么一定存在G的一棵最小生成树，它以(u,v)为其中一条边。这个性质有时也称为**MST性质**。

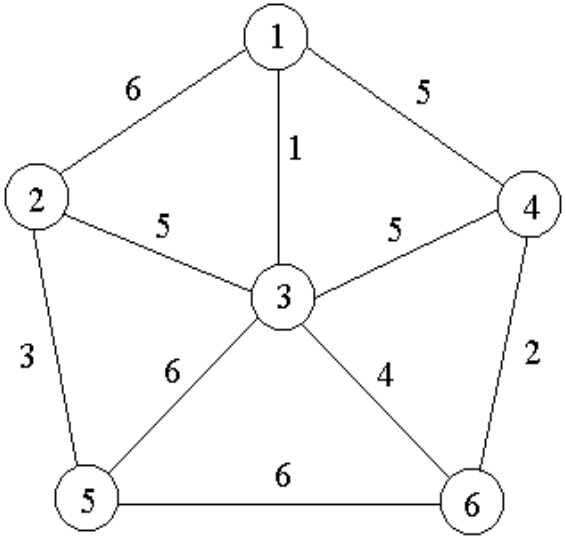
Prim算法

设G=(V,E)是连通带权图，V={1,2,…,n}。

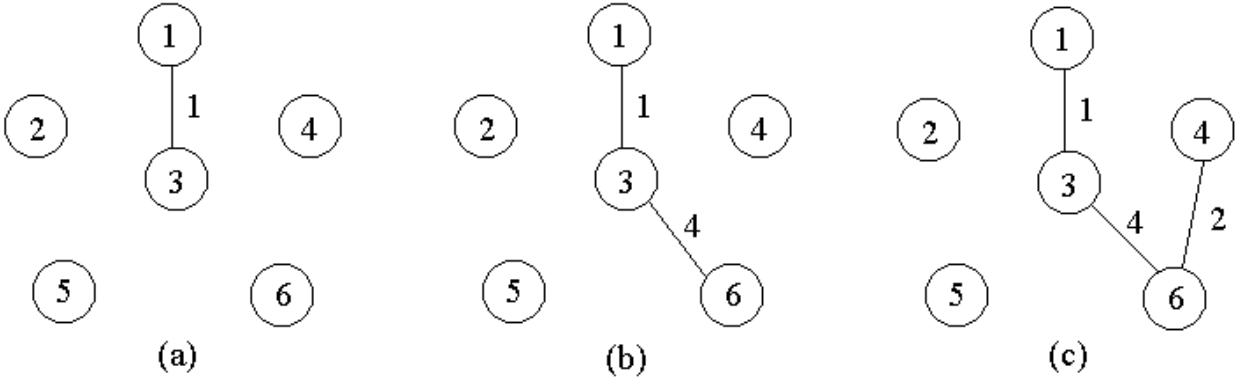
构造G的最小生成树的Prim算法的基本思想是：首先置S={1}，然后，只要S是V的真子集，就作如下的**贪心选择**：选取满足条件i∈S，j∈V-S，且c[i][j]最小的边，将顶点j添加到S中。这个过程一直进行到S=V时为止。

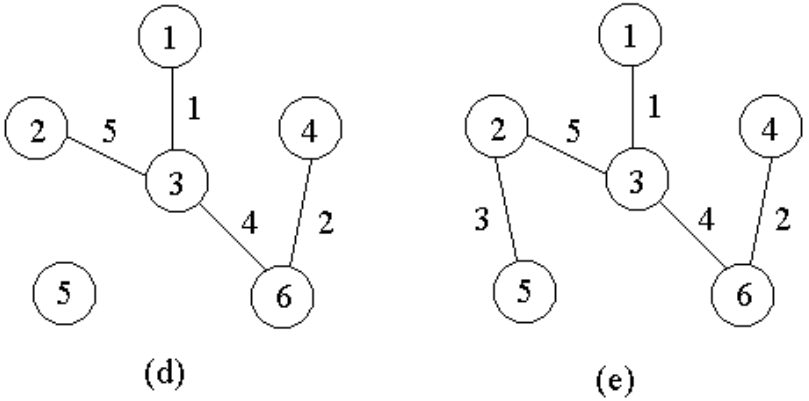
在这个过程中选取到的所有边恰好构成G的一棵最小生成树。

Prim算法



例如，对于右图中的带权图，按Prim算法选取边的过程如下所示。





Prim算法

在上述Prim算法中，还应当考虑**如何有效地找出满足条件i∈S, j∈V-S，且权c[i][j]最小的边(i, j)**。实现这个目的的较简单的办法是设置2个数组closest和lowcost。closest[j]记录S中与j最近的顶点，lowcost[j]记录j与S中最近顶点的距离。

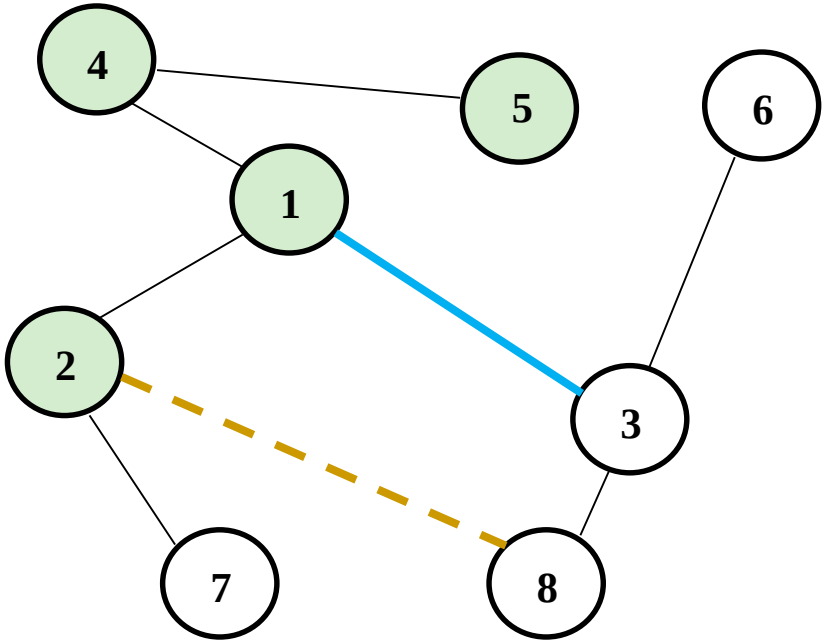
在Prim算法执行过程中，先找出V-S中使lowcost值最小的顶点 j，然后根据数组closest选取边(j, closest[j]），最后将j添加到S中，并对closest和lowcost作必要的修改。

用这个办法实现的Prim算法所需的计算时间为$O(n^2)$。

Prim算法

算法的正确性

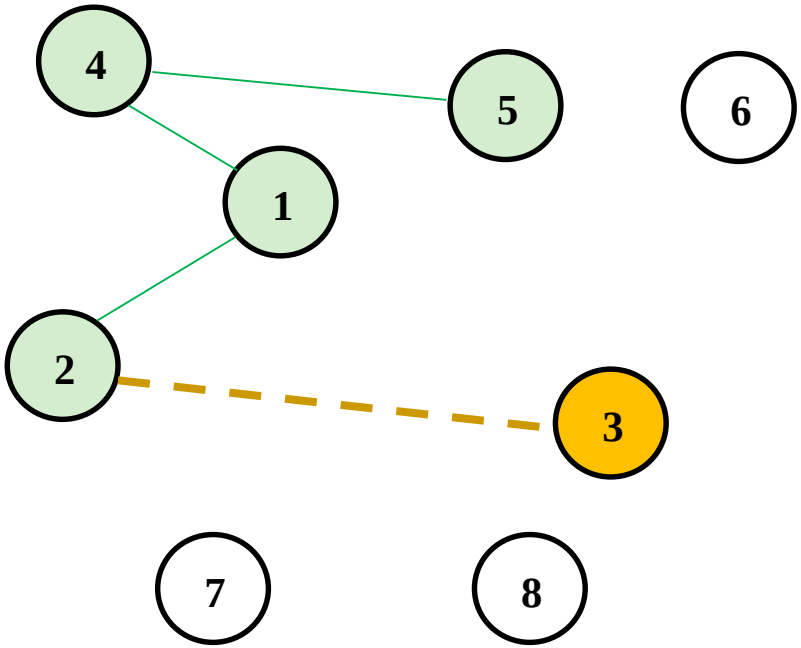
1）贪心选择性质



Prim算法

算法的正确性

2）最优子结构性质



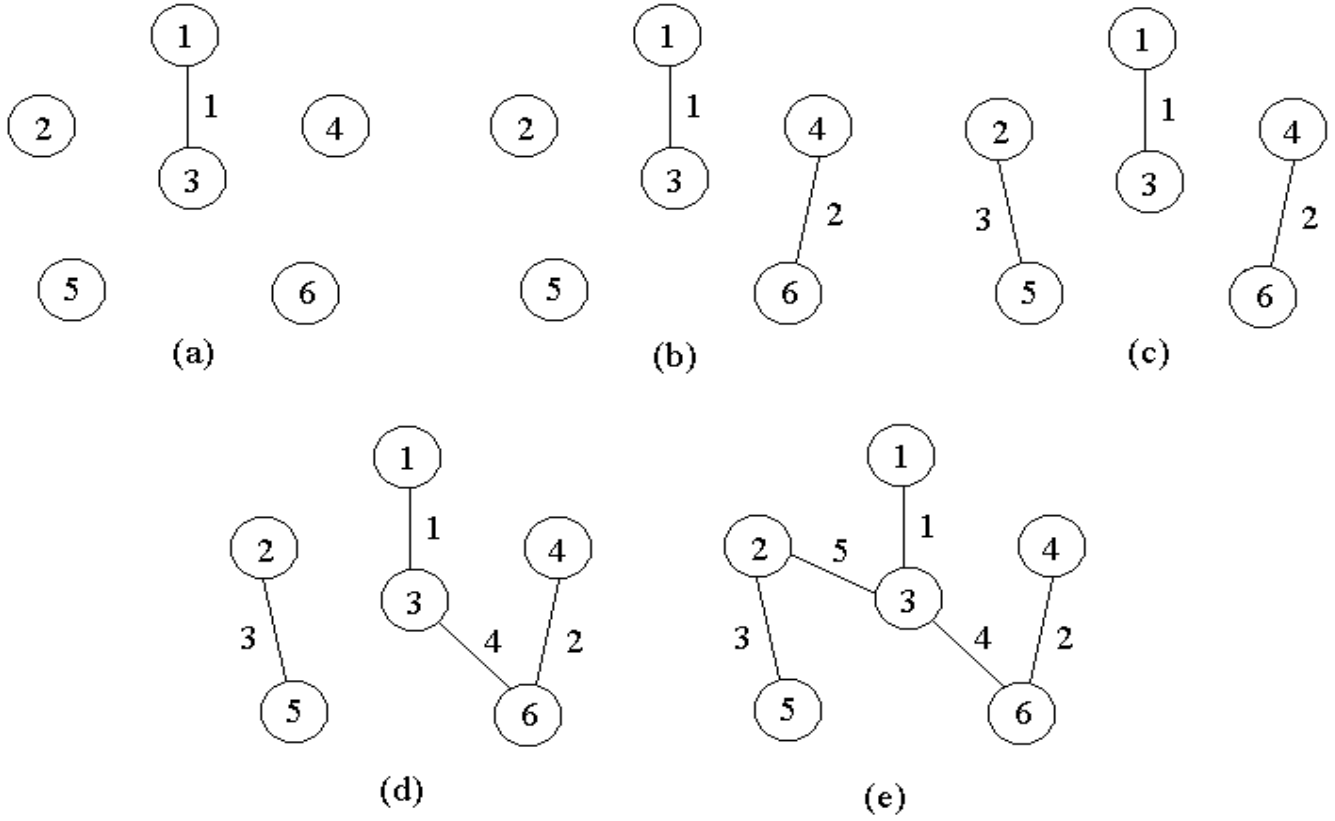
Kruskal算法

Kruskal算法构造G的最小生成树的基本思想是：

- 首先将G的n个顶点看成n个孤立的连通分支。将所有的边按权从小到大排序。
- 然后从第一条边开始，依边权递增的顺序查看每一条边，并按下述方法连接2个不同的连通分支：
  - 当查看到第k条边(v,w)时，如果端点v和w分别是当前2个不同的连通分支T1和T2中的顶点时，就用边(v,w)将T1和T2连接成一个连通分支，然后继续查看第k+1条边；
  - 如果端点v和w在当前的同一个连通分支中，就直接再查看第k+1条边。这个过程一直进行到只剩下一个连通分支时为止。

Kruskal算法

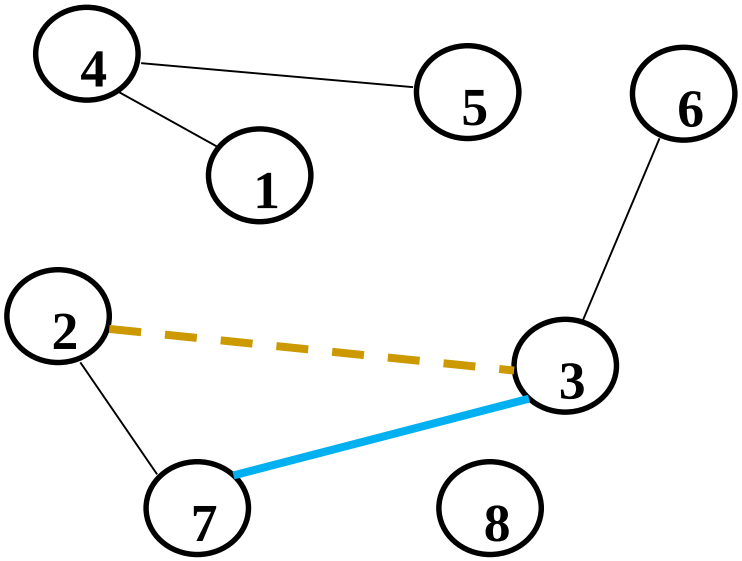
举例：对前面的连通带权图，按Kruskal算法顺序得到的最小生成树上的边如下图所示。



Kruskal算法

算法的正确性

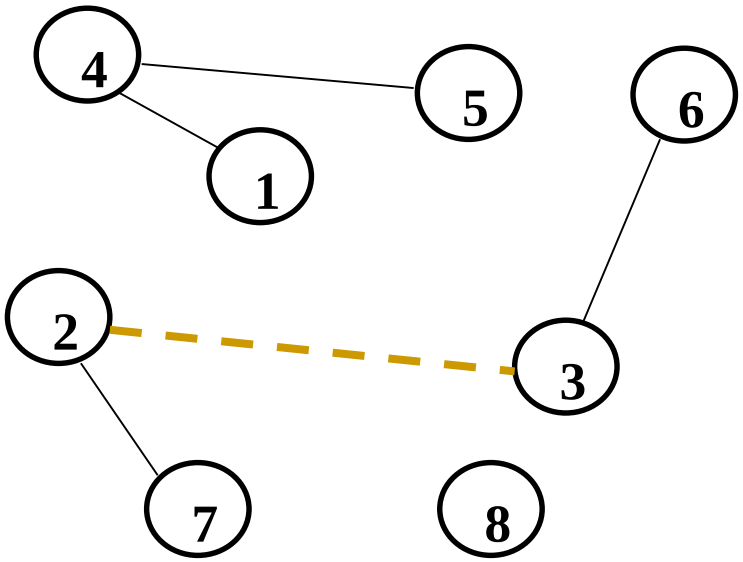
1）贪心选择性质



Kruskal算法

算法的正确性

1）最优子结构性质



4.7 多机调度问题

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。

约定，每个作业均可在任何一台机器上加工处理，但未完工前不允许中断处理。作业不能拆分成更小的子作业。

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**。

4.7 多机调度问题

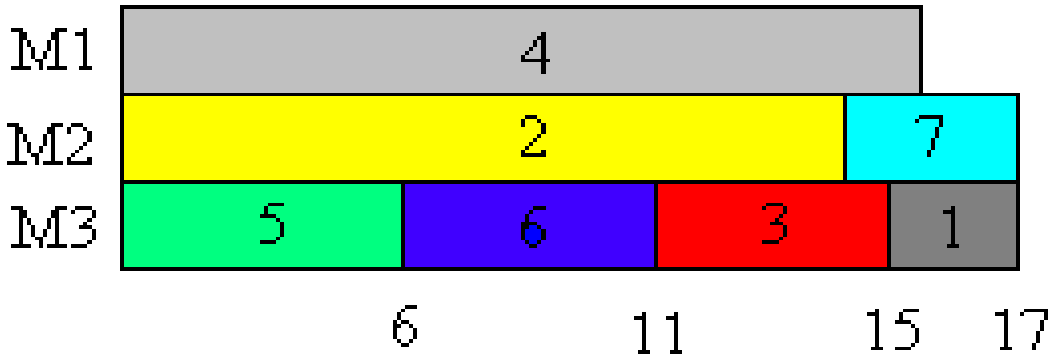
采用**最长处理时间作业优先**的贪心选择策略可以设计出解多机调度问题的较好的**近似算法**。

按此策略，当$n \leq m$时，只要将机器i的[0, ti]时间区间分配给作业i即可，算法只需要**O(1)**时间。

当$n > m$时，首先将n个作业依其所需的处理时间从大到小排序。然后依此顺序将作业分配给空闲的处理机。算法所需的计算时间为**O(nlogn)**。

4.7 多机调度问题

例如，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}，处理时间从长到短对作业排序得到{4,2,5,6,3,7,1}。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。



算法设计策略的比较与选择

- Divide-and-conquer: Break up a problem into independent subproblems, solve each subproblem, and combine solutions of subproblems to form solution to original problem.
- Dynamic programming: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.
- Greedy： Build up a solution incrementally, myopically optimizing some local criterion.

分治算法

- 特征
  - 规模如果很小，则很容易解决；//一般问题都能满足
  - 大问题可以分为若干规模小的相同问题；//前提
  - 利用子问题的解，可以合并成该问题的解；//关键
  - 分解出的各个子问题相互独立，子问题不再包含公共子问题；//效率高低
- 实质
  - 递归求解（各子问题要互相独立）；
- 缺点
  - 如果子问题不独立，需要重复求公共子问题；

动态规划算法

- 特征
  - 不仅求出了当前状态最优值，而且同时求出了中间状态的最优值；
  - 自底向上，从小子问题处理至大子问题；
  - 每一步都要做出选择，这些选择依赖于子问题的解；
- 实质
  - 分治思想和解决冗余；//各子问题不相互独立，各子问题包含公共的子问题，对每个子问题只求解一次，将其结果保存在一张表中，避免重新计算
- 缺点
  - 空间需求大；

贪心算法

- 特征
  - 依赖于当前已经做出的所有选择；
  - 所做的选择看起来都是当前最佳的，期望通过局部最优选择来产生一个全局最优解；
  - 自顶向下，先做出选择再求解子问题；
- 缺点
  - 对大多数优化问题来说能产生最优解，但也不一定总是这样

分治、动态规划、贪心

	分治	动态规划	贪心
适应类型	优化问题 或 通用问题		
子问题结构	子问题是否重复		
最优子结构	是否需要满足最优子结构		
子问题数	解决几个子问题		
子问题在最优解里	子问题是否全部在最优解里		
选择与求解次序	先选择还是先求解		

分治、动态规划、贪心

	分治	动态规划	贪心
适应类型	通用问题	优化问题	优化问题
子问题结构	每个子问题不同	很多子问题重复	只有一个子问题
最优子结构	不需要	必须满足	必须满足
子问题数	全部子问题都要解决	全部子问题都要解决	只要解决一个子问题
子问题在最优解里	全部	部分	部分
选择与求解次序	先选择后解决子问题	先解决子问题后选择	先选择后解决子问题