# 第二章 贪心法

## 教学目标

- 理解贪心法的概念
- 掌握贪心法的基本思想和要素
- 理解贪心法的正确性证明方法
- 通过对实例的学习，掌握贪心法的设计策略及解题步骤

A greedy algorithm always makes the choice that looks best at the moment.That is,it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

——《算法导论》

在众多的算法设计策略中，贪心法可以算得上是最接近人们日常思维的一种解题策略，它以其简单、直接和高效而受到重视。尽管该方法并不是从整体最优方面来考虑问题，而是从某种意义上的局部最优角度做出选择，但对范围相当广泛的许多实际问题它通常都能产生整体最优解，如单源最短路径问题、最小生成树等等。在一些情况下，即使采用贪心法不能得到整体最优解，但其最终结果却是最优解的很好近似解。正是基于此，该算法在对 NP 完全问题的求解中发挥着越来越重要的作用。另外，近年来贪心法在各级各类信息学竞赛、ACM 程序设计竞赛中经常出现，竞赛中的一些题目常常需要选手经过细致的思考后得出高效的贪心算法。为此，学习该算法具有很强的实际意义和学术价值。

## 2.1 概述

### 2.1.1 贪心法的基本思想

贪心法是一种稳扎稳打的算法，它从问题的某一个初始解出发，在每一个阶段都根据贪心策略来做出当前最优的决策，逐步逼近给定的目标，尽可能快地求得更好的解。当达到算法中的某一步不能再继续前进时，算法终止。贪心法可以理解为以逐步的局部最优，达到最终的全局最优。

从算法的思想中，很容易得出以下结论：

（1）贪心法的精神是“今朝有酒今朝醉”。每个阶段面临选择时，贪心法都做出对眼前来讲是最有利的选择，不考虑该选择对将来是否有不良影响。

（2）每个阶段的决策一旦做出，就不可更改，也即该算法不允许回溯。

（3）贪心法是根据贪心策略来逐步构造问题的解。如果所选的贪心策略不同，则得到的贪心算法就不同，贪心解的质量当然也不同。因此，该算法的好坏关键在于正确地选择贪心策略。

贪心策略是依靠经验或直觉来确定一个最优解的决策。该策略一定要精心确定，且在使用之前最好对它的可行性进行数学证明，只有证明其能产生问题的最优解后再使用，不要被表面上看似正确的贪心策略所迷惑。

（4）贪心法具有高效性和不稳定性，因为它可以非常迅速地获得一个解，但这个解不一定是最优解，即便不是最优解，也一定是最优解的近似解。

### 2.1.2 贪心法的基本要素

何时能、何时应该采用贪心法呢？一般认为，凡是经过数学归纳法证明可以采用贪心法的情况都应该采用它，因为它具有高效性。可惜的是，它需要证明后才能真正运用到问题的求解中。

那么能采用贪心法的问题具有怎样的性质呢？这个提问很难给予肯定的回答。但是，从许多可以用贪心法求解的问题中，可以看到这些问题一般都具有两个重要的性质：最优子结构性质和贪心选择性质。换句话说，如果一个问题具有这两大性质，那么使用贪心法来对其求解总能求得最优解。

（1）最优子结构性质

当一个问题的最优解一定包含其子问题的最优解时，称此问题具有最优子结构性质。换句话说，一个问题能够分解成各个子问题来解决，通过各个子问题的最优解能递推到原问题的最优解。那么原问题的最优解一定包含各个子问题的最优解，这是能够采用贪心法来求解问题的关键。因为贪心法求解问题的流程是依序研究每个子问题，然后综合得出最后结果。而且，只有拥有最优子结构性质才能保证贪心法得到的解是最优解。

在分析问题是否具有最优子结构性质时，通常先设出问题的最优解，给出子问题的解一定是最优的结论。然后，采用反证法证明“子问题的解一定是最优的”结论成立。证明思路是：设原问题的最优解导出的子问题的解不是最优的，然后在这个假设下可以构造出比原问题的最优解更好的解，从而导致矛盾。

（2）贪心选择性质

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获得，即通过一系列的逐步局部最优选择使得最终的选择方案是全局最优的。其中每次所做的选择，可以依赖于以前的选择，但不依赖于将来所做的选择。

可见，贪心选择性质所做的是一个非线性的子问题处理流程，即一个子问题并不依赖于另一个子问题，但是子问题间有严格的顺序性。

在实际应用中，至于什么问题具有什么样的贪心选择性质是不确定的，需要具体问题具体分析。对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所做的贪心选择能够最终导致问题的一个整体最优解。首先考察问题的一个整体最优解，并证明可修改这个最优解，使其以贪心选择开始。而且做了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步做贪心选择，最终可得到问题的一个整体最优解。其中，证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

### 2.1.3 贪心法的解题步骤及算法设计模式

利用贪心法求解问题的过程通常包含如下 3 个步骤：

（1）分解：将原问题分解为若干个相互独立的阶段。

（2）解决：对于每个阶段依据贪心策略进行贪心选择，求出局部的最优解。

（3）合并：将各个阶段的解合并为原问题的一个可行解。

依据该步骤，设计出的贪心法的算法设计模式如下：

```
Greedy (A，n)
  {
   //A[0:n-1]包含 n 个输入，即 A 是问题的输入集合；
```

```
将解集合 solution 初始化为空;
for(i=0; i<n; i++) //原问题分解为 n 个阶段
  {
     x=select (A);  //依据贪心策略做贪心选择，求得局部最优解
     if (x 可以包含在 solutioin) //判断解集合 solution 在加入 x 后是否满足约束条件
        solution=union (solution,x); //部分局部最优解进行合并
  }
return (解向量 solution );//n 个阶段完成后，得到原问题的最优解
}
```

贪心法是在少量运算的基础上做出贪心选择而不急于考虑以后的情况，一步一步地进行解的扩充，每一步均是建立在局部最优解的基础上。

## 2.2 会场安排问题

会场安排问题来源于实际，事实上无论任何与时间分配有关的问题都要考虑安排问题，来达到占用公共资源最少且花费时间最短的要求。类似会场安排问题的情况较为普遍，例如公司会议安排，要求开会的时间不能冲突而会议室又有限，如何安排最佳的顺序来开会；学校课程安排及体育场场地分配问题等等。贪心法为这类问题提供了一个简单、漂亮且有效的方法，使得尽可能多的参与者兼容地使用公共资源。

1．问题描述

设有 n 个会议的集合 C={1,2,…,n}，其中每个会议都要求使用同一个资源（如会议室），而在同一时间内只能有一个会议使用该资源。每个会议 i 都有要求使用该资源的起始时间 $b_i$ 和结束时间 $e_i$，且 $b_i<e_i$。如果选择了会议 i 使用会议室，则它在半开区间$[b_i, e_i)$内占用该资源。如果$[b_i, e_i)$与$[b_j, e_j)$不相交，则称会议 i 与会议 j 是相容的。也就是说，当 $b_i \geq e_j$ 或 $b_j \geq e_i$ 时，会议 i 与会议 j 相容。会场安排问题要求在所给的会议集合中选出最大的相容活动子集，也即尽可能地选择更多的会议来使用资源。

2．贪心策略

贪心法求解会场安排问题的关键是如何设计贪心策略，使得算法在依照该策略的前提下按照一定的顺序来选择相容会议，以便安排尽量多的会议。根据给定的会议开始时间和结束时间，会场安排问题至少有三种看似合理的贪心策略可供选择。

（1）每次从剩下未安排的会议中选择具有最早开始时间且不会与已安排的会议重叠的会议来安排。这样可以增大资源的利用率。

（2）每次从剩下未安排的会议中选择使用时间最短且不会与已安排的会议重叠的会议来安排。这样看似可以安排更多的会议。

（3）每次从剩下未安排的会议中选择具有最早结束时间且不会与已安排的会议重叠的会议来安排。这样可以使下一个会议尽早开始。

到底选用哪一种贪心策略呢？选择策略（1），如果选择的会议开始时间最早，但使用时间无限长，这样只能安排 1 个会议来使用资源；选择策略（2），如果选择的会议的开始时间最晚，那么也只能安排 1 个会议来使用资源；由策略（1）和策略（2），人们容易想到一种更好的策略：“选择开始时间最早且使用时间最短的会议”。根据“会议结束时间=会议开始时间+使用资源时间”可知，该策略便是策略（3）。直观上，按这种策略选择相容会议可以给未安排的会议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会议。

3．算法的设计和描述

根据问题描述和所选用的贪心策略，对贪心法求解会议安排问题的 GreedySelector 算法设计思路如下：

步骤 1：初始化。将 n 个会议的开始时间存储在数组 B 中；将 n 个会议的结束时间存储在数组 E 中且按照结束时间的非减序排序：$e_1 \le e_2 \le \ldots \le e_n$，数组 B 需要做相应调整；采用集合 A 来存储问题的解，即所选择的会议集合，会议 i 如果在集合 A 中，当且仅当 A[i]=true；

步骤 2：根据贪心策略，算法 GreedySelector 首先选择会议 1，即令 A[1]=true；

步骤 3：依次扫描每一个会议，如果会议 i 的开始时间不小于最后一个选入集合 A 中的会议的结束时间，即会议 i 与 A 中会议相容，则将会议 i 加入集合 A 中；否则，放弃会议 i，继续检查下一个会议与集合 A 中会议的相容性。

设会议 i 的起始时间 $b_i$ 和结束时间 $e_i$ 的数据类型为自定义结构体类型 sturct time；则 GreedySelector 算法描述如下：

```
void GreedySelector(int n，struct time B[]，struct time E[]，bool A[])
  {
   E 中元素按非减序排列，B 中对应元素做相应调整；
   int i,j;
   A[1]=true；//初始化选择会议的集合 A，即只包含会议 1
   j=1；i=2；//从会议 i 开始寻找与会议 j 相容的会议
   while（i<=n）
       if (B[i]>=E[j]）{A[i]=true;j=i;}
       else   A[i]=false；
}
```

**【例 2-1】**设有 11 个会议等待安排，用贪心法找出满足目标要求的会议集合。这些会议按结束时间的非减序排列如表 2-1 所示。

表 2-1 11 个会议按结束时间的非减序排列表

| 会议 i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 开始时间 $b_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| 结束时间 $e_i$ | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

根据贪心策略可知，算法每次从剩下未安排的会议中选择具有最早的完成时间且不会与已安排的会议重叠的会议来安排。具体的求解过程如图 2-1 所示。

| 会议 i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 开始时间 $b_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| 结束时间 $e_i$ | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

图 2-1 会议安排问题的贪心法求解过程示意图

因为会议 1 具有最早的完成时间，因此 GreedySelector 算法首先选择会议 1 加入解集合 A。由于 $b_2<e_1$、$b_3<e_1$，显然会议 2 和会议 3 与会议 1 不相容，所以放弃它们，继续向后扫描，由于 $b_4>e_1$，可见会议 4 与会议 1 相容，且在剩下未安排会议中具有最早完成时间，符合贪心策略，因此将会议 4 加入解集合 A。然后在剩下未安排会议中选择具有最早完成时间且与会议 4 相容的会议。依此类推，最终选定的解集合 A 为{1,0,0,1,0,0,0,1,0,0,1}，即选定的会议集合为{1,4,8,11}。

4．算法分析

从 GreedySelector 算法的描述可以看出，该算法的时间主要消耗在将各个活动按结束时间从小到大进行排列。若采用快速排序算法进行排序，算法的时间复杂性为 O(nlogn)。显然该算法的空间复杂性是常数阶，即 S(n)=O(1)。

5．算法的正确性证明

前面已经介绍过，使用贪心法并不能保证最终的解就是最优解。但对于会议安排问题，贪心法 GreedySelector 却总能求得问题的最优解，即它最终所确定的相容活动集合 A 的规模最大。

贪心算法的正确性证明需要从贪心选择性质和最优子结构性质两方面进行。因此，GreedySelector 算法的正确性证明只需要证明会场安排问题具有贪心选择性质和最优子结构性质即可。下面采用数学归纳法来对该算法的正确性进行证明。

（1）贪心选择性质

贪心选择性质的证明即证明会场安排问题存在一个以贪心选择开始的最优解。设 C={1,2,…,n}是所给的会议集合。由于 C 中的会议是按结束时间的非减序排列，故会议 1 具有最早结束时间。因此，该问题的最优解首先首先选择会议 1。

设 $C^*$是所给的会议安排问题的一个最优解，且 $C^*$中会议也按结束时间的非减序排列，$C^*$中的第一个会议是会议 k。若 k=1，则 $C^*$就是一个以贪心选择开始的最优解。若 k>1，则设 $C'= C^*-\{k\} \cup \{1\}$。由于 $e_1 \le e_k$，且 $C^*-\{k\}$中的会议是互为相容的且它们的开始时间均大于等于 $e_k$，故 $C^*-\{k\}$中的会议的开始时间一定大于等于 $e_1$，所以 C’中的会议也是互为相容的。又由于 C’中会议个数与 $C^*$中会议个数相同且 $C^*$是最优的，故 C’也是最优的。即 C’是一个以贪心选择活动 1 开始的最优会议安排。因此，证明了总存在一个以贪心选择开始的最优会议安排方案。

（2）最优子结构性质

进一步，在做了贪心选择，即选择了会议 1 后，原问题就简化为对 C 中所有与会议 1 相容的会议进行会议安排的子问题。即若 A 是原问题的一个最优解，则 A’=A-{1}是会议安排问题 $C_1=\{i \in C | b_i \ge e_1\}$的一个最优解。

证明（反证法）：假设 A’ 不是会议安排问题 $C_1$ 的一个最优解。设 $A_1$ 是会议安排问题 $C_1$ 的一个最优解，那么$|A_1|>|A'|$。令 $A_2=A_1 \bigcup \{1\}$，由于 $A_1$ 中的会议的开始时间均大于等于 $e_1$，故 $A_2$ 是会议安排问题 C 的一个解。又因为$|A_2=A_1 \bigcup \{1\}|>|A' \bigcup \{1\}=A|$，所以 A 不是会议安排问题 C 的最优解。这与 A 是原问题的最优解矛盾，所以 A’是会议安排问题 C 的一个最优解。

## 2.3 单源最短路径问题

1．问题描述

给定一个有向带权图 G=(V，E)，其中每条边的权是一个非负实数。另外，给定 V 中的一个顶点，称为源点。现在要计算从源点到所有其它各个顶点的最短路径长度，这里的路径长度是指路径上经过的所有边上的权值之和。这个问题通常称为单源最短路径问题。

2．Dijkstra 算法思想及算法设计

（1）Dijkstra 算法思想

对于一个具体的单源最短路径问题，如何求得该最短路径呢？一个传奇人物的出现使得该问题迎刃而解，他就是迪杰斯特拉(Dijkstra)。他提出按各个顶点与源点之间路径长度的递增次序，生成源点到各个顶点的最短路径的方法，即先求出长度最短的一条路径，再参照它求出长度次短的一条路径，依此类推，直到从源点到其它各个顶点的最短路径全部求出为止，

该算法俗称 Dijkstra 算法。Dijkstra 对于它的算法是这样说的:“这是我自己提出的第一个图问题，并且解决了它。令人惊奇的是我当时并没有发表。但这在那个时代是不足为奇的，因为那时，算法基本上不被当作一种科学研究的主题。”

（2）Dijkstra 算法设计

假定源点为 u。顶点集合 V 被划分为两部分：集合 S 和 V-S，其中 S 中的顶点到源点的最短路径的长度已经确定，集合 V-S 中所包含的顶点到源点的最短路径的长度待定，称从源点出发只经过 S 中的点到达 V-S 中的点的路径为特殊路径。Dijkstra 算法采用的贪心策略是选择特殊路径长度最短的路径，将其相连的 V-S 中的顶点加入到集合 S 中。

Dijkstra 算法的求解步骤设计如下：

步骤 1：设计合适的数据结构。设置带权邻接矩阵 C，即如果< u, x >∈E，令 C[u][x]=<u, x >的权值，否则，C[u][x]=∞；采用一维数组 dist 来记录从源点到其它顶点的最短路径长度，例如 dist[x]表示源点到顶点 x 的路径长度；采用一维数组 p 来记录最短路径；

步骤 2：初始化。令集合 S={u}，对于集合 V-S 中的所有顶点 x，设置 dist[x]=C[u][x](注意，x 只是一个符号，它可以表示集合 V-S 中的任一个顶点)；如果顶点 i 与源点相邻，设置 p[i]=u，否则 p[i]=-1；

步骤 3：在集合 V-S 中依照贪心策略来寻找使得 dist[x]具有最小值的顶点 t，即：$dist[t]=\min\{dist[x]\mid x\in(V-S)\}$，满足该公式的顶点 t 就是集合 V-S 中距离源点 u 最近的顶点。

步骤 4：将顶点 t 加入集合 S 中，同时更新集合 V-S；

步骤 5：如果集合 V-S 为空，算法结束；否则，转步骤 6；

步骤 6：对集合 V-S 中的所有与顶点 t 相邻的顶点 x，如果 dist[x]>dist[t]+C[t][x]，则 dist[x]=dist[t]+C[t][x]并设置 p[x]=t。转步骤 3。

由此，可求得从源点 u 到图 G 的其余各个顶点的最短路径及其长度。

3．单源最短路径问题的构造实例

**【例 2-2】在**如图 2-2 所示的有向带权图中，求源点 0 到其余顶点的最短路径及最短路径长度。

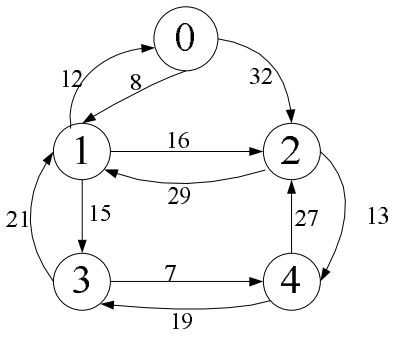



图 2-2 有向带权图

根据算法思想和求解步骤，很容易得出算法的执行过程：初始时，设置集合 S={0}，对所有与源点 0 相邻的顶点 1 和 2，设置 p[1]=p[2]=0，而与源点 0 不相邻的顶点 3 和 4，则设置 p[3]=p[4]=-1。然后在集合 V-S 的各个顶点中，由于 dist[1]最小，可见距离源点 0 最近的顶点为顶点 1，因此将 1 加入集合 S，并将其从集合 V-S 中删去，同时更新所有与顶点 1 相邻的顶点到源点 0 的最短路径长度，其中 dist[2]=24、dist[3]=23，因此 p[2]=1，p[3]=1。接下来，再次从 V-S 的各个顶点中，找出距离源点 0 最近的顶点，容易看出 dist[3]最小，因此，将顶点 3 加入集合 S，同时将其从 V-S 中删去，并更新所有与顶点 3 相邻的顶点到源点 0 的

最短路径长度，其中 dist[4]=30，设置 p[4]=3；继续从 V-S 的各个顶点中，找出距离源点 0 最近的顶点，易得 dist[2]最小，因此，将顶点 2 加入集合 S，并从 V-S 中删去，并更新所有与顶点 2 相邻的顶点到源点 0 的最短路径长度。由于 V-S 中只剩下顶点 4，则将其加入集合 S，并从 V-S 中删去，算法结束。执行过程中各量的变化情况如表 2-2、2-3 所示。

表 2-2　Dijkstra 算法的求解过程

| | S | V-S | dist[1] | dist[2] | dist[3] | dist[4] | t |
|---|---|---|---|---|---|---|---|
| 初始 | {0} | {1,2,3,4} | 8 | 32 | max | max | 1 |
| 1 | {0,1} | {2,3,4} | - | 24 | 23 | max | 3 |
| 2 | {0,1,3} | {2,4} | - | 24 | - | 30 | 2 |
| 3 | {0,1,3,2} | {4} | - | - | - | 30 | 4 |
| 4 | {0,1,3,2,4} | {} | | | | | |

表 2-3 前驱数组 p 变化情况表

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 初始 | -1 | 0 | 0 | -1 | -1 |
| 1 | -1 | 0 | 1 | 1 | -1 |
| 2 | -1 | 0 | 1 | 1 | 3 |
| 3 | -1 | 0 | 1 | 1 | 3 |
| 4 | -1 | 0 | 1 | 1 | 3 |

注：表 2-3 中的行表示顶点编号；列表示求解步骤；表中数据表示 p[i](i 为顶点编号)。

对于【例 2-2】，经 Dijkstra 算法计算后，数组 dist 记录了源点到其它各个顶点的最短路径长度。其中，dist[1]=8，dist [2]=24，dist [3]=23，dist [4]=30；数组 p 记录了最短路径，p[1]=0，p[2]=1，p[3]=1，p[4]=3。如果要找出从源点 0 到顶点 4 的最短路径，可以从数组 p 得到顶点 4 的前驱顶点为 3，而顶点 3 的前驱顶点为 1，顶点 1 的前驱顶点为 0。于是从源点 0 到顶点 4 的最短路径为 0，1，3，4。同理，可找出从源点 0 到顶点 3 的最短路径为：0，1，3；而源点 0 到顶点 2 的最短路径为 0，1，2；源点 0 到顶点 1 的最短路径为 0，1。

4．Dijkstra 算法描述

n：顶点个数；u：源点、C[n][n]：带权邻接矩阵；dist[]：记录某顶点与源点 u 的最短路径长度；p[]：记录某顶点到源点的最短路径上的该顶点的前驱顶点。

```
void Dijkstra(int n，int u，float dist[]，int p[]，int C[n][n])
  {
    bool s[n];    //如果 s[i]等于 true,说明顶点 i 已加入集合 S；否则，顶点 i 属于集合 V-S
    for(int i=1;i<=n;i++)
       {
         dist[i]=C[u][i];  //初始化源点 u 到其它各个顶点的最短路径长度
         s[i]=false;
         if (dist[i]==∞)  p[i]=-1;    //满足条件,说明顶点 i 与源点 u 不相邻,设置 p[i]=-1
         else   p[i]=u;  //说明顶点 i 与源点 u 相邻，设置 p[i]=u
       }//for  循环结束
   dist[u]=0; s[u]=true;      //初始时，集合 S 中只有一个元素：源点 u
   for(i=1;i<=n;i++)
      {
        int temp=∞; int t=u;
       for(int j=1;j<=n;j++)   //在集合 V-S 中寻找距离源点 u 最近的顶点 t
           if((!s[j])&&(dist[j]<temp))
```

```
            {
              t=j;
              temp=dist[j];
            }
        if(t==u) break;     //找不到 t，跳出循环
        s[t]=true;      //否则，将 t 加入集合 S
        for(j=1;j<=n;j++)     //更新与 t 相邻接的顶点到源点 u 的距离
          if((!s[j])&&(C[t][j]< ∞ )
            if (dist[j]>(dist[t]+C[t][j]))
                {
                  dist[j]=dist[t]+C[t][j];
                  p[j]=t;
                }
        }
    }
```

5．算法分析

从算法的描述中，不难发现语句 if((!s[j])&&(dist[j]<temp))对算法的运行时间贡献最大，因此选择将该语句作为基本语句。当外层循环标号为 1 时，该语句在内层循环的控制下，共执行 n 次，外层循环从 1~n，因此，该语句的执行次数为 $n*n=n^2$，算法的时间复杂性为 $O(n^2)$。

实现该算法所需的辅助空间包含为数组 s 和变量 i、j、t 和 temp 所分配的空间，因此，Dijkstra 算法的空间复杂性为 O(n)。

6．算法的正确性证明

Dijkstra 算法的正确性证明，也即证明该算法满足贪心选择性质和最优子结构性质。

（1）贪心选择性质

Dijkstra 算法是应用贪心法设计策略的又一个典型例子。它所做的贪心选择是从集合 V-S 中选择具有最短路径的顶点 t，从而确定从源点 u 到 t 的最短路径长度 dist[t]。这种贪心选择为什么能导致最优解呢?换句话说，为什么从源点到 t 没有更短的其它路径呢?

事实上，假设存在一条从源点 u 到 t 且长度比 dist[t]更短的路，设这条路径初次走出 S 之外到达的顶点为 x∈V-S，然后徘徊于 S 内外若干次，最后离开 S 到达 t。

在这条路径上，分别记 d(u，x)，d(x，t)和 d(u，t)为源点 u 到顶点 x，顶点 x 到顶点 t 和源点 u 到顶点 t 的路径长度，那么，依据假设容易得出：

dist[x]≤d(u，x)

d(u，x)+d(x，t)=d(u，t)<dist[t]

利用边权的非负性，可知 d(x，t)≥0，从而推得 dist[x]<dist[t]。此与前提矛盾，从而证明了 dist[t]是从源点到顶点 t 的最短路径长度。

（2）最优子结构性质

要完成 Dijkstra 算法正确性的证明，还必须证明最优子结构性质，即算法中确定的 dist[t]确实是当前从源点到顶点 t 的最短路径长度。为此，只要考察算法在添加 t 到 S 中后，dist[t]的值所起的变化就行了。将添加 t 之前的 S 称为老 S。当添加了 t 之后，可能出现一条到顶点 j 的新的特殊路径。如果这条新路径是先经过老的 S 到达顶点 t，然后从 t 经一条边直接到达顶点 j，则这条路径的最短长度是 dist[t]+C[t][j]。这时，如果 dist[t]+C[t][j]＜dist[j]，则算法中用 dist[t]+C[t][j]做为 dist[j]的新值。如果这条新路径经过老的 S 到达 t 后，不是从 t 经一条边直接到达 j，而是先回到老的 S 中某个顶点 x，最后才到达顶点 j，那么由于 x 在老的 S 中，因此 x 比 t 先加入 S，故从源点到 x 的路径长度比从源点到 t，再从 t 到 x 的路径长

度小。于是当前 dist[j]的值小于从源点经 x 到 j 的路径长度，也小于从源点经 t 和 x，最后到达 j 的路径长度。因此，在算法中不必考虑这种路径。可见，无论算法中 dist[t]的值是否有变化，它总是关于当前顶点集 S 到顶点 t 的最短路径长度。

## 2.4 哈夫曼编码

哈夫曼编码是一种编码方式，属于可变字长编码(VLC)的一种，该编码方式是数学家 D.A.Huffman 于 1952 年提出，其完全依据字符出现概率来构造异字头的平均长度最短的码字。简言之，哈夫曼编码算法是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，有时称之为最佳编码，一般就叫作 Huffman 编码。

哈夫曼编码是广泛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，常用的 JPEG 图片格式就是采用哈夫曼编码压缩的。

1．哈夫曼编码问题的引出

解决远距离通信以及大容量存储问题时，经常涉及到字符的编码和信息的压缩问题。一般来说，较短的编码能够提高通信的效率且节省磁盘存储空间。通常的编码方法有固定长度编码和不等长度编码两种。

（1）固定长度编码方法

假设所有字符的编码都等长，则表示 n 个不同的字符需要 logn 位，ASCII 码就是固定长度的编码。如果每个字符的使用频率相等的话，固定长度编码是空间效率最高的方法。但在信息的实际处理过程中，每个字符的使用频率有着很大的差异，现在的计算机键盘中的键的不规则排列，就是源于这种差异。

（2）不等长度编码方法。

不等长编码方法是今天广泛使用的文件压缩技术，其思想是：利用字符的使用频率来编码，使得经常使用的字符编码较短，不常使用的字符编码较长。这种方法既能节省磁盘空间，又能提高运算与通信速度。

但采用不等长编码方法要注意一个问题：任何一个字符的编码都不能是其它字符编码的前缀（即前缀码特性），否则译码时将产生二义性。那么如何来设计前缀编码呢？我们自然地想到利用二叉树来进行设计。具体做法是：约定在二叉树中用叶子结点表示字符，从根结点到叶子结点的路径中，左分支表示“0”，右分支表示“1”。那么，从根结点到叶子结点的路径分支所组成的字符串做为该叶子结点字符的编码，可以证明这样的编码一定是前缀编码，这棵二叉树即为编码树。

剩下的问题是怎样保证这样的编码树所得到的编码总长度最小？哈夫曼提出了解决该问题的方法，由此产生的编码方案称为哈夫曼算法。

2．哈夫曼算法的构造思想及设计

（1）算法的构造思想

该算法的基本思想是以字符的使用频率做权构建一棵哈夫曼树，然后利用哈夫曼树对字符进行编码，俗称哈夫曼编码。具体来讲，是将所要编码的字符作为叶子结点，该字符在文件中的使用频率作为叶子结点的权值，以自底向上的方式、通过执行 n-1 次的“合并”运算后构造出最终所要求的树，即哈夫曼树，它的核心思想是让权值大的叶子离根最近。

那么，在执行合并运算的过程中，哈夫曼算法采取的贪心策略是每次从树的集合中取出双亲为 0 且权值最小的两棵树作为左、右子树，构造一棵新树，新树根结点的权值为其左右孩子结点权之和，将新树插入到树的集合中。依照该贪心策略，执行 n-1 次合并后最终可构造出哈夫曼树。

（2）算法的设计

根据算法的构造思想所设计的哈夫曼算法求解步骤如下：

步骤 1：确定合适的数据结构。由于哈夫曼树中没有度为 1 的结点，则一棵有 n 个叶子结点的哈夫曼树共有 2n-1 个结点；构成哈夫曼树后，为求编码需从叶子结点出发走一条从叶子到根的路径；而译码则需从根出发走一条从根到叶子的路径。对每个结点而言，既需要知道双亲的信息，又需要知道孩子结点的信息，因此数据结构的选择要考虑这方面的情况；

步骤 2：初始化。构造 n 棵结点为 n 个字符的单结点树集合 $F=\{T_1, T_2,\ldots, T_n\}$，每棵树中只有一个带权的根结点，权值为该字符的使用频率；

步骤 3：如果 F 中只剩下一棵树，则哈夫曼树构造成功，转步骤 6；否则，从集合 F 中取出双亲为 0 且权值最小的两棵树 $T_i$ 和 $T_j$，将它们合并成一棵新树 $Z_k$，新树以 $T_i$ 为左儿子，$T_j$ 为右儿子（反之也可以）。新树 $Z_k$ 的根结点的权值为 $T_i$ 与 $T_j$ 的权值之和；

步骤 4：从集合 F 中删去 $T_i$、$T_j$，加入 $Z_k$；

步骤 5：重复步骤 3 和 4；

步骤 6：从叶子结点到根结点逆向求出每个字符的哈夫曼编码（约定左分支表示字符“0”，右分支表示字符“1”）。则从根结点到叶子结点路径上的分支字符组成的字符串即为叶子字符的哈夫曼编码。算法结束。

3．哈夫曼算法的构造实例

**【例 2-3】**已知某系统在通信联络中只可能出现 8 种字符，分别为 a，b，c，d，e，f，g，h，其使用频率分别为 0.05，0.29，0.07，0.08，0.14，0.23，0.03，0.11，试设计哈夫曼编码。

设权 w=(5，29，7，8，14，23，3，11)，n=8，按哈夫曼算法的设计步骤构造一棵哈夫曼编码树，具体过程如下：

（1）构造 8 棵结点为 8 种字符的单结点树，每棵树中只有一个带权的根结点，权值为该字符的使用频率，如图 2-3 所示。

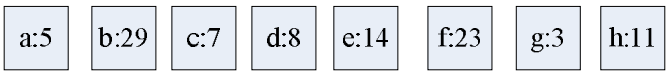



图 2-3　8 棵单结点树的集合

（2）从树的集合中取出两棵双亲为 0 且权值最小的两棵树，并将它们作为左、右子树合并成一棵新树，在树的集合中删去所选的两棵树，并将新树加入集合。

即从 8 棵树的集合中选出权值为 5 和 3 的两棵树，合并成根结点权值为 8 的新树，如图 2-4 所示，同时更新树的集合。此时，树的集合中共有 7 棵树，其根结点的权值分别为：8,29,7,8,14,23,11。

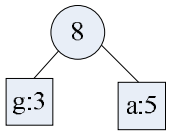



图 2-4 构造的一棵根结点权值为 8 的新树

按照步骤（2）的构造思想，依次类推，最终可构造出哈夫曼树。

（3）在 7 棵树的集合中选取根结点权值为 7 和 8 的两棵树，合并成根结点权值为 15 的新树，如图 2-5 所示，更新树的集合。此时，树的集合中共有 6 棵树，其根结点的权值为：8,29,15,14,23,11。

（4）从 6 棵树的集合中选取根结点权值为 8 和 11 的两棵树，合并成根结点权值为 19 的新树，如图 2-6 所示，更新树的集合。此时，树的集合中共有 5 棵树，其根结点的权值为：19, 29,15,14,23。

（5）从 5 棵树的集合中选取根结点权值为 15 和 14 的两棵树，合并成根结点权值为 29 的新树，如图 2-7 所示，更新树的集合。此时，树的集合中共有 4 棵树，其根结点的权值为：

19,29,29,23。

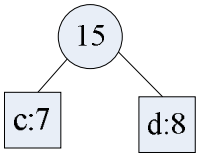



图 2-5 根结点权值为 15 的新树

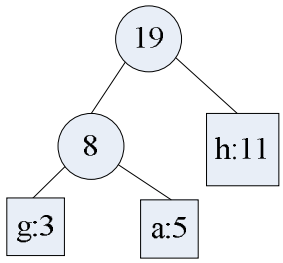



图 2-6 根结点权值为 19 的新树

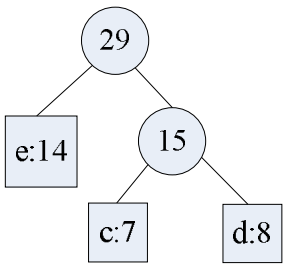



图 2-7 根结点权值为 29 的新树

（6）从 4 棵树的集合中选取根结点权值为 19 和 23 的两棵树，合并成根结点权值为 42 的新树，如图 2-8 所示，并更新树的集合。此时，树的集合中共有 3 棵树，其根结点的权值为：42,29,29。

（7）从 3 棵树的集合中选取根结点权值为 29 的两棵树，合并成根结点权值为 58 的新树，如图 2-9 所示，并更新树的集合。此时，树的集合中共有 2 棵树，根结点的权值为：42,58。

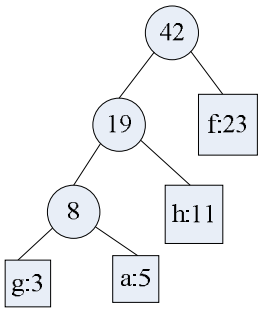



图 2-8 根结点权值为 42 的新树

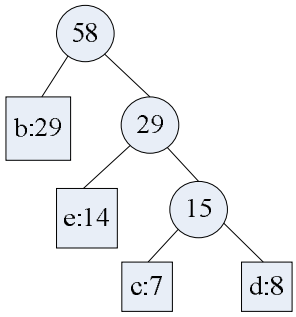



图 2-9 根结点权值为 55 的新树

（8）将树的集合中的两棵树合并成根结点权值为 100 的一棵树，即为哈夫曼树，如图 2-10 所示。

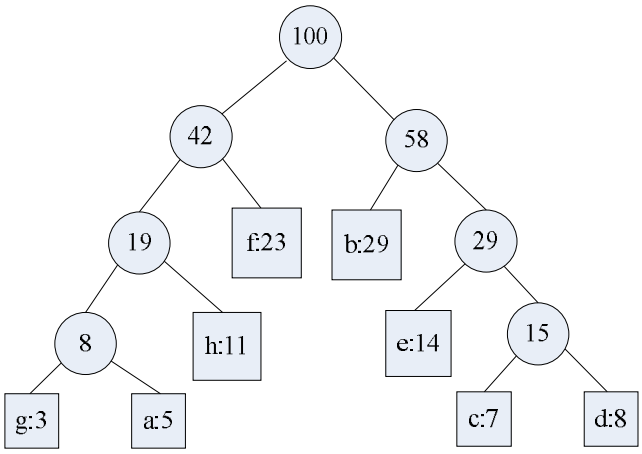



图 2-10 哈夫曼树

（9）哈夫曼编码树的构造

依据约定：左分支表示字符“0”，右分支表示字符“1”，获得的哈夫曼编码树如图 2-11 所示。

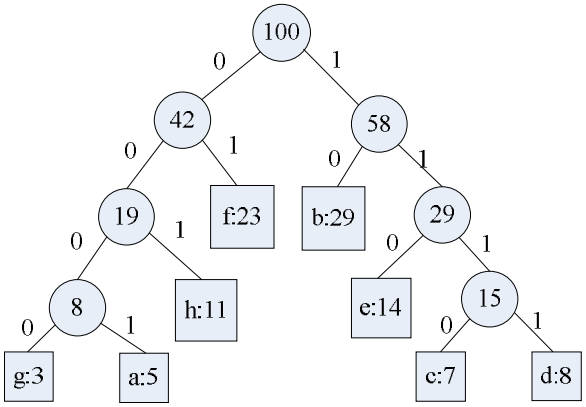



图2-11 哈夫曼编码树

由于从根结点到叶子结点路径上的分支字符组成的字符串即为叶子字符的哈夫曼编码，所以各个字符的哈夫曼编码分别为 g:0000；a:0001；h:001；f:01；b:10；e:110；c:1110；d:1111。

4．算法描述及分析

（1）算法所采用的数据结构

（a）哈夫曼树中结点的存储结构

考虑到有 n 个叶子结点的哈夫曼树共有 2n-1 个结点，并且需要进行 n-1 次合并操作。为了便于选取根结点权值最小的二叉树以及进行合并操作，树中每个结点的存储结构为：

```
struct HtNode{
double weight;
int parent,lchild,rchild;};
```

其中，weight 域表示结点的权值，即为该字符的使用频率；parent 域表示结点的双亲结点在数组中的下标。lchild 域表示结点的左孩子结点在数组中的下标。rchild 域表示结点的右孩子结点在数组中的下标。

（b）哈夫曼树的存储结构

```
struct HtTree{
        struct HtNode ht[MaxNode];//MaxNode 表示哈夫曼树中结点的总个数
        int root; //树根结点在数组中的下标};
typedef struct HtTree *PHtTree;
```

（2）线性结构上实现的哈夫曼树算法

假设 n 个叶子结点的权值 weight 已存放在数组 double w[n]中。

```
PHtTree Huffman (int n,double w[n]) //构造具有 n 个叶子结点的哈夫曼树
 {
  PHtTree pht;
  int i,j,p1,p2；// p1,p2 用于记录权值最小的两棵树在数组中的位置
  double min1,min2; // min1,min2 用于记录两个最小的权值
  if(n<=1)   return;
  pht=(PHtTree)malloc(sizeof(struct HtTree)); //动态分配哈夫曼树的空间
  pht->ht=(struct HtNode*)malloc (sizeof(struct HtNode)*(2*n-1));
  for( i=0;i<2*n-1;i++) //初始化，设置 ht 数组的初始值
     {
     pht->ht[i].parent=0;
     if(i<n)
         pht->ht[i].weight=w[i];
```

```
        else
            pht->ht[i].weight=-1;}
        for(i=0;i<n-1;i++) //执行 n-1 次合并操作，即构造哈夫曼树的 n-1 个内部结点
            {
                p1=p2=0;    min1=min2=∞; //相关变量赋初值
                for(j=0;j<n+i;j++) //在 ht 中选择权值最小的两个结点
                if (pht->ht[j].parent==0)
                    if(pht->ht[j].weight<min1) //查找权值最小的结点，用 p1 记录其下标
                     {
                       min2=min1;
                       min1=pht->ht[j].weight;
                       p2=p1;
                       p1=j;
                     }
                    else if(pht->ht[j].weigth<min2) //查找权值次小的结点，p2 记录其下标
                       {
                         min2=pht->ht[j].weight;
                         p2=j;
                       }
                pht->ht[p1].parent=n+i;     //将 ht[p1]和 ht[p2]进行合并，其双亲是 i
                pht->ht[p2].parent=n+i;
                pht->ht[n+i].weight=min1+min2;
                pht->ht[n+i].lchild=p1;pht->ht[n+i].rchild=p2;
             }
         return pht;//返回哈夫曼树指针
    }
```

（3）哈夫曼编码算法描述

注：从叶子到根逆向求每个字符的哈夫曼编码算法。

```
void HuffmanCode(char**HC, int n, PHtTree pht)
 {
  char c[n]; //每个字符的编码最大长度不会超过 n
  HC=(char**)malloc(n*sizeof(char*));   //分配 n 个字符指针数组的空间
  c[n-1]=“\0”;   //n-1 单元存储编码串的结束符
  for (i=1;i<=n;i++)   //逐个对 n 个字符求其哈夫曼编码
      { start=n-1;
       for(intk=i,f=pht->ht[i].parent;f!=0;k=f,f=pht->ht[k].parent) //k,f 是两个工作指针，f 指向 k 的父亲
           if(pht->ht[f].lchild==k)   c[--start]=”0”;
           else   c[--start]=”1”;
       HC[i]=(char*)malloc((n-start)*sizeof(char));//为第 i 个字符编码分配空间
       strcpy(HC[i],&c[start]);
        }
}
```

译码的过程是：从根出发，依照字符“0”或“1”确定找左孩子或右孩子，直至叶子结点，便可求得该编码串相应的字符。具体算法请读者自行考虑并设计。

（4）算法分析

从 Huffman 算法描述中可以看出，语句 if (pht->ht[j].parent==0)为基本语句，当外层循环标号 i=0 时，该语句执行 n 次，当 i=1 时，该语句执行 n+1 次，依此类推……，当 i=n-2 时，该语句执行 n+n-2 次。因此该语句共执行的次数为：n+(n+1)+…+(n+(n-2))=(n-1)*(3n-2)/2。因此，算法的时间复杂性为 $O(n^2)$。HuffmanCode 算法的基本语句为 if(pht->ht[f].lchild==k)，该语句最多执行 $n^2$ 次，其复杂性为 $O(n^2)$。

5．算法的改进

Huffman 算法耗时主要集中在从树的集合中取出两棵权值最小的树，因此算法改进以此入手，选用极小堆来实现该操作。

（1）改进算法中所采用的数据结构

（a）哈夫曼树中结点的存储结构同上。

（b）哈夫曼树的存储结构

```
struct HtTree{
    int n;//叶子结点的个数
    int root; //哈夫曼树的根结点在数组中的下标
    struct HtNode *ht;};
typedef struct HtTree *PHtTree;
```

（c）极小堆的类型声明如下：

```
typedef int element_type;
struct heap_struct{
    unsigned int max_heap_size; //堆中最多容纳的元素个数
    unsigned int size; //堆中目前的元素个数
    element_type *elements;};
typedef struct heap_struct *PRIORITY_QUEUE;
```

（2）采用极小堆实现的哈夫曼改进算法描述如下：

```
PHtTree Huffmane (int n,double w[n]) //构造具有 n 个叶子结点的哈夫曼树
  {
   PHtTree pht; PRIORITY_QUEUE pq; int i ,p1,p2;
   if(n<=1) return;
   pht=(PHtTree)malloc(sizeof(struct PHtTree)); //动态分配哈夫曼树的空间
   pht->ht=(struct HtNode*)malloc (sizeof(struct HtNode)*(2*n-1));
   for( i=0;i<2*n-1;i++) //初始化，设置 ht 数组的初始值
     {
      pht->ht[i].parent=0;
      if(i<n)
          pht->ht[i].weight=w[i];
      else
          pht->ht[i].weight=-1;
       }// end for
      pq=creat_pq(n);//创建优先队列 pq
      for(i=0;i<n;i++)
           insert(pht->ht[i].weight,pq);//用 n 个字符的使用频率来初始化优先队列 pq
       for(i=0;i<n-1;i++) //构造哈夫曼树，每循环一次构造出一个内部结点
          {
```

```
            p1=p2=0; //结点在优先队列中的下标
            p1=delete_min(pq); p2=delete_min(pq);//从优先队列 pq 中取出权值最小的两个结点
            pht->ht[p1].parent=n+i;pht->ht[p2].parent=n+i;
            pht->ht[n+i].weight=pht->ht[p1].weight+ pht->ht[p2].weight;
            pht->ht[n+i].lchild=p1;pht->ht[n+i].rchild=p2;
            insert(pht->ht[n+i].weight,pq);   //将新构造的结点插入优先队列 pq 中
        }
    pht->root=2*n-2; return pht;   //pht 指向根结点
}
```

（3）算法分析

改进算法是采用极小堆来实现哈夫曼树中结点的存储和操作，那么初始化 creat_pq 需要O(n)的时间，由于极小堆的 delete_min 和 insert 运算均需 O(logn)时间，n-1 次的合并操作总共需要 O(nlogn)的时间，因此，关于 n 个字符的哈夫曼算法的时间复杂性为 O(nlogn)。

## 2.5 最小生成树

假设要在 n 个城市之间建立通信网络，则连通 n 个城市至少需要 n-1 条线路。n 个城市之间，最多可能设置 n(n-1)/2 条线路。这时，自然会考虑一个问题：如何在这些可能的线路中选择 n-1 条，使得在最节省费用的前提下建立该通信网络？

该问题用无向连通带权图 G=(V，E)来表示通信网络，图的顶点表示城市，顶点与顶点之间的边表示城市之间的通信线路，边的权值表示线路的费用。对于 n 个顶点的连通网可以建立许多包含 n-1 条通信线路且各个城市互连通的通信网，该通信网称为图 G 的一棵生成树。现在要选择一棵使得总的耗费最少的生成树，这就是构造连通网的最小费用生成树问题。一棵生成树的费用就是树上各边的权值之和。

怎样找出这个最小生成树呢？最直观的方法是将所有情况都枚举出来之后通过排序比较找出。但是可以预料这种方法所需要的时间是非常大的。而且由于没有一个固定的标准，很难用代码来实现枚举。

在这样的情况下，Prim 算法和 Kruskal 算法就应运而生了，它们是使用贪心法设计策略的典型算法，并且最终都会产生一个最优解。这两大算法主要利用了最小生成树的一条很重要的性质来进行问题的求解。该性质是：假设 G=(V，E)是一个连通网，U 是顶点集 V 中的一个非空子集。若(u，v)是一条具有最小权值的边，其中 $u \in U$，$v \in V-U$，则必存在一棵包含边(u，v)的最小生成树。

### 2.5.1 Prim 算法

该算法是 R. C. Prim 在 1957 年提出的，不过他并不全是这个算法最最祖先的人，早在 1930 年捷克人 V. Jarník 就在文章中提出了该算法，因此有人也把这个算法叫做 Prim-Jarnik 算法。该算法因其简单有效，至今仍被人们津津乐道。

1．算法的基本思想

设 G=(V，E)是无向连通带权图，V={1,2,…,n}；设最小生成树 T=(U，TE)，算法结束时 U=V，$TE \subseteq E$。构造最小生成树 T 的 Prim 算法思想是：首先，令 $U=\{u_0\}$，$u_0 \in V$，TE={}。然后，只要 U 是 V 的真子集，就做如下贪心选择：选取满足条件 $i \in U$，$j \in V-U$，且边(i，j)是连接 U 和 V-U 的所有边中的最短边，即该边的权值最小。然后，将顶点 j 加入集合 U，

边(i，j)加入集合 TE。继续上面的贪心选择一直进行到 U=V 为止，此时，选取到的所有边恰好构成 G 的一棵最小生成树 T。需要注意的是，贪心选择这一步骤在算法中应执行多次，每执行一次，集合 TE 和 U 都将发生变化，即分别增加一条边和一个顶点。因此，TE 和 U 是两个动态的集合，这一点在理解算法时要密切注意。

2．算法设计

从算法的基本思想可以看出，实现 Prim 算法的关键是如何找到连接 U 和 V-U 的所有边中的最短边？要实现这一点，必须知道 V-U 中的每一个顶点与它所连接的 U 中的每一个顶点的边的信息，该信息可通过设置两个数组 closest 和 lowcost 来体现。其中，closest[j]表示 V-U 中的顶点 j 在集合 U 中的最近邻接顶点，lowcost[j]表示 V-U 中的顶点 j 到 U 中的所有顶点的最短边的权值，即边(j，closest[j])的权值。

算法的求解步骤设计如下：

步骤 1：确定合适的数据结构。设置带权邻接矩阵 C，如果图 G 中存在边(u，x)，令 C[u][x]=边(u，x)上的权值，否则，C[u][x]=∞；bool 数组 s[]，如果 s[i]=true，说明顶点 i 已加入集合 U；设置两个数组 closest[]和 lowcost[]；

步骤 2：初始化。令集合 U={$u_0$}，$u_0 \in V$，TE={}，并初始化数组 closest、lowcost 和 s；

步骤 3：在集合 V-U 中寻找使得 lowcost 具有最小值的顶点 t，即：$lowcost[t] = \min\{lowcost[x] \mid x \in (V-U)\}$，满足该公式的顶点 t 就是集合 V-U 中连接集合 U 中的所有顶点中最近的邻接顶点；

步骤 4：将顶点 t 加入集合 U，边(t，closest[t])加入集合 TE；

步骤 5：如果集合 V=U，算法结束，否则，转步骤 6；

步骤 6：对集合 V-U 中的所有顶点 k，更新其 lowcost 和 closest，用下面的公式更新：if (C[t][k]<lowcost[k]) {lowcost[k]=C[t][k];closest[k]=t;}，转步骤 3。

按照上述步骤，最终可求得一棵各边上的权值之和最小的生成树。

3．Prim 算法的构造实例

**【例 2-4】**按 Prim 算法对如图 2-12 所示的无向连通带权图构造一棵最小生成树。

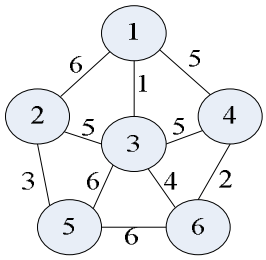



图 2-12 无向连通带权图

假定初始顶点为顶点 1，即设定最小生成树 T 的顶点集合 U={1}；t：集合 V-U 中距离集合 U 最近的顶点。构造过程如下：

（1）构造过程中各个辅助变量如表 2-4 所示：

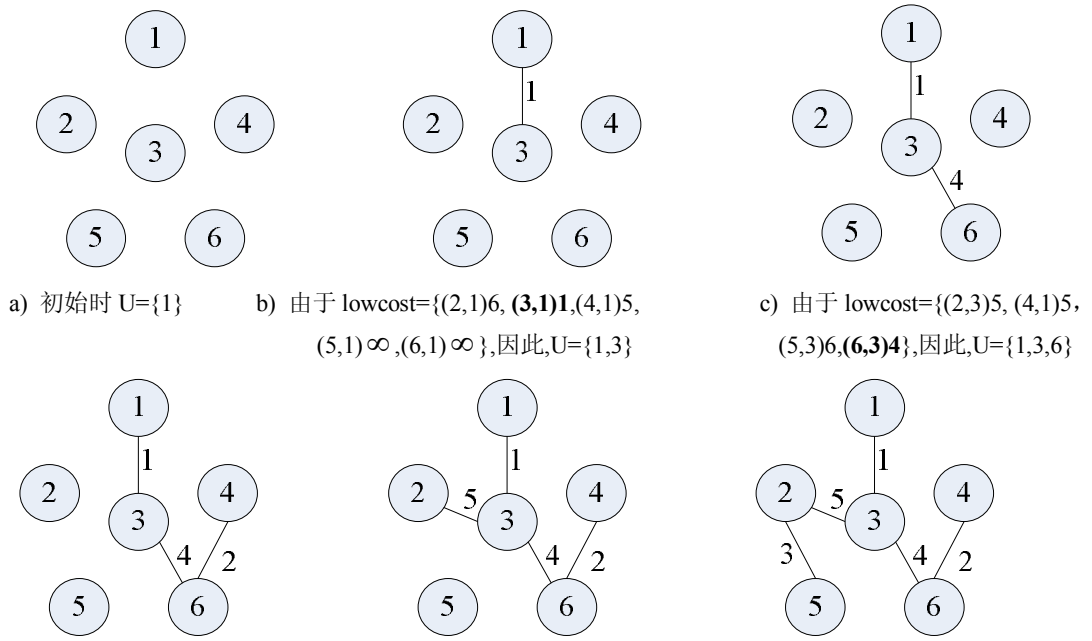
表 2-4 辅助变量值

| | U | V-U | 顶点编号 | | | | | t |
|---|---|---|---|---|---|---|---|---|
| | | | 2 | 3 | 4 | 5 | 6 | |
| closest<br>lowcost | {1} | {2,3,4,5,6} | 1<br>6 | 1<br>1 | 1<br>5 | 1<br>∞ | 1<br>∞ | 3 |
| closest<br>lowcost | {1,3} | {2,4,5,6} | 3<br>5 | -<br>- | 1<br>5 | 3<br>6 | 3<br>4 | 6 |
| closest | {1,3,6} | {2,4,5} | 3 | - | 6 | 3 | - | 4 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| lowcost | | | 5 | - | 2 | 6 | - | |
| closest<br>lowcost | {1,3,4,6} | {2,5} | 3<br>5 | -<br>- | -<br>- | 3<br>6 | -<br>- | 2 |
| closest<br>lowcost | {1,2,3,4,6} | {5} | -<br>- | -<br>- | -<br>- | 2<br>3 | -<br>- | 5 |
| closest<br>lowcost | {1,2,3,4,5,6} | {} | -<br>- | -<br>- | -<br>- | -<br>- | -<br>- | |

（2）构造过程的图示法

按照算法思想及设计步骤，构造最小生成树的过程如图 2-13 所示。




a) 初始时 U={1}　　b) 由于 lowcost={(2,1)6, **(3,1)1**,(4,1)5, (5,1)∞,(6,1)∞},因此,U={1,3}　　c) 由于 lowcost={(2,3)5, (4,1)5, (5,3)6,**(6,3)4**},因此,U={1,3,6}

d) 由于 lowcost ={(2,3)5, **(4,6)2**, (5,3)6},因此,U={1,3,6,4}　　e) 由于 lowcost={**(2,3)5**,(5,3)6}, 因此,U={1,3,6,4,2}　　f) 由于 lowcost={(5,2)3} 因此,U={1,3,6,4,2,5},结束

图 2-13　Prim 算法构造最小生成树的过程示意图

4．算法描述

```
void Prim(int n，int u0，int C[n][n]) //顶点个数 n、开始顶点 u0、带权邻接矩阵 C[n][n]
  { //如果 s[i]=true，说明顶点 i 已加入最小生成树的顶点集合 U；否则顶点 i 属于集合 V-U
    bool s[n]; int closest[n]; double lowcost[n];
    s[u0]=1;    //初始时，集合 U 中只有一个元素，即顶点 u0
    for(int i=0;i<n; i++)
      if(i!= u0)
        {
          lowcost[i]=C[u0][i];
          closest[i]=u0;
          s[i]=false;
        }
   for (i=0;i<n; i++) //在集合 V-U 中寻找距离集合 U 最近的顶点并且更新 lowcost 和 closest
      {
        double temp=∞; int t=u0;
        for(int j=0;j<n;j++)   //在集合 V-U 中寻找距离集合 U 最近的顶点 t
```

```
            if((!s[j])&&(lowcost[j]<temp))
              {
                t=j;
                temp=lowcost[j];
              }
          if(t==u0)
              break;   //找不到 t，跳出循环
          s[t]=true;   //否则，将 t 加入集合 U
          for(j=0;j<n; j++) //更新 lowcost 和 closest
              if((!s[j])&&(C[t][j]<lowcost[j])
                {
                  lowcost[j]=C[t][j];
                  closest[j]=t;
                }
      }
  }
```

5．算法分析

从算法的描述可以看出，语句 if((!s[j])&&(lowcost[j]<temp))是算法的基本语句，该语句的执行次数为 $n^2$，由此可得 Prim 算法的时间复杂度为 $O(n^2)$。显然该复杂性与图中的边数无关，因此，Prim 算法适合于求边稠密的网的最小生成树。

容易看出，Prim 算法和 Dijkstra 算法的用法非常相似，它们都是从余下的顶点集合中选择下一个顶点来构建一棵扩展树，但是千万不要把它们混淆了。由于解决的问题不同，计算的方式也有所不同，Dijkstra 算法比较路径的长度，因此必须把边的权值相加，而 Prim 算法则直接比较给定的边的权值。

### 2.5.2 Kruskal 算法

该算法是由 J.B.Kruskal 在 1956 年的论文里面提出来的，也是一个经典的算法。它从边的角度出发，每一次将图中的权值最小的边取出来，在不构成环的情况下，将该边加入最小生成树。重复这个过程，直到图中所有的顶点都加入到最小生成树中，算法结束。

1．算法的基本思想

设 G=(V，E)是无向连通带权图，V={1,2,…,n}；设最小生成树 T=(V，TE)，该树的初始状态为只有 n 个顶点而无边的非连通图 T=(V，{})，Kruskal 算法将这 n 个顶点看成是 n 个孤立的连通分支。它首先将所有的边按权从小到大排序。然后，只要 T 中的连通分支数目不为 1，就做如下的贪心选择：在边集 E 中选取权值最小的边(i，j)，如果将边(i，j)加入集合 TE 中不产生回路（或环），则将边(i，j)加入边集 TE 中，即用边(i，j)将这两个连通分支合并连接成一个连通分支；否则继续选择下一条最短边。在这两种情况下，都把边(i，j)从集合 E 中删去。继续上面的贪心选择直到 T 中所有顶点都在同一个连通分支上为止。此时，选取到的 n-1 条边恰好构成 G 的一棵最小生成树 T。

2．算法设计

Kruskal 算法俗称避环法。可见，该算法的实现关键是要注意避开环路问题，算法的基本思想恰好说明了这一点。那么，怎样判断加入某条边后图 T 会不会出现回路呢？该算法对于手工计算十分方便，因为用肉眼可以很容易看到挑选哪些边能够避免构成回路，但使用计算机程序来实现时，还需要一种机制来进行判断。Kruskal 算法用了一个非常聪明的方法，

就是运用集合的性质进行判断：如果所选择加入的边的起点和终点都在 T 的集合里，那么就可以断定一定会形成回路。因为，根据集合的性质，如果两个点在一个集合里，就是包含的关系，那么反映到图上就一定是包含关系。

算法的求解步骤设计如下：

步骤 1：初始化。将图 G 的边集 E 中的所有边按权从小到大排序，边集 TE={}，把每个顶点都初始化为一个孤立的分支，即一个顶点对应一个集合；

步骤 2：在 E 中寻找权值最小的边(i，j)；

步骤 3：如果顶点 i 和 j 位于两个不同的连通分支，则将边(i，j)加入边集 TE，并执合并操作将两个连通分支进行合并；

步骤 4：将边(i，j)从集合 E 中删去，即 E=E-{(i，j)}；

步骤 5：如果连通分支数目不为 1，转步骤 2；否则，算法结束，生成最小生成树 T。

3．Kruskal 算法的构造实例

**【例 2-5】**按 Kruskal 算法对如图 2-12 所示的无向连通带权图构造一棵最小生成树。

首先，将图的边集 E 中的所有边按权从小到大排序为：1(1，3)、2(4，6)、3(2，5)、4(3，6)、5(1，4) 和(2，3)和(3，4)、6(1，2)和(3，5)和(5，6)。

依照 Kruskal 算法思想和求解步骤可知，图 2-12 中边权为 1，2，3，4 的 4 条边满足要求，则在图 2-14a)~图 2-14d)中先后把它们加入到边集 TE 中，边权为 5 的两条边(1，4) 和(3，4) 被舍去，因为它们依附的两个顶点在同一连通分支上，若把它们加入边集 TE 中，则会产生回路，而另一条边权为 5 的边(2，3)依附的两个顶点则在不同的连通分支上，则将其加入边集 TE 中，如图 2-24e)所示。至此，一棵最小生成树就构造成功了。

具体构造过程如图 2-15 所示。

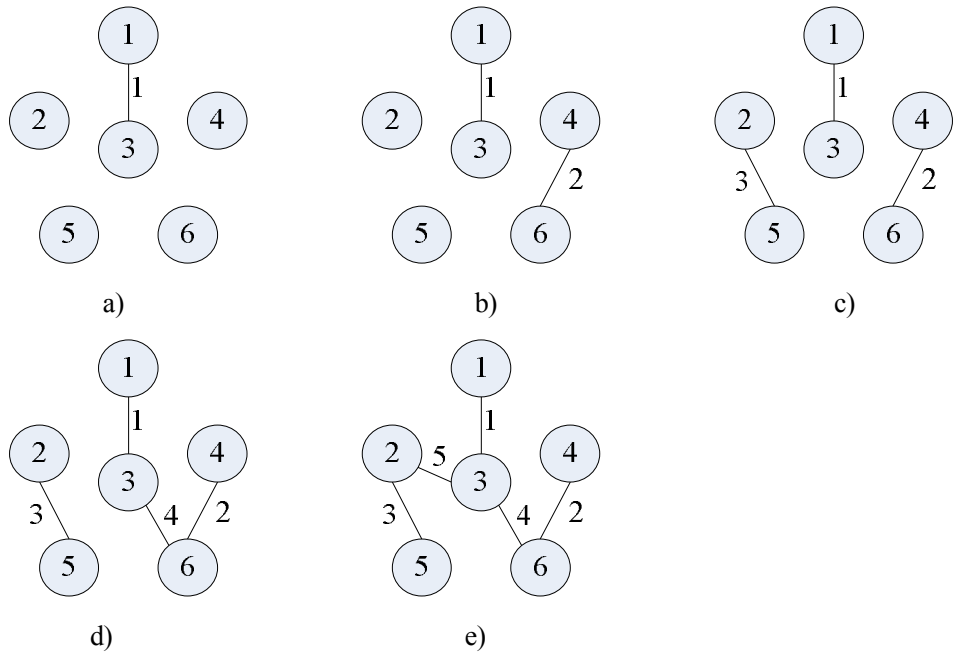



图 2-14　Kruskal 算法构造最小生成树的过程示意图

4．算法描述

从算法的基本思想和设计步骤可以看出，要实现 Kruskal 算法必须解决以下两个关键问题：（1）选取权值最小的边的同时，要判断加入该条边后树中是否出现回路；（2）不同的连通分支如何进行合并。

为了便于解决这两大问题，必须了解每条边的信息。那么，在 Kruskal 算法中所设计的

每条边的结点信息存储结构如下：

```
struct edge{
        double weight;   //边的权值
        int u，v;   // u 为边的起点，v 为边的终点。
 }bian[];
```

Kruskal 算法描述如下：

```
void Kruskal(int n，struct edge bian[]，double C[n][n])//  顶点个数 n、带权邻接矩阵 C[n][n]
{
   int nodeset[n];//顶点所属的集合
   int count=1;bool flag[n+1];
   if(n==1) return;
   for(int i=1;i<=n;i++)
    {
      nodeset[i]=i;flag[i]=false;
      for(int j=1;j<=n;j++) //将图中所有边存于数组 bian 中
        if(C[i][j]< ∞ )
          {
           bian[count].u=i;
           bian[count].v=j;
           bian[count].weight=C[i][j];
           count++;
           }
     }
   sort(bian+1,bian+count);//sort 函数将数组 bian 中的元素按 weight 的大小进行排列
   count=1; int edgeset=0; //存储最小生成树中边的个数
   int w=0;//最小生成树的耗费
   while (edgeset<(n-1)) //不足 n-1 条边
    {
     if(!flag[bian[count].u])&&(flag[bian[count].v]))//u 未加入任何集合,v 已加入某个集合
        {
         w+=bian[count].weight; edgeset++;
         flag[bian[count].u]=true;//将 u 加入某一集合
         nodeset[bian[count].u]=nodeset[bian[count].v];
        }
//记录 u 点所加入的集合
       else if ((flag[bian[count].u]) && (!flag[bian[count].v]))
            {
             w+=bian[count].weight; edgeset++; flag[bian[count].v]=true;
             nodeset[bian[count].v]=nodeset[bian[count].u];
            }
         else if((!flag[bian[count].u]) && (!flag[bian[count].v])) //u、v 均未加入任何集合
              {
               w+=bian[count].weight; edgeset++; flag[bian[count].u]=true;
               flag[bian[count].v]=true;nodeset[bian[count].u]=nodeset[bian[count].v];
```

```
            }
        else //两端顶点都加入了 set，判断新加入的边是否使结点形成了环
            if(nodeset[bian[count].u]!=nodeset[bian[count].v])//若无环
              {
                w+=bian[count].weight;edgeset++; int tmp=nodeset[bian[count].v];
                for(int i=1; i <= n; i++) //将两个集合中的元素合到一个集合中
                  if(nodeset[i]==tmp)  nodeset[i]=nodeset[bian[count].u];
              }
      count ++;
    }//end while
}// end Kruskal
```

5．算法分析

假设无向连通带权图 G 中包含了 n 个顶点和 e 条边。

从算法的描述可以看出，Kruskal 算法为了提高贪心选择时查找最短边的效率，首先将图 G 中的边按权值从小到大排序，因此算法中耗时最大的语句是：sort(bian+1,bian+count)；那么，将 e 条边按权值从小到大的顺序排列时，sort 函数所需的时间是 O(eloge)，由此可得，Kruskal 算法的时间复杂性为 O(eloge)。此外，如果图 G 是一个完全图，那么有 e=n(n-1)/2，则用顶点个数 n 来衡量算法所花费的时间为 $O(n^2 logn)$；如果图 G 是一个平面图，即 e=O(n)，则算法花费的时间为 O(nlogn)。

此外，算法的辅助空间包含为数组 nodeset 和 flag 所分配的空间，其余辅助变量占用的空间为 O(1)，因此，算法的空间复杂性为 O(n)。

### 2.5.3 两种算法的比较

设无向连通带权图 G 具有 n 个顶点和 e 条边。

（1）从算法的思想可以看出，如果图 G 中的边数较小时，可以采用 Kruskal，因为 Kruskal 算法每次查找最短的边；边数较多可以用 Prim 算法，因为它是每次加一个顶点。可见，Kruskal 适用于稀疏图，而 Prim 适用于稠密图。

（2）从时间上讲，Prim 算法的时间复杂度为 $O(n^2)$，Kruskal 算法的时间复杂度为 O(eloge)。

（3）从空间上讲，显然在 Prim 算法中，只需要很小的空间就可以完成算法，因为每一次都是从个别点开始出发进行扫描的，而且每一次扫描也只扫描与当前顶点集对应的边。但在 Kruskal 算法中，因为时刻都得知道当前边集中权值最小的边在哪里，这就需要对所有的边进行排序，对于很大的图而言，Kruskal 算法需要占用比 Prim 算法大得多的空间。

## 阅读材料 2——遗传算法

遗传算法（Genetic Algorithm，简写为遗传算法）试图从解释自然系统中生物的复杂适应过程入手，采用生物进化的机制来构造人工系统模型。它是从达尔文进化论中得到灵感和启迪，借鉴自然选择和自然进化的原理，模拟生物在自然界中的进化过程所形成的一种优化求解方法。它最初由 J.Holland 教授和他的学生于 1975 年提出来，并且他们出版了颇有影响的专著《Adaptation in Natural and Artificial Systems》。自此，遗传算法这个名称才逐渐为人所知，J.Holland 教授所提出的遗传算法通常称为基本遗传算法。

遗传算法自从被提出之后，经历了一个相对平衡的发展时期。但在进入 90 年代之后，遗传算法迎来了兴盛时期，无论是理论研究还是应用研究都成了十分热门的课题，尤其是遗传算法的应用研究显得格外活跃，原因在于该算法是一种自适应随机搜索方法，它对优化对象既不要求连续，也不要求可微，并具有极强的鲁棒性和内在的并行计算机制。因此，随着研究的不断深入，它的应用领域逐渐扩大，而且利用遗传算法进行优化和规则学习的能力也显著提高。此外，一些新的理论和方法在应用研究中亦得到了迅速发展，这些无疑给遗传算法增添了新的活力。

多年的不懈研究证明，遗传算法特别适合于非凸空间中复杂的多极值优化和组合优化问题。它在机器学习、自动控制、机器人技术、电气自动化以及计算机和通信等领域已取得了非凡的成就。

1．遗传算法的基本思想

遗传算法从代表问题的可能潜在解集的一个种群(population)出发，一个种群由一定数目的个体(individual)组成，每个个体实际上是染色体带有特征的实体。初始群体产生后，按照适者生存和优胜劣汰的原理，逐代演化产生越来越好的个体。在每一代，根据个体的适应度大小挑选个体，并借助于自然遗传学的遗传算子进行组合交叉和变异，产生出新的种群。整个过程类似于自然的进化，最后末代种群中的最优个体经过解码，可以作为问题的近似最优解。

遗传算法中使用适应度这个概念来度量种群中的各个个体在优化过程中有可能达到最优解的优良程度，度量个体适应度的函数称为适应度函数，该函数的定义一般与具体问题有关。

2．遗传算法的求解步骤

从算法的基本思想可知遗传算法是一种自适应寻优技术，可用来处理复杂的线性、非线性问题。但它的工作机理十分简单，主要采用了如选择，交叉，变异等自然进化操作。遗传算法的求解步骤如下：

步骤 1：构造满足约束条件的染色体。由于遗传算法不能直接处理解空间中的解，所以必须通过编码将解表示成适当的染色体。实际问题的染色体有多种编码方式，染色体编码方式的选取应尽可能地符合问题约束，否则将影响计算效率。

步骤 2：随机产生初始群体。初始群体是搜索开始时的一组染色体，其数量应适当选择。

步骤 3：计算每个染色体的适应度。适应度是反映染色体优劣的唯一指标，遗传算法就是要寻找适应度最大的染色体。

步骤 4：使用复制、交叉和变异算子产生子群体。这三个算子是遗传算法的基本算子，其中复制体现了优胜劣汰的自然规律，交叉体现了有性繁殖的思想，变异体现了进化过程中的基因突变。

步骤 5：若满足终止条件，则输出搜索结果；否则返回步骤 3。

上述算法中，适应度是对染色体（个体）进行评价的指标，是遗传算法进行优化所用的主要信息，它与个体的目标值存在一种对应关系；复制操作通常采用比例复制，即复制概率正比于个体的适应度；交叉操作通过交换两父代个体的部分信息构成后代个体，使得后代继承父代的有效模式，从而有助于产生优良个体；变异操作通过随机改变个体中某些基因而产生新个体，有助于增加种群的多样性，避免早熟收敛。

将遗传算法应用于实际问题中，通常需做以下工作：

（1）寻求有效的编码方法，将问题的可能解直接或间接地编码成有限位字符串；

（2）产生一组问题的可行解；

（3）确定遗传算子的类型及模式，如复制、交叉、变异、重组、漂移等等，并设计其模式；

（4）确定算子所涉及的参数，如种群规模 $n$，交叉概率 $p_c$，变异概率 $p_m$ 等等；

（5）根据编码方法和问题的实际需求，定义或设计一个适应度函数，以测量和评价各个解的性能优劣；

（6）确定算法的收敛判据；

（7）根据编码方法，设计译码方法。

3．遗传算法的特点

遗传算法利用生物进化和遗传的思想实现优化过程，区别于传统优化算法，它具有以下特点：

（1）遗传算法对问题参数编码成“染色体”后进行进化操作，而不是针对参数本身，这使得遗传算法不受函数约束条件的限制，如连续性、可微性；

（2）遗传算法的搜索过程是从问题解的一个集合开始的，而不是从单个个体开始的，具有隐含并行搜索特性，从而大大减小了陷入局部极小的可能；

（3）遗传算法使用的遗传操作均是随机操作，同时遗传算法根据个体的适应度信息进行搜索，无需其他信息，如导数信息等；

（4）遗传算法具有全局搜索能力，最善于搜索复杂问题和非线性问题。

遗传算法的优越性主要表现在：

（1）算法进行全空间并行搜索，并将搜索重点集中于性能高的部分，即使在所定义的适应函数是不连续的、非规则的或有噪声的情况下，它也能以很大的概率找到整体最优解，不易陷入局部极小，从而能够提高效率；

（2）算法具有固有的并行性，通过对种群的遗传处理可处理大量的模式，并且容易并行实现。

4．遗传算法的理论基础

遗传算法理论研究的主要内容有分析遗传算法的编码策略、全局收敛性和搜索效率的数学基础、遗传算法的新结构研究、基因操作策略及其他算法的比较研究等。

（1）数学基础

Holland 的模式理论（定义长度短、低阶并且适应度高于群体平均适应度的模式数量随代数的增加呈指数增长）奠定了遗传算法的数学基础，根据隐含并行性可得出每代处理的有效模式是 $O(n^3)$，其中 $n$ 是种群规模。Bertoni 和 Dorigo 推广了该研究，获得 $n=2^{\beta l}$，$\beta$ 为任意值时，处理多少有效模式的表达式，其中 $l$ 为染色体长度。

模式定理中模式适应度难以计算和分析，Bethke 运用 Walsh 函数和模式转换发展了有效的分析工具，Holland 扩展了这种计算。后来 Frantz 首先觉察到了一种常使遗传算法从全局最优解发散出去的问题，称为遗传算法-欺骗问题。Goldberg 首先运用 Walsh 模式转换设计了最小遗传算法-欺骗问题，并进行了详细分析。

近几年来，在遗传算法全局收敛分析方面取得了突破。Goldberg 和 Segrest 首先使用马尔可夫链分析了遗传算法，Eiben 等用马尔可夫链证明了保留最优个体的遗传算法的全局收敛性，Rudolph 用齐次有限马尔可夫链证明了带有复制、交叉、变异操作的标准遗传算法收敛不到全局最优解，不适合于静态函数优化问题，建议改变复制策略以达到全局收敛。Back 和 Muhlenberg 研究了达到全局最优解的遗传算法的时间复杂性问题。以上收敛性分析是基于简化了的遗传算法模型，复杂遗传算法的收敛性分析仍是困难的。

Holland 模式定理建议采用二进制编码。由于浮点数编码有精度高、便于大空间搜索的优点，浮点数编码越来越受到重视，Michalewicz 比较了两种编码的优缺点。 Qi 和 Palmieri 对浮点数编码的遗传算法进行了严密的数学分析，用马尔可夫链建模，进行了全局收敛性分析，但其结果是基于群体无穷大这一假设的。另外，Kazz 还发展了用计算机程序

来编码，开创了新的应用领域。

（2） 算法结构研究

包括在遗传算法的基本结构中插入迁移、显性、倒拉等其它高级遗传算子和启发式知识，以及针对复杂约束问题，研究遗传算法并行实现的结构等，如 Gretenstette 提出的同步主从式、半同步主从式、非同步分布及网络式结构形式。

（3）遗传算子

遗传算子包括复制、交叉和变异，适应度尺度变换和最佳保留等策略可视为遗传复制的一部分，另外，还有许多用的较少、作用机理尚不明或没有普遍意义的高级遗传算子。这部分内容的研究在遗传算法理论研究中最为丰富多彩。

（4）遗传算法参数选择

遗传算法中需要选择的参数主要有染色体长度、种群规模、交叉概率和变异概率等，这些参数对遗传算法性能影响很大。许多学者对此进行了研究，建议的最优参数范围是：$n$=20~200，$p_c$=0.75~0.95，$p_m$=0.005~0.01。

（5）与其它算法的综合及比较研究

由于遗传算法的结构是开放式的，与问题无关，所以容易和其它算法综合。例如 Lin 等把遗传算法和模拟退火进行综合，构成模拟遗传算法，在解决一些 NP 难问题时显示了良好的性能，时间复杂性为 $O(n^3)$。

（6） 遗传算法的应用研究

遗传算法的应用研究总的来说可分为优化计算、机器学习和神经网络三大类。其中优化计算是遗传算法最直接的应用，应用面也最广，目前在运筹学、机械优化、电网设计、生产管理、结构优化、VLSL 设计等应用学科中都尝试着用遗传算法解决现实优化计算问题。

5．遗传算法的典型应用及研究动向

上述内容表明：遗传算法提供了一种求解复杂系统问题的通用框架，即不依赖于问题的具体领域，对问题的种类有很强的鲁棒性，所以它被广泛应用于许多科学。下面介绍一下遗传算法的主要应用领域。

（1）函数优化。函数优化是遗传算法应用最早也是最经典的领域，也是进行遗传算法性能评价的常用算例。许多人构造出了各种各样复杂形式的测试函数：连续函数和离散函数、凸函数和凹函数、低维函数和高维函数、单峰函数和多峰函数等。对于一些非线性、多模型、多目标的函数优化问题，用其它优化方法较难求解，而遗传算法可以方便的得到较好的结果。

（2）组合优化。例如遗传算法在旅行商问题方面的应用，背包，装箱问题等等。

（3）人工生命。遗传算法在人工生命以及复杂系统的模拟设计等方面有很广阔的应用前景。

（4）机器学习。基于遗传算法的机器学习，在许多领域得到了应用。例如利用遗传算法调节神经网络的权值，还可用于神经网络的优化设计等等。

此外，遗传算法在图像处理，自动控制，生产调度等方面也有广泛应用。

随着应用领域的不断扩展，遗传算法的研究出现了几个引人注目的新动向。

（5）基于遗传算法的机器学习。这一新的研究课题把遗传算法从历来离散的搜索空间的优化搜索算法扩展到具有独特的规则生成功能的崭新的机器学习算法，这对于解决人工智能中知识获取和知识优化精炼的瓶颈难题带来了希望。

（6）遗传算法正日益和神经网络、模糊推理以及混沌理论等其它智能计算方法相互渗透和结合。这对开拓 21 世纪中新的智能计算技术将具有重要的意义。

（7）并行处理的遗传算法研究。这一研究不仅对遗传算法本身的发展，而且对于新一

代智能计算机体系结构的研究都是十分重要的。

（8）遗传算法和人工生命研究领域的渗透。所谓人工生命即是用计算机模拟自然界丰富多彩的生命现象，其中生物的自适应、进化和免疫等现象是人工生命的重要研究对象，而遗传算法在这方面将会发挥一定的作用。

（9）遗传算法、进化规划以及进化策略等进化计算理论日益结合。它们几乎是和遗传算法同时独立发展起来的，同遗传算法一样，它们也是模拟自然界生物进化机制的智能计算方法，即同遗传算法具有相同之处，也有各自的特点。目前，这三者之间的比较研究和彼此结合的探讨正形成热点。

参考文献

[1]周春光, 梁艳春. 计算智能. 长春: 吉林大学出版社, 2001.
[2]黄岚, 物流配送中的车辆路由算法的研究.吉林大学博士学位论文. 2003.

## 习题 2

2-1 简述贪心法的基本思想和求解步骤。

2-2 会场安排问题。假设要在足够多的会场里安排一批活动，并希望使用尽可能少的会场。设计一个有效的贪心算法进行安排。对于给定的 k 个待安排的活动，编程计算使用最少会场的时间表。

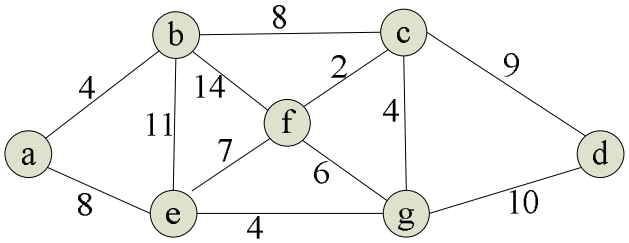
2-3 汽车加油问题。一辆汽车加满油后可行驶 n 公里。旅途中有若干个加油站。设计一个有效算法，指出应在哪些加油站停靠加油，使沿途加油次数最少。

2-4 给定 n 位正整数 a，去掉其中任意 k(k≤n)个数字后，剩下的数字按原次序排列组成一个新的正整数。对于给定的 n 位正整数 a 和正整数 k，设计一个算法找出剩下数字组成的新数最小的删数方案。

2-5 数列极差问题。在黑板上写了 N (N<2000)个正数组成的一个数列，进行如下操作：每一次擦去其中 2 个数（设为 a 和 b），然后在数列中加入一个数 a*b+1，如此下去直至黑板上只剩一个数。不同的操作方式最后得到的数中，最大的数记为 max，最小的数记为 min，则该数列的极差 M 定义为 M=max-min。设计一个算法求出该极差。

2-6 登山机器人问题。登山机器人是一个极富挑战性的高技术密集型科学研究项目，它为研究发展多智能体系统和多机器人之间的合作与对抗提供了生动的研究模型。登山机器人可以携带有限的能量。在登山过程中，登山机器人需要消耗一定能量，连续攀登的路程越长，其攀登速度就越慢。在对 n 种不同类型的机器人做性能测试时，测定出每个机器人连续攀登 1 米,2 米,…,k 米时所用的时间。现在要对这 n 个机器人做综合性能测试，举行机器人接力攀登演习。攀登的总高度为 m 米。规定每个机器人至少攀登 1 米，最多攀登 k 米，而且每个机器人攀登的高度必须是整数，即只能在整米处接力。安排每个机器人攀登适当的高度，使完成接力攀登用的时间最短。编写一个算法求出该攀登方案。

2-7 对下图分别用 Kruskal 算法和 Prim 算法求最小生成树。



2-8 设 $A_n=\{a_1,a_2,\ldots a_n\}$是 n 种不同邮票的集合，面值也记为 $a_i$，且有 $a_1>a_2>\cdots>a_n=1$ 现有总数为 c 的邮资要用 $A_n$的邮票来贴付，每种邮票用数不限，但要求贴的邮票数达到最少。试设计一种基于贪心法的算法，并讨论这样所产生的方案是否为最佳?

2-9 n 个事件需共享同一资源，而这一资源同一时刻只能被一个事件所使用，每个事件有一起始时间和终止时间。如何恰当地选择事件可以使最多的事件能使用资源称为事件选择问题。试采用贪心法设计寻找最优解的方法，并证明。

2-10 设字符 $m_1$，$m_2$，$\cdots m_{10}$的查阅频率依次为：0.05，0.01，0.01，0.10，0.03，0.17，0.02，0.24，0.31，0.06。试构造对应的哈夫曼(Haffman)编码，并画出相应的编码树，同时写出 $m_1,m_2,\ldots m_{10}$的编码。

2-11 设有 n 个程序{1,2,…, n }要存放在长度为 L 的磁带上。程序 i 存放在磁带上的长度是 $l_i$，$1\leqslant i\leqslant n$，这 n 个程序的读取概率分别是 $p_1,p_2,\cdots,p_n$ ，且$\sum_{i=1}^{n} p_i=1$。如果将这 n 个程序按 $i_1,i_2,\cdots,i_n$,的次序存放，则读取程序 $i_r$ 所需的时间 $t_r=c*\sum P_{i_k} l_{i_k}$ 。这 n 个程序的平均读取时间为$\sum_{r=1}^{n} t_r$。磁带最优存储问题要求确定这 n 个程序在磁带上的一个存储次序，使平均读取时间达到最小(假设取 c=1)。试设计一个解此问题的算法，并分析算法的正确性和计算复杂性。