第三部分　算法设计方法

第13章　贪 婪 算 法

离开了数据结构的世界，现在进入算法设计方法的世界。

从本章开始，我们来研究一些算法设计方法。虽然设计一个好的求解算法更像是一门艺术，而不像是技术，但仍然存在一些行之有效的能够用于解决许多问题的算法设计方法，你可以使用这些方法来设计算法，并观察这些算法是如何工作的。一般情况下，为了获得较好的性能，必须对算法进行细致的调整。但是在某些情况下，算法经过调整之后性能仍无法达到要求，这时就必须寻求另外的方法来求解该问题。

本书的第13~17章提供了五种基本的算法设计方法：贪婪算法、分而治之算法、动态规划、回溯和分枝定界。而其他的常用高级方法如：线性规划、整数规划、遗传算法、模拟退火等则没有提及。有关这些方法的详细描述请参见相关书籍。

本章首先引入最优化的概念，然后介绍一种直观的问题求解方法：贪婪算法。最后，应用该算法给出货箱装船问题、背包问题、拓扑排序问题、二分覆盖问题、最短路径问题、最小代价生成树等问题的求解方案。

13.1　最优化问题

本章及后续章节中的许多例子都是最优化问题（optimization problem），每个最优化问题都包含一组限制条件（constraint）和一个优化函数（optimization function），符合限制条件的问题求解方案称为可行解（feasible solution），使优化函数取得最佳值的可行解称为最优解（optimal solution）。

例13-1　[渴婴问题] 有一个非常渴的、聪明的小婴儿，她可能得到的东西包括一杯水、一桶牛奶、多罐不同种类的果汁、许多不同的装在瓶子或罐子中的苏打水，即婴儿可得到 n 种不同的饮料。根据以前关于这 n 种饮料的不同体验，此婴儿知道这其中某些饮料更合自己的胃口，因此，婴儿采取如下方法为每一种饮料赋予一个满意度值：饮用1盎司第 i 种饮料，对它作出相对评价，将一个数值 s_i 作为满意度赋予第 i 种饮料。

通常，这个婴儿都会尽量饮用具有最大满意度值的饮料来最大限度地满足她解渴的需要，但是不幸的是：具有最大满意度值的饮料有时并没有足够的量来满足此婴儿解渴的需要。设 a_i 是第 i 种饮料的总量（以盎司为单位），而此婴儿需要 t 盎司的饮料来解渴，那么，需要饮用 n 种不同的饮料各多少量才能满足婴儿解渴的需求呢？

设各种饮料的满意度已知。令 x_i 为婴儿将要饮用的第 i 种饮料的量，则需要解决的问题是：找到一组实数 x_i（$1 \le i \le n$），使 $\sum_{i=1}^{n} s_i x_i$ 最大，并满足：$\sum_{i=1}^{n} x_i = t$ 及 $0 \le x_i \le a_i$。

需要指出的是：如果 $\sum_{i=1}^{n} a_i < t$，则不可能找到问题的求解方案，因为即使喝光所有的饮料也不能使婴儿解渴。

对上述问题精确的数学描述明确地指出了程序必须完成的工作，根据这些数学公式，可以对输入 / 输出作如下形式的描述：

输入：n，t，s_i，a_i（其中$1\le i\le n$，n 为整数，t、s_i、a_i 为正实数）。

输出：实数x_i（$1\le i\le n$），使$\sum_{i=1}^{n} s_i x_i$ 最大且$\sum_{i=1}^{n} x_i = t$（$0\le x_i\le a_i$）。如果$\sum_{i=1}^{n} a_i < t$，则输出适当的错误信息。

在这个问题中，限制条件是$\sum_{i=1}^{n} x_i = t$且$0\le x_i\le a_i$，$1\le i\le n$。而优化函数是$\sum_{i=1}^{n} s_i x_i$。任何满足限制条件的一组实数x_i都是可行解，而使$\sum_{i=1}^{n} s_i x_i$最大的可行解是最优解。

例13-2 [装载问题] 有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第 i 个货箱的重量为w_i（$1\le i\le n$），而货船的最大载重量为c，我们的目的是在货船上装入最多的货物。

这个问题可以作为最优化问题进行描述：设存在一组变量x_i，其可能取值为0或1。如x_i为0，则货箱i 将不被装上船；如x_i 为1，则货箱i 将被装上船。我们的目的是找到一组 x_i，使它满足限制条件 $\sum_{i=1}^{n} w_i x_i \le c$ 且$x_i \in \{0, 1\}, 1\le i\le n$。相应的优化函数是$\sum_{i=1}^{n} x_i$。

满足限制条件的每一组x_i都是一个可行解，能使$\sum_{i=1}^{n} x_i$取得最大值的方案是最优解。

例13-3 [最小代价通讯网络] 这个问题曾在例12-2介绍过。城市及城市之间所有可能的通信连接可被视作一个无向图，图的每条边都被赋予一个权值，权值表示建成由这条边所表示的通信连接所要付出的代价。包含图中所有顶点（城市）的连通子图都是一个可行解。设所有的权值都非负，则所有可能的可行解都可表示成无向图的一组生成树，而最优解是其中具有最小代价的生成树。

在这个问题中，需要选择一个无向图中的边集合的子集，这个子集必须满足如下限制条件：所有的边构成一个生成树。而优化函数是子集中所有边的权值之和。

13.2 算法思想

在贪婪算法（greedy method）中采用逐步构造最优解的方法。在每个阶段，都作出一个看上去最优的决策（在一定的标准下）。决策一旦作出，就不可再更改。作出贪婪决策的依据称为贪婪准则（greedy criterion）。

例13-4 [找零钱] 一个小孩买了价值少于1美元的糖，并将1美元的钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为 25美分、10美分、5美分、及1美分的硬币。售货员分步骤组成要找的零钱数，每次加入一个硬币。选择硬币时所采用的贪婪准则如下：每一次选择应使零钱数尽量增大。为保证解法的可行性（即：所给的零钱等于要找的零钱数），所选择的硬币不应使零钱总数超过最终所需的数目。

假设需要找给小孩67美分，首先入选的是两枚25美分的硬币，第三枚入选的不能是25美分的硬币，否则硬币的选择将不可行（零钱总数超过 67美分），第三枚应选择10美分的硬币，然后是5美分的，最后加入两个1美分的硬币。

贪婪算法有种直觉的倾向，在找零钱时，直觉告诉我们应使找出的硬币数目最少（至少是接近最少的数目）。可以证明采用上述贪婪算法找零钱时所用的硬币数目的确最少（见练习 1）。

例13-5 [机器调度] 现有n 件任务和无限多台的机器，任务可以在机器上得到处理。每件任务的开始时间为s_i，完成时间为f_i，$s_i < f_i$。$[s_i, f_i]$ 为处理任务i 的时间范围。两个任务i，j 重叠是

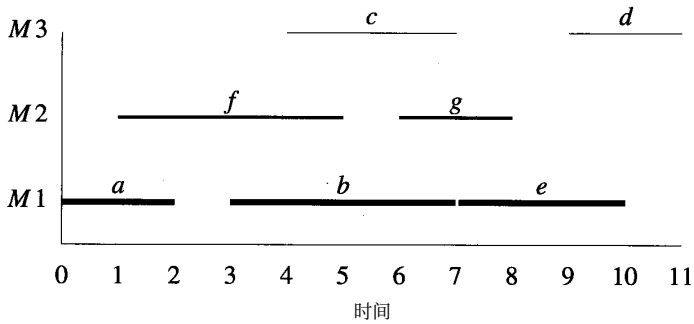
指两个任务的时间范围区间有重叠，而并非是指 i，j 的起点或终点重合。例如：区间[1，4]与区间[2，4]重叠，而与区间[4，7]不重叠。一个可行的任务分配是指在分配中没有两件重叠的任务分配给同一台机器。因此，在可行的分配中每台机器在任何时刻最多只处理一个任务。最优分配是指使用的机器最少的可行分配方案。

假设有n=7件任务，标号为a 到g。它们的开始与完成时间如图13-1a 所示。若将任务a分给机器$M1$，任务b 分给机器$M2$，. . .，任务g 分给机器$M7$，这种分配是可行的分配，共使用了七台机器。但它不是最优分配，因为有其他分配方案可使利用的机器数目更少，例如：可以将任务a、b、d分配给同一台机器，则机器的数目降为五台。

一种获得最优分配的贪婪方法是逐步分配任务。每步分配一件任务，且按任务开始时间的非递减次序进行分配。若已经至少有一件任务分配给某台机器，则称这台机器是旧的；若机器非旧，则它是新的。在选择机器时，采用以下贪婪准则：根据欲分配任务的开始时间，若此时有旧的机器可用，则将任务分给旧的机器。否则，将任务分配给一台新的机器。

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

a)



b)

图13-1 任务及三台机器的调度

a) 7个任务 b) 调度

根据例子中的数据，贪婪算法共分为n=7步，任务分配的顺序为a、f、b、c、g、e、d。第一步没有旧机器，因此将 a 分配给一台新机器（比如 $M1$）。这台机器在0到2时刻处于忙状态（如图13-1b所示）。在第二步，考虑任务f。由于当f 启动时旧机器仍处于忙状态，因此将f 分配给一台新机器(设为$M2$)。第三步考虑任务b, 由于旧机器$M1$在S_b=3时刻已处于闲状态，因此将b分配给$M1$执行，$M1$下一次可用时刻变成f_b=7，$M2$的可用时刻变成f_f=5。第四步，考虑任务c。由于没有旧机器在S_c=4时刻可用，因此将c 分配给一台新机器（$M3$），这台机器下一次可用时间为f_c=7。第五步考虑任务g，将其分配给机器$M2$，第六步将任务e 分配给机器$M1$，最后在第七步，任务2分配给机器$M3$。（注意：任务d 也可分配给机器$M2$）。

上述贪婪算法能导致最优机器分配的证明留作练习（练习 7）。可按如下方式实现一个复杂性为O($n$log$n$)的贪婪算法：首先采用一个复杂性为 O($n$log$n$)的排序算法（如堆排序）按 S_i 的递增次序排列各个任务，然后使用一个关于旧机器可用时间的最小堆。

例13-6 [最短路径] 给出一个如图13-2所示的有向网络，路径的长度定义为路径所经过的各边的耗费之和。要求找一条从初始顶点s到达目的顶点d的最短路径。

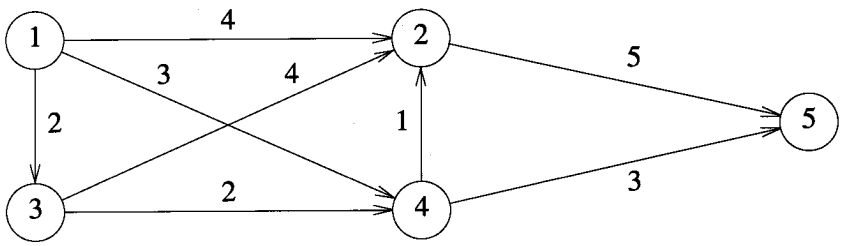


图13-2 有向图

贪婪算法分步构造这条路径，每一步在路径中加入一个顶点。假设当前路径已到达顶点q，且顶点q并不是目的顶点d。加入下一个顶点所采用的贪婪准则为：选择离q最近且目前不在路径中的顶点。

这种贪婪算法并不一定能获得最短路径。例如，假设在图13-2中希望构造从顶点1到顶点5的最短路径，利用上述贪婪算法，从顶点1开始并寻找目前不在路径中的离顶点1最近的顶点。到达顶点3，长度仅为2个单位，从顶点3可以到达的最近顶点为4，从顶点4到达顶点2，最后到达目的顶点5。所建立的路径为1,3,4,2,5，其长度为10。这条路径并不是有向图中从1到5的最短路径。事实上，有几条更短的路径存在，例如路径1，4，5的长度为6。

根据上面三个例子，回想一下前几章所考察的一些应用，其中有几种算法也是贪婪算法。例如，9.5.3节中的霍夫曼树算法，利用n-1步来建立最小加权外部路径的二叉树，每一步都将两棵二叉树合并为一棵，算法中所使用的贪婪准则为：从可用的二叉树中选出权重最小的两棵。

9.5.2节的LPT调度规则也是一种贪婪算法，它用n步来调度n个作业。首先将作业按时间长短排序，然后在每一步中为一个任务分配一台机器。选择机器所利用的贪婪准则为：使目前的调度时间最短。将新作业调度到最先完成的机器上（即最先空闲的机器）。

注意到在9.5.2节中的机器调度问题中，贪婪算法并不能保证最优，然而，那是一种直觉的倾向且一般情况下结果总是非常接近最优值。它利用的规则就是在实际环境中希望人工机器调度所采用的规则。算法并不保证得到最优结果，但通常所得结果与最优解相差无几，这种算法也称为启发式方法（heuristics)。因此LPT方法是一种启发式机器调度方法。定理9-2陈述了LPT调度的完成时间与最佳调度的完成时间之间的关系，因此LPT启发式方法具有限定性能（bounded performance）。具有限定性能的启发式方法称为近似算法（approximation algorithm）。

10.5.1节阐述了解决箱子装载问题的几种具有限定性能的启发式方法（即近似算法），每一种启发式方法都是贪婪启发法，9.5.2节的LPT法也是一种贪婪启发法。所有这些启发式方法都具有直觉倾向，并且在实际应用中，这些方法所得到的结果比使用9.5.2节中限定方法所得到的结果更接近最优解。

本章的其余部分将介绍几种贪婪算法的应用。在有些应用中，贪婪算法所产生的结果总是最优的解决方案。但对其他一些应用，生成的算法只是一种启发式方法，可能是也可能不是近似算法。

练习

1. 证明找零钱问题（例13-4）的贪婪算法总能产生具有最少硬币数的零钱。

2. 考虑例13-4的找零钱问题，假设售货员只有有限的 25美分， 10美分， 5美分和1美分的硬币，给出一种找零钱的贪婪算法。这种方法总能找出具有最少硬币数的零钱吗？证明结论。

3. 扩充例13-4的算法，假定售货员除硬币外还有50, 20, 10, 5, 和1美元的纸币，顾客买价格为x 美元和y 美分的商品时所付的款为u 美元和v 美分。算法总能找出具有最少纸币与硬币数目的零钱吗？证明结论。

4. 编写一个C++程序实现例13-4的找零钱算法。假设售货员具有面值为 100， 20， 10， 5和1美元的纸币和各种硬币。程序可包括输入模块（即输入所买商品的价格及顾客所付的钱数），输出模块（输出零钱的数目及要找的各种货币的数目）和计算模块（计算怎样给出零钱）。

5. 假设某个国家所使用硬币的币值为14,2,5和1分，则例13-4的贪婪算法总能产生具有最少硬币数的零钱吗？证明结论。

6. 1) 证明例13-5的贪婪算法总能找到最优任务分配方案。

2) 实现这种算法，使其复杂性为O($n$log$n$)（提示：根据完成时间建立最小堆）。

*7. 考察例13-5的机器调度问题。假定仅有一台机器可用，那么将选择最大数量的任务在这台机器上执行。例如，所选择的最大任务集合为 {a,b,e}。解决这种任务选择问题的贪婪算法可按步骤选择任务，每步选择一个任务，其贪婪准则如下：从剩下的任务中选择具有最小的完成时间且不会与现有任务重叠的任务。

1) 证明上述贪婪算法能够获得最优选择。

2) 实现该算法，其复杂性应为O($n$log$n$)。（提示：采用一个完成时间的最小堆）

13.3 应用

13.3.1 货箱装船

这个问题来自例13-2。船可以分步装载，每步装一个货箱，且需要考虑装载哪一个货箱。根据这种思想可利用如下贪婪准则：从剩下的货箱中，选择重量最小的货箱。这种选择次序可以保证所选的货箱总重量最小，从而可以装载更多的货箱。根据这种贪婪策略，首先选择最轻的货箱，然后选次轻的货箱，如此下去直到所有货箱均装上船或船上不能再容纳其他任何一个货箱。

例13-7 假设n=8, [w_1, ... w_8]=[100,200,50,90,150,50,20,80], c=400。利用贪婪算法时，所考察货箱的顺序为7,3,6,8,4,1,5,2。货箱7,3,6,8,4,1的总重量为390个单位且已被装载，剩下的装载能力为10个单位，小于剩下的任何一个货箱。在这种贪婪解决算法中得到[x_1, ..., x_8]=[1,0,1,1,0,1,1,1]且$\sum x_i$ =6。

定理13-1 利用贪婪算法能产生最佳装载。

证明 可以采用如下方式来证明贪婪算法的最优性：令 x=[x_1, ..., x_n]为用贪婪算法获得的解，令y=[y_1, ..., y_n]为任意一个可行解，只需证明 $\sum_{i=1}^{n} x_i \geq \sum_{i=1}^{n} y_i$。不失一般性，可以假设货箱都排好了序：即$w_i \leq w_{i+1}$ （$1 \leq i \leq n$）。然后分几步将y 转化为x，转换过程中每一步都产生一个可行的新y，且$\sum_{i=1}^{n} y_i$ 大于等于未转化前的值，最后便可证明 $\sum_{i=1}^{n} x_i \geq \sum_{j=1}^{n} y_i$。

根据贪婪算法的工作过程，可知在[0, n] 的范围内有一个k，使得x_i=1, $i \leq k$且x_i=0, $i > k$。寻

找[1,n]范围内最小的整数j，使得$x_j \neq y_j$。若没有这样的j 存在，则 $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$。如果有这样的$j$ 存在，则$j \leq k$，否则y 就不是一个可行解，因为$x_j \neq y_j$，x_j=1且y_j=0。令y_j=1，若结果得到的y 不是可行解，则在[j+1,n]范围内必有一个l 使得y_l=1。令y_l=0，由于$w_j \leq w_l$，则得到的y 是可行的。而且，得到的新y 至少与原来的y 具有相同数目的1。

经过数次这种转化，可将y 转化为x。由于每次转化产生的新y 至少与前一个y 具有相同数目的1，因此x 至少与初始的y 具有相同的数目1。

货箱装载算法的C++代码实现见程序13-1。由于贪婪算法按货箱重量递增的顺序装载，程序13-1首先利用间接寻址排序函数IndirectSort对货箱重量进行排序（见3.5节间接寻址的定义），随后货箱便可按重量递增的顺序装载。由于间接寻址排序所需的时间为 O($n$log$n$)（也可利用9.5.1节的堆排序及第14章的归并排序），算法其余部分所需时间为O(n)，因此程序13-1的总的复杂性为O($n$log$n$)。

程序13-1 货箱装船

```
template<class T>
void ContainerLoading(int x[],  T w[], T c, int n)
{// 货箱装船问题的贪婪算法
 // x[i] = 1 当且仅当货箱 i被装载,   1<=i<=n
 // c是船的容量, w 是货箱的重量

   // 对重量按间接寻址方式排序
   // t 是间接寻址表
   int *t = new int [n+1];
   IndirectSort(w, t, n);
   // 此时, w[t[i]] <= w[t[i+1]], 1<=i<n

   // 初始化 x
   for (int i = 1; i <= n; i++)
     x[i] = 0;

   // 按重量次序选择物品
   for (i = 1; i <= n && w[t[i]] <= c; i++) {
     x[t[i]] = 1;
     c -= w[t[i]];}  // 剩余容量

   delete [] t;
}
```

13.3.2 0/1背包问题

在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为w_i，价值为p_i。对于可行的背包装载，背包中物品的总重量不能超过背包的容量，最佳装载是指所装入的物品价值最高，即 $\sum_{i=1}^{n} p_i x_i$ 取得最大值。约束条件为 $\sum_{i=1}^{n} w_i x_i \leq c$ 和 x_i [0,1](1≤i≤n)。

在这个表达式中，需求出x_t 的值。x_i=1表示物品i 装入背包中，x_i=0 表示物品i 不装入背包。

0/1背包问题是一个一般化的货箱装载问题，即每个货箱所获得的价值不同。货箱装载问题转化为背包问题的形式为：船作为背包，货箱作为可装入背包的物品。

例13-8 在杂货店比赛中你获得了第一名，奖品是一车免费杂货。店中有 n 种不同的货物。规则规定从每种货物中最多只能拿一件，车子的容量为 c，物品 i 需占用 w_i 的空间，价值为 p_i。你的目标是使车中装载的物品价值最大。当然，所装货物不能超过车的容量，且同一种物品不得拿走多件。这个问题可仿照0/1背包问题进行建模，其中车对应于背包，货物对应于物品。

0/1背包问题有好几种贪婪策略，每个贪婪策略都采用多步过程来完成背包的装入。在每一步过程中利用贪婪准则选择一个物品装入背包。一种贪婪准则为：从剩余的物品中，选出可以装入背包的价值最大的物品，利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。这种策略不能保证得到最优解。例如，考虑 n=2, w=[100,10,10], p=[20,15,15], c=105。当利用价值贪婪准则时，获得的解为 x=[1,0,0]，这种方案的总价值为20。而最优解为[0,1,1]，其总价值为30。

另一种方案是重量贪婪准则是：从剩下的物品中选择可装入背包的重量最小的物品。虽然这种规则对于前面的例子能产生最优解，但在一般情况下则不一定能得到最优解。考虑 n=2, w=[10,20], p=[5,100], c=25。当利用重量贪婪策略时，获得的解为 x=[1,0], 比最优解[0,1]要差。

还可以利用另一方案，价值密度 p_i/w_i 贪婪算法，这种选择准则为：从剩余物品中选择可装入包的 p_i/w_i 值最大的物品，这种策略也不能保证得到最优解。利用此策略试解 n=3, w=[20,15,15], p=[40,25,25], c=30 时的最优解。

我们不必因所考察的几个贪婪算法都不能保证得到最优解而沮丧， 0/1背包问题是一个NP - 复杂问题（见9.5.2节对NP - 复杂问题的讨论）。对于这类问题，也许根本就不可能找到具有多项式时间的算法。虽然按 p_i/w_i 非递（增）减的次序装入物品不能保证得到最优解，但它是一个直觉上近似的解。我们希望它是一个好的启发式算法，且大多数时候能很好地接近最后算法。在600个随机产生的背包问题中，用这种启发式贪婪算法来解有 239题为最优解。有583个例子与最优解相差10%，所有600个答案与最优解之差全在25%以内。该算法能在O($n$log$n$)时间内获得如此好的性能。

我们也许会问，是否存在一个 x (x<100)，使得贪婪启发法的结果与最优值相差在 x%以内。答案是否定的。为说明这一点，考虑例子 n=2, w=[1,y], p=[10,9y], 和 c=y。贪婪算法结果为 x=[1,0], 这种方案的值为10。对于 y≥10/9，最优解的值为9y。因此，贪婪算法的值与最优解的差对最优解的比例为((9y-10)/9y*100)%，对于大的 y，这个值趋近于100%。

但是可以建立贪婪启发式方法来提供解，使解的结果与最优解的值之差在最优值的 x%(x<100) 之内。首先将最多 k 件物品放入背包，如果这 k 件物品重量大于 c，则放弃它。否则，剩余的容量用来考虑将剩余物品按 p_i/w_i 递减的顺序装入。通过考虑由启发法产生的解法中最多为 k 件物品的所有可能的子集来得到最优解。

例13-9 考虑 n=4, w=[2,4,6,7], p=[6,10,12,13], c=11。当 k=0时，背包按物品价值密度非递减顺序装入，首先将物品1放入背包，然后是物品2，背包剩下的容量为5个单元，剩下的物品没有一个合适的，因此解为 x=[1,1,0,0]。此解获得的价值为16。

现在考虑 k=1时的贪婪启发法。最初的子集为{1},{2},{3},{4}。子集{1},{2}产生与 k=0时相同的结果，考虑子集{3}，置 x_3 为1。此时还剩5个单位的容量，按价值密度非递增顺序来考虑如何利用这5个单位的容量。首先考虑物品1，它适合，因此取 x_1 为1，这时仅剩下3个单位容量

了，且剩余物品没有能够加入背包中的物品。通过子集{3}开始求解得结果为x=[1,0,1,0]，获得的价值为18。若从子集{4}开始，产生的解为x=[1,0,0,1]，获得的价值为19。考虑子集大小为0和1时获得的最优解为[1,0,0,1]。这个解是通过k=1的贪婪启发式算法得到的。

若k=2，除了考虑k<2的子集，还必需考虑子集{1,2},{1,3},{1,4},{2,3},{2,4}和{3,4}。首先从最后一个子集开始，它是不可行的，故将其抛弃，剩下的子集经求解分别得到如下结果：[1,1,0,0],[1,0,1,0],[1,0,0,1],[0,1,1,0]和[0,1,0,1]，这些结果中最后一个价值为23，它的值比k=0和k=1时获得的解要高，这个答案即为启发式方法产生的结果。

这种修改后的贪婪启发方法称为k阶优化方法（*k-optimal*）。也就是，若从答案中取出k件物品，并放入另外k件，获得的结果不会比原来的好，而且用这种方式获得的值在最优值的(100/(k+1))%以内。当k=1时，保证最终结果在最佳值的50%以内；当k=2时，则在33.33%以内等等，这种启发式方法的执行时间随k的增大而增加，需要测试的子集数目为$O(n^k)$，每一个子集所需时间为$O(n)$，因此当k>0时总的时间开销为$O(n^{k+1})$。

实际观察到的性能要好得多，图13-3给出了600种随机测试的统计结果。

	偏差百分比				
k	0	1%	5%	10%	25%
0	239	390	528	583	600
1	360	527	598	600	
2	483	581	600		

图13-3 600个例子中差值在x%以内的数目

13.3.3 拓扑排序

一个复杂的工程通常可以分解成一组小任务的集合，完成这些小任务意味着整个工程的完成。例如，汽车装配工程可分解为以下任务：将底盘放上装配线，装轴，将座位装在底盘上，上漆，装刹车，装门等等。任务之间具有先后关系，例如在装轴之前必须先将底板放上装配线。任务的先后顺序可用有向图表示——称为顶点活动（Activity On Vertex, AOV）网络。有向图的顶点代表任务，有向边(i, j)表示先后关系：任务j开始前任务i必须完成。图13-4显示了六个任务的工程，边（1,4）表示任务1在任务4开始前完成，同样边（4,6）表示任务4在任务6开始前完成，边（1,4）与（4,6）合起来可知任务1在任务6开始前完成，即前后关系是传递的。由此可知，边（1,4）是多余的，因为边（1,3）和（3,4）已暗示了这种关系。

在很多条件下，任务的执行是连续进行的，例如汽车装配问题或平时购买的标有“需要装配”的消费品（自行车、小孩的秋千装置，割草机等等）。我们可根据所建议的顺序来装配。在由任务建立的有向图中，边（i, j）表示在装配序列中任务i在任务j的前面，具有这种性质的序列称为拓扑序列（topological orders或topological sequences)。根据任务的有向图建立拓扑序列的过程称为拓扑排序（topological sorting）。

图13-4的任务有向图有多种拓扑序列，其中的三种为123456，132456和215346，序列142356就不是拓扑序列，因为在这个序列中任务4在3的前面，而任务有向图中的边为（3,4），这种序列与边（3,4）及其他边所指示的序列相矛盾。

可用贪婪算法来建立拓扑序列。算法按从左到右的步骤构造拓扑序列，每一步在排好的序

列中加入一个顶点。利用如下贪婪准则来选择顶点：从剩下的顶点中，选择顶点 w，使得w 不存在这样的入边（v,w），其中顶点v 不在已排好的序列结构中出现。注意到如果加入的顶点 w 违背了这个准则（即有向图中存在边（v,w）且v 不在已构造的序列中），则无法完成拓扑排序，因为顶点v 必须跟随在顶点w 之后。贪婪算法的伪代码如图13-5所示。while 循环的每次迭代代表贪婪算法的一个步骤。

现在用贪婪算法来求解图 13-4的有向图。首先从一个空序列 V开始，第一步选择V的第一个顶点。此时，在有向图中有两个候选顶点 1和2，若选择顶点 2，则序列V=2，第一步完成。第二步选择 V的第二个顶点，根据贪婪准则可知候选顶点为1和5，若选择5，则V=25。下一步，顶点1是唯一的候选，因此 V=251。第四步，顶点3是唯一的候选，因此把顶点 3加入V得到V=2513。在最后两步分别加入顶点 4和6，得V=251346。

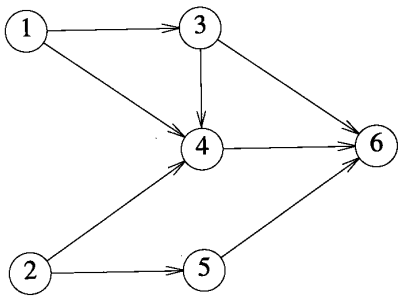


图13-4 任务有向图

```
设n是有向图中的顶点数
设V是一个空序列
while(true) {
  设w不存在入边（v,w），其中顶点v不在V中
  如果没有这样的w，break。
  把w添加到V的尾部
 }
if(V中的顶点数少于n) 算法失败
else V是一个拓扑序列
```

图13-5 拓扑排序

1. 贪婪算法的正确性

为保证贪婪算法算的正确性，需要证明： 1) 当算法失败时，有向图没有拓扑序列； 2) 若算法没有失败，V即是拓扑序列。2) 即是用贪婪准则来选取下一个顶点的直接结果， 1) 的证明见定理13-2，它证明了若算法失败，则有向图中有环路。若有向图中包含环 $q_j\ q_{j+1}\cdots q_k\ q_j$，则它没有拓扑序列，因为该序列暗示了$q_j$一定要在$q_j$开始前完成。

定理13-2 如果图13-5算法失败，则有向图含有环路。

证明 注意到当失败时| V |<n，且没有候选顶点能加入V中，因此至少有一个顶点q_1 不在V中，有向图中必包含边（q_2，q_1）且q_2 不在V中，否则，q_1 是可加入V的候选顶点。同样，必有边（q_3，q_2）使得q_3 不在V中，若$q_3=q_1$ 则$q_1\ q_2\ q_3$ 是有向图中的一个环路；若$q_3\neq q_1$，则必存在q_4 使（q_4，q_3）是有向图的边且q_4 不在V中，否则，q_3 便是V的一个候选顶点。若q_4 为q_1，q_2，q_3 中的任何一个，则又可知有向图含有环，因为有向图具有有限个顶点数 n，继续利用上述方法，最后总能找到一个环路。

2. 数据结构的选择

为将图13-5用C++代码来实现，必须考虑序列V的描述方法，以及如何找出可加入V的候选顶点。一种高效的实现方法是将序列 V用一维数组v 来描述的，用一个栈来保存可加入 V的候选顶点。另有一个一维数组InDegree，InDegree[j]表示与顶点j相连的节点i 的数目，其中顶点i不是V中的成员，它们之间的有向图的边表示为（ i, j）。当InDegree[j]变为0时表示j 成为一个候选节点。序列V初始时为空。InDegree[j]为顶点j 的入度。每次向V中加入一个顶点时，所有与新加入顶点邻接的顶点j，其InDegree[j]减1。

对于有向图13-4，开始时InDegree[1:6]=[0,0,1,3,1,3]。由于顶点1和2的InDegree值为0，因此它们是可加入V的候选顶点，由此，顶点1和2首先入栈。每一步，从栈中取出一个顶点将其加入V，同时减去与其邻接的顶点的 InDegree值。若在第一步时从栈中取出顶点 2并将其加入V，便得到了v[0]=2，和InDegree[1:6]=[0,0,1,2,0,3]。由于InDegree[5]刚刚变为0，因此将顶点5入栈。

程序13-2给出了相应的C++代码，这个代码被定义为 Network的一个成员函数。而且，它对于有无加权的有向图均适用。但若用于无向图（不论其有无加权）将会得到错误的结果，因为拓扑排序是针对有向图来定义的。为解决这个问题，利用同样的模板来定义成员函数 AdjacencyGraph, AdjacencyWGraph，LinkedGraph和LinkedWGraph。这些函数可重载Network中的函数并可输出错误信息。如果找到拓扑序列，则 Topological 函数返回true；若输入的有向图无拓扑序列则返回false。当找到拓扑序列时，将其返回到v[0:n-1]中。

3. Network:Topological 的复杂性

第一和第三个for循环的时间开销为$\Theta(n)$。若使用（耗费）邻接矩阵,则第二个for 循环所用的时间为$\Theta(n^2)$；若使用邻接链表,则所用时间为$\Theta(n+e)$。在两个嵌套的while 循环中，外层循环需执行n次，每次将顶点w 加入到v 中，并初始化内层while 循环。使用邻接矩阵时，内层while循环对于每个顶点w 需花费$\Theta(n)$的时间；若利用邻接链表, 则这个循环需花费d_w^{out}的时间，因此，内层while 循环的时间开销为$\Theta(n^2)$或$\Theta(n+e)$。所以，若利用邻接矩阵，程序 13-2的时间复杂性为$\Theta(n^2)$，若利用邻接链表则为$\Theta(n+e)$。

程序13-2 拓扑排序

```
bool Network::Topological(int v[])
{// 计算有向图中顶点的拓扑次序
 // 如果找到了一个拓扑次序，则返回 true，此时，在v[0:n-1]中记录拓扑次序
 // 如果不存在拓扑次序，则返回 false

  int n = Vertices();

  // 计算入度
  int *InDegree = new int [n+1];
  InitializePos(); // 图遍历器数组
  for (int i = 1; i <= n; i++) // 初始化
    InDegree[i] = 0;
  for (i = 1; i <= n; i++) {// 从i 出发的边
    int u = Begin(i);
    while (u) {
      InDegree[u]++;
```

```
            u = NextVertex(i);}
        }

    // 把入度为0的顶点压入堆栈
    LinkedStack<int> S;
    for (i = 1; i <= n; i++)
        if (!InDegree[i]) S.Add(i);

    // 产生拓扑次序
    i = 0;  // 数组 v   的游标
    while (!S.IsEmpty()) {// 从堆栈中选择
        int w;              // 下一个顶点
        S.Delete(w);
        v[i++] = w;
        int u = Begin(w);
        while (u) {// 修改入度
            InDegree[u]--;
            if (!InDegree[u]) S.Add(u);
            u = NextVertex(w);}
        }

    DeactivatePos();
    delete [] InDegree;
    return (i == n);
}
```

13.3.4 二分覆盖

二分图（见例12-3）是一个无向图，它的n 个顶点可二分为集合A和集合B，且同一集合中的任意两个顶点在图中无边相连（即任何一条边都是一个顶点在集合A中，另一个在集合B中）。当且仅当B中的每个顶点至少与A中一个顶点相连时，A的一个子集A'覆盖集合B（或简单地说，A'是一个覆盖）。覆盖A'的大小即为A'中的顶点数目。当且仅当A'是覆盖B的子集中最小的时，A'为最小覆盖。

例13-10 考察如图13-6所示的具有17个顶点的二分图，A={1, 2, 3, 16, 17}和B={4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}，子集A'={1,16,17}是B的最小覆盖。

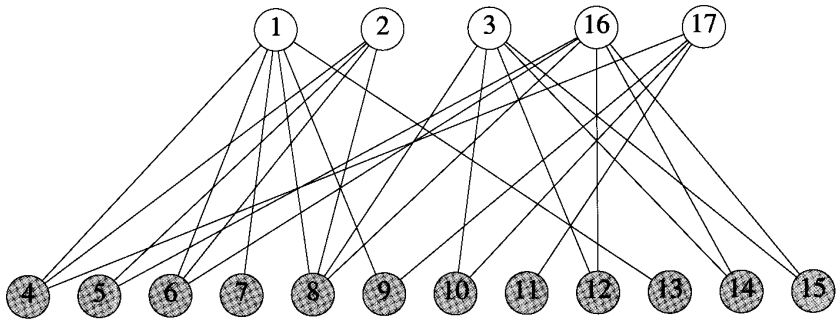


图13-6 例13-10所使用的图

在二分图中寻找最小覆盖的问题为二分覆盖（bipartite-cover）问题。在例12-3中说明了最小覆盖是很有用的，因为它能解决“在会议中使用最少的翻译人员进行翻译”这一类的问题。二分覆盖问题类似于集合覆盖（set-cover）问题。在集合覆盖问题中给出了 k 个集合$S=\{S_1, S_2, \cdots, S_k\}$，每个集合$S_i$中的元素均是全集$U$中的成员。当且仅当 $\bigcup_{i\in S'} S_i = U$ 时，S的子集S'覆盖U，S'中的集合数目即为覆盖的大小。当且仅当没有能覆盖 U的更小的集合时，称S'为最小覆盖。可以将集合覆盖问题转化为二分覆盖问题（反之亦然），即用A的顶点来表示$S_1, \cdots, S_k$，B中的顶点代表U中的元素。当且仅当 S的相应集合中包含 U中的对应元素时，在A与B的顶点之间存在一条边。

例13-11 令$S=\{S_1, \ldots, S_5\}$, $U=\{4, 5, \ldots, 15\}$, $S_1=\{4, 6, 7, 8, 9, 13\}$，$S_2=\{4, 5, 6, 8\}$，$S_3=\{8, 10, 12, 14, 15\}$，$S_4=\{5, 6, 8, 12, 14, 15\}$，$S_5=\{4, 9, 10, 11\}$。$S'=\{S_1, S_4, S_5\}$是一个大小为3的覆盖，没有更小的覆盖，$S'$即为最小覆盖。这个集合覆盖问题可映射为图13-6的二分图，即用顶点1，2，3，16和17分别表示集合S_1，S_2，S_3，S_4和S_5，顶点j表示集合中的元素j，$4\leq j\leq 15$。

集合覆盖问题为NP-复杂问题。由于集合覆盖与二分覆盖是同一类问题，二分覆盖问题也是NP-复杂问题。因此可能无法找到一个快速的算法来解决它，但是可以利用贪婪算法寻找一种快速启发式方法。一种可能是分步建立覆盖 A'，每一步选择A中的一个顶点加入覆盖。顶点的选择利用贪婪准则：从A中选取能覆盖B中还未被覆盖的元素数目最多的顶点。

例13-12 考察图13-6所示的二分图，初始化$A'=\phi$且B中没有顶点被覆盖，顶点1和16均能覆盖B中的六个顶点，顶点3覆盖五个，顶点2和17分别覆盖四个。因此，在第一步往A'中加入顶点1或16，若加入顶点16，则它覆盖的顶点为{5,6,8,12,14,15}，未覆盖的顶点为{4,7,9,10,11,13}。顶点1能覆盖其中四个顶点（{4,7,9,13}），顶点2 覆盖一个({4})，顶点3覆盖一个（{10}），顶点16覆盖零个，顶点17覆盖四个{4,9,10,11}。下一步可选择1或17加入A'。若选择顶点1，则顶点{10,11} 仍然未被覆盖，此时顶点1，2，16不覆盖其中任意一个，顶点3覆盖一个，顶点17覆盖两个，因此选择顶点17，至此所有顶点已被覆盖，得$A'=\{16,1,17\}$。

图13-7给出了贪婪覆盖启发式方法的伪代码，可以证明：1) 当且仅当初始的二分图没有覆盖时，算法找不到覆盖；2) 启发式方法可能找不到二分图的最小覆盖。

```
A' =ϕ
while（更多的顶点可被覆盖）
 把能覆盖未被覆盖的顶点数目最多的顶点加入A'
if（有些顶点未被覆盖）失败
else（找到一个覆盖）
```

图13-7 贪婪覆盖启发式方法的伪代码

1. 数据结构的选取及复杂性分析

为实现图13-7的算法，需要选择A'的描述方法及考虑如何记录A中节点所能覆盖的B中未覆盖节点的数目。由于对集合A'仅使用加法运算，则可用一维整型数组C来描述A'，用m来记录A'中元素个数。将A'中的成员记录在$C[0:m-1]$中。

对于A中顶点i，令New_i为i所能覆盖的B中未覆盖的顶点数目。逐步选择New_i值最大的顶

点。由于一些原来未被覆盖的顶点现在被覆盖了，因此还要修改各 New_i 值。在这种更新中，检查B中最近一次被V覆盖的顶点，令j 为这样的一个顶点，则A中所有覆盖j 的顶点的New_i 值均减1。

例13-13 考察图13-6，初始时(New_1, New_2, New_3, New_{16}, New_{17})=(6,4,5,6,4)。假设在例13-12中，第一步选择顶点16，为更新New_i 的值检查B中所有最近被覆盖的顶点，这些顶点为5,6,8,12,14和15。当检查顶点5时，将顶点2和16的New_i 值分别减1，因为顶点5不再是被顶点2和16覆盖的未覆盖节点；当检查顶点6时，顶点1,2,和16的相应值分别减1；同样，检查顶点8时，1，2，3和16的值分别减1；当检查完所有最近被覆盖的顶点，得到的 New_i 值为（4，1，0，4）。下一步选择顶点1，最新被覆盖的顶点为4，7，9和13；检查顶点4时，New_1，New_2，和New_{17} 的值减1；检查顶点7时，New_1 的值减1，因为顶点1是覆盖7的唯一顶点。

为了实现顶点选取的过程，需要知道 New_i 的值及已被覆盖的顶点。可利用一个二维数组来达到这个目的，New是一个整型数组，New[i] 即等于New_i，且cov为一个布尔数组。若顶点i未被覆盖则cov[i]等于false，否则cov[i]为true。现将图13-7的伪代码进行细化得到图13-8。

```
m=0; //当前覆盖的大小
对于A中的所有i，New[i]=Degree[i]
对于B中的所有i，Cov[i]=false
while (对于A中的某些i,New[i]>0) {
  设v是具有最大的New[i]的顶点；
  C[m++]=v;
  for (所有邻接于v的顶点j) {
   if (!Cov[j]) {
     Cov[j]= true;
     对于所有邻接于j的顶点，使其New[k]减1
   }}}
if (有些顶点未被覆盖) 失败
else 找到一个覆盖
```

图13-8 图13-7的细化

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，则需花 $\Theta(n^2)$ 的时间来寻找图中的边，若用邻接链表，则需 $\Theta(n+e)$ 的时间。实际更新时间根据描述方法的不同为O(n^2) 或O($n+e$)。

逐步选择顶点所需时间为 $\Theta(SizeOfA)$，其中$SizeOfA$=| A |。因为A的所有顶点都有可能被选择，因此所需步骤数为 O($SizeOfA$)，覆盖算法总的复杂性为 O($SizeOfA^2+n^2$)=O(n^2)或O($SizeOf A^2+n+e$)。

2. 降低复杂性

通过使用有序数组New_i、最大堆或最大选择树（max selection tree）可将每步选取顶点v的复杂性降为$\Theta(1)$。但利用有序数组，在每步的最后需对 New_i 值进行重新排序。若使用箱子排序，则这种排序所需时间为 $\Theta(SizeOfB)$ ($SizeOfB$=|B|)（见3.8.1节箱子排序）。由于一般$SizeOfB$比$SizeOfA$大得多，因此有序数组并不总能提高性能。

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$\Theta(1)$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($n+e$)。

若利用最大选择树，每次更新New值时需要重建选择树，所需时间为Θ(log *SizeOfA*)。重建的最好时机是在每步结束时，而不是在每次 New值减1时，需要重建的次数为O(e)，因此总的重建时间为O(e log *SizeOfA*)，这个时间比最大堆的重建时间长一些。

然而，通过维持具有相同New值的顶点箱子，也可获得和利用最大堆时相同的时间限制。由于New的取值范围为0到*SizeOfB*，需要*SizeOfB*+1个箱子，箱子*i* 是一个双向链表，链接所有New值为*i* 的顶点。在某一步结束时，假如New[6]从12变到4，则需要将它从第12个箱子移到第4个箱子。利用模拟指针及一个节点数组 node（其中node[i]代表顶点i，node[i].left和node[i].right为双向链表指针），可将顶点6从第12个箱子移到第4个箱子，从第12个箱子中删除node[0]并将其插入第4个箱子。利用这种箱子模式，可得覆盖启发式算法的复杂性为 O(n^2)或O($n+e$)。（取决于利用邻接矩阵还是线性表来描述图）。

3. 双向链接箱子的实现

为了实现上述双向链接箱子，图13-9定义了类Undirected的私有成员。NodeType是一个具有私有整型成员left和right的类，它的数据类型是双向链表节点，程序13-3给出了Undirected的私有成员的代码。

```
void CreateBins (int b, int n)
创建b个空箱子和n个节点

void DestroyBins() { delete [] node;
                     delete [] bin;}

void InsertBins(int b, int v)
 在箱子b中添加顶点v

void MoveBins(int bMax, int ToBin, int v)
  从当前箱子中移动顶点v到箱子ToBin

int *bin;
 bin[i]指向代表该箱子的双向链表的首节点

NodeType *node;
 node[i]代表存储顶点i的节点
```

图13-9 实现双向链接箱子所需的Undirected私有成员

程序13-3 箱子函数的定义

```
void Undirected::CreateBins(int b, int n)
{// 创建b个空箱子和n个节点
  node = new NodeType [n+1];
```

```
    bin = new int [b+1];
    // 将箱子置空
    for (int i = 1; i <= b; i++)
      bin[i] = 0;
  }

  void Undirected::InsertBins(int b, int v)
  {// 若b不为0，则将 v 插入箱子 b
    if (!b) return;  // b为 0，不插入
    node[v].left = b; // 添加在左端
    if (bin[b]) node[bin[b]].left = v;
    node[v].right = bin[b];
    bin[b] = v;
  }

  void Undirected::MoveBins(int bMax, int ToBin, int v)
  {// 将顶点 v 从其当前所在箱子移动到 ToBin.
    // v的左、右节点
    int l = node[v].left;
    int r = node[v].right;

    // 从当前箱子中删除
    if (r) node[r].left = node[v].left;
    if (l > bMax || bin[l] != v) // 不是最左节点
      node[l].right = r;
    else bin[l] = r;            // 箱子 l的最左边

    // 添加到箱子 ToBin
    InsertBins(ToBin, v);
  }
```

函数CreateBins动态分配两个数组：node和bin，node[i]表示顶点i, bin[i]指向其New值为i的双向链表的顶点,for循环将所有双向链表置为空。

如果b≠0，函数InsertBins 将顶点v 插入箱子b 中。因为b 是顶点v 的New 值，b=0意味着顶点v 不能覆盖B中当前还未被覆盖的任何顶点，所以，在建立覆盖时这个箱子没有用处，故可以将其舍去。当b≠0时，顶点n 加入New 值为b 的双向链表箱子的最前面，这种加入方式需要将node[v] 加入bin[b] 中第一个节点的左边。由于表的最左节点应指向它所属的箱子，因此将它的node[v].left 置为b。若箱子不空，则当前第一个节点的 left 指针被置为指向新节点。node[v] 的右指针被置为bin[b]，其值可能为0或指向上一个首节点的指针。最后， bin[b]被更新为指向表中新的第一个节点。

MoveBins 将顶点v 从它在双向链表中的当前位置移到New 值为ToBin 的位置上。其中存在bMax，使得对所有的箱子bin[j]都有：如j>bMax，则bin[j]为空。代码首先确定node[v]在当前双向链表中的左右节点，接着从双链表中取出 node[v]，并利用InsertBins函数将其重新插入到bin[ToBin]中。

4. Undirected::BipartiteCover的实现

函数的输入参数L用于分配图中的顶点（分配到集合A或B）。L[i]=1表示顶点i在集合A中，L[i]=2则表示顶点在B中。函数有两个输出参数：C和m，m为所建立的覆盖的大小， C[0,m-1]

是*A*中形成覆盖的顶点。若二分图没有覆盖，函数返回 false；否则返回true。完整的代码见程序13-4。

程序13-4 构造贪婪覆盖

```
bool Undirected::BipartiteCover(int L[], int C[], int& m)
{// 寻找一个二分图覆盖
 // L 是输入顶点的标号, L[i] = 1 当且仅当 i 在A中
 // C 是一个记录覆盖的输出数组
 // 如果图中不存在覆盖，则返回false
 // 如果图中有一个覆盖，则返回true;
 // 在 m中返回覆盖的大小; 在C[0:m-1]中返回覆盖

  int n = Vertices();

  // 插件结构
  int SizeOfA = 0;
  for (int i = 1; i <= n; i++) // 确定集合 A的大小
    if (L[i] == 1) SizeOfA++;
  int SizeOfB = n - SizeOfA;
  CreateBins(SizeOfB, n);
  int *New = new int [n+1]; //顶点i覆盖了B中New[i]个未被覆盖的顶点
  bool *Change = new bool [n+1]; // Change[i]为true当且仅当 New[i] 已改变
  bool *Cov = new bool [n+1]; // Cov[i] 为 true 当且仅当顶点 i 被覆盖
  InitializePos();
  LinkedStack<int> S;

  // 初始化
  for (i = 1; i <= n; i++) {
    Cov[i] = Change[i] = false;
    if (L[i] == 1) {// i 在 A中
      New[i] = Degree(i); // i 覆盖了这么多
      InsertBins(New[i], i);}}

  // 构造覆盖
  int covered = 0,       // 被覆盖的顶点
     MaxBin = SizeOfB;  // 可能非空的最大箱子
  m = 0;                 // C的游标
  while (MaxBin > 0) {   // 搜索所有箱子
    // 选择一个顶点
    if (bin[MaxBin]) {      // 箱子不空
      int v = bin[MaxBin]; // 第一个顶点
      C[m++] = v;          // 把v 加入覆盖
      // 标记新覆盖的顶点
      int j = Begin(v), k;
      while (j) {
        if (!Cov[j]) {// j尚未被覆盖
```

```
            Cov[j] = true;
            covered++;
            // 修改 New
            k = Begin(j);
            while (k) {
              New[k]--;    // j 不计入在内
              if (!Change[k]) {
                S.Add(k); // 仅入栈一次
                Change[k] = true;}
              k = NextVertex(j);}
            }
          j = NextVertex(v);}

        // 更新箱子
        while (!S.IsEmpty()) {
          S.Delete(k);
          Change[k] = false;
          MoveBins(SizeOfB, New[k], k);}
        }
      else MaxBin--;
      }

    DeactivatePos();
    DestroyBins();
    delete [] New;
    delete [] Change;
    delete [] Cov;
    return (covered == SizeOfB);
  }
```

程序13-4首先计算出集合A和B的大小、初始化必要的双向链表结构、创建三个数组、初始化图遍历器、并创建一个栈。然后将数组 Cov和Change初始化为false，并将A中的顶点根据它们覆盖B中顶点的数目插入到相应的双向链表中。

为了构造覆盖，首先按 SizeOfB 递减至1的顺序检查双向链表。当发现一个非空的表时，就将其第一个顶点v 加入到覆盖中，这种策略即为选择具有最大New 值的顶点。将所选择的顶点加入覆盖数组 C并检查B中所有与它邻接的顶点。若顶点 j 与v 邻接且还未被覆盖，则将Cov[j]置为true，表示顶点j 现在已被覆盖，同时将已被覆盖的B中的顶点数目加1。由于j 是最近被覆盖的，所有A中与j 邻接的顶点的New 值减1。下一个while 循环降低这些New 值并将New 值被降低的顶点保存在一个栈中。当所有与顶点 v邻接的顶点的Cov 值更新完毕后，New值反映了A中每个顶点所能覆盖的新的顶点数，然而 A中的顶点由于New 值被更新，处于错误的双向链表中，下一个while 循环则将这些顶点移到正确的表中。

13.3.5 单源最短路径

在这个问题中，给出有向图G，它的每条边都有一个非负的长度（耗费） a[i][j]，路径的长度即为此路径所经过的边的长度之和。对于给定的源顶点 s，需找出从它到图中其他任意顶点（称为目的）的最短路径。图13-10a 给出了一个具有五个顶点的有向图，各边上的数即为长度。假设源顶点s 为1，从顶点1出发的最短路径按路径长度顺序列在图13-10b 中，每条路径前

面的数字为路径的长度。

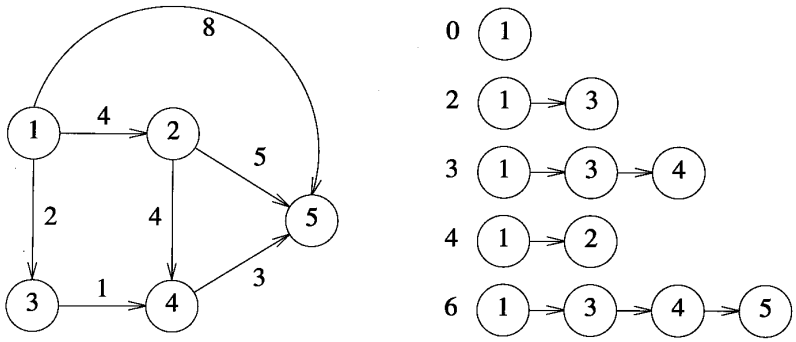


图13-10 最短路径举例

a) 图 b) 最短路径

利用E. Dijkstra发明的贪婪算法可以解决最短路径问题，它通过分步方法求出最短路径。每一步产生一个到达新的目的顶点的最短路径。下一步所能达到的目的顶点通过如下贪婪准则选取：在还未产生最短路径的顶点中，选择路径长度最短的目的顶点。也就是说， Dijkstra的方法按路径长度顺序产生最短路径。

首先最初产生从s 到它自身的路径，这条路径没有边，其长度为0。在贪婪算法的每一步中，产生下一个最短路径。一种方法是在目前已产生的最短路径中加入一条可行的最短的边，结果产生的新路径是原先产生的最短路径加上一条边。这种策略并不总是起作用。另一种方法是在目前产生的每一条最短路径中，考虑加入一条最短的边，再从所有这些边中先选择最短的，这种策略即是Dijkstra算法。

可以验证按长度顺序产生最短路径时，下一条最短路径总是由一条已产生的最短路径加上一条边形成。实际上，下一条最短路径总是由已产生的最短路径再扩充一条最短的边得到的，且这条路径所到达的顶点其最短路径还未产生。例如在图 13-10中，b 中第二条路径是第一条路径扩充一条边形成的；第三条路径则是第二条路径扩充一条边；第四条路径是第一条路径扩充一条边；第五条路径是第三条路径扩充一条边。

通过上述观察可用一种简便的方法来存储最短路径。可以利用数组 p，p[i]给出从s 到达i 的路径中顶点i 前面的那个顶点。在本例中p[1:5]=[0,1,1,3,4]。从s 到顶点i 的路径可反向创建。从i 出发按p[i],p[p[i]],p[p[p[i]]], …的顺序，直到到达顶点s 或0。在本例中，如果从i=5开始，则顶点序列为p[i]=4, p[4]=3, p[3]=1=s，因此路径为1,3,4,5。

为能方便地按长度递增的顺序产生最短路径，定义 d[i]为在已产生的最短路径中加入一条最短边的长度，从而使得扩充的路径到达顶点i。最初，仅有从s 到s 的一条长度为0的路径，这时对于每个顶点i，d[i]等于a[s][i]（a 是有向图的长度邻接矩阵）。为产生下一条路径，需要选择还未产生最短路径的下一个节点，在这些节点中 d值最小的即为下一条路径的终点。当获得一条新的最短路径后，由于新的最短路径可能会产生更小的 d值，因此有些顶点的d值可能会发生变化。

综上所述，可以得到图 13-11所示的伪代码， 1) 将与s 邻接的所有顶点的p 初始化为s，这个初始化用于记录当前可用的最好信息。也就是说，从 s 到i 的最短路径，即是由s到它自身那

条路径再扩充一条边得到。当找到更短的路径时， p[i]值将被更新。若产生了下一条最短路径，需要根据路径的扩充边来更新d 的值。

1) 初始化$d[i]=a[s][i]$（$1\le i\le n$），
对于邻接于s的所有顶点i，置$p[i]=s$, 对于其余的顶点置$p[i]=0$；
对于$p[i]\neq 0$的所有顶点建立L表。
2) 若L为空，终止，否则转至3)。
3) 从L中删除d值最小的顶点。
4) 对于与i 邻接的所有还未到达的顶点j，更新$d[j]$值为$min\{d[j], d[i]+a[i][j]\}$；若$d[j]$发生了变化且$j$ 还未在L中，则置$p[j]=1$，并将j 加入L，转至2。

图13-11 最短路径算法的描述

1. 数据结构的选择

我们需要为未到达的顶点列表L选择一个数据结构。从L中可以选出d 值最小的顶点。如果L用最小堆（见9.3节）来维护，则这种选取可在对数时间内完成。由于 3) 的执行次数为O(n)，所以所需时间为O(nlogn)。由于扩充一条边产生新的最短路径时，可能使未到达的顶点产生更小的d 值，所以在4) 中可能需要改变一些d 值。虽然算法中的减操作并不是标准的最小堆操作，但它能在对数时间内完成。由于执行减操作的总次数为： O(有向图中的边数)= $O(n^2)$，因此执行减操作的总时间为$O(n^2 \log n)$。

若L用无序的链表来维护，则3) 与4) 花费的时间为$O(n^2)$，3) 的每次执行需O(|L |)=O(n)的时间，每次减操作需$\Theta(1)$的时间（需要减去d[j] 的值，但链表不用改变）。

利用无序链表将图 13-11的伪代码细化为程序 13-5，其中使用了 Chain(见程序 3-8)和ChainIterator类（见程序3-18）。

程序13-5 最短路径程序

```
template<class T>
void AdjacencyWDigraph<T>::ShortestPaths(int s, T d[], int p[])
{// 寻找从顶点 s出发的最短路径, 在d中返回最短距离
 // 在p中返回前继顶点
  if (s < 1 || s > n) throw OutOfBounds();
  Chain<int> L; // 路径可到达顶点的列表
  ChainIterator<int> I;
  // 初始化d, p, L
  for (int i = 1; i <= n; i++){
    d[i] = a[s][i];
    if (d[i] == NoEdge) p[i] = 0;
    else {p[i] = s;
          L.Insert(0,i);}
    }

  // 更新d, p
  while (!L.IsEmpty()) {// 寻找具有最小d的顶点v
    int *v = I.Initialize(L);
```

```
        int *w = I.Next();
        while (w) {
          if (d[*w] < d[*v]) v = w;
          w = I.Next();}

        // 从L中删除通向顶点v的下一最短路径并更新d
        int i = *v;
        L.Delete(*v);
        for (int j = 1; j <= n; j++) {
          if (a[i][j] != NoEdge && (!p[j] ||
                  d[j] > d[i] + a[i][j])) {
            // 减小d[j]
            d[j] = d[i] + a[i][j];
            // 将j加入L
            if (!p[j]) L.Insert(0,j);
            p[j] = i;}
          }
        }
    }
```

若NoEdge足够大，使得没有最短路径的长度大于或等于 NoEdge，则最后一个for 循环的if条件可简化为：

```
if (d[j] > d[i] + a[i][j]))
```

NoEdge 的值应在能使d[j]+a[i][j] 不会产生溢出的范围内。

2. 复杂性分析

程序13-5的复杂性是$O(n^2)$，任何最短路径算法必须至少对每条边检查一次，因为任何一条边都有可能在最短路径中。因此这种算法的最小可能时间为 O(e)。由于使用耗费邻接矩阵来描述图，仅决定哪条边在有向图中就需 $O(n^2)$的时间。因此，采用这种描述方法的算法需花费$O(n^2)$的时间。不过程序13-5作了优化（常数因子级）。即使改变邻接表，也只会使最后一个 for 循环的总时间降为O(e)（因为只有与i 邻接的顶点的d 值改变）。从L中选择及删除最小距离的顶点所需总时间仍然是$O(n^2)$。

13.3.6 最小耗费生成树

在例12-2及13-3中已考察过这个问题。因为具有 n 个顶点的无向网络G的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。至少可以采用三种不同的贪婪策略来选择这$n-1$条边。这三种求解最小生成树的贪婪算法策略是： Kruskal算法，Prim算法和Sollin算法。

1. Kruskal算法

(1) 算法思想

Kruskal算法每次选择n-1条边，所使用的贪婪准则是：从剩下的边中选择一条不会产生环路的具有最小耗费的边加入已选择的边的集合中。注意到所选取的边若产生环路则不可能形成一棵生成树。Kruskal算法分e 步，其中e 是网络中边的数目。按耗费递增的顺序来考虑这 e 条边，每次考虑一条边。当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

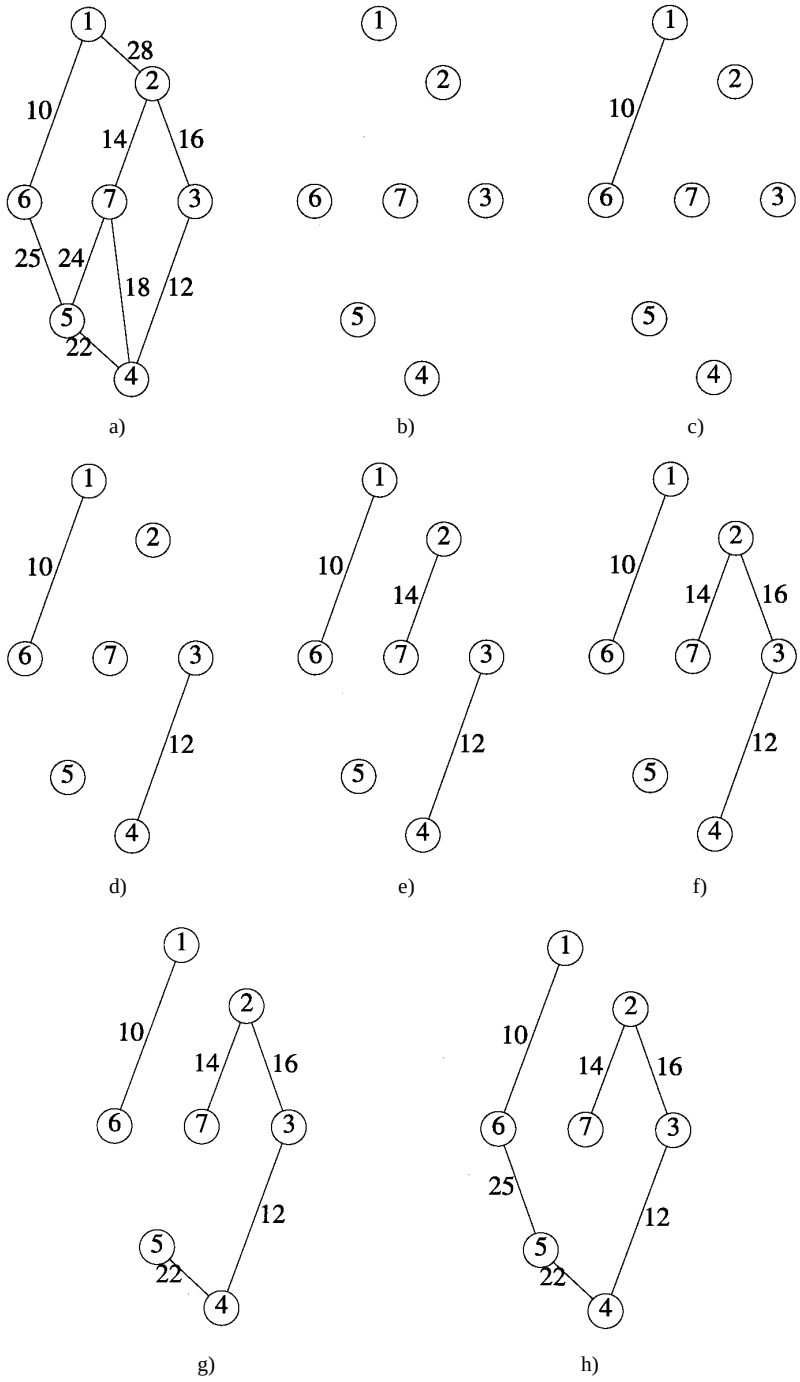


图13-12 构造最小耗费生成树

考察图13-12a 中的网络。初始时没有任何边被选择。图 13-12b 显示了各节点的当前状态。边（1,6）是最先选入的边，它被加入到欲构建的生成树中，得到图 13-12c。下一步选择边（3，4）并将其加入树中（如图 13-12d所示）。然后考虑边 (2，7)，将它加入树中并不会产生环路，于是便得到图 13-12e。下一步考虑边（2，3）并将其加入树中（如图 13-12f所

示)。在其余还未考虑的边中，(7，4）具有最小耗费，因此先考虑它，将它加入正在创建的树中会产生环路，所以将其丢弃。此后将边（ 5，4）加入树中，得到的树如图 13-12g 所示。下一步考虑边（ 7，5)，由于会产生环路，将其丢弃。最后考虑边（ 6，5）并将其加入树中，产生了一棵生成树（如图 13-12h所示)，其耗费为99。图13-13给出了Kruskal算法的伪代码。

```
//在一个具有n 个顶点的网络中找到一棵最小生成树
令T为所选边的集合，初始化 T=ϕ
令E 为网络中边的集合
while(E≠ϕ )&&(| T |≠n-1){
令(u,v)为E中代价最小的边
E=E-{(u,v)}//从E中删除边
if((u,v)加入T中不会产生环路）将（u,v）加入T
}
if(| T |==n-1) T是最小耗费生成树
else 网络不是互连的，不能找到生成树
```

图13-13 Kruskao算法的伪代码

(2) 正确性证明

利用前述装载问题所用的转化技术可以证明图 13-13的贪婪算法总能建立一棵最小耗费生成树。需要证明以下两点：1) 只要存在生成树，Kruskal算法总能产生一棵生成树；2) 产生的生成树具有最小耗费。

令G为任意加权无向图（即G是一个无向网络)。从12.11.3节可知当且仅当一个无向图连通时它有生成树。而且在Kruskal 算法中被拒绝（丢弃）的边是那些会产生环路的边。删除连通图环路中的一条边所形成的图仍是连通图，因此如果 G在开始时是连通的，则T与E中的边总能形成一个连通图。也就是若G开始时是连通的，算法不会终止于$E=\phi$ 和$|T|<n-1$。

现在来证明所建立的生成树 T具有最小耗费。由于 G具有有限棵生成树，所以它至少具有一棵最小生成树。令 U为这样的一棵最小生成树，T与U都刚好有n-1条边。如果$T=U$, 则T就具有最小耗费，那么不必再证明下去。因此假设$T\neq U$，令$k(k>0)$ 为在T中而不在U中的边的个数，当然k 也是在U中而不在T中的边的数目。

通过把U变换为T来证明U与T具有相同的耗费，这种转化可在 k 步内完成。每一步使在T而不在U中的边的数目刚好减 1。而且U的耗费不会因为转化而改变。经过 k 步的转化得到的U将与原来的 U具有相同的耗费，且转化后 U中的边就是T中的边。由此可知，T具有最小耗费。

每步转化包括从T中移一条边e 到U中，并从U中移出一条边f。边e 与f 的选取按如下方式进行：

1) 令e 是在T中而不在U中的具有最小耗费的边。由于$k>0$，这条边肯定存在。

2) 当把e 加入U时，则会形成唯一的一条环路。令 f 为这条环路上不在T中的任意一条边。由于T中不含环路，因此所形成的环路中至少有一条边不在T中。

从e 与f 的选择方法中可以看出，$V=U+\{e\}-\{f\}$ 是一棵生成树，且T中恰有k-1条边不在V中出现。现在来证明V的耗费与U的相同。显然，V的耗费等于U的耗费加上边e 的耗费再减去

边f的耗费。若e的耗费比f的小，则生成树V的耗费比U的耗费小，这是不可能的。如果e的耗费高于f，在Kruskal算法中f会在e之前被考虑。由于f不在T中，Kruskal算法在考虑f能否加入T时已将f丢弃，因此f和T中耗费小于或等于f的边共同形成环路。通过选择e，所有这些边均在U中，因此U肯定含有环路，但是实际上这不可能，因为U是一棵生成树。e的代价高于f的假设将会导致矛盾。剩下的唯一的可能是e与f具有相同的耗费，由此可知V与U的耗费相同。

(3) 数据结构的选择及复杂性分析

为了按耗费非递减的顺序选择边，可以建立最小堆并根据需要从堆中一条一条地取出各边。当图中有e条边时，需花$\Theta(e)$的时间初始化堆及$O(\log e)$的时间来选取每一条边。

边的集合T与G中的顶点一起定义了一个由至多n个连通子图构成的图。用顶点集合来描述每个子图，这些顶点集合没有公共顶点。为了确定边（u,v）是否会产生环路，仅需检查u,v是否在同一个顶点集中（即处于同一子图）。如果是，则会形成一个环路；如果不是，则不会产生环路。因此对于顶点集使用两个Find操作就足够了。当一条边包含在T中时，2个子图将被合并成一个子图，即对两个集合执行Union操作。集合的Find和Union操作可利用8.10.2节的树（以及加权规则和路径压缩）来高效地执行。Find操作的次数最多为$2e$，Union操作的次数最多为n-1（若网络是连通的，则刚好是n-1次）。加上树的初始化时间，算法中这部分的复杂性只比$O(n+e)$稍大一点。

对集合T所执行的唯一操作是向T中添加一条新边。T可用数组t来实现。添加操作在数组的一端进行，因为最多可在T中加入n-1条边，因此对T的操作总时间为$O(n)$。

总结上述各个部分的执行时间，可得图13-13算法的渐进复杂性为$O(n+e\log e)$。

(4) 实现

利用上述数据结构，图13-13可用C++代码来实现。首先定义EdgeNode类（见程序13-6），它是最小堆的元素及生成树数组t的数据类型。

程序13-6 Kruskal算法所需要的数据类型

```
template <class T>
class EdgeNode {
  public:
    operator T() const {return weight;}
  private:
    T weight;//边的高度
    int u, v;//边的端点
};
```

为了更简单地使用8.10.2节的查找和合并策略，定义了UnionFind类，它的构造函数是程序8-16的初始化函数，Union是程序8-16的加权合并函数，Find是程序8-17的路径压缩搜索函数。

为了编写与网络描述无关的代码，还定义了一个新的类UNetWork，它包含了应用于无向网络的所有函数。这个类与Undirected类的差别在于Undirected类中的函数不要求加权边，而UNetWork要求边必须带有权值。UNetWork中的成员需要利用Network类中定义的诸如Begin和NextVertex的遍历函数。不过，新的遍历函数不仅需要返回下一个邻接的顶点，而且要返回到达这个顶点的边的权值。这些遍历函数以及有向和无向加权网络的其他函数一起构成了WNetwork类（见程序13-7）。

程序13-7 WNetwork类

```
template<class T>
class WNetwork : virtual public Network
{
  public :
    virtual void First(int i, int& j, T& c)=0;
    virtual void Next(int i, int& j, T& c)=0;
};
```

象Begin和NextVertex一样，可在AdjacencyWDigraph及LinkedWDigraph类中加入函数First与Next。现在AdjacencyWDigraph及LinkedWDigraph类都需要从WNetWork中派生而来。由于AdjacencyWGraph类和LinkedWGraph类需要访问UNetwork的成员，所以这两个类还必须从UNetWork中派生而来。UNetWork::Kruskal的代码见程序13-8，它要求将Edges()定义为NetWork类的虚成员，并且把UNetWork定义为EdgeNode的友元）。如果没有生成树，函数返回false，否则返回true。注意当返回true时，在数组t中返回最小耗费生成树。

程序13-8 Kruskal算法的C++代码

```
template<class T>
bool UNetwork<T>::Kruskal(EdgeNode<T> t[])
{// 使用Kruskal算法寻找最小耗费生成树
 // 如果不连通则返回false
 // 如果连通，则在t[0:n-2]中返回最小生成树

  int n = Vertices();
  int e = Edges();
  //设置网络边的数组
  InitializePos();  // 图遍历器
  EdgeNode<T> *E = new EdgeNode<T> [e+1];
  int k = 0;        // E的游标
  for (int i = 1; i <= n; i++) { // 使所有边附属于 i
    int j;
    T c;
    First(i, j, c);
    while (j) {    // j 邻接自 i
      if (i < j) {// 添加到达 E的边
        E[++k].weight = c;
        E[k].u = i;
        E[k].v = j;}
      Next(i, j, c);
      }
    }

  // 把边放入最小堆
  MinHeap<EdgeNode<T> > H(1);
  H.Initialize(E, e, e);

  UnionFind U(n); // 合并/搜索结构
```

```
    // 根据耗费的次序来抽取边
    k = 0;  // 此时作为t的游标
    while (e && k < n - 1) {
      // 生成树未完成，尚有剩余边
      EdgeNode<T> x;
      H.DeleteMin(x); // 最小耗费边
      e--;
      int a = U.Find(x.u);
      int b = U.Find(x.v);
      if (a != b) {// 选择边
        t[k++] = x;
        U.Union(a,b);}
      }

    DeactivatePos();
    H.Deactivate();
    return (k == n - 1);
  }
```

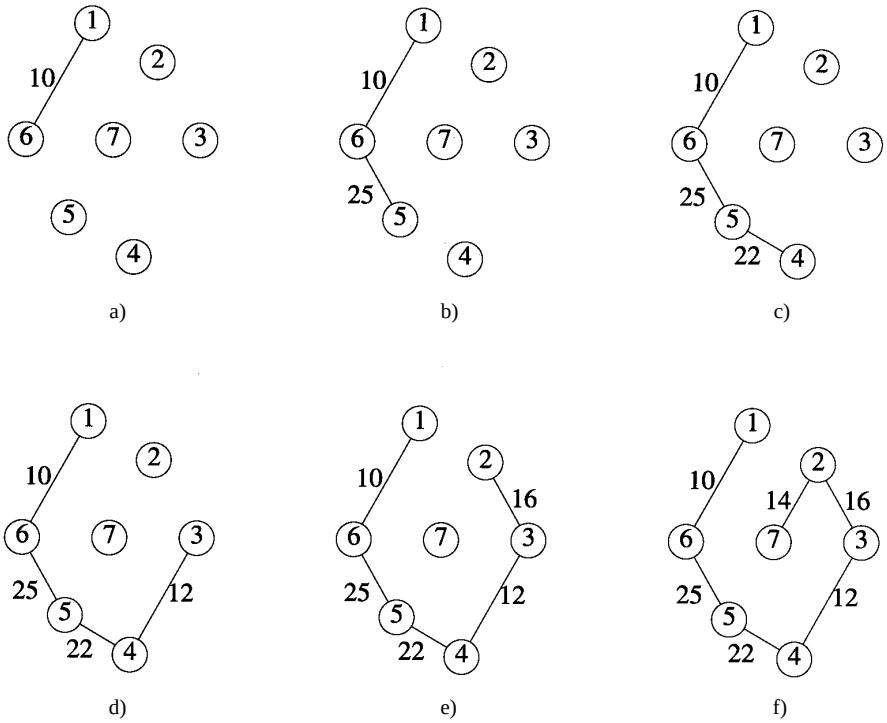
2. Prim算法

与Kruskal算法类似，Prim算法通过每次选择多条边来创建最小生成树。选择下一条边的贪婪准则是：从剩余的边中，选择一条耗费最小的边，并且它的加入应使所有入选的边仍是一棵树。最终，在所有步骤中选择的边形成一棵树。相反，在 Kruskal 算法中所有入选的边集合最终形成一个森林。

Prim算法从具有一个单一顶点的树T开始，这个顶点可以是原图中任意一个顶点。然后往T中加入一条代价最小的边（u,v）使T $\{(u,v)\}$仍是一棵树，这种加边的步骤反复循环直到T中包含n-1条边。注意对于边（u,v），u、v 中正好有一个顶点位于T中。Prim算法的伪代码如图13-14所示。在伪代码中也包含了所输入的图不是连通图的可能，在这种情况下没有生成树。图13-15显示了对图13-12a 使用Prim算法的过程。把图13-14的伪代码细化为C++程序及其正确性的证明留作练习（练习31）。

```
//假设网络中至少具有一个顶点
设T为所选择的边的集合，初始化 T=ϕ
设TV为已在树中的顶点的集合，置 TV={1}
令E为网络中边的集合
while(E<>ϕ)&&(| T | < >n-1) {
令（u,v）为最小代价边，其中u∈TV, v∉TV
if（没有这种边） break
E=E-{(u,v)}//从E中删除此边
在T中加入边（u,v）
}
if (| T |==n-1) T是一棵最小生成树
else 网络是不连通的，没有最小生成树
```

图13-14 Prim最小生成树算法



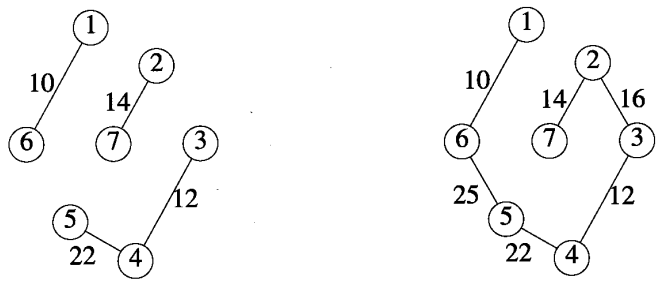
a) b) c) d) e) f)

图13-15 Prim算法的步骤

如果根据每个不在*TV*中的顶点*v* 选择一个顶点*near* (*v*)，使得*near* (*v*) *TV* 且*cost* (*v*, *near* (*v*)) 的值是所有这样的*near* (*v*) 节点中最小的，则实现Prim算法的时间复杂性为O(n^2)。下一条添加到*T*中的边是这样的边：其*cost* (*v*, *near* (*v*)) 最小，且*v*∉*TV*。

3. Sollin算法

Sollin算法每步选择若干条边。在每步开始时，所选择的边及图中的 *n*个顶点形成一个生成树的森林。在每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。注意一个森林中的两棵树可选择同一条边，因此必须多次复制同一条边。当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边，在这种情况下，必须丢弃其中的一条边。开始时，所选择的边的集合为空。若某一步结束时仅剩下一棵树或没有剩余的边可供选择时算法终止。



a) b)

图13-16 Sollin算法的步骤

图13-6给出了初始状态为图13-12a 时，使用Sollin算法的步骤。初始入选边数为0时的情形如图13-12a 时,森林中的每棵树均是单个顶点。顶点1，2，…，7所选择的边分别是(1.6), (2,7), (3,4), (4,3), (5,4), (6,1), (7,2)，其中不同的边是(1,6)，(2,7)，(3,4) 和(5,4)，将这些边加入入选边的集合后所得到的结果如图13-16a所示。下一步具有顶点集{1,6}的树选择边(6,5)，剩下的两棵树选择边(2,3)，加入这两条边后已形成一棵生成树，构建好的生成树见图 13-6b。Sollin算法的C++程序实现及其正确性证明留作练习（练习32)。

练习

8. 针对装载问题，扩充贪婪算法，考虑有两条船的情况，算法总能产生最优解吗？

9. 已知n 个任务的执行序列。假设任务i 需要t_i 个时间单位。若任务完成的顺序为1，2，…，n，则任务i 完成的时间为$c_i=\sum_{j=1}^{i}t_j$。任务的平均完成时间（Averge Completion Time, ACT）为$\frac{1}{n}\sum_{i=1}^{n}c_i$。

1) 考虑有四个任务的情况，每个任务所需时间分别是（4，2，8，1)。若任务的顺序为1，2，3，4，则ACT是多少？

2) 若任务顺序为2，1，4，3，则ACT是多少？

3) 创建具有最小ACT的任务序列的贪婪算法分n 步来构造该任务序列，在每一步中，从剩下的任务里选择时间最小的任务。对于1)，利用这种策略获得的任务顺序为4，2，1，3，这种顺序的ACT是多少？

4) 写一个C++程序实现3) 中的贪婪策略，程序的复杂性应为O($n$log$n$)，试证明之。

5) 证明利用3) 中的贪婪算法获得的任务顺序具有最小的ACT。

10. 若有两个工人执行练习9中的n个任务，需将任务分配给他们，同时他们具有自己的任务执行顺序。任务完成时间及 ACT的定义同练习9。使ACT最小化的一种可行的贪婪算法是：两个工人轮流选择任务，每次从剩余的任务中选择时间最小的任务。每个人按照自己所选任务的顺序执行任务。对于练习9中的例子，假定工人1首先选择任务4，然后工人2选择任务2，工人1选择任务1，最后工人2选择任务3。

1) 利用C++程序实现这种策略，其时间复杂性为多少？

2) 上述的贪婪策略总能获得最小的ACT吗？证明结论。

11. 1) 考虑有m 个人可以执行任务，扩充练习10中的贪婪算法。

2) 算法能保证获得最优解吗？证明结论。

3) 用C++程序实现此算法，其复杂性是多少？

12. 考虑例4-4的堆栈折叠问题。

1) 设计一个贪婪算法，将堆栈折叠为最小数目的子堆栈，使得每个子堆栈的高度均不超过H。

2) 算法总能保证得到数目最少的子堆栈吗？证明结论。

3) 用C++代码实现1) 的算法。

4) 代码的时间复杂性是多少？

13. 编写C++程序实现0/1背包问题，使用如下启发式方法：按价值密度非递减的顺序打包。

14. 根据k=1的性能受限启发式方法编写一个C++程序来实现0/1背包问题。

15. 对于k=1的情况证明用性能受限的启发式方法求解0/1背包问题会发生边界错误。

16. 根据k=2的性能受限启发式方法编写一个C++程序来实现0/1背包问题。

17. 考虑$0 \le x_i \le 1$而不是$x_i \in \{0,1\}$的连续背包问题。一种可行的贪婪策略是：按价值密度非递减的顺序检查物品，若剩余容量能容下正在考察的物品，将其装入；否则，往背包中装入此物品的一部分。

1) 对于n=3, w=[100,10,10], p=[20,15,15]及c=105,上述装入方法获得的结果是什么？

2) 证明这种贪婪算法总能获得最优解。

3) 用一个C++程序实现此算法。

18. 例13-1的渴婴问题是练习17中连续背包问题的一般化，将练习17的贪婪算法用于渴婴问题，算法能保证总能得到最优解吗？证明结论。

19. 1) 证明当且仅当二分图没有覆盖时，图13-7的算法找不到覆盖。

2) 给出一个具有覆盖的二分图，使得图13-7的算法找不到最小覆盖。

20. 当第一步选择了顶点1时，给出图13-7的工作过程。

21. 对于二分图覆盖问题设计另外一种贪婪启发式方法，可使用如下贪婪准则：如果B中的某一个顶点仅被A中一个顶点覆盖，选择A中这个顶点；否则，从A中选择一个顶点，使得它所覆盖的未被覆盖的顶点数目最多。

1) 给出这种贪婪算法的伪代码。

2) 编写一个C++函数作为Undirected类的成员来实现上述贪婪算法。

3) 函数的复杂性是多少？

4) 验证代码的正确性。

22. 令G为无向图，S为G中顶点的子集，当且仅当S中的任意两个顶点都有一条边相连时，S为完备子图（clique），完备子图的大小即S中的顶点数目。最大完备子图（maximum clique）即具有最大顶点数目的完备子图。在图中寻找最大完备子图的问题（即最大完备子图问题）是一个NP-复杂问题。

1) 给出最大完备子图问题的一种可行的贪婪算法及其伪代码。

2) 给出一个能用1) 中的启发式算法求解最大完备子图的图例，以及不能用该算法求解的一个图例。

3) 将1) 中的启发式算法实现为Undirected::Clique(int C, int m) 共享成员，其中最大完备子图的大小返回到m中，最大完备子图的顶点返回到C中。

4) 代码的复杂性是多少？

23. 令G为一无向图，S为G中顶点的子集，当且仅当S中任意两个顶点都无边相连时，S为无关集（independent set）。最大无关集即是顶点数目最多的无关集。在一幅图中寻找最大无关集是一个NP-复杂问题。按练习22的要求解决最大无关集问题。

24. 对无向图G着色的方法是：为G中的顶点编号（{1,2,…}），使得由一条边相连的两个顶点具有不同的编号。在图的着色问题中，要求利用最少的相互不同的颜色（编号）来给图G着色。图的着色问题也是一个NP-复杂问题。按练习22的要求解决图着色问题。

25. 证明当按路径长度的顺序产生一条最短路径时，所产生的下一条最短路径总是由已产生的一条最短路径扩充一条边得到。

26. 证明对于具有一条或多条具有负长度的边，图13-11的贪婪算法不一定能正确地计算出最短路径的长度。

27. 编写一个Path(p,s,i)函数，利用函数ShortestPaths计算出的p值，输出从顶点s到顶点i的一条最短路径。函数的复杂性是多少？

28. 若把有向图作为LinkedwDigraph类的一个成员，重写程序13-5，函数应作为该类的一

个成员。函数的复杂性是多少？

29. 若把有向图作为LinkedWDigraph类的一个成员且仅有O(n)条边，重写程序13-5，L用最小堆来实现。函数的复杂性是多少？

30. 从Network类（见程序12-15）派生出一个新的模板类DNetwork（有向网络），这个类仅包含应用于有向网络的所有函数。为该类定义一个 ShortestPaths函数，使得它与有向网络的描述形式无关，尤其适用于耗费邻接矩阵及邻接链表描述方法。在函数的实现过程中可利用原来的遍历函数，也可根据需要定义新的遍历函数。函数的复杂性应为 O(n^2)，其中n 是顶点的数目，试证明之。

*31. 1) 给出Prim算法（如图13-14所示）的一种正确性证明。

2) 将图13-14细化为一个C++程序UNetwork::Prim，其复杂性应为O(n^2)。

3) 证明程序的复杂性确实是O(n^2)。

*32. 1) 证明对于任意连通无向图，Sollin算法总能找到一个最小耗费生成树。

2) 在Sollin算法中，最大的步骤数是多少？试用图中顶点数n 来表示。

3) 编写一个C++程序UNetwork::Sollin，使用Sollin算法找到一棵最小生成树。

4) 程序的复杂性是多少？

*33. 令T为一棵每条边均带有长度的树（不一定是二叉树）。令S为T中顶点的子集，并令T/S为从T中删除S中的顶点所得到的森林。我们希望能找到具有最小走势的子集 S，使得T/S中没有从根到叶的距离大于d的森林。

1) 给出一种寻找最小走势子集S的贪婪算法（提示：从叶节点开始向根移动）。

2) 证明算法的正确性。

3) 算法的复杂性是多少？如果它不是 T中顶点数的线性函数，则重新设计算法，使其复杂性是线性的。

*34. 令T/S表示将S中的每个顶点复制两份而获得的森林，其中父节点的指针指向一个复本，而另一复本的指针指向其儿子。针对这种情况再做练习 33。

13.4 参考及推荐读物

V.Paschos. A Survey of Approximately Optimal Solutions to Some Covering and Packing Problems. *ACM Computing Surveys*, 29, 2, 1997, 171~209。其中描述了几种问题的近似贪婪算法。

B. Moret, H. Shapiro. An Empirical Assessment of Algorithms for Constructing a Minimum Spanning tree. *DIMACS Series in Discrete Mathematics*, 15, 1994, 99~117。它讲述了最小代价生成树问题所使用的贪婪算法的实验估计。