



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 $E=\{1,2,\dots,n\}$ ，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 s_i 和一个结束时间 f_i ，且 $s_i < f_i$ 。如果选择了活动 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 i 与活动 j 是相容的。也就是说，当 $s_i \geq f_j$ 或 $s_j \geq f_i$ 时，活动 i 与活动 j 相容。



4.1 活动安排问题

下面所给出的解活动安排问题的贪心算法greedySelector:

```
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

- 由于输入的活动以其完成时间的非减序排列，所以算法greedySelector每次总是选择具有最早完成时间的相容活动加入集合A中。
- 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 $O(n\log n)$ 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：

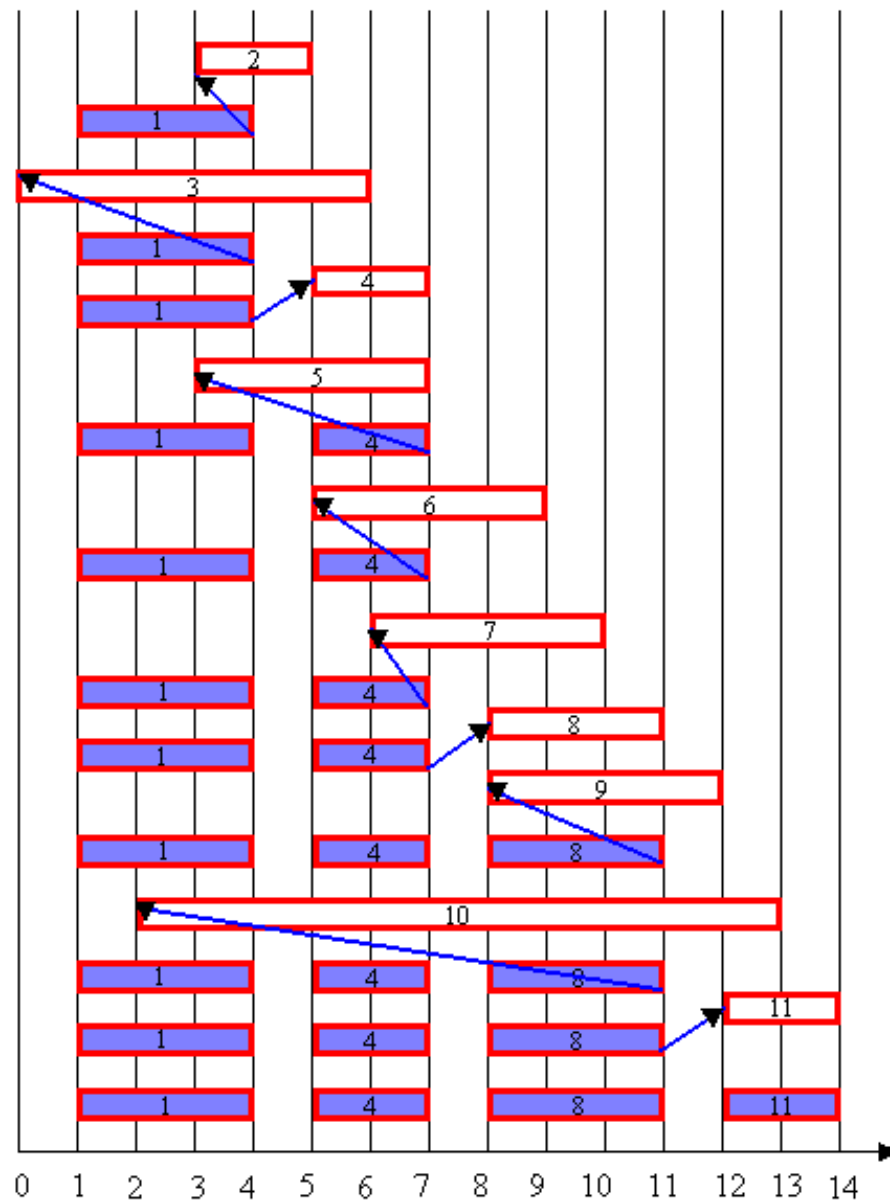
i	1	2	3	4	5	6	7	8	9	10	11
s[i]	1	3	0	5	3	5	6	8	8	2	12
f[i]	4	5	6	7	8	9	10	11	12	13	14



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 s_i 小于最近选择的活动的结束时间 f_j ，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 $E=\{1,2,\dots,n\}$ 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 $A \subseteq E$ 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 $B=A-\{k\} \cup \{1\}$ 。由于 $f_1 \leq f_k$ 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 $E' = \{i \in E: s_i \geq f_1\}$ 的最优解。

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。

证明: 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。



4.2 贪心算法的基本要素

2. 最优子结构性质

当一个问题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 贪心算法的基本要素

3. 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性质，这是2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。



4.2 贪心算法的基本要素

■ 0-1背包问题:

- 给定 n 种物品和一个背包。物品 i 的重量是 W_i ，其价值为 V_i ，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？
- 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 。



4.2 贪心算法的基本要素

■ 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， $1 \leq i \leq n$ 。

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。



4.2 贪心算法的基本要素

用贪心算法解背包问题的基本步骤:

首先计算每种物品单位重量的价值 V_i/W_i ，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。

具体算法可描述如下页:



4.2 贪心算法的基本要素

```
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 $O(n\log n)$ 。



4.2 贪心算法的基本要素

- 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。
- 事实上，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。
- 实际上也是如此，动态规划算法的确可以有效地解0-1背包问题。



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，有一个候选的对象的集合C，如活动安排问题中的活动集合 $E = \{1, 2, 3, \dots, n\}$ 。

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。



贪心算法的基本步骤

- 3) 有一个函数 $\text{solution}(S)$ 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。
- 4) 还有一个函数 $\text{feasible}(S)$ 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。



贪心算法的基本步骤

- 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 =  $\Phi$ ; /*集合S是构造解*/  
    while (C !=  $\Phi$  && ! solution(S) )  
        x = select(C);  
        C = C - {x};  
        if ( feasible(S  $\cup$  {x}) ) S = S  $\cup$  {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 W_i 。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。

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。具体算法描述如下页。



4.3 最优装载

2. 贪心选择性质

证明：设集装箱已依其重量从小到大排列， (x_1, x_2, \dots, x_n) 是最优装载问题的一个最优解。又设 $k = \min_{1 \leq i \leq n} \{i | x_i = 1\}$ 。已知，如果给定的最优装载问题有解，则 $1 \leq k \leq n$ 。

1) 当 $k=1$ 时， (x_1, x_2, \dots, x_n) 是满足贪心选择性质的最优解。



4.3 最优装载

2. 贪心选择性质

证明:

2) 当 $k > 1$ 时, 取 $y_1 = 1, y_k = 0, y_i = x_i, 1 < i \leq n, i \neq k$ 。

则

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

因此, (y_1, y_2, \dots, y_n) 是所给最优装载问题的可行解。

由 $\sum_{i=1}^n y_i = \sum_{i=1}^n x_i$ 知, (y_1, y_2, \dots, y_n) 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 (x_1, x_2, \dots, x_n) 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， $x_1 = 1$ ，且 (x_2, x_3, \dots, x_n) 是轮船载重量为 $c - w_1$ ，待装船集装箱为 $(2, 3, \dots, n)$ 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 **$O(n\log n)$** 。

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。

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。



4.4 哈夫曼编码

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总是一棵**完全二叉树**，即树中任一结点都有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 。



哈夫曼编码算法

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 $n - 1$ 次的合并后，优先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 $O(\log n)$ 时间， $n - 1$ 次的合并总共需要 $O(n \log n)$ 计算时间。因此，关于 n 个字符的哈夫曼算法的计算时间为 $O(n \log n)$ 。



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**。

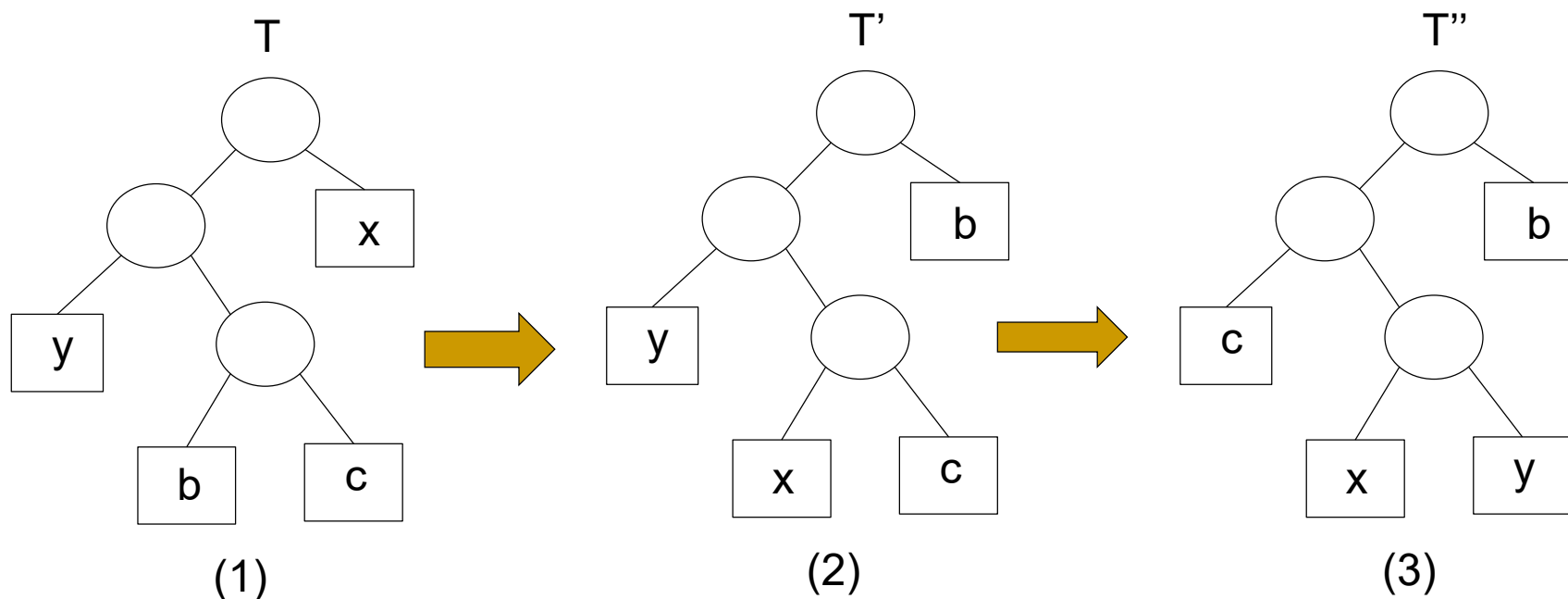
设 C 是编码字符集， C 中字符 c 的频率为 $f(c)$ 。又设 x 和 y 是 C 中具有最小频率的两个字符，则存在 C 的最优前缀码使 x 和 y 具有相同码长且最后一位编码不同。设 $f(x) \leq f(y)$ 。



4.4 哈夫曼编码

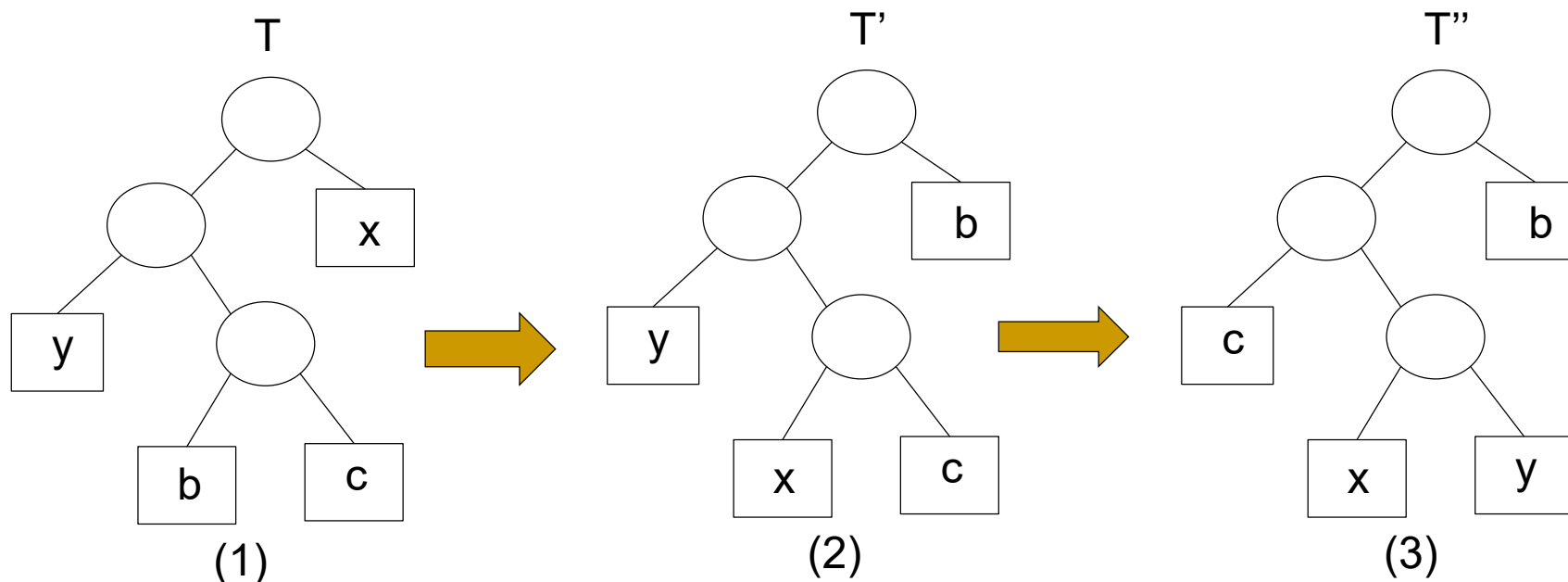
1) 贪心选择性质

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 $f(b) \leq f(c)$ 。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

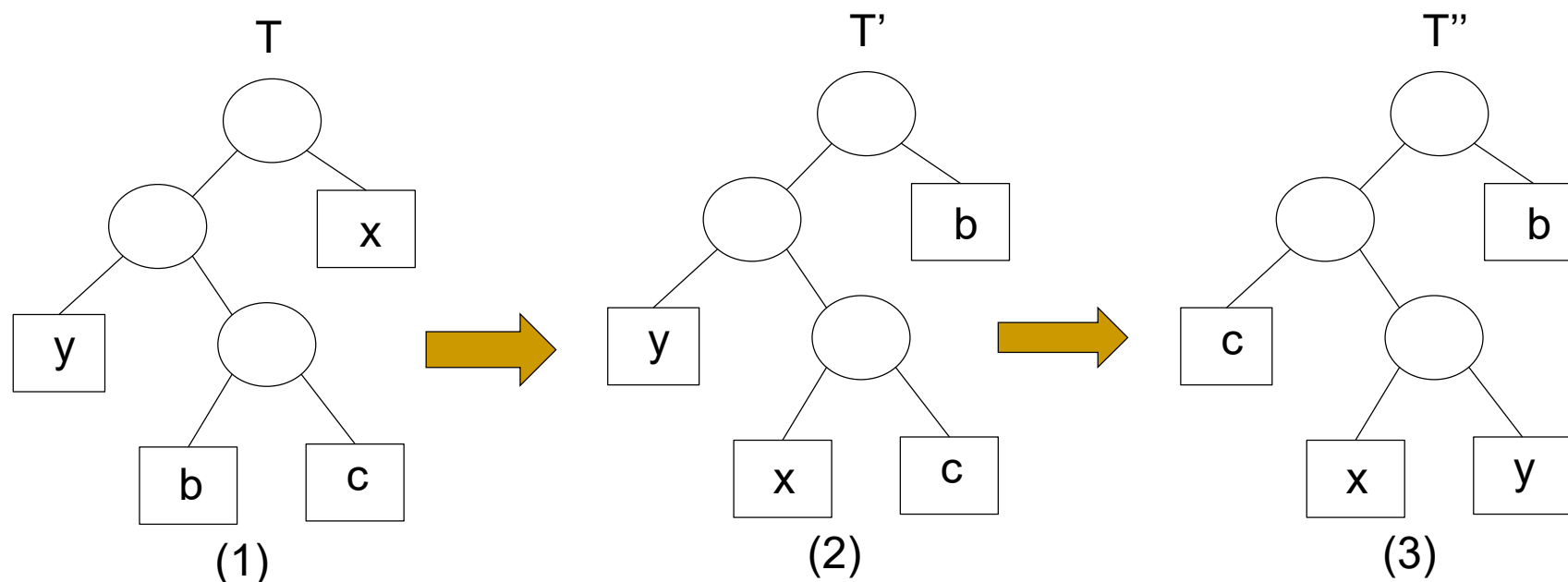


T和T'的平均码长之差为：

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\ &= (f(b) - f(x))(d_T(b) - d_T(x)) \geq 0 \end{aligned}$$



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

2) 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 $C'=C-\{x,y\}\cup\{z\}$ 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 $c \in C-\{x,y\}$ 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

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= f(x) + f(y) + f(z)d_{T'}(z) \end{aligned}$$

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

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

$$\begin{aligned} B(T''') &= B(T'') + f(x) + f(y) \\ &< B(T') + f(x) + f(y) \\ &= B(T) \end{aligned}$$

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。



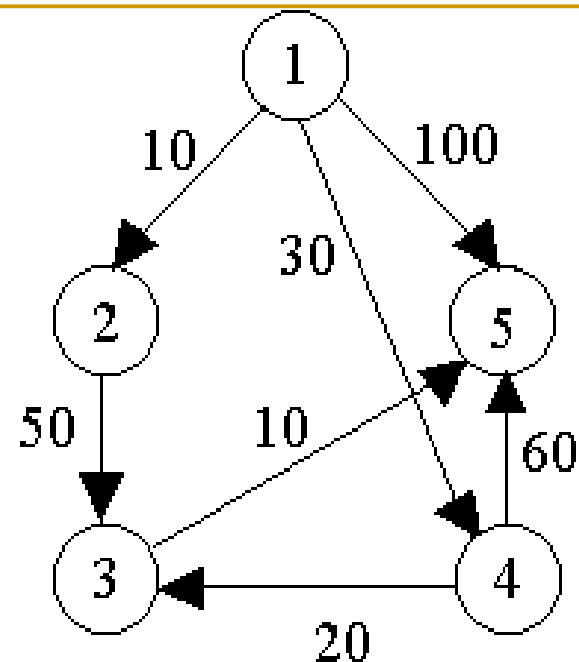
4.5 单源最短路径

- 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



4.5 单源最短路径

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 $\text{dist}[u]$ 。

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？

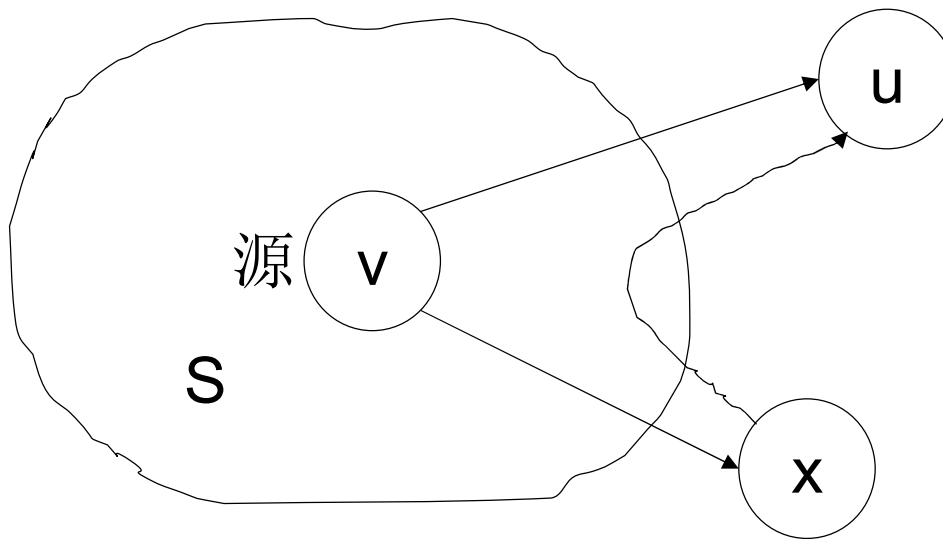
假设存在一条从源到 u 且长度比 $\text{dist}[u]$ 更短的路，设这条路初次走出 S 之外到达的顶点 $x \in V-S$ ，然后徘徊于 S 内外若干次，最后离开 S 到达 u 。



4.5 单源最短路径

2. 算法的正确性

1) 贪心选择性质

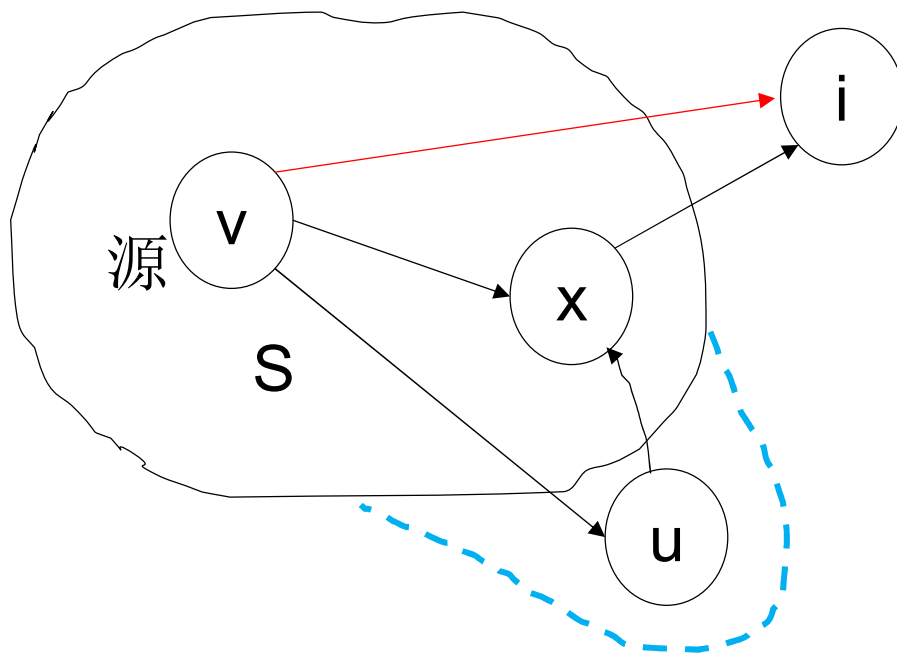




4.5 单源最短路径

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 $(u,v) \in E$ ，且 $u \in U$ ， $v \in V-U$ ，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, $V=\{1,2,\dots,n\}$ 。

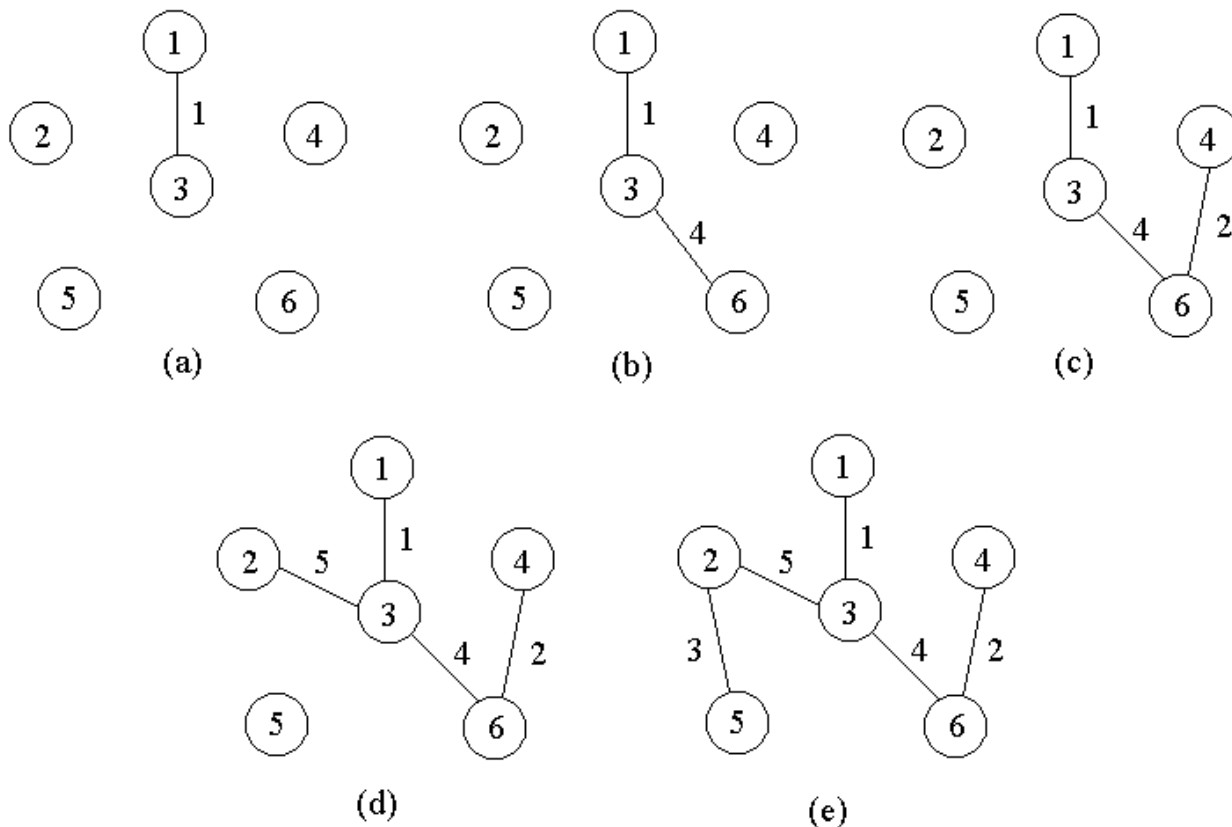
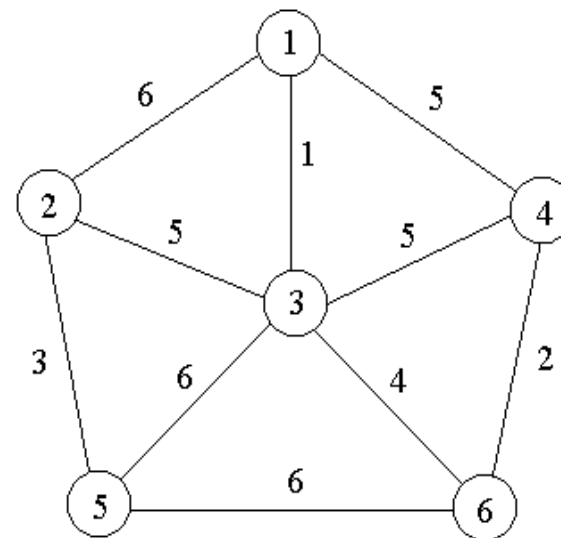
构造 G 的最小生成树的Prim算法的**基本思想**是：首先置 $S=\{1\}$, 然后, 只要 S 是 V 的真子集, 就作如下的**贪心选择**：选取满足条件 $i \in S, j \in V-S$, 且 $c[i][j]$ 最小的边, 将顶点 j 添加到 S 中。这个过程一直进行到 $S=V$ 时为止。

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 $i \in S, j \in V-S$ ，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 $O(n^2)$



Kruskal 算法

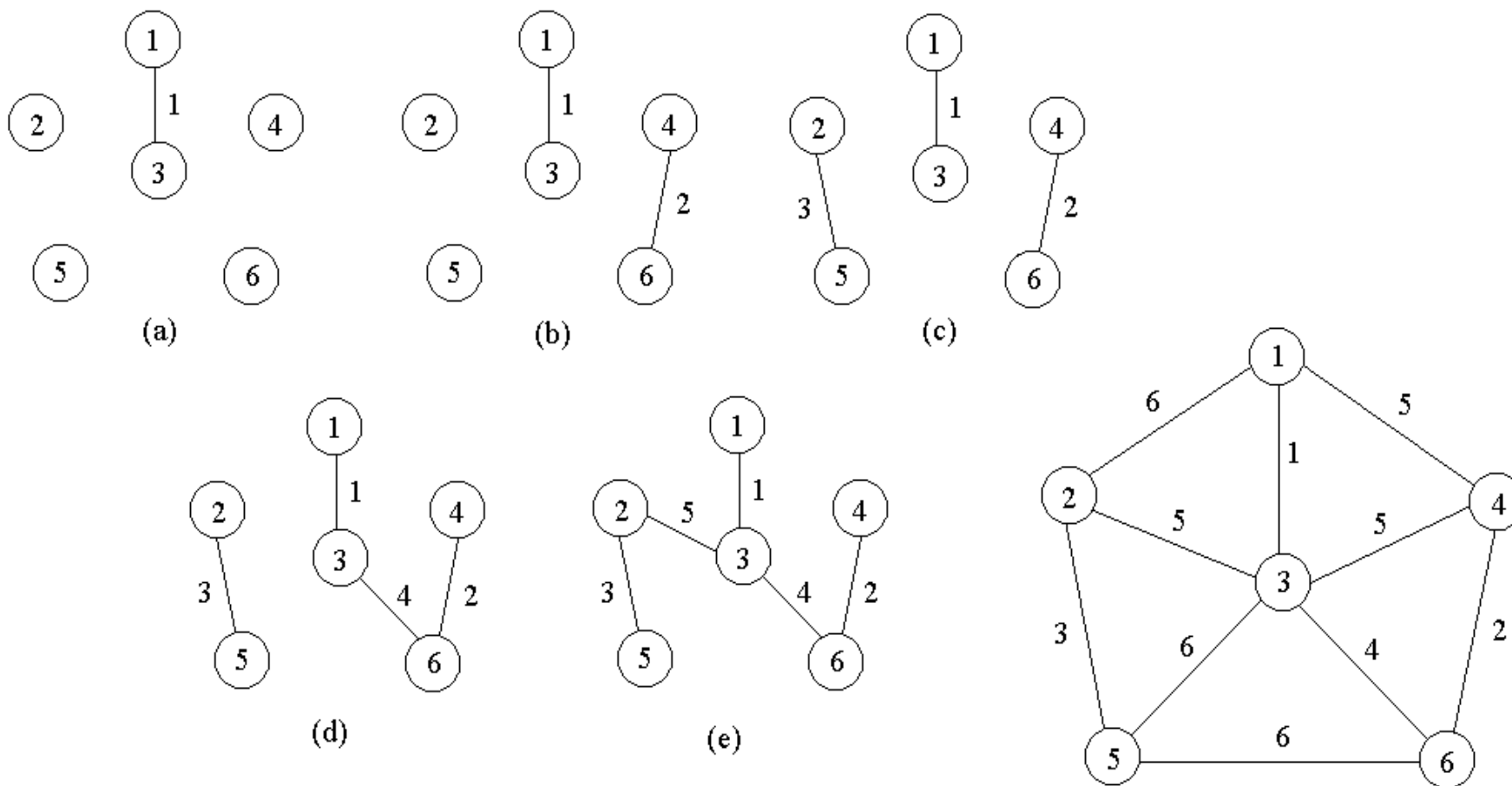
Kruskal算法构造G的最小生成树的基本思想是：

- 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 T_1 和 T_2 中的顶点时，就用边 (v,w) 将 T_1 和 T_2 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。





最小生成树

- 当图的边数为 e ，顶点数为 n 时，Kruskal算法所需的计算时间复杂性是 $O(e \log e)$ ，Prim算法是 $O(n^2)$ 。
- 当 $e = \Omega(n^2)$ 时，即边数多时，Prim算法优。
- 当 $e = o(n^2)$ 时，即边数少时，Kruskal算法优。



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 $[0, t_i]$ 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 **$O(n \log n)$** 。



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}。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。

