

第 12 章 算法设计基本方法

前面我们已经介绍了几种常见的算法设计方法和程序设计策略（逐步求精、穷举法、递归法、分治法等），本章再介绍几种更深入一些的算法设计方法，包括回溯法、限界剪枝法、动态规划、贪心法等。

§ 12.1 回溯法与限界剪枝法

§ 12.1.1 基本思想

回溯法主要针对一类“求解”问题，这类问题的每个结果（一个解），一般都能分别被抽象为一个 n 元组 (x_1, x_2, \dots, x_n) 。每个解通常需要满足一定的条件（约束条件）。约束条件有时用判定函数 $B(x_1, x_2, \dots, x_n)$ 表达。不一定满足约束条件的解称为可能解。

有时，需要对解的“好坏”做出评价。解的“好坏”衡量标准就称为目标函数。最优解就是那些可使目标函数取极值的可能解。

回溯法对任一解的生成，一般都采用逐步的方式（逐步扩大解）。每进行一步，都试图在当前部分解的基础上扩大该部分解。扩大时，首先检查扩大后是否违反了约束条件，若不违反，则扩大之，然后在此基础上按类似方法进行，直至其成为完整解；若违反，则放弃该步（以及它所生成的部分解），然后按类似方法尝试其他的可能的扩大方式，此时若发现已尝试过所有方式，则结束，表示该解的生成失败。

一般情况下，解可能不唯一，此时也可能需要求出全部解（求全解），也可能按某种目标筛选最优解，等等。

若求全解，则在得到一个解后，不终止，而回退一步，继续按类似的方法尝试新的解（可能多次回退），如此进行，直到所有的可能都尝试过（回退到开始处，且开始处的各种情况也都尝试完）。

回溯法一般需要解的约束条件具有完备性。所谓解的约束条件的完备性，是指若部分解 (x_1, x_2, \dots, x_i) 满足约束条件中仅涉及 x_1, x_2, \dots, x_i 的所有条件，则对所有的 $1 \leq j \leq i$ ， (x_1, x_2, \dots, x_i) 也满足约束条件中仅涉及 x_1, x_2, \dots, x_j 的所有条件，这里， $i=1, 2, \dots, n$ 。换言之，若存在 $1 \leq j \leq n-1$ ，使 (x_1, x_2, \dots, x_j) 不满足约束条件中仅涉及 x_1, x_2, \dots, x_j 的某个条件，则任何以 (x_1, x_2, \dots, x_j) 为前缀的 n 元组 (x_1, x_2, \dots, x_j) 也一定不满足仅涉及 x_1, x_2, \dots, x_j 的某个条件。显然，只有满足约束完备性，才能在只得到部分解的情况下，判别该部分解是

否可被丢弃（否则有可能丢弃了正确的部分解！）。

回溯法与穷举法有某些联系，它们都是基于试探。但是，穷举法要将一个解的各部分全部生成（没有逐步生成）后，才检查是否满足条件，若不满足，则直接放弃该（完整）解，然后从头再来，没有逐步回退。而对回溯法，一个解的各部分是逐步生成的，当发现当前生成的某部分不满足约束条件，就放弃该步所作的工作，退到上一步进行新的尝试，而不是放弃整个解重来。显然，回溯法要比穷举法效率高。

在回溯法中，每次扩大当前部分解时，都面临一个可选的状态集合，新的部分解就通过在该集合中进行选择构造而成。这样的状态集合，结构上是一棵多叉树，每个树结点代表一个可能的部分解，它的儿子是在它的基础上生成的其他部分解。树根为初始状态。这样的状态集合，称为**状态空间树**。因此，逐步生成解的过程，也可以看作是一个搜索过程----在逻辑上存在的状态空间树中搜索。

回溯法的适应范围很广，许多问题都可以用回溯法解决。比较经典的回溯法运用例子有 8-皇后问题、迷宫问题、稳定婚姻问题、子集和问题、图的可着色问题、哈密顿回路、背包问题，等等。下面就介绍其中几个例子。

§ 12.1.2 迷宫问题

(一) 问题陈述

老鼠走迷宫是心理学中的一个经典试验。问题是这样的，设有一只无盖大箱，箱中设置一些隔壁，形成一些曲曲弯弯的通道，做为迷宫。箱子设有一个入口和出口。实验时，在出口处放一些奶酪之类的可以吸引老鼠的食物，然后将一只老鼠放到入口处，这样，老鼠受到美味的吸引，向出口处走。心理学家就观察老鼠如何由入口到达出口。

这个实验可以用来考察老鼠记忆力的强弱。如果它记忆力很好，那么，在迷宫中对以前尝试过的失败路径就不会再去尝试。

我们的任务是，编写一个计算机程序，模拟老鼠走迷宫。该程序假定老鼠具有稳定记忆力（记为 A 级假设智能），能记住以前走过的失败路径，而不会重蹈覆辙。如果计算机模拟的走迷宫路径与老鼠的实际走迷宫路径基本吻合，则说明老鼠具有计算机程序的假设智能。当然，如果不吻合，可以改变计算机程序的假设智能，直到其与老鼠的实际轨迹吻合。我们这里只考虑编写模拟 A 级假设智能的程序。

(二) 基本实现方法

根据我们对老鼠的智慧的假设（A 级假设智能），老鼠走迷宫的方式为：

试探----回溯

尝试----纠错

即每到达一个位置，就试探当前方向是否可行，若可行，则到达下个位置，继续按类似的方法试探；若不可行，有两种情况：a)当存在下个未尝试的方向时，转下一个方向并按类似方式试探；b)当不存在下个未尝试的方向时，回退一步，转下个方向并按类似方

式试探, 若此位置已无下个未尝试的方向, 则按类似的方式回退, 直到找到未尝试的方向, 或退到开始处, 而且开始处的各种方向都已尝试过, 此时, 表示当前解的生成失败。如此一直进行, 直到找到出口, 或者所有可能都尝试。该 (求单解) 过程描述如下:

```
设定一个记录当前走过的路径的栈 path;
令 pos 表示当前老鼠位置, 初始设置为入口位置;
令 dire 表示当前试探方向, 初始置为某一选定方向;
pos 与 dire 进 path;

while (未到达出口 || 存在可供选择的下一位置)
{
    if (存在新方向)
    {
        求出 pos 上的 dire 方向上的邻接点 pos1;
        if (pos1 为通路 && pos1 未被试探过)
        { //走到 pos1
            为 pos1 置试探标志;
            pos1 与 dire 进 path;
            置 pos 为 pos1, 置 dire 初始方向; //下次仍从初始方向起试探
            if (pos 为出口) 结束;
        }
        else 置 dire 为下一方向; //下次循环中, 试探下个方向
    } // if (存在新方向)
    else //当前位置的各方向均已试探, 回退一步
    {
        栈顶元素的 dire 的值赋 dire; //以在下次循环中, 从上个位置的下个方向起试探
        从 path 弹出一个元素;
        新栈顶元素的值送 pos
    }
} //while
```

在求全解的情况下, 当找到一个出口时不结束, 而回退一步, 按类似的方法继续找其他解, 直到回退到入口处, 且入口处的各个方向都已尝试过。

如果要将该过程细化为计算机程序, 则需要设计相应的数据结构了。

(三) 数据结构设计

为了得到一个具体的计算机程序, 需考虑问题中涉及的实体 (如迷宫、路径、位置、

方向等) 的表示方法。

(1) 迷宫

迷宫实质上是由多条路径纵横交错而成。每条路径相当于一条平面曲线，而平面曲线可用点阵表示。具体表示法为：每个点可有两种状态，分别表示“通”与“不通”。对每个“通”点，若它的相邻周围（上下左右即各对角，共 8 个方向）有“通”点，则表示从它到每个相邻“通”点都分别有通路。若用直线连接各相邻“通”点，则这些首尾相接的直线段就表示路径曲线。显然，点阵愈密，精度愈高。

当然，迷宫的表示也可从另一角度考虑：由于我们只关心路径上的交汇点及转弯情况（拓扑关系），故迷宫中路径可用若干站点表示，每个站点有“通”与“不通”两种状态，若某站点 A 的某个邻接站点 B 为“通”，则表示可从 A 到达 B，即 A 到 B 有通路。任意两点间的一条路径，是它们之间的“通”点的序列。

这两种理解法都归纳到同一种表示法：点阵。由于平面点阵每个点最多有 8 个邻接点，故从每个点出发，最多有 8 个方向可供选择。若实际需要多于 8 个方向，则显然可用增加点的密度的方法解决。所以，使用点阵表示迷宫，不失一般性。图 12-0 给出一个迷宫点阵的例子。

点阵可简单地用二维数组表示。数组的每个元素代表一个点，用 0 和 1 分别表示“通”与“不通”。为了编程方便，二维数组的四周都置 1，表示迷宫的墙壁。入口设在左上角，出口设在右下角。

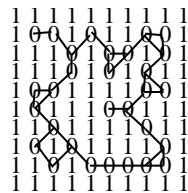


图 12-1 用点阵表示的迷宫
0 表示通，1 表示不通，四周的 1 表示墙壁，左上角为入口，右小角为出口

(2) 路径纪录器

从算法输出结果的要求看，至少需要一个路径记录器保存所找到的路径。由于一条路径可以用构成它的点的序列表示，所以，路径记录器可以是以点为元素的线性表。

因为我们使用的是回溯法，所以要求记下各回退点。这些回退点恰好是当前已找到的部分路径（部分解）的倒序，因此，也可用路径记录器实现回溯。由于记录路径的次序与回退的次序正好相反，所以路径记录器是做为栈使用。

在点阵迷宫中，每个点可用行号和列号表示。另外，为了处理方便，对每个点，除了记下它的行号和列号外，也可以设置一个表示方向的值（不是必要的）。若 A 是路径上某点，A' 是该路径中 A 的上个邻接点，则 A 的方向值表示在点阵中，A 相对于 A' 的方位，即表示 A 来自于 A' 的什么方向。至于方向的表示，用整数 0~7 表示。

因此，路径记录器中每个点用一个三元组表示：

（行号，列号，方向）

如果要求全解并保存它们，则路径记录器需要多个。

(3) 方向和方向增量

在点阵中，每个非边界点在东、东南、南、西南、西、西北、北、东北各有一个邻

点，也就是说最多有 8 个不同方向，故可分别用整数 0~7 表示这 8 种不同的方向。

为了能方便地由任一给定点的坐标（行号，列号），求得它的任一方向上的邻点的坐标，设立一个 8 行 2 列的增量表（二维数组），表的行 i 上的两个元素分别表示到达方向 d 上的邻点所需的 x 增量与 y 增量。显然，增量表中的值为整数常量。增量计算说明与增量表具体内容如图 12-0。例如，该图中左图中，从中心点（其坐标为 (i,j) ）到达点 $(i+1,j-1)$ ，需要沿方向 3 前进一步，行坐标增 1，列坐标减 1，因此，在增量表（右图）中的方向号为 3 的行中，其他两列内的值（行增量和列增量）分别为 1 和 -1。

显然，有了增量表（设增量表为二维数组 $incr$ ）后，可方便地由某点 (i, j) ，计算出

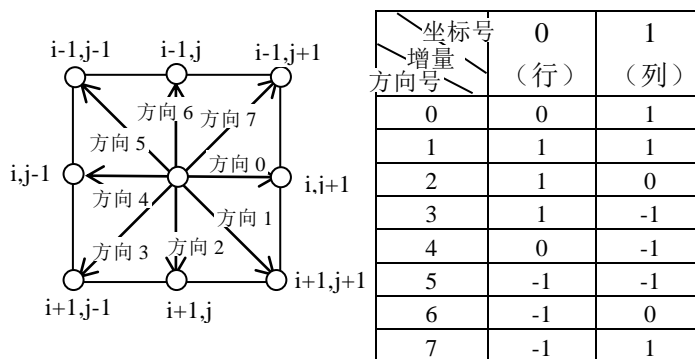


图 12-2 增量计算示意（左图）与增量表（右图）
设左图中的中心点的坐标为 (i,j)

它的任一方向 d 上的点的坐标：

$$(i+incr[d][0], j+incr[d][1])$$

(4) 试探标志

在一次路径的寻找过程中，以前试探过的站点不应再试探，以避免重蹈覆辙。这样，就需要在算法进行中，对已试探过的站点进行标记，使得算法可以识别它们。

试探标志的设置可有多种方法，一种是设置与迷宫同规模的数据结构（如二维数组），其每个元素分别做为一个站点的试探标志，取两种值，分别表示“未试探”和“已试探”。另一方法是不单独设置试探标志，而是利用迷宫数组。每当试探过一个点，就将迷宫中对应点置为试探标记。这种方法的优点显然是节省存储空间，但当迷宫还另有别用时，需要在算法结束后恢复之。

(四) 算法实现

根据我们对数据结构的设计，先讨论一下前面给出的迷宫程序伪码中的一些主要步骤的实现。

首先，方向用 0~7 的整数表示，所以，伪码中的 $dire$ 为 0~7 整数，超过此范围表示不存在的方向。

由于老鼠走过的路径用三元组（行号，列号，相对于上个点的方向）序列表示，所

以，伪码中的当前站点 `pos` 及 `pos1` 为三元组。

显然，若在每次 `while` 循环时，发现栈 `path` 已空，则表示已回退到了入口，此时，若各方向均已试探过，则表示解的所有可能都试探过了，算法结束，此时全部解已求出（包括无解的情况），因此，`while` 循环的条件应为：

```
while (path 不空 || dire<8)
```

而由当前位置 `pos=(i, j, dire)` 求下一位置 `pos1` (即 `(i,j)` 的 `dire` 方向的站点) 的坐标可用增量数组 `incr`：

```
pos1 = ( i+incr[dire][0], j+incr[dire][1] )
```

另外，为了节省空间，我们借用迷宫矩阵本身做为试探标志。迷宫矩阵中，用 0 和 1 分别表示“通”与“不通”。我们在程序中规定，对已试探过（走到过）的点，在迷宫矩阵中置其为 -1（表示“已试探”）。

至此，我们可以写出完整的子程序了。

```
typedef unsigned int TMazeDimension;
struct TMazePos // 定义路径栈的元素类型
{
    TMazeDimension row, col;
    char dire;
};
const int CNST_MaxNumCols=10;
unsigned int MazePath(char maze[][CNST_MaxNumCols], TMazeDimension n,
TMazeDimension m, TMazePos *path)
{//求迷宫路径（单解）
    // maze: 输入量，n 行 m 列二维数组，代表迷宫。四周为全 1,表示墙壁
    // path: 输出量，一维数组，存放所找到的路径，在外部分配空间，最大为 n*m 个元素
    //返回所找到的路径的长度（点数）

    TMazeDimension i, j, g, h, top;
    char dire;
    char incr[8][2] = //方向增量数组
    {
        {0, 1},
        {1, 1},
        {1, 0},
        {1, -1},
        {0, -1},
        {-1, -1},
        {-1, 0},
        {-1, 1}
```

```
};

top = 1;
i = 1; j = 1; dire = 0; //置前点和当前方向
path[top].row = i; // 当前点和当前方向进栈
path[top].col = j;
path[top].dire = dire;
maze[i][j] = -1;

while (top > 0 || dire < 8)
{
    if (dire < 8)
    {
        g = i + incr[dire][0];
        h = j + incr[dire][1];
        if (maze[g][h] != 1 && maze[g][h] != -1) //试探过的点置为-1, 表示试探标志
        {
            maze[g][h] = -1; //置试探标志
            top++;
            path[top].row = g;
            path[top].col = h;
            path[top].dire = dire;

            i = g; j = h; dire = 0;
            if (i == n - 2 && j == m - 2) return top; //已达到出口, 找到了一条路径, 结束
        }
        else dire++; //换个方向试探
    } //if (dire < 8)
    else //dire >= 8, 回溯
    {
        dire = path[top].dire + 1;
        top--;
        if (top > 0)
        {
            i = path[top].row;
            j = path[top].col;
        }
    } //if (dire < 8)
}
```

```
    } //while  
  
    return 0;  
  
} //Mazepath()
```

(五) 程序改进

上面给出的程序只求一个解。若要求全解，程序还要在此基础上适当调整。主要有两点要调整，一是结束条件，在找到一个解后不结束，而是象往常一样回退一步，继续找其他路径；另一点是试探标志的处理，因为允许一些点出现在多条路径上（即路径交汇），所以，以前标为“已试探”的点，应在求其他解的过程中允许重新试探。

为了逼真地看到老鼠走迷宫的过程，算法中可增加轨迹输出功能。为了看清楚走迷宫的轨迹，需对走迷宫的过程延时。这可用适当的延时函数，或简单地使用空循环。

我们在算法中将迷宫做为输入参数。具体使用时，最好有个迷宫生成函数。迷宫的生成可有两种方式，一是自动生成（随机生成），另一是操作员在屏幕象画画儿一样，用鼠标器画路径。

上面几点改进的实现，留作练习。

§ 12.1.3 稳定婚姻问题*

从问题模型讲，稳定婚姻问题是个配位问题；从实现技术讲，是个回溯程序设计问题。本节以稳定婚姻这个具体例子，说明配位问题的回溯程序设计方法。

(一) 问题描述

设有若干个男子与若干个女子，分别用集合 M 和 W 表示，每个男子和女子都各有自己的选择志愿。现要为集合 M 中每个男子分别根据其志愿在集合 W 中选一个不同的女子作为恋人，同时要使得所作的选择是“可行”的（稳定的）。我们称这种选择为配位。

男子的志愿是这样的：对 M 中每个男子，若他第 $i(i=1,2,\dots)$ 喜欢 W 中的 w_j ，则称该男子的第 i 志愿为女子 w_j 。女子的志愿与男子类似。

在为男子选择恋人时，同时要兼顾女子的志愿，即考虑为男子 m_i 选择女子 w_j 时，要考查原来选择 w_j 的那位男子 m_k ，若 w_j 对 m_i 的喜欢程度胜于对 m_k 的喜欢程度，则可将 w_j 许配给 m_i ，否则不可。

可行的搭配方案可能有多种，一般要求出最佳搭配。最佳搭配也可分几种，一是平均最佳搭配，二是男子或女子最佳搭配，三是相对某几个人的最佳搭配。

我们将按照上述策略，分别为 M 中每个男子，在 W 中选择一个不同的女子。该问题就称为稳定婚姻问题。

一般来讲, 该类问题可抽象为: 设有两个集合 M 和 W , M 的基数小于或等于 W 的基数, 按照某种条件建立一个 M 到 W 的内射 (即为 M 中每个元素, 在 W 中找恰好一个与其搭配, 而 W 中每个元素, 最多能与 M 中一个搭配)。考生选择学校, 学生选课, 八皇后问题及其它一些配位问题均属于此类问题。

(二) 实现思想

对此类问题的解决, 一般没有解析方法, 我们仍然采用下列回溯策略:

试探——回溯

尝试——纠错

即对 M 中每个元素, 依次分别按条件在 W 中找一个不同的搭配, 当某次为 M 中某元素 m 进行配位时发现无法进行, 就撤消对 M 中的上个元素的搭配, 为它选择下一种搭配, 然后按同样方式进行。若对 m 的上个元素不能做出新的选择, 则考虑 m 的上上个元素, 如此一直进行, 直到退到可进行新的选择的某个元素, 按同样方式往下进行, 或退到 M 的第 1 个元素, 但若此元素不能有新选择, 则结束。


如果是求一个解, 则当对 M 中每个元素分别做了一个合适的选择后, 即可停止。若是求全解, 则应对 m 中每个元素的每个可能的选择都分别进行试探, 将其中所有可行者记录下来。

(1) 递归算法

对上述基本思想的描述, 最简洁的方式是使用递归。下面给出的用伪码描述的递归过程 $\text{Try}(m)$ 用于求全解。 $\text{Try}(m)$ 中的参数 m 是 M 中的某个元素。 $\text{Try}(m)$ 表示在前面的配位的基础上 (即假定 m 前面的元素都进行了配位) 对 m 和 m 之后的元素分别配位。显然, 求全解时, 用第一个元素调用 $\text{Try}()$ 。下面是递归求全解的伪码:

```
Try(m)
{
    for (m 的每个志愿 w)
        if (m 与 w 的搭配是可行的)
        {
            将 w 许配给 m;
            if (m 不是 M 中最后一个元素)
                Try (m 的下个元素);
            else 记录该组解;
            撤消 m 与 w 的搭配; //以求下一个解
        }
}
```

这个算法描述简单得可能有些出乎意料，其关键所在是将递归调用放到了 for 语句内。

 另一个需强调的问题是，算法中的“撤消 m 与 w 的搭配”一句的重要性，若无此句，是无法求出全解的。

(2) 非递归算法

非递归方法稍稍复杂些，但比较容易理解。下面是非递归求全解的伪码。

Arrange()

```
{//假定 M 与 W 集合中元素已编号(从 1 起的自然数)
  //nm 与 nw 分别为集合 M 与 W 的基数(元素个数)
  m=1; r=1; //先考虑 M 中第 i 个元素的第 1 志愿
  while (nm>0)  //当尚有未尝试的选择时
  {
    求出 m 的第 r 个志愿 w;
    if (m 与 w 的搭配是可行的)
    {
      将 w 许配给 m; 记录此许配;

      if (m 不是最后一人)
      { //准备为下个人配位
        m=m+1; //令 m 代表下一个元素
        r=1;  //下次仍从第 1 志愿起考查
      }
      else
      { //已找到一个解
        输出当前找到的解;
        准备从 m 的下个志愿起重新试探，以求另一个解
      } // if (m 不是最后一人)
    }
    else  //m 与 w 的搭配不可行
    if (r<nw && m<=nm)
      r = r+1; // m 尚有志愿未考虑时，试探它的下个志愿
    else  //m 在当前状态下得不到任何搭配，回溯
    {
      找 m 前面最近的且不是以最后志愿配位的男子 pm;
      并同时撤销 pm~m 之间的各个男子的配位;
```

```

    置 m 为 pm;
    置 r 为 pm 的下个志愿;
  } // if (m 与 w 的搭配是可行的)
} // while
}

```

(三) 数据结构设计

为了用计算机语言描述上面给出的基本思想，必须设计相应的数据结构。首先，为了处理方便，我们用编号（从 1 起自然数）表示集合 M 与 W 中的元素。

(1) 志愿表

分男子志愿与女子志愿，用来存放男子/女子的恋人。

1) 男子志愿表 wmr ：二维数组， $wmr[m][r]=w$ 表示男子 m 的第 r 志愿是女子 w （即 m 对 w 的喜欢程度排行第 r ）。

2) 女子志愿表 mwr ：与男子志愿表类似， $mwr[w][r]=m$ 表示女子 w 的第 r 志愿是男子 m 。

以上两个表是本问题的原始输入数据。

(2) 配偶表

分男子配偶表与女子配偶表，表示搭配结果（算法的输出结果）。

男子配偶表 $x[]$ ：一维数组， $x[m]=w$ 表示已为男子 m 选择了女子 w 。

女子婚配表 $y[]$ ：一维数组， $y[w]=m$ 表示已为女子 w 选择了男子 m 。

这两个表并不是独立的，可由其中任意一个得到另一个，设立两个表是为了处理方便。

(3) 爱度表

爱度表表示男女之间的爱情程度的大小。处理时，爱情程度应数字化。我们这里用志愿序号表示爱情度，志愿值越小，表示爱情度越大。

爱度表分男子爱度表与女子爱度表。男子爱度表设计成二维数组 $rmw[][]$ ，它的含意是： $rmw[m][w]=r$ 表示男子 m 对女子 w 的爱度为 r ，这个 r 就是 m 对 w 的志愿序号，即 m 的第 r 志愿是 w 。女子爱度表与男子爱度表类似，用二维数组 $rwm[][]$ 表示。 $rwm[w][m]=r$ 表示女子 w 对男子 m 的爱情程度为 r ，该表也可由女子志愿表 mwr 生成。

显然，爱度表与志愿表存在下列关系：

若 $wmr[m][r]=w$ ，则 $rmw[m][w]=r$ ；

若 $mwr[w][r]=m$ ，则 $rwm[w][m]=r$ ；

因此，爱度表 rmw 与 rwm 可分别由志愿表 wmr 与 mwr 生成，具体方法是：

```
for (i=1; i<=nm; i++)
```

```

for (j=1; j<=nw; j++)
    rmw[i][wmr[i][j]] = j;

for (i=1; i<=nw; i++)
    for (j=1; j<=nm; j++)
        rwm[i][mwr[i][j]] = j;

```

爱度表是中间数据结构，设置它们主要是为了方便判断配位的稳定性。

(4) 婚配表


婚配表用来表示某人是否已被配位。我们用一维数组 `single[]` 表示女子婚配表，`single[w]` 为 0 时表示女子 `w` 已被配位，否则表示尚未配位。


由于该算法是从男子出发（以男子为主动者）进行选配的，故男子的婚况可不考虑。

(四) 算法实现

有了上面的设计，我们就可以给出具体的计算机程序了。在程序中，使用了一个函数 `Stable(m,w)`，当 `m` 与 `w` 的搭配可行时，它返回非 0，否则返回 0。它的实现涉及到选择规则（策略），我们将在下节中专门介绍。

由于该问题的程序需分为几个子程序，而它们共享较多的数据（结构），对普通 C，一般采用参数传递或外部变量共享的方法。但是，前者效率低，并导致接口复杂，后者使用较多的外部变量，导致程序结构不紧凑。鉴于这些原因，我们在这里将稳定婚姻问题的解决设计为一个类，将各子程序的共享变量封装在类中。

 与 Pascal 及 Ada 之类的语言相比，C 语言有个严重的缺陷：子程序不能嵌套。所谓子程序（包括函数和过程）嵌套，是指在子程序中定义（注意，不是调用）其他子程序，这样，子程序定义也构成多层次结构。处在内（下）层的子程序可以共享它们的外（上）层的子程序中定义的变量，好像是在使用全局变量一样。这种功能对简化子程序接口、限制全局变量的无序扩张、提高执行效率等起到很好的作用。

 在 C++ 中，这种缺陷得到缓解。这并不是说 C++ 增加了子程序嵌套功能，而是可用对象封装的方式解决。当多个子程序需要共享数据时，将它们连同需共享的数据封装在一起做为一个类。如果要实现多层嵌套，则还可使用类的继承。利用面向对象语言的这种功能，可实现全局变量的“局部化”。这种与“对象”并不紧密联系的用法，也是面向对象语言的一个应用亮点。

```

const int NM=8;
const int NW=8;

class TMarriageArrange
{

```

```
public:
```

```
    int nw,nm;
```

```
    int wmr[NM+1][NW+1]; //志愿表。在实际中，可考虑使用动态结构
```

```
    int mwr[NW+1][NM+1];
```

```
    int x[NM+1], y[NW+1]; //配位结果表
```

```
private:
```

```
    int rmw[NM+1][NW+1], rwm[NW+1][NM+1]; //爱度表
```

```
    char single[NW+1];
```

```
private:
```

```
    void Try(int m);
```

```
    char Stable(int m, int w);
```

```
public:
```

```
    TMarriageArrange();
```

```
    int Arrange1();
```

```
    int Arrange2();
```

```
    void OutArrange();
```

```
};
```

```
TMarriageArrange::TMarriageArrange()
```

```
{//初始化数据与输入量的生成。这里的设计是为了调试程序方便。
```

```
    //实际中，原始数据的读入可采用其他方法。
```

```
    nw=nm=8;
```

```
    wmr[1][1]=7;wmr[1][2]=2;wmr[1][3]=6;wmr[1][4]=5;wmr[1][5]=1;wmr[1][6]=3;wmr[1][7]=8;wmr[1][8]=4;
```

```
    wmr[2][1]=4;wmr[2][2]=3;wmr[2][3]=2;wmr[2][4]=6;wmr[2][5]=8;wmr[2][6]=1;wmr[2][7]=7;wmr[2][8]=5;
```

```
    wmr[3][1]=3;wmr[3][2]=2;wmr[3][3]=4;wmr[3][4]=1;wmr[3][5]=8;wmr[3][6]=5;wmr[3][7]=7;wmr[3][8]=6;
```

```
    wmr[4][1]=3;wmr[4][2]=8;wmr[4][3]=4;wmr[4][4]=2;wmr[4][5]=5;wmr[4][6]=6;wmr[4][7]=7;wmr[4][8]=1;
```

```
    wmr[5][1]=8;wmr[5][2]=3;wmr[5][3]=4;wmr[5][4]=5;wmr[5][5]=6;wmr[5][6]=1;wmr[5][7]=7;wmr[5][8]=2;
```

```
    wmr[6][1]=8;wmr[6][2]=7;wmr[6][3]=5;wmr[6][4]=2;wmr[6][5]=4;wmr[6][6]=3;wmr[6][7]=1;wmr[6][8]=6;
```

```
    wmr[7][1]=2;wmr[7][2]=4;wmr[7][3]=6;wmr[7][4]=3;wmr[7][5]=1;wmr[7][6]=7;wmr[7][7]=5;wmr[7][8]=8;
```

```
    wmr[8][1]=6;wmr[8][2]=1;wmr[8][3]=4;wmr[8][4]=2;wmr[8][5]=7;wmr[8][6]=5;wmr[8][7]=3;wmr[8][8]=8;
```

```
    mwr[1][1]=4;mwr[1][2]=6;mwr[1][3]=2;mwr[1][4]=5;mwr[1][5]=8;mwr[1][6]=1;mwr[1][7]=3;mwr[1][8]=7;
```

```

mwr[2][1]=8;mwr[2][2]=5;mwr[2][3]=3;mwr[2][4]=1;mwr[2][5]=6;mwr[2][6]=7;mwr[2][7]=4;mwr[2][8]=2;
mwr[3][1]=6;mwr[3][2]=8;mwr[3][3]=1;mwr[3][4]=2;mwr[3][5]=3;mwr[3][6]=4;mwr[3][7]=7;mwr[3][8]=5;
mwr[4][1]=3;mwr[4][2]=2;mwr[4][3]=4;mwr[4][4]=7;mwr[4][5]=6;mwr[4][6]=8;mwr[4][7]=5;mwr[4][8]=1;
mwr[5][1]=6;mwr[5][2]=3;mwr[5][3]=1;mwr[5][4]=4;mwr[5][5]=5;mwr[5][6]=7;mwr[5][7]=2;mwr[5][8]=8;
mwr[6][1]=2;mwr[6][2]=1;mwr[6][3]=3;mwr[6][4]=8;mwr[6][5]=7;mwr[6][6]=4;mwr[6][7]=6;mwr[6][8]=5;
mwr[7][1]=3;mwr[7][2]=5;mwr[7][3]=7;mwr[7][4]=2;mwr[7][5]=4;mwr[7][6]=1;mwr[7][7]=8;mwr[7][8]=6;
mwr[8][1]=7;mwr[8][2]=2;mwr[8][3]=8;mwr[8][4]=4;mwr[8][5]=5;mwr[8][6]=6;mwr[8][7]=3;mwr[8][8]=1;

```

```

int i, j;
for ( i=1; i<=nw; i++) single[i]=1;

for (i=1; i<=nm; i++) //生成男子爱度表
    for (j=1; j<=nw; j++)
        rmw[i][wmr[i][j]] = j;

for (i=1; i<=nw; i++) //生成女子爱度表
    for (j=1; j<=nm; j++)
        rwm[i][mwr[i][j]] = j;

};

```

```

void TMarriageArrange::OutArrange()
{ //输出一个解。这里的输出只设计为显示
    cout<<"\n";
    for (int i=1; i<=nm; i++)
        cout<<' '<<x[i];
}

```

```

char TMarriageArrange::Stable(int m, int w)
{ //判断 m 与 w 的搭配是否稳定
    int pm, pw, i, r, r2;
    char s;

    s=1;
    i=1;
    r = rmw[m][w];
    while (i<r && s) //检查 m 更喜欢(与 w 相比) 的每个女子
    {

```

```

    pw=wmr[m][i]; //求出 m 的第 i 志愿 pw, m 对 pw 的喜欢度胜过对 w 的喜欢度 (因为 i<r)
    i++;
    if (!single[pw]) //pw 喜欢 m 胜过喜欢她的原配时, 置 s 为 0, 表示 m 与 w 不稳定
        s = (rwm[pw][m] > rwm[pw][y[pw]]);
} //while

i=1;
r2=rwm[w][m];
while (i<r2 && s) //检查 w 更喜欢(与 m 相比) 的每个男子
{
    pm=mwr[w][i]; //求出 w 的第 i 志愿 pm, w 对 pm 的喜欢度胜过对 m 的喜欢度 (因为 i<r2)
    i++;
    if (pm<m) //若 pm 已配位
        s = (rmw[pm][w] > rmw[pm][x[pm]]);
    //pm 喜欢 w 胜过喜欢她的原配时, 置 s 为 0, 表示 m 与 w 不稳定
} //while

return s;
}

void TMarriageArrange::Try (int m)
{ //递归求全解
    int r,w;

    for ( r=1; r<=nw; r++)
    {
        w=wmr[m][r];
        if (single[w])
            if (Stable(m,w))
            {
                x[m]=w; y[w]=m;
                single[w]=0;
                if (m<nm)
                    Try(m+1);
                else OutArrange();
                single[w]=1;
            }
    }
} //for

```

```
//Try

int TMarriageArrange::Arrange1()
{ //递归求全解
    Try(1);
}

int TMarriageArrange::Arrange2()
{ //稳定婚姻问题的非递归程序（求全解）。返回解的个数
    int m, w, r;
    int cnt;

    cnt=0;
    m=1; //先考虑第一个男子
    r=1; //先考虑第一第一志愿
    while(m>0)
    {
        w = wmr[m][r]; //求出 m 的第 r 志愿
        if (single[w] && Stable(m,w) )
        { //将 w 许配给 m
            x[m]=w; y[w]=m; single[w]=0;

            if (m<nm)
            { //考虑下个男子，仍从第一志愿开始
                m++; r=1;
            }
            else
            { //已找到一个解
                OutArrange(); cnt++; //输出解

                single[x[m]]=1; //撤销对 m 的配位，从他的下个志愿起寻找另外的选择（解）
                r++;
            }
        }
        else
        {
            if (r<nw && m<=nm)
                r++; //考虑 m 的下个志愿
```



```

else
{
    //m 的所有志愿都不可行，回溯
    m--;
    while (m>0 && rmw[m][x[m]]==nw)
    {
        //按原路回退，并撤销相应的配位，直到找到一个可有下个志愿的男子
        single[x[m]]=1;
        m--;
    } //while
    if (m>0)
    {
        single[x[m]]=1; //撤销对 m 的配位
        r = rmw[m][x[m]]+1; //准备下次从 m 的下个志愿起试探
    }
}

} //while(m>0)
return cnt;
}

```

(五) 稳定性判别

在前面的算法中，需要判别某个男子与某个女子的搭配是否可行（稳定），我们用函数 $\text{Stable}(m,w)$ 表示这种判别。这里就讨论 $\text{Stable}(m,w)$ 的实现思想。

m 与 w 的搭配是“可行”的，应同时满足下列两条：

a) w 应是单身（尚未许配给人），这可由婚恋表 single 获悉。

b) m 与 w 的搭配应是稳定的。

要搞清什么是稳定的，先应明白什么情况是不稳定。设我们考虑将 w 许配给 m 是否稳定， w 是 m 的第 r 志愿，则下列两种情况均为不稳定。

a) 存在一个女子 pw ， m 对她比对 w 更中意（爱情度更大），同时，她(pw)对 m 比对它的原配更中意；

b) 存在一个男子 pm ， w 对他比对 m 更中意（爱情度更大），同时，他(pm)对 w 比对它的原配更中意。

如果有上列两种情况之一，则表示不能将 w 许配给 m ，即 m 与 w 的搭配是不稳定的，否则认为是稳定的。所以，稳定性的识别只需检查是否存在上列两种情况。

§ 12.1.4 N 皇后问题

(一) 基本方法

N 皇后问题最早由著名数学家高斯(Gauss, 1777-1855, 德国数学家)提出（当时是以八皇后为例），但他并未完全解决，这是不足为奇的，因为该问题不宜解析求解。事实上，使用回溯法可方便地解决，但这依赖于高速计算机。

所谓 n 皇后问题，是指如何从 $n \times n$ 的方格阵列（原为 8×8 的国际象棋棋盘）中放置 n 个皇后，使其中的任意两个都不冲突。这里，任意两个皇后不冲突，是指她们都不同行、不同列，不同对角线（包括各正反向斜线）。这个问题一般要求给出所有的不同放置方法。

N 皇后的解决，仍然用回溯法。由于一列只容许放置一个皇后，所以，可以以列为单位考虑，即依次为第 1、第 2、...、第 n 列放置皇后。为每一列放置时，都是从第 1 行开始考虑，若放置在第 1 行与目前其他列的皇后冲突，则用类似的方法考虑下一行。当某一列的各行都不能放置时，则说明前面几列的放置不合理，此时应回溯，找到有新选择（即未放置到最后一列）的最近一列，将该列的皇后移到下一行重新检查。如果最后一列安置好了，则说明已求出一个解；如果回溯到了第一列，但该列已没有其他选择（各行都试探过了），则表示没有其他解了，应该结束。

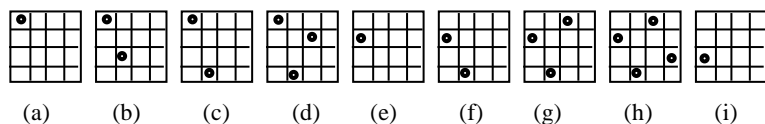


图 12-3 4-皇后回溯求解

图 12-0 以 4 皇后为例说明这种回溯解法。图中各步说明如下：

- (a) 安排第 1 列：先放到第 1 行；
- (b) 安排第 2 列：第 1 行同行冲突，第 2 行斜线冲突，所以安排在第 3 行；
- (c) 安排第 3 列：所有行都冲突，**回溯**到第 2 列(重新**安排第 2 列**)，安排在（第 2 列的）第 4 行；
- (d) 安排第 3 列：第 1 行同行冲突，安排在第 2 行；
- (e) 安排第 4 列：所有行都冲突，**回溯**到第 3 列(重新**安排第 3 列**)，但该列上第 3 和第 4 行也冲突，故继续回溯，**回溯**到第 2 列，该列上已无其他选择，故继续回溯，**回溯**到第 1 列(重新**安排第 1 列**)，该列的第 2 行不冲突，安排在第 2 行；
- (f) 安排第 2 列：第 1 行斜线冲突，第 2 行同行冲突，第 3 行反斜线冲突，故安排在第 4 行；
- (g) 安排第 3 列：安排在第 1 行；
- (h) 安排第 4 列：第 1 行和第 2 行都同行冲突，故安排在第 3 行；至此，得到一个解；
- (i) 得到一个解后，重新安排第 4 列（最后一列），以求下一解。但第 4 列上的下个位置（第 4 行）同行冲突，故回溯，**回溯**路过第 3 列、第 2 列，但它们都没有合适位置，一直**回溯**到第 1 列，将第 1 列上皇后安排在下个位置（第 3 行）；

此后，与前面过程类似，安排其他列。当第 1 列的各行都试探过后，不在有其他解了，结束。

4 皇后问题共有 2 个不同的解（含对称解）。易知，1 皇后有 1 个解，2 皇后和 3 皇后无解。8 皇后有 98 个解。解的个数将随 n 快速增长。

(二) 算法与数据结构设计考虑

该问题的解法与前面介绍的稳定婚姻问题和迷宫问题类似（应该还比这两个问题简单些），因此算法设计也十分类似。区别比较大的是数据结构和约束条件的判别。

首先我们看看如何表示一个解。由于每列要放置一个皇后，所以，可以设置一个一维数组 x （称其为解数组），令其每个元素对应一列，其值等于该列上皇后所在的行号，即

$$x[i] == k$$

表示第 i 列上的第 k 行放置皇后。

显然，要保存多个解时，可以设立多个这样的一维数组（或一个二维数组）。

下面考虑棋盘的表示。棋盘在这里是原始输入数据。但该问题与一般不同，只要知道棋盘的规模（即 $n \times n$ ），棋盘就完全确定了。此外，皇后的放置状态已用解数组 x 表示，因此，如果没有其他用途，棋盘的 $n \times n$ 的阵列表示是不需要显式给出的。

下面考虑冲突判别（约束条件）。如果已设置 $n \times n$ 阵列棋盘的表示，则可以对已放置了皇后的格子打标记，然后根据标记判别是否冲突。若不设置 $n \times n$ 阵列，能否判别冲突？回答是肯定的，即如果有解数组 x ，则完全可以判别冲突。因为解数组已包含了全部的放置状态信息。

为此，我们先分析一下。设目前要在 i 行 j 列放置皇后，考虑如何判断其是否与其他皇后冲突。冲突判别有下列几种：

“是否为同列”：由于我们以列为单位放置皇后，所以不需要判断“是否为同列”；

“是否为同行”：这可以通过扫描解数组 x 得到，即查找 $x[1] \sim x[j-1]$ 中，是否有值为 i 的元素；

“是否为同斜线”：注意， $n \times n$ 的阵列中有两种斜线：正斜线(/)和反斜线(\)，各有 $2n-1$ 条。同一条正斜线上，各元素的行号和列号之和相等；同一条反斜线上，各元素的行号和列号之差相等。因此，若有两个皇后的放置位置分别为 (i, j) 和 (g, h) ，则如果

$$i-j == g-h \text{ 或 } i+j == g+h$$

则表示它们在同一斜线上，有冲突。这两个式子等价于

$$i-g == j-h \text{ 或 } i-g == h-j$$

其又等价于

$$|i-g| == |j-h|$$

因此，判断 (i, j) 是否与当前放置的皇后冲突，可以通过扫描解数组 x 来完成，即检查 x 的每个元素 $x[k]$ （其表示位置 $(x[k], k)$ 上有皇后， $k < j$ ），若下式成立

$$|i-x[k]| == |j-k|$$

则表明有冲突。

如果为了提高判别速度，可以专门设置三个一维数组 a 、 b 和 c ，分别做为行、正斜线、反斜线上的放置皇后的标志，这样，直接判别标志就可以确定是否有冲突。行标志数组 a 很简单， $a[i]$ 就是 i 行的放置标志。由于每条正斜线上的元素的行号和列号的和分别为一个不同的固定值（从左上角到右下角的值分别为 2 、 3 、 \dots 、 $2n$ ），因此， b 可定义为（第一个元素不使用）

$\text{int } b[2*n+1]$

这样，若判断 (i,j) 是否与正斜线上元素冲突（正斜线上是否有元素），只要检查元素 $b[i+j]$ 的值即可。

反斜线的情况也类似。每条反斜线上的行号与列号之差，都分别等于一个固定值，范围是（从右上角到左下角）： $-(n-1)$ 、 $-(n-2)$ 、 \dots 、 -1 、 0 、 1 、 2 、 \dots 、 $(n-1)$ 。我们称这些值为差值。因此， c 亦定义为（第一个元素不使用）

$\text{int } c[2*n+1]$

但使用时，需下标换算，即 $c[k]$ 代表差值为 $n-k$ 的斜线。换言之，若位置 (i, j) 放置了皇后，则置 $c[n-(i-j)]$ 为放置标志。

分析到这里，完整的 n 皇后问题的程序就不难写出了。具体编程，留作练习。

§ 12.1.5 限界剪枝法简介*

限界剪枝法又称**分枝限界法**(Branch-and-Bound)，类似于回溯法，也可以看作是回溯法的改进。在回溯法中，是在整个状态空间树中搜索解，并用约束条件判断搜索进程，一旦发现不可能产生问题的解的部分解，就中止对相应子树的搜索，从而避免不必要的工作。

限界剪枝主要在两个方面与回溯法不同：解的评判与解搜索方式。

解的评判：回溯法一般只使用约束函数评判得到的部分解，若满足约束函数，则继续扩大该解，否则丢弃之，重新搜索。在限界剪枝法中，除使用约束函数外，还使用更有效的评判函数---目标函数（对解的“好坏”有衡量标准就称为**目标函数**）来控制搜索进程，使搜索能更好地向有最优解的分枝（方向）上推进，而及时丢弃不可能的分枝，以尽快得到最优解。

解的搜索方式：在回溯法中，搜索一般是以深度优先方式进行。在限界剪枝法中，搜索一般是广度优先方式进行，即先检测当前结点的每个儿子结点，并将其中满足约束条件与目标函数的儿子结点做为“活”结点，加入到活结点表中。当前结点的各个儿子处理完后，从活结点表中按某种方式取一结点，按类似方式处理。

活结点表的组织，根据具体问题的需要，可有栈式（FILO）、队式(FIFO)和优先队列式(PQ)。

在限界剪枝法中，将目标函数转换为代价函数 $C(x)$ 。这里， x 为部分解。 $C(x)$ 应满足单调性。如果 $C(x)$ 是不可及时计算的，则要用它的一个下界估价函数 $LC(x)$ 代替，即 $LC(x) \leq C(x)$ 。同时，也为 $C(x)$ 构造一个可即时计算的上界函数 $UC(x)$ ，这样有 $LC(x) \leq C(x) \leq UC(x)$ 。有了 $LC(x)$ 和 $UC(x)$ ，在搜索中，就可以把在 $[LC(x), UC(x)]$ 范围外的（认

为是不可能产生最优解的)子树剪掉(有点象园艺师那样对果树剪枝,这就是限界剪枝法的名称由来),使搜索集中在较小的范围。具体实例分析,限于篇幅就不在这里介绍了。

§ 12.2 动态规划法

动态规划(Dynamic Programming)法与分治法和回溯法都有某些类似,也是基于问题的划分解决(多步决策、递增生成子解)的。但在递增生成子解的过程中,力图朝最优方向进行,而且也不回溯。因此,动态规划法效率较高,且常用来求最优解,而不象回溯法那样可直接求全解。

§ 12.2.1 动态规划法要素与最优性原理

如果只从过程来看,可以简单地把动态规划法看作是回溯法去掉了“回溯”,或严格迭代分治法。也就是说,把一个问题的解决(求解),分成多个步骤(子问题),每个步骤都是在前面结果的基础上,得到一个新的子解(部分解),子解随着步骤的进行而逐步“扩大”,最后一步得到完整的解。在任一步骤(初始除外),子解都是在前面得到的子解的基础上生成的,而且生成方式一般是递推的,即第 i 步的子解与它前面的子解存在递推关系。

那么,动态规划法求出的解(获得的结果)是否一定正确(最优)?回答是:不一定。这主要决定于问题的性质和划分方式。能使用动态规划的问题必须满足一定的条件,即最优子结构和重叠子计算。

所谓**最优子结构**问题,是指满足**最优性原理**(Principle of Optimality)的问题。最优性原理是贝尔曼(Richard Bellman)在 20 世纪 50 年代针对多步决策问题提出的。它的基本意思是,对一个多步决策(求解)问题,不论初始状态和初始决策是什么,其余的决策必须相对于初始决策所产生的状态构成一个最优决策序列。

通俗地讲,最优性原理是指,在多步决策中,各子问题只与它的前面的子问题的解相关,而与如何得到子问题的过程无关,而且各子问题的解都是相对于当前状态的最优解,换言之,整个问题的最优解是由各个子问题的最优解构成。此外,第 i 步的子解与它前面步骤的子解存在递推关系。

显然,如果一个问题满足最优性原理,那么,它的最优解就可递推地得到,而不需要在解空间搜索。因此,会有很高的计算效率。

重叠子计算是指在多步决策中,每步的计算(求解)可分成若干子任务,这些子任务有的在前面已出现过,并已解决(计算)。

显然,如果问题具有重叠子计算的性质时,则可以将重叠子计算的结果保存下来,后面的步骤可以直接引用,不必重新计算,从而节省计算时间。

下面通过几个具体的例子说明。

§ 12.2.2 最长公共子序列

(一) 基本概念

字符串的子序列的概念不同于子串的概念。简单地讲，一个字符串的一个**子序列**，是在该字符串中删除若干字符后所成部分。因此，子串是子序列，但子序列不一定是子串。

形式化地讲，设 $X = "x_1x_2...x_n"$ 和 $Y = "y_1y_2...y_m"$ 是两个字符串，若存在一个严格的递增下标序列 $i_1, i_2, ..., i_m$ ，使得对于所有的 $j=1, 2, ..., m$ ，有 $y_j = x_{i_j}$ ，则称 Y 是 X 的子序列。

称 $(i_1, i_2, ..., i_m)$ 为该子序列的下标序列。

例如， $Y = "baeg"$ 是 $X = "abacefgd"$ 的一个子序列，相应的递增下标序列为 $(2, 3, 5, 7)$ 。

我们规定，空字符串（长度为 0）为任何字符串的子序列。

显然，如果串中存在相同字符，则下标序列不同的子序列也可能相同。例如，对串 $X = "abacefgd"$ ，下标序列 (1) 和 (3) 对应的子序列都是 $"a"$ 。

显然，子序列是集合的“成员”包含关系的推广，可以说是多成员序列包含关系，可以对应到实际应用中的许多问题。

给定两个字符串 X 和 Y ，若 Z 既是 X 的子序列，又是 Y 的子序列，则称 Z 是 X 与 Y 的**公共子序列**。两个给定的字符串可能有多个公共子序列（当然也可能没有），那么，我们称长度最大的那个为**最长公共子序列**。例如，若 $X = "2135"$ ， $Y = "16923655821"$ ，则它们的公共子序列有 2, 21, 23, 235, 1, 13, 135, 15, 3, 35, 5，因此，最长公共子序列为 235 和 135，长度均为 3。

显然，下列结论成立：

- ✓ 若 X 的长度为 n ，则 X 最多有 2^n 个不同的子序列，当 X 中每个字符都不相同时恰好有 2^n 个不同的子序列；
- ✓ 空串是任何两个串的公共子序列（从而，两个串不存在非空公共子序列时，可以称它们的公共子序列为空串）；
- ✓ 空串和其他串的最长公共子序列为空串；
- ✓ 若 X' 是 X 的子串，则 X' 的子序列也是 X 的子序列；

对于第一个结论，若串中每个字符都不相同时，则 n 长的字符串，共有 n 个不同的字符位置，子序列就是这 n 个位置上的字符的依序组合，每个位置可以出现，也可以不出现，倘若出现，次序不能颠倒。因此，每个位置有 2 种选择，则 m 个位置有 2^n 种选择。如果串中有重复字符，则不同的子序列个数显然小于 2^n 。

我们的问题是，给定两个字符串，找出其一个最长公共子序列。

找最长公共子序列，也许最容易想到的方法是穷举法，即求 X 和 Y 的最长公共子序列时，对 X 的每个子序列，检查其是否出现在 Y 中，若是，则表明该子序列是个公共子序列，并将它与当前记录到的最长公共子序列（当前最长公共子序列）比较，若它更长，

则记录它（即将它做为新的当前最长公共子序列）。初始时，可将空序列做为当前最长公共子序列。由于长为 n 的串的子序列个数是 2^n ，故穷举法的时间复杂度是指数级的。但是，用动态规划法可以获得 $O(nm)$ 的时间复杂度的算法（ n 与 m 分别是两个输入字符串的长度）。

(二) 公共子序列问题的最优子结构

要使用动态规划法，首先要考察问题是否满足最优性原理，即找出它的最优子结构。首先给出一个定理。

定理 12-1(最长公共子序列的最优子结构)：设字符串 $Z = "z_1z_2 \dots z_k"$ 是字符串 $X = "x_1x_2 \dots x_n"$ 和 $Y = "y_1y_2 \dots y_m"$ 的一个最长公共子序列，则

- a) 若 $x_n = y_m$ ，则 $z_k = x_n = y_m$ ，且 Z_{k-1} 是 X_{n-1} 和 Y_{m-1} 的一个最长公共子序列；
- b) 若 $x_n \neq y_m$ 且 $z_k \neq x_n$ ，则 Z 是 X_{n-1} 和 Y 的最长公共子序列；
- c) 若 $x_n \neq y_m$ 且 $z_k \neq y_m$ ，则 Z 是 X_n 和 Y_{m-1} 的最长公共子序列；

这里， $Z_{k-1} = "z_1z_2 \dots z_{k-1}"$ ， $X_{n-1} = "x_1x_2 \dots x_{n-1}"$ ， $Y_{m-1} = "y_1y_2 \dots y_{m-1}"$

证：

a) 用反证法。设 $z_k \neq x_n$ ，则 $S = "z_1z_2 \dots z_kx_n"$ 显然是 X 的子序列，考虑到 $x_n = y_m$ ，则它也是 Y 的子序列，所以， S 是 X 和 Y 的长度为 $k+1$ 的公共子序列，这与 Z 是 X 和 Y 的最长公共子序列矛盾。因此必有 $z_k = x_n = y_m$ ，所以， Z_{k-1} 是 X_{n-1} 和 Y_{m-1} 的一个长度为 $k-1$ 的公共子序列。若 X_{n-1} 和 Y_{m-1} 存在长度大于 $k-1$ 的公共子序列 T ，则将 x_m 加到 T 的尾部得到 T' ，则由于 $z_k = x_n = y_m$ ， T' 仍然为 X_{n-1} 和 Y_{m-1} 的公共子序列，但长度大于 k ，这与 Z 是最长公共子序列的假设矛盾，这表明， Z_{k-1} 是 X_{n-1} 和 Y_{m-1} 的一个最长公共子序列。

b) 由于 $z_k \neq x_n$ ，所以 Z 是 X_{n-1} 和 Y 的一个公共子序列。若 X_{n-1} 和 Y 有一个长度大于 k 的公共子序列 T ，则 T 也必为 X 和 Y 的一个公共子序列（因为 X_{n-1} 是 X 的前缀），这与 Z 是最长公共子序列（长为 k ）的假设矛盾，这表明 Z 是 X_{n-1} 和 Y 的一个最长公共子序列。

c) 证明方法与 b) 类似。

这个定理表明，两个串的最长公共子序列，包含了这两个串的最大真前缀（长度比原串小一的前缀）的最长公共子序列，或真前缀之一与另一原串的最长公共子序列，从而也包含了这两个串的所有前缀的最长公共子序列。因此，若将前缀做为各个子解，则最长子序列问题存在最优子结构，满足最优性原理。

(三) 递推关系

由上面的定理知，求 $X = "x_1x_2 \dots x_n"$ 和 $Y = "y_1y_2 \dots y_m"$ 的一个最长公共子序列，可按如下方法递归完成：

- a) 当 X 或 Y 为空时，空串是它们的最长公共子序列；
- b) 当 $x_n = y_m$ 时，先找出 X_{n-1} 和 Y_{m-1} 的最长公共子序列，然后在其后加上 x_n ，所成串

即为 X 与 Y 的最长公共子序列；

c) 当 $x_n \neq y_m$ 时，先找出 X_{n-1} 和 Y 的最长公共子序列 T_1 ，再找出 X 和 Y_{m-1} 的最长公共子序列 T_2 ，那么， T_1 和 T_2 中最长者即为 X 和 Y 的最长公共子序列。

对于 b)，结论是显然的。对于 c)，显然 T_1 和 T_2 也都是 X 与 Y 的公共子序列，再根据上面的定理， X 与 Y 的最长公共子序列，或者是 T_1 ，或者是 T_2 ，故应该是它们中最长者。

如果我们用 $lcsL[i][j]$ 表示串 " $x_0x_1 \dots x_{i-1}$ " 和 " $y_0y_1 \dots y_{j-1}$ " (原串的前缀) 的最长公共子序列的长度，则上面的递推关系可进一步表达为：

$$lcsL[i][j] = \begin{cases} 0 & \text{若 } i=0 \text{ 或 } j=0 \\ lcsL[i-1][j-1]+1 & \text{若 } x[i-1]=y[j-1] \\ \text{MAX}(lcsL[i][j-1], lcsL[i-1][j]) & \text{其他} \end{cases}$$

这是个求最长公共子序列的长度的递推式，而不是求最长公共子序列本身的递推式。但是，有了找个式子，就不难求最长公共子序列了。如果直接找出最长公共子序列本身的递推式，当然也可以，但是比较困难。通过求“长度”的递推式求最长公共子序列本身，将难点分散了，是一种值得学习的思路。

有了这个递推关系，就可以写出具体的实现程序了。

(四) 求长度算法实现

我们这里通过求最长公共子序列的长度，来求最长公共子序列本身。而求最长公共子序列的长度的问题，已归结到了计算上面给出的递推关系。有两种计算方法，一是自顶向下，即使用递归方法；另一是自底向上，即采用非递归方法。不论那种方法，都要记下 $L[i][j]$ 的值，只是递归方法隐含用栈记录，同时还要记下相关子问题的界限（状态等），所以需要额外的存储空间，而非递归方法不需要。我们这里采用非递归方法。

非递归方法计算，从递推量 i 和 j 的最小值开始，依次计算各相应的 X_i 和 Y_j ，并保存相应的计算结果供后面计算引用。下面是具体的 C 程序。

```
int LCSLen2(char *x, char *y, int lcsL[][CNST_LCS_MaxM+1])
{ //求 x 与 y 的最长公共子序列长度，返回其。
  //并且，将 x 与 y 的各前缀间的最长公共子序列长度求出，存于 lcsL
  //lcsL[i][j]---前缀"x[0]~x[i-1]"与前缀"y[0]~y[j-1]"的最长公共子序列的长度

  int n,m,k;
  int i, j;

  n=strlen(x);
  m=strlen(y);
```



```

//空串和任何串的公共子序列为空
for (i=0;i<=n;i++) lcsL[i][0]=0; //令 lcsL[i][0]表示  $X_i$  与空串的最长公共子序列长度;
for (i=0;i<=m;i++) lcsL[0][i]=0; //令 lcsL[0][i]表示  $Y_i$  与空串的最长公共子序列长度;

for (i=1;i<=n; i++)
{
    for (j=1; j<=m; j++)
        if (x[i-1]==y[j-1])
            lcsL[i][j] = lcsL[i-1][j-1]+1;
        else
            if (lcsL[i-1][j] > lcsL[i][j-1])
                lcsL[i][j] = lcsL[i-1][j];
            else
                lcsL[i][j] = lcsL[i][j-1];

} //for (i=

return lcsL[n][m];
}

```

(五) 求最长子序列算法

上面给出的求长度算法 `LCSLen2()` 并没有找出具体的最长的公共子序列。但有了它的结果(`lcsL[i][j]`), 就可以方便地求出具体的最长公共子序列了。

根据前面给出的递推关系可知, 若 $lcsL[i][j]=lcsL[i-1][j-1]+1$, 则表示 X_i 和 Y_i 的最长公共子序列是在 X_{i-1} 和 Y_{j-1} 的最长公共子序列尾部加 x_i 得到, 否则, 如果 $lcsL[i][j]=lcsL[i-1][j]$, 则 X_i 和 Y_i 的最长公共子序列等于 X_{i-1} 和 Y_j 的最长公共子序列, 否则, 等于 X_i 和 Y_{j-1} 的最长公共子序列。据此可写出下面的程序。

```

int LCSOut(int lcsL[][CNST_LCS_MaxM+1], char *x, int i, int j, char *lcs)
{ //输出串 x[0]~x[i]和串 y[0]~y[j]的一个最长公共子序列到 lcs,返回其长度
    //lcsL: 二维数组, 输入量, 存储 x 与 y 的各前缀的间的最长公共子序列的长度,
    //其由算法 LCSLen 产生
    //x: 一维数组, 输入量, 代表求公共子序列的两个字符串之一, 另一串 y 在这里不需要给出
    //i, j: 输入量, 分别表示 x 与 y 的当前待求 LCS 的前缀的尾下标 (长度)
    //lcs: 一维数组, 输出量, 存放所求出的最长公共子序列

    int cnt=0;

```

```

if (i==0 || j==0) return 0;
if (lcsL[i+1][j+1]==lcsL[i+1][j]+1)
{
    cnt = cnt + LCSOut(lcsL,x,i-1,j-1,lcs);
    lcs[cnt++]= x[i];
}
else
if (lcsL[i+1][j+1]==lcsL[i][j+1])
    cnt = cnt + LCSOut(lcsL,x,i-1,j,lcs);
else cnt = cnt + LCSOut(lcsL,x,i,j-1,lcs);

return cnt;
}

```

显然，若调用 `LCSOut(lcsL, X, n-1,m-1,lcs)`，即可得到完整字符串的最长公共子序列（ n 和 m 分别为两个串的长度）。

(六) 算法分析与改进

对于求最长公共子序列的长度的算法 `LCSLen2()`，由于主要由循环次数固定的两层嵌套循环构成，所以时间复杂度显然为 $O(nm)$ ，这里， n 与 m 为两个待求最大公共子序列的串的长度，下同。至于空间复杂度，主要用二维数组 `lcsL[][]` 存放中间和最后计算结果，所以复杂度也为 $O(nm)$ 。

至于求最长公共子序列的算法 `LCSOut()`，由于在递归阶段每次 i 和 j 至少有一个要减少 1，而且减到 0 时不再递归，所以，时间复杂度为 $O(n+m)$ 。空间复杂度决定于递归深度，显然最大递归深度为 $MAX(n,m)$ 。

算法 `LCSLen2()` 在空间复杂度方面还可以改进。从递推关系看出，在递推计算中，对任一步，只需要上一步的结果和同一步中较前面的结果，因此，可以只保留这些可能用得到的结果，而不必保留每一步的计算结果（即数组 `lcsL[][]`）。显然，用两个一维数组可以分别保留上一步和当前一步的计算结果，具体程序留作练习。

但是，不生成 `lcsL[][]` 时，算法 `LCSOut()` 不能工作。所以求具体的最长公共子序列方法也要改变。一种直接的方法是，在求长度的同时求具体序列，具体程序留作练习。

§ 12.2.3 流水线调度问题*

(一) 基本问题

一条**流水线**是这样一种加工处理设备，它由多个部件（**子设备，处理机**） P_1, \dots, P_m 构成。每个部件可独立工作，完成一定的功能。各部件往往有顺序关系，某部件接收它

前面部件产生的结果（已完成的子任务），对其进行加工处理，完成后送到它后面的部件继续处理。

流水线上处理的独立完整工作称为**作业**。每个作业 T_i 顺序分成 m 个**子作业**（也称**任务**） $T_{i1}, T_{i2}, \dots, T_{im}, 1 \leq i \leq n$ ，子作业 T_{ij} 只能在处理机 P_j 上执行，而且，子作业 T_{ij} 完成前， $T_{i,j+1}$ 不能开工。此外，每个处理机在任何同一时刻最多只能执行一项子作业。作业 T_{ij} 在 P_j 上的处理时间是 t_{ij} 。设有 n 个作业要在流水线上完成，那么，流水线调度问题是为 n 个作业安排在流水线上的工作时间表，使它们能在最少的时间内完成。

在调度方式上，分为优先调度和非优先调度两种。**非优先调度**（也称**不可剥夺调度**）（non preemptive schedule）是指把某台处理机分配给某个子作业后，在未完成该子作业时不容许中断而转去处理其他子作业。而**优先调度**（也称**可剥夺调度**）（preemptive schedule）是指为各作业分别指定一个优先数，允许优先数高的作业（的子作业）中断优先数低的作业，被中断的子作业在优先数高的子作业执行完后恢复执行。

流水线调度问题是许多实际问题的抽象。如果一个作业需要顺序经过 m 个步骤（处理机）处理，则可以看作该作业由 m 个子作业构成，每个处理步骤就是一部处理机，这样就可以对应到流水线调度。例如，除了工厂的各种流水线调度外，由多个处理器构成的计算机上多道程序（每道程序相当于一个作业）的运行也属于该问题。

例如，设有一个三处理机两作业调度问题，各作业所需的处理机时间由下面的矩阵给出。

$$T = \begin{pmatrix} 2 & 3 & 5 \\ 0 & 3 & 2 \end{pmatrix}$$

图 12-0 给出了两种不同的调度方案。左图方案用了 12 个单位时间，而右图用了 11 单位时间。因此，调度方案直接影响整体速度。

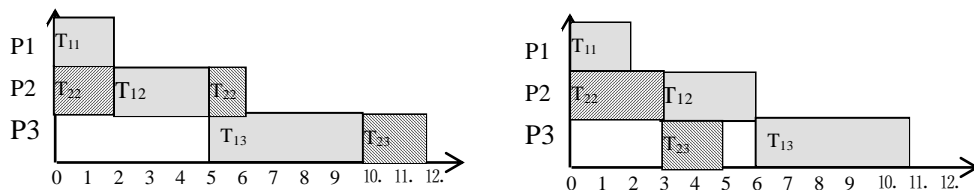


图 12-4 作业调度示例(左：优先方式；右：非优先方式)

为了讨论方便，我们定义几个概念。

作业 i 在某调度方式 s 下的**完工时间** $f_i(s)$ 定义为

$f_i(s)$ = 作业 i 的所有子作业都完成的时间

例如，在上面的例子中，左图对应的调度下， $f_1(s_1)=10, f_2(s_1)=12$ ，右图对应的调度下， $f_1(s_2)=11, f_2(s_2)=5$ 。

调度 s 的**总完工时间** 定义为：

$$F(s) = \text{MAX}_i \{ f_i(s) \}$$

调度 s 的**平均完工时间** $MFT(s)$ 定义为：

$$MFT(s) = \frac{1}{n} \sum_i f_i(s)$$

给定作业集 T 的**最佳完工时间** **OFT** 定义为各种调度方案下的总完工时间的最小值，即

$$OFT(T) = \min_i \{ F(s_i) \}$$

而相应的调度 s_k （取得最小值的调度）称为 T 的**最佳调度**。

给定作业集 T 的**最佳平均完工时间** **OMFT** 定义为各种调度方案下的平均完工时间的最小值，即

$$OMFT(T) = \min_i \{ MFT(s_i) \}$$

而相应的调度 s_k （取得最小值的调度）称为 T 的**最佳平均调度**。

定义的这几个量是衡量调度方案的指标。若从全局考虑，我们一般期望得到能达到最佳平均完工时间 **OMFT**（或最佳完工时间 **OFT**）的调度（最佳(平均)调度）。

但是，作业的子作业个数 m 大于 2（流水线上处理机个数大于 2）时，求 **OFT/OMFT** 是相当困难的。当 $m=2$ 时，可使用动态规划法有效地得到 **OFT**。下面我们就讨论 $m=2$ 这一特殊情况。

(二) 最优子结构

由于我们只考虑流水线只有两台处理机（每个作业有两个任务）的情况，所以，可简单地将 n 个作业的任务分为两组， A 组包括各作业的在处理机 A 上执行的各任务，即 $\{T_{11}, T_{12}, \dots, T_{1n}\}$ ， B 组包括各作业的在处理机 B 上执行的各任务，即 $\{T_{21}, T_{22}, \dots, T_{2n}\}$ 。同时为了方便，用 a_i 和 b_i 分别表示作业 i 在处理机 A 和 B 上的执行时间，即 t_{i1} 和 t_{i2} 。

在只有两台处理机的情况下，有个重要的性质：**两台处理机按同样的顺序处理作业，不会比其他顺序更浪费时间**。也就是说，设有 n 个作业，若处理机 A 处理的作业的次序是 $(T_{1,p1}, T_{1,p2}, \dots, T_{1,pn})$ ，那么，若处理机 B 的执行作业次序是 $(T_{2,p1}, T_{2,p2}, \dots, T_{2,pn})$ ，则总的执行时间不会比处理机 B 用任何其他次序的执行时间更多。若 A 与 B 对调，情况亦如此。这里 $(p1, p2, \dots, pn)$ 是 $1 \sim n$ 的一个全排列。我们把这种调度称为**同序调度**。

根据这个性质，我们只需要为一台处理机安排作业执行顺序，而另一台取相同的次序即可。注意，这并不意味着在安排作业时忽略另一台的存在。事实上，若我们规定只为处理机 A 安排作业次序，则显然不论取何种次序，各作业（即各作业的 A 处理机的任务，下同）之间应该没有间隔时间，但是，由于同一作业的两个任务存在次序关系，所以， B 处理机上各作业一般很难达到没有间隔的情况（若都没有间隔，这这种调度肯定是最优的），显然，总间隔最小的那种调度，肯定是最优的。那么， B 上总间隔的大小决定于什么呢？显然决定于 A 上作业次序。因此，考虑 A 上的作业次序时，当然要涉及到 B 的情况。

例如，设有 4 个作业，每个作业分 2 个任务。4 个作业在处理机 A 和 B 上的处理时

间分别如下:

$$A=(a_1, a_2, a_3, a_4) = (5, 1, 2, 3)$$

$$B=(b_1, b_2, b_3, b_4) = (3, 2, 5, 1)$$

图 12-0 给出了 2 种不同的同序调度。对第一种调度 (图的上部分), 我们先为 A 处理机安排作业, 依次按 a_2, a_1, a_4, a_3 的次序安排, 那么, B 处理机也按同样次序安排: b_2, b_1, b_4, b_3 , 由于对于每个作业, 只有它的 a 任务完成后, 才能开始 b 任务, 所以, 我们看到, 这种安排使处理机 B 上有 3 个空隙 (间隔)。图 12-0 的下图是另一种调度方式, 它只有开始的一个空隙, 易知, 这种调度是最佳调度。下面给出这两种调度的几个度量值:

第一种调度 s_1 (上图)

$$\text{总完工时间: } F(s_1) = \text{MAX}\{f_1(s_1), f_2(s_1), f_3(s_1), f_4(s_1)\}$$

$$= \{9, 3, 16, 10\} =$$

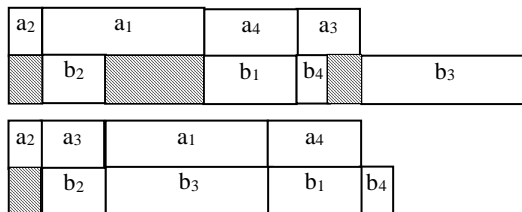


图 12-5 一个 4 作业 2 处理机的两种调度的例子

16

$$\text{平均完工时间: } \text{MFT}(s_1) = (9+3+16+10)/4 = 9.5$$

第二种调度 s_2 (下图)

$$\text{总完工时间: } F(s_2) = \text{MAX}\{f_1(s_2), f_2(s_2), f_3(s_2), f_4(s_2)\}$$

$$= \{12, 3, 8, 13\} = 13$$

$$\text{平均完工时间: } \text{MFT}(s_2) = (12+3+8+13)/4 = 6.5$$

由于各作业在每台处理机上的排列是线性的, 所以, 作业排列显然满足最优性原理: **最佳排列中各前缀排列都是最佳的**。也就是说, 若 (p_1, p_2, \dots, p_n) 是一个最佳排列, 则对任何 $1 \leq k \leq n$, k -前缀排列 (p_1, p_2, \dots, p_k) 是一个最佳排列。这里, k -前缀排列 (p_1, p_2, \dots, p_k) 是假定 k 以后的作业存在的情况下做出的。

由该性质知, 我们只要知道 $k=1$ 时的最优选择方法, 然后依次计算各个 k -前缀排列就可以了。这需要找到递推关系。

设对每个同序调度 (由一个 a 任务序列和一个 b 任务序列构成), 处理机 A 和 B 处理它的两个任务序列中前 k 个任务所需时间分别为 f_1 和 f_2 , 则 $t=f_2-f_1$ 表示处理机 B 上完成前 k 个 b 任务的 (相对于 A 的完成的) 滞后时间 (时间差), 即在 A 完成前 k 个任务后, B 再经过 t 时间才完成 b 任务的全部前 k 个任务。 t 反映了调度的好坏, t 越小, 调度越好。

对上述意义下的 k 和 t , 设 k 以后的任务集为 S , 令 $g(S, t)$ 表示在前 k 个任务完成后 (两个处理机的时间差为 t), 对 S 所做的最佳调度的完工时间。显然, 整个作业集合 $\{1, 2, \dots, n\}$ 的最佳调度完工就表示为 $g(\{1, 2, \dots, n\}, 0)$ 。

由调度的最优性知:

$$g(\{1, 2, \dots, n\}, 0) = \text{MIN}_i \{a_i + g(\{1, 2, \dots, n\} - \{i\}, b_i)\} \quad (1 \leq i \leq n)$$

下面考虑将该式推广到一般情况。根据 $g(S, t)$ 的定义，我们有

$$g(\Phi, t) = \text{MAX}(0, t)$$

$$g(S, t) = \text{MIN}_i \{ a_i + g(S - \{i\}, t') \} \quad (i \in S)$$

上面第一式是显然的，因为对空作业集调度不需要额外的时间。对第二式，是由最优性原理得来的： S 是最佳排列，则它中第一个（即 a_i ）以及随后的排列也是应该是最优的，这显然应该取各个 $a_i + g(S - \{i\}, t')$ 中最小者。

下面考察上式中的 t' 的具体形式。按上面的定义知，这里的 t' 为两台处理机执行完任务 i （及 i 以前的任务）后的时间差，故

$$t' = f_2 - f_1 = b_i + \text{MAX}\{a_i, t\} - a_i = b_i + \text{MAX}\{0, t - a_i\}$$

因此， g 的递推计算式为：

$$g(\Phi, t) = \text{MAX}(0, t)$$

$$g(S, t) = \text{MIN}_i \{ a_i + g(S - \{i\}, b_i + \text{MAX}\{t - a_i, 0\}) \} \quad (i \in S)$$

(三) 算法设计

调度算法的中心问题是计算上面给出的递推关系 $g(S, t)$ ，这可以从空集开始，依次为 S 增加元素（每次加一个元素），并进行相应的计算。显然，任务集的各种子集都要尝试，由于 $\{1, 2, \dots, n\}$ 共有 2^n 个子集，所以，这种计算是指数级的。为了减少计算量，我们对递推式进行变换。

现设 R 是任务集的子集 S 上的任一调度为。 i 和 j 是 R 中排在最前面的两个任务，变换 $g(S, t)$ 的递推式：

$$\begin{aligned} g_R(S, t) &= a_i + g_R(S - \{i\}, b_i + \text{MAX}\{t - a_i, 0\}) \\ &= a_i + a_j + g_R(S - \{i, j\}, b_j + \text{MAX}\{b_i + \text{MAX}\{t - a_i, 0\} - a_j, 0\}) \end{aligned}$$

记 $t_{ij} = b_j + \text{MAX}\{b_i + \text{MAX}\{t - a_i, 0\} - a_j, 0\}$ ，便有

$$g_R(S, t) = a_i + a_j + g(S - \{i, j\}, t_{ij})$$

化简 t_{ij} ：

$$\begin{aligned} t_{ij} &= b_j + \text{MAX}\{b_i + \text{MAX}\{t - a_i, 0\} - a_j, 0\} \\ &= b_j + (b_i - a_j) + \text{MAX}\{\text{MAX}\{t - a_i, 0\}, a_j - b_i\} \quad // \text{注意: } \text{MAX}(x+y, 0) = x + \text{MAX}(y, -x) \\ &= b_j + b_i - a_j + \text{MAX}\{t - a_i, a_j - b_i, 0\} \\ &= b_j + b_i - a_j - a_i + \text{MAX}\{t, a_i + a_j - b_i, a_i\} \end{aligned}$$

如果作业 i 和 j 易位，则得 S 的另一种调度 R' ，相应的完工时间为

$$g_R(S, t) = a_j + a_i + g_R(S - \{i, j\}, t_{ji})$$

其中 t_{ji} 为

$$t_{ji} = b_j + b_i - a_j - a_i + \text{MAX}\{t, a_i + a_j - b_j, a_j\}$$

比较式 $g_R(S, t)$ 和 $g_{R'}(S, t)$ ，它们的主要差异在 t_{ij} 和 t_{ji} ，显然，如果下式成立

$$\text{MAX}\{t, a_i + a_j - b_i, a_i\} < \text{MAX}\{t, a_i + a_j - b_j, a_j\}$$

则有

$$g_R(S, t) < g_R(S, t)$$

为了让上式对任意 t 都成立，应该有：

$$\text{MAX}\{a_i + a_j - b_i, a_i\} < \text{MAX}\{a_i + a_j - b_j, a_j\}$$

注意性质：若 $\text{MAX}(a, b) < \text{MAX}(x, y)$ ，则 $c + \text{MAX}(a - c, b - c) < c + \text{MAX}(x - c, y - c)$ 。

从而有 $a_i + a_j + \text{MAX}\{-b_i, -a_j\} < a_i + a_j + \text{MAX}\{-b_j, -a_i\}$

$$\text{MIN}\{b_i, a_j\} \geq \text{MIN}\{b_j, a_i\} \quad (\text{F1})$$

该式告诉我们，在一个调度排列 $\{p_1, p_2, \dots, p_{k-1}, p_k, p_{k+1}, \dots, p_n\}$ 中，若 $\{p_1, p_2, \dots, p_{k-1}\}$ 已是最佳排列（的前缀），则要使 $\{p_1, p_2, \dots, p_{k-1}, p_k\}$ 成为更长的最佳排列（的前缀）， p_k 与它的后邻 p_{k+1} 应该满足上式（ p_k 与 p_{k+1} 分别做为上式的 i 和 j ）。因此，我们可以这样生成调度：

■ 如果 $\text{MIN}\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\} = a_i$ ，则任务 i 是调度中的第一个任务。这是因为，在这种情况下，对所有 $k \neq i, 1 \leq k \leq n$ ，有 $\text{MIN}\{b_i, a_k\} \geq \text{MIN}\{a_i, b_k\} = a_i$ ，由式(F1)知，在当前考虑的调度中， a_i 是第一个任务。

■ 如果 $\text{MIN}\{a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_n\} = b_j$ ，则任务 j 是调度中的最后一个任务。这是因为，在这种情况下，对所有 $k \neq j, 1 \leq k \leq n$ ，有 $\text{MIN}\{b_k, a_j\} \geq \text{MIN}\{b_j, a_k\} = b_j$ ，由式(F1)知，在当前考虑的调度中，每个 a_k 都应该是比 b_j 在先。

■ 这样，已经对这 n 项任务的调度序列找到了第一和最后一个任务，接下来，对剩余的任务按类似的方法调度。

将这个过程进一步归纳，得到下面的调度方法：

设当前调度序列为空，令 $\text{top}=1; \text{rear}=n$;

对所有的 $1 \leq i \leq n$ ，令 $\text{tAB}_i = \text{MIN}\{a_i, b_i\}$;

将 $\text{tAB}_1, \text{tAB}_2, \dots, \text{tAB}_n$ 排成非递减序列；

for ($k=1; k \leq n; k++$)

if ($\text{tAB}_k == a_i$) {将任务 i 安排在 $S[\text{top}]; \text{top}++$ }

else {将任务 i 安排在 $S[\text{rear}]; \text{rear}--$ } //此时 $\text{tAB}_k == b_i$

(四) 算法实现

在上面的讨论的基础上，给出计算程序就很容易了，具体程序如下。程序中用到一个排序子程序，可用任意排序算法，这里就不具体给出它的实现了。

```
int JobScheduling(int *a, int *b, int n, int *c)
```

```
{//两处理机流水线调度
```

```
//a,b----输入量，一维数组，n 个元素，
```

```
//分别代表 n 个作业的两个任务: A 处理机任务和 B 处理机任务
```

//c----一维数组, 输出量, 存放调度结果(调度序列), c[i]表示在处理机上第 i 次执行的任务

```
int top, rear, k;
```

```
int *tABTime, *tABNo;
```

```
tABTime = new int[n]; //存放算法中任务序列 tAB 的时间值
```

```
tABNo = new int[n];
```

```
//存放算法中任务序列 tAB 的时间值对应的任务号(数组 a 中的序号)
```

```
//若对应的时间值属于 B 任务,则令该值为任务号的负值,以示区别
```

```
top=0;
```

```
for (k=0; k<n; k++) //生成算法中的任务序列 tAB
```

```
    if (a[k] < b[k])
```

```
    {
```

```
        tABTime[top]=a[k];
```

```
        tABNo[top]=k+1;
```

```
        top++;
```

```
    }
```

```
    else
```

```
    {
```

```
        tABTime[top] =b[k];
```

```
        tABNo[top]=-(k+1);
```

```
        top++;
```

```
    }
```

```
SortArr2(tABTime,tABNo, n); //对 tABTime 排序,同时也相应地调整 tABNO 的次序
```

```
top=0; rear=n-1;
```

```
for (k=0; k<n; k++) //根据 tAB 序列构造任务序列
```

```
    if (tABNo[k] > 0) c[top++] = tABNo[k];
```

```
    else c[rear--] = -tABNo[k];
```

```
delete[] tABTime, tABNo;
```

```
return n;
```

```
}
```

分析该算法,其主要耗时的地方是排序和构造任务序列。排序算法时间复杂度可以

低到 $O(n \log_2 n)$ ，而构造任务序列时间复杂度为 $O(n)$ ，故整个算法的时间复杂度为 $O(n \log_2 n)$ 。

在该算法的基础上，也可增加计算最佳完工时间 OFT 的功能，具体留作练习。

这里讨论的是处理机个数 n （子作业数）为 2 的情况，对于 $n > 2$ 的情况，没有这样理想，要找出一个最佳调度，时间复杂度仍然指数级的。

§ 12.2.4 多源最短路径的 Floyd 算法

多源最短路径是关于图结构的问题。有关图的概念，我们已在图结构一章中介绍，这里就不再给出了。图结构的路径问题包括三个方面，一是任意两点间是否有路径？二是任意两点间的一条路径长为多少？三是任意两点间的最短路径长是多少？这里，假定边带权，路径长就定义为路径上边权之和。对它们的解答是图结构的重要操作。

在贪心法中，我们介绍 Dijkstra 单源最短路径算法，它每次只求一个源点到其余各点间的最短路径。显然，若要求每对顶点间的最短路径（多源最短路径），可调用 n 次 Dijkstra 算法，时间复杂度为 $O(n^3)$ 。

但对此问题，Floyd 发现了一个更为直接的算法，我们称之为 Floyd 多源最短路径，它的时间复杂度也是 $O(n^3)$ 。Floyd 算法属于动态规划法，所以我们在这里介绍。

(一) 基本思想

Floyd 指出，如果 $A^k(i, j)$ 表示结点 i 到 j 的中间结点序号不大于 k 的最短路径长度 ($k=1, \dots, n$)，那么对它做 n 次迭代，第 n 次迭代后， $A^n(i, j)$ 即为 i 到 j 的最短路径值。具体地讲，Floyd 算法递推地产生一个矩阵序列（逻辑上）。

$$A^1, A^2, \dots, A^n$$

其中， $A^0 = g$ （即图的邻接矩阵）。显然， $A^n(i, j)$ 是 i 到 j 的最短路径长度，因为它中间经过的结点没受限制（即允许为 $1 \sim n$ ）。事实上，对每个 k ， $A^k(i, j)$ 都表示 i 到 j 的最短路径长，但该最短路径有个条件限制：中间结点序号不大于 k 。随着 k 的增大，条件越来越放宽（对中间结点限制越来越少），直至 $k=n$ 时，限制完全取消。所以，Floyd 算法也可以看作是一种迭代算法。

要进行这种迭代， $A^k(i, j)$ 必须满足最优性原理，即要找出具体的迭代（递推）关系，下面我们接着讨论。

(二) 递推公式及其正确性

首先我们指出，最短路径满足最优性原理，这是可使用动态规划法的必要条件。

定理 12-2 假定图中边权均非负，那么最短路径满足最优性原理。

证：考察图中任意两不同点 i 和 j 间的最短路径。最短路径由 i 出发，中间经过一些其他结点（也可能不经过），在 j 处结束。首先我们肯定，在起点和终点不同的情况下，

最短路径肯定不含回路，否则，我们可以通过删除某些边来打断环，而不增加长度（因为边权非负）。如果结点 k 在 i 到 j 的最短路径上，那么，由路径的顺序性知， i 到 j 的最短路径 $\text{path}(i,j)$ 可以看作由两段连接而成： i 到 k 的路径 $\text{path}(i,k)$ 和 k 到 j 的路径 $\text{path}(k,j)$ ，即 $\text{path}(i,j)=\text{path}(i,k)+\text{path}(k,j)$ 。如果 $\text{path}(i,k)$ 不是 i 到 k 的最短路径，或者 $\text{path}(k,j)$ 不是 k 到 j 的最短路径，那么，我们可以将 i 到 k 和 k 到 j 的最短路径首尾相连，构成 i 到 j 的最短路径，按假定这条路径不同于 $\text{path}(i,k)+\text{path}(k,j)$ ，矛盾。这个矛盾表明，最短路径 $\text{path}(i,j)$ 的子路径 $\text{path}(i,k)$ 和 $\text{path}(k,j)$ 也是最短子路径，故最优性原理成立。

下面要寻找最优子结构，讨论如何递推 A^k ，即假定 A^{k-1} 已求得，那么如何得到 A^k ？这由下面的定理给出。

定理 12-3 设图结点从 1 起编连续自然数，每条边 $\langle i,j \rangle$ 都有一个非负的权 $\text{cost}(i,j)$ ，边 $\langle i,j \rangle$ 不存在时 $\text{cost}(i,j)=\infty$ ，并假定不存在自己到自己的边，即对所有 i ，不存在边 $\langle i,i \rangle$ ，此时令 $\text{cost}(i,i)=0$ ；对任意不同的结点 i 和 j ，定义 $A^k(i,j)$ 为 i 到 j 的中间经过的结点的编号不大于 k 的最短路径的长度，那么

$$\begin{aligned} A^0(i,j) &= \text{cost}(i,j) \\ A^k(i,j) &= \min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\} \quad (k \geq 1) \end{aligned}$$

证：先考虑第一式。按定义， $i \neq j$ 时， $A^0(i,j)$ 就表示 i 到 j 的中间不经过任何结点的最短路径，也就是说是边 $\langle i,j \rangle$ ，显然有 $A^0(i,j)=\text{cost}(i,j)$ 。当 $i=j$ 时，由于 $\text{cost}(i,i)=0$ ，所以 $A^0(i,j)=\text{cost}(i,j)$ 亦成立。

下面考虑第二式，分下列情况：

(a) $i=j$ ：由于 $A^0(i,i)=\text{cost}(i,i)=0$ ，且 $A^{k-1}(i,k) + A^{k-1}(k,i)$ 永远非负，所以 $A^1(i,i)$ 为 0，以此为基础，容易归纳出对所有的 k 有 $A^k(i,i)=0$ 。

(b) $i=k$ 或 $j=k$ （但不同时等于，否则就 $i=j$ ，是(a)中情况）：比如， $i=k$ ，此时，按定义 $A^k(k,j)$ 表示 k 到 j 的中间经过结点编号不超过 k 的最短路径。事实上，我们已假定对任意 x ，边 $\langle x,x \rangle$ 不存在，所以， $A^k(k,j)$ 实质上表示 k 到 j 的中间经过结点编号不超过 $k-1$ 的最短路径，即

$$A^k(k,j) = A^{k-1}(k,j)$$

另一方面，

$$A^{k-1}(k,j) = \min\{A^{k-1}(k,j), A^{k-1}(k,k) + A^{k-1}(k,j)\}$$

即 $A^k(k,j) = \min\{A^{k-1}(k,j), A^{k-1}(k,k) + A^{k-1}(k,j)\}$ ，这正是要证明的结论。对 $j=k$ 的情况可类似证明。

(c) $i \neq k$ 且 $j \neq k$ ：使用归纳法（对 k 归纳）。当 $k=1$ 时，按定义， $A^k(i,j)$ 表示 i 到 j 的中间只经过编号不超过 1 的结点的最短路径，这只有两种情况：情况 A，中间不经过任何结点；情况 B，中间只经过结点 1。具体是那种情况，要看两者中路径长谁较小，而取小者。易知：

$$\min\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\} = \min\{A^0(i,j), A^0(i,1) + A^0(1,j)\}$$

$$= \text{MIN}\{\text{cost}(i,j), \text{cost}(i,1) + \text{cost}(1,j)\}。$$

当 $i \neq 1$ 且 $j \neq 1$ 时, $A^0(i,1) + A^0(1,j)$ 显然是路径 $(i-1-j)$ 的长度, 即情况 B 的路径长, 这表明 $A^k(i,j) = \text{MIN}\{A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j)\}$ 在 $k=1$ 时成立。

现设 $k=m$ 时定理成立。考虑 $k=m+1$ 的情况。此时, i 到 j 的最短路径, 有两种情况: 情况 A: 中间不经过结点 $(m+1)$, 那么按定义 $A^{m+1}(i,j) = A^m(i,j)$; 情况 B: 中间经过结点 $(m+1)$, 此时, i 到 j 最短路径由两段连接而成, 一段是 i 到 $m+1$ 路径 $\text{path}(i,m+1)$, 中间经过的结点的编号不超过 m (注意, 最短路径上显然没有环路, 所以每个结点在最短路径中最多出现一次), 另一段是 $m+1$ 到 j 的路径 $\text{path}(m+1,j)$, 中间经过的结点的编号也不超过 m 。由最优性原理 (定理 12-2) 知, $\text{path}(i, m+1)$ 是 i 到 k 的最短路径, $\text{path}(m+1,j)$ 是 $m+1$ 到 j 的最短路径, 而 i 到 j 的最短路径长是这两个最短路径长度的和。根据归纳假设, $\text{path}(i, m+1)$ 的长度为 $A^m(i, m+1)$, $\text{path}(m+1,j)$ 的长度为 $A^m(m+1,j)$, 因此, 情况 B 下, i 到 j 最短路径长度为 $A^m(i,m+1) + A^m(m+1,j)$ 。综合情况 A 和 B, 应有 $A^{m+1}(i,j) = \text{MIN}\{A^m(i,j), A^m(i,m+1) + A^m(m+1,j)\}$, 这表明, $k=m+1$ 时定理成立。

(三) Floyd 算法实现

上面给出的定理, 实质上就是 Floyd 算法的理论表述。具体实现时, 主要问题是如何计算递推式 $A^k(i,j)$ 。

首先, 根据 Floyd 算法的特点, 我们用邻接矩阵表示图。在邻接矩阵中, 用 ∞ 代表无边标志。但是, 由于在上面的定理中, 规定矩阵的对角元素为 0, 所以, 边 $\langle i,i \rangle$ 权应置为 0。

接着看如何存储 $A^k(i,j)$ 。算法逻辑上要产生一个递推序列 A^0, A^1, \dots, A^n , 但在算法运行中, 没必要保存这个序列, 只在一个矩阵上迭代即可, 所以, 我们可以用一个二维数组 A 表示 $A^k(i,j)$ 。 $A[i][j]$ 的第 k 次迭代就代表 $A^k(i,j)$ 。

另外, 为了记下各最短路径上的结点序列, 我们引进一个与 A 规模相同的路径矩阵 path , 它的作用类似于 Dijkstra 算法中的路径数组。算法结束后, $\text{path}[i][j]$ 表示 i 到 j 的最短路径的前趋路径 (最短路径的长度最大的最左真子路径称为前趋路径) 的终点序号, 顺着它即可找到任一最短路径的结点序列。 path 也是迭代生成的。 $\text{path}^k[i][j]$ 表示 i 到 j 的中间结点序号不大于 k 的最短路径的前趋路径的终点。

具体的算法实现见下面的程序。显然该程序的时间复杂度为 $O(n^3)$

```
void FloydPath(float g[][CNST_NumGrphNodes3], long n, long maxWeight,
               float A[][CNST_NumGrphNodes3], long path[][CNST_NumGrphNodes3])
{ //g---输入量, 表示图的邻接矩阵。
  //n---输入量图结点个数;
  //maxWeight---输入量, 表示逻辑最大权值
  //A---输出量, 规模同g的二维数组, 存放各对顶点间的最短路径的值;
  //path---输出量, 规模同g的二维数组, 为各对顶点间的最短路径的结点序列的链接。
```

```

long i, j, k;

for (i=0; i<n; i++) //初始化, 令A的初值为邻接矩阵g
    for (j=0; j<n; j++)
    {
        A[i][j] = g[i][j];
        if (A[i][j] < maxWeight) path[i][j] = i;
        else path[i][j] = -1; //-1表示不存在路径
    }
for (i=0; i<n; i++) //将对角线上元素置为0
    { A[i][i] = 0; }

for (k=0; k<n; k++) //迭代求A
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (A[i][k] + A[k][j] < A[i][j] )
            {
                A[i][j] = A[i][k] + A[k][j];
                path[i][j] = path[k][j];
            }
    }//FloydPath

```

为了帮助理解该程序, 我们给出表 12-1, 它中给出了针对图 12-0 的 Floyd 算法的各次迭代结果

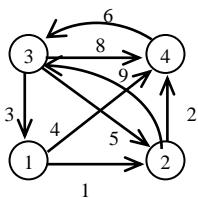


图 12-6 一个有向图

表 12-1 Floyd 算法求解迭代过程示例

	A ⁰				A ¹				A ²				A ³				A ⁴			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	∞	4	0	1	10	3	0	1	10	3	0	1	9	3
2	∞	0	9	2	∞	0	9	2	∞	0	9	2	12	0	9	2	11	0	8	2
3	3	5	0	8	3	4	0	7	3	4	0	6	3	4	0	6	3	4	0	6
4	∞	∞	6	0	∞	∞	6	0	∞	∞	6	0	9	10	6	0	9	10	6	0
	path ⁰				path ¹				path ²				path ³				path ⁴			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
1	-1	0	-1	0	-1	0	-1	0	-1	0	1	1	-1	0	1	1	-1	0	3	1
2	-1	-1	1	1	-1	-1	1	1	-1	-1	1	1	2	-1	1	1	2	-1	3	1
3	2	2	-1	2	2	0	-1	0	2	0	-1	1	2	0	-1	1	2	0	-1	1
4	-1	-1	3	-1	-1	-1	3	-1	-1	-1	3	-1	2	0	3	-1	2	0	3	-1

§ 12.2.5 0-1 背包问题

(一) 基本问题

设有一个容量(重量)为 M 的背包和 n 种物品, 每种物品都有一定的重量和装包利润 (装入背包后所获得的利润), 如果背包的剩余容量容许, 则每种物品都可装入背包, 但要求每种要么全部装入背包, 要么不装入。0-1 背包问题是, 如何从这 n 种物品中选择若干种装入背包, 使得总的获利最大。

0-1 背包问题的形式化描述是: 已知一个容量(重量)为 M 的背包和 n 种物品, 每种物品 i 的重量为 w_i , 假定将物品 i 全部装入背包, 会得到 p_i 的效益, 这里, $1 \leq i \leq n, p_i > 0$ 。现在的问题是, 选择哪些物品来装包, 才能使总的收益达到最大? 也就是说, 要求找出一个 n 元向量 (x_1, x_2, \dots, x_n) , $x_i \in \{0, 1\}$, $1 \leq i \leq n$, 使得

$$\sum_{i=1}^n w_i x_i \leq M \quad (\text{约束条件}), \quad \text{且总利润} \sum_{i=1}^n p_i x_i \quad (\text{目标函数}) \text{ 达到最大}$$

0-1 背包问题是许多实际问题的抽象, 如货物装运、任务调度、存储 (包括仓库) 使用等。

0-1 背包问题也属于组合问题。任意的一个长度为 n 的 0/1 串都是一个可能解, 显然, 这种组合总数为 2^n , 因此, 若按穷举法解, 时间复杂度为 $O(2^n)$ 。

为了讨论方便, 我们用 $\text{Knap}(i, j, C)$ 表示背包问题, 其中, C 表示背包容量, i 和 j 表示要考虑装包的物品的编号, $\text{Knap}(i, j, C)$ 表示对编号从 i 到 j 的物装入容量为 C 的装包的 0-1 背包问题。

(二) 0-1 背包问题最优性和递推关系

定理 12-4 0-1 背包问题满足最优性原理，即若 $\{y_1, y_2, \dots, y_n\}$ 是 $\text{Knap}(1, n, M)$ 的一个最优序列 (0/1 序列)，则 $\{y_2, \dots, y_n\}$ 是 $\text{Knap}(2, n, M)$ 的一个最优序列。

证：设 $\{y_1, y_2, \dots, y_n\}$ 是 $\text{Knap}(1, n, M)$ 的一个最优序列 (0/1 序列)。若 $y_1=0$ ，则最优序列为 $\{0, y_2, \dots, y_n\}$ ，这表示在没有装入 y_1 的情况下背包获得了最大利润，最优解可忽略 y_1 的存在，只由 y_1 以外的其他物品的选择获得，因此， $\{y_2, \dots, y_n\}$ 必然相对于 $\text{Knap}(2, n, M)$ 构成一个最优序列。

若 $y_1=1$ ， $\{y_2, \dots, y_n\}$ 也必然相对于 $\text{Knap}(2, n, M-w_1)$ 构成一个最优序列。如若不然，必另有一个 0/1 序列 z_2, z_3, \dots, z_n ，使得 $\sum_{2 \leq i \leq n} w_i z_i \leq M-w_1$ ，且 $\sum_{2 \leq i \leq n} p_i z_i > \sum_{2 \leq i \leq n} p_i y_i$ 因此 $1 + \sum_{2 \leq i \leq n} p_i z_i > 1 + \sum_{2 \leq i \leq n} p_i y_i$ ，这与 $\{y_1, y_2, \dots, y_n\}$ 是最优序列矛盾。

定理 12-5 若用 $f_k(C)$ 表示 $\text{Knap}(1, k, C)$ 的最优解的值，则

$$f_0(C)=0 \quad \text{当 } C \geq 0$$

$$f_0(C)=-\infty \quad \text{当 } C < 0$$

$$f_k(C) = \text{MAX}\{f_{k-1}(C), f_{k-1}(C-w_k)+p_k\} \quad k > 0$$

证：对 0-1 背包问题，我们的任务可以认为是决策一个序列 $\{y_1, y_2, \dots, y_n\}$ 。对任一变量 y_k 的决策 ($k > 0$)，是决定 $y_k=0$ 还是 $y_k=1$ 。在对 $k-1$ 决策后 (决策 k 时)，问题处于两种状态之一：一是背包剩余容量不足以装入 y_k 物品，不装入 y_k ，不产生新的利润；二是背包剩余容量可以装入 y_k 物品，利润增加 p_k 。这两种情况的利润最大者应为对 y_k 决策后的利润值 $f_k(X)$ ，因此有

$$f_k(C) = \text{MAX}\{f_{k-1}(C), f_{k-1}(C-w_k)+p_k\}$$

为了能够递推，还必须考虑递推的基础。如果对 $C \geq 0$ ，取 $f_0(C)=0$ ，对 $C < 0$ ，取 $f_0(C)=-\infty$ ，又在开始时肯定有 $C \geq 0$ ，则

$$\text{MAX}\{f_0(C), f_0(C-w_1)+p_1\} = \text{MAX}\{0, f_0(C-w_1)+p_1\}。$$

这里，若 $C \geq w_1$ (表示初始时，背包可以装得下 w_1)， $\text{MAX}\{f_0(C), f_0(C-w_1)+p_1\} = p_1$ ，而 $C < w_1$ 时 (表示初始时，背包不可以装得下 w_1)， $\text{MAX}\{f_0(C), f_0(C-w_1)+p_1\} = f_0(C)=0$ ，这两种情况与 $f_1(C)$ 的定义吻合。

这个递推式的计算是复杂的。例如，设 $n=3$ ， $M=6$ ， $(w_1, w_2, w_3)=(2, 3, 4)$ ， $(p_1, p_2, p_3)=(1, 2, 5)$ ，则 $f_k(C)$ 的递推计算如下：

$$f_0(C) = -\infty \quad \text{当 } C < 0$$

$$f_0(C) = 0 \quad \text{当 } C \geq 0$$

$$\begin{aligned} f_1(C) &= -\infty && \text{当 } C < 0 \\ f_1(C) &= \text{MAX}\{0, -\infty+1\}=0 && \text{当 } 0 \leq C < 2 \\ f_1(C) &= \text{MAX}\{0, 0+1\}=1 && \text{当 } C \geq 2 \end{aligned}$$

$$\begin{aligned} f_2(C) &= -\infty && \text{当 } C < 0 \\ f_2(C) &= 0 && \text{当 } 0 \leq C < 2 \\ f_2(C) &= 1 && \text{当 } 2 \leq C < 3 \\ f_2(C) &= \text{MAX}\{1, 0+2\}=2 && \text{当 } 3 \leq C < 5 \\ f_2(C) &= \text{MAX}\{1, 1+2\}=3 && \text{当 } C \geq 5 \end{aligned}$$

$$f_3(M) = \text{MAX}\{3, 1+5\}=6$$

利用定理给出的递推式，就可以编写实现 0-1 背包问题的计算机程序了，具体程序编写，留作练习。

需要指出的是，0-1 背包问题一般是指指数复杂度问题（NP-难题），但也已证明，0-1 背包问题属于 NP-完全问题（它可在多项式时间复杂度内解决，当且仅当所有其他 NP-完全问题可在多项式时间复杂度内解决）。

在“贪心法”中，我们要介绍部分背包问题，它的时间复杂度是多项式的。实验证明，许多情况下，对 0-1 背包问题使用部分背包问题的解法，可获得很好的近似解。因此，在精度要求不是绝对的情况下，对 0-1 背包问题可以使用部分背包的解法（贪心法）。

§ 12.3 贪心法

动态规划法是一种按多步决策、逐步递增局部解的方式求最优解的方法，局部解的扩大，需要按最优子结构进行，因此，使用该方法的前提是需要找出最优子结构，这项工作有时候很困难，那么，能否有一种直接的方法呢？回答是，对许多问题，可以使用一种特殊的方法——贪心(Greedy)法，就可免去找最优子结构。

但是，按贪心法找出的解，并不总是最优的（有时是最优解的很好的近似解），因此，一个问题能否使用贪心法，需要进行证明。

贪心法的典型例子有最小生成树、Dijkstra 单源最短路径、背包问题、多处理机调度问题、哈夫曼树问题等。

§ 12.3.1 基本思想

贪心法和动态规划法一样，也是一种基于逐步求解的方法，即一个完整的解通过多步决策（选择）逐步递增生成，前面每步生成的是部分解（子解），最后求出完整解。但与动态规划不同的是，不需要找出最优子结构，而是每次扩大（递增）子解时，不需要

依赖前面的子解，都是按当前最优的目标进行（不按最优子结构进行，只追求当前生成的子解在当前是最优的）。

通过一个简单例子说明。设某种货币有 5 元、2 元、1 元三种面值，现设某服务员要给顾客找回 18 元，要求找回的币的数目最少。显然，服务员可以先分别拿出 3 张 5 元的，然后后拿出 1 张 2 元的，再拿出 1 张 1 元的，共找出 5 张币。显然这是最优的方案（全局最优，找出的币的总数最少）。这就是个分步决策方法，所求的解是找回的若干张币。该问题的解决，共用了 6 个步骤（每步找回一张），每步都选择面值尽可能大的币（以期获得局部最优）。所以这是一种贪心法。

但是，这种找币的方法并不是总能获得全局最优解。例如，在上面的问题中，若增加一种 4 元面值，则按贪心法，找回的仍然是（3*5 元，1*2 元，1*1 元），但是，最好的方案显然是（2*5 元，2*4 元），共 4 张币，这个方案显然不是按贪心法得到的。这个例子表明，用贪心法是否能得到最优解，要经过证明才能确认。

§ 12.3.2 背包问题

我们先通过背包（Knapsack Problem）问题说明贪心法。

（一）基本思想

背包问题是这样的，已知一个容量(重量)大小为 M 的背包和 n 种物品，每种物品 i 的重量为 w_i ，其可以分开装包，假定将物品 i 的一部分 x_i （对总重量 w_i 的比例）放入背包，会得到 $p_i x_i$ 的效益，这里， $0 \leq x_i \leq 1$, $p_i > 0$ 。这里， p_i 为物品 i 的单位重量装包利润。现在的问题是，选择哪些物品，及它们分别选择多少来装包，才能使总的收益达到最大？由于每种物品都可以只装入一部分，所以这种背包问题也称**部分背包问题**，以与前面介绍的 0-1 背包问题区别。

背包问题是许多问题的抽象，例如，可以把货船看作是背包，那么装运多种货物，以求最大效益就是背包问题。当考虑优化问题时，库房或存储器安排，也可以看作是背包问题。

在背包问题中，约束条件为

$$\sum_{i=1}^n w_i x_i \leq M \quad 0 \leq x_i \leq 1, \quad p_i > 0, \quad w_i > 0, \quad 1 \leq i \leq n$$

衡量解的好坏的目标函数是：

$$\sum_{i=1}^n p_i x_i \quad p_i > 0, \quad 1 \leq i \leq n$$

显然，任何满足 $0 \leq x_i \leq 1$ 的向量 (x_1, x_2, \dots, x_n) 都是一个可能解。这样的可能解有无穷多个。我们的任务是，找出满足约束条件的使目标函数取最大值的 X 向量。对该问题，用穷举法或回溯法显然不行。

下面举例说明。给定 $n=3$, $M=40$, $(w_1, w_2, w_3)=(28, 15, 24)$, $(p_1, p_2, p_3)=(35, 25, 24)$, 下面给出 5 个可能解。下面我们将知道, 其中最后一个解就是全局最优解。

(x_1, x_2, x_3)	$\sum_{i=1}^n w_i x_i$	$\sum_{i=1}^n p_i x_i$
(1, 4/5, 0)	40	55
(1/2, 1, 1/3)	37	50.5
(1/28, 1, 1)	40	50.25
(5/7, 1, 5/24)	40	55
(25/28, 1, 0)	40	56.25

现在我们考虑如何求最优解。首先, 若 $\sum_{i=1}^n w_i \leq M$, 最优解是选择全部, 即取一切

$x_i=1$ 。因此若 $\sum_{i=1}^n w_i > M$, 则不可能取一切 $x_i=1$, 此时, 任何最优解都必须将背包装满。

这总是可以办到的, 因为每种物品都可以取任一大小的一部分装包。

我们考虑使用贪心法。先考察利润优先的贪心策略: 每次选择利润最大的物品装包, 当装到某一物品 k 时, 其由于包容量的限制不能全部装包, 则取适当的 $x_k < 1$, 用其将包塞满。这样, 就使目标函数增加最快。上面的第 1 种解就是按这种策略得到的, 显然, 得到的不是最优解, 这表明这种贪心策略不奏效。之所以这样, 是因为这种方法会使背包很快装满, 装包的物品的种类不多。

再考察另一种贪心策略---品种数目优先: 即尽可能使包容量消耗慢, 装入尽可能多种物品, 亦即按 w_i 的非递减次序选择品种。上面第 3 种解就是用这种策略求得的, 显然也不是最优的, 原因也很简单: 没有考虑利润因素。

对该例, 这两种贪心策略都不能取得最优解, 这就提示我们, 必须兼顾利润和种类个数这两个因素。因此, 我们采用这种贪心策略: 按物品的利润和重量之比 p_i/w_i 的非递增(先大后小)次序选择物品。上面的最后一个解就是用这种策略求得的。

(二) 算法实现

我们考虑背包问题的贪心法实现。贪心策略是按物品的利润和重量之比 p_i/w_i 的非递增次序选择物品。这个实现是很简单的。

```
int KnapsackGreedy(float *w, float *p, int n, float M, float *x)
{ //背包问题的贪心实现
  //w[], p[]--输入量, 一维数组, 分别存放各物品的重量与利润
  //n--输入量, 物品种数
  //M--输入量, 背包容量
```

//x[]--输出量，存放解向量，x[i]的值代表物品 i 应装包的比例，空间由调用者提供
//返回值：所选择装包的物品的种数，即 x[]中非 0 元素个数

```
float mM;  
int i,k;  
float *pw;  
int *pwi;
```

```
pw = new float[n];  
pwi = new int[n];
```

```
for (i=0; i<n; i++)  
    pw[i] = p[i]/w[i]; //将利润重量比存入数组 pw[]
```

```
//对 pw[]排序(降序)，pwi[i]存放"利润重量比"第 i 大的物品号（在 w[]中的下标）  
SortArr1(pw, n, pwi);
```

```
for (i=0; i<n; i++) x[i]=0; //初始化结果向量  
mM = M; //mM 表示当前包的剩余容量  
i=0;  
k=pwi[i]; //k 为当前"利润重量比"最大的物品的编号  
while (w[k] <= mM) //当背包的剩余容量足以容纳当前欲装包的物品时循环  
{ //将当前的"利润重量比"最大的物品（完全）装包  
    x[k] = 1; //表示完全装包  
    mM = mM-w[k];  
    i++;  
    k=pwi[i]; //k 为利润重量比下一个最大的物品的编号  
}
```

```
x[k] = mM/w[k]; //处理最后一装入的包，只能装入部分。
```

```
delete [] pw, pwi;  
return i;  
}
```

```
int SortArr1(float *a, int n, int *b)
```

```
{ //使用简单选择排序，对 a[]排降序，使 b[i]存放 a[]中第 i 大的元素的下标值
```

```

int i, j, k, s;
float x;

for (i=0; i<n; i++) b[i]=i;
for (i=0; i<n-1; i++)
{ //每次从 a[i]~a[n-1]中选择一个最大者, 将其对应下标存入 b[i]
  k=i;
  for (j=i+1; j<n; j++)
    if (a[k] < a[j]) k=j;
  x=a[k]; a[k]=a[i];a[i]=x;
  s = b[k]; b[k]=b[i]; b[i]=s;
}
return n;
}

```

(三) 背包问题的贪心法的正确性证明

定理 12-6: 对背包问题, 设背包容量为 M , 物品种数为 n , 物品 i 的重量为 w_i , 将物 i 装入背包可获的利润为 p_i , 现设物品编号按利润-重量比 p_i/w_i 的非升次序排好, 即 $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$, 则依次取物品 1、2、...、 k 装包 ($1 \leq k \leq n$), 前面的物品都完整装入, 最后一种 k 由于背包容量的限制只装入一部分 (将包补满为止), 则这种装包法产生的解是最优的 (获得最大的装包利润)。

证: 设这种装包法产生的解是 $X=(x_1, x_2, \dots, x_n)$, 则如果对所有 i , $x_i=1$, 则显然 X 是最优的。现设存在某个 j , $1 \leq j \leq n$, 有 $x_1=x_2=\dots=x_{j-1}=1$, 而 $0 < x_j < 1$, $x_{j+1}=x_{j+2}=\dots=x_n=0$ 。由

定理的条件知, 解 X 必须满足 $\sum_{i=1}^n w_i x_i = M$ 。

设 $Y=(y_1, y_2, \dots, y_n)$ 是问题的某个最优解, 其显然也满足 $\sum_{i=1}^n w_i y_i = M$ 。现在我们证明

$X=Y$ 。用反证法。如果 $X \neq Y$, 一定存在一个 k , $1 \leq k \leq n$, 对一切满足 $1 \leq i < k$ 的 i , 有 $x_i=y_i$, 但 $x_k \neq y_k$ 。这只有下列两种情况:

1) $x_k > y_k$ 。此时, 因为 $\sum_{i=1}^{k-1} w_i x_i = \sum_{i=1}^{k-1} w_i y_i$, 所以有 $\sum_{i=1}^k w_i x_i > \sum_{i=1}^k w_i y_i$ 。又 $\sum_{i=1}^n w_i x_i = M$,

所以 $\sum_{i=1}^k w_i x_i \leq M$, 从而, $\sum_{i=1}^k w_i y_i < M$, 且 $y_{k+1}, y_{k+2}, \dots, y_n$ 不全为 0, 否则在这种情况下

就没可能有 $\sum_{i=1}^n w_i y_i = M$.

现在, 由于 $y_k < x_k \leq 1$, 我们增大 y_k , 同时, 从 $y_{k+1}, y_{k+2}, \dots, y_n$ 中减小某些值, 以平衡对 y_k 的增加, 这样, 就产生了一个新的解 $Y' = (y'_1, y'_2, \dots, y'_n)$, 这里, 对 $1 \leq i < k$, 有

$y'_i = y_i$; $y'_k > y_k$; 对 $k < i \leq n$, $y'_i \leq y_i$, 且 $\sum_{i=1}^k w_i y'_i = M$. 这样, Y' 与 Y 相比, 利润较大的物品

的重量增加了, 而重量减少的是利润较小的, 这表明, 解 Y' 优于 Y , 这与 Y 是最优解的假设矛盾。

2) $x_k < y_k$. 此时, 必有 $x_k < 1$ (否则 $y_k > 1$), 此时, 如果 $x_k = 0$, 由解 X 的结构知, $\sum_{i=1}^{k-1} w_i x_i = M$

成立, 因此, $\sum_{i=1}^{k-1} w_i y_i = M$ 亦成立, 这表明, 物品 k 已不能装包, 即必有 $y_k = 0$, 这与 $0 = x_k < y_k$

矛盾. 如果 $0 < x_k < 1$, 则由解 X 的结构知, $\sum_{i=1}^k w_i x_i = M$ 成立. 再考虑到 $\sum_{i=1}^{k-1} w_i x_i = \sum_{i=1}^{k-1} w_i y_i$,

$x_k < y_k$, 可得 $\sum_{i=1}^k w_i y_i > \sum_{i=1}^k w_i x_i = M$, 这表明, Y 不是问题的解, 与假设矛盾。

综合这两种情况, 都表明 $X \neq Y$ 是不可能的, 定理得证。

该定理表明, 算法 KnapsackGreedy() 是正确的。

§ 12.3.3 Prim 最小生成树

求最小生成树是无向图的一种基本操作。由于对应的算法属于贪心法, 故我们在这里介绍。

(一) 基本概念

无向图的无回路的连通的生成图称为该无向图的**生成树**。这种树是无向无根树。一个无向图可以有多个生成树。如果给图的各边分别赋予一个权, 则边权之和取极值的生成树常常是具体问题所关心的对象。我们称边权之和最小的生成树为**最小生成树** (Minimum Spanning Tree), 它是图的一种最优化问题的抽象, 在诸如网络构造 (如何构

造连通 n 个站点的网络, 如通讯线路网、交通运输网、电路连接……) 之类的应用中占重要地位。

(二) 最小生成树构造基础

使用贪心法可以构造出有效的最小生成树算法, 有 Prim 算法、Kruskal 算法、Sollin 算法等, 它们都基于下面的最小生成树性质(MST 性质):

定理 12-7 (MST 性质): 设 $G=\langle V, E \rangle$ 是一个连通带边权的图, U 是 V 的一个真子集。如果 $(u, v) \in E$, $u \in U$, $v \in V-U$, 且在所有的这样的连接 U 和 $V-U$ 的边 (也称这种边为**割边**) 中, (u, v) 的权 $\text{cost}(u, v)$ 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。

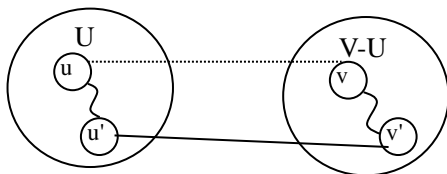


图 12-7 MST 证明

证: 用反证法。考虑 G 的任一最小生成树 T , 设其都不含边 (u, v) , 那么, 由于生成树是连通的 (即 u 到 v 有通路), 则如果将边 (u, v) 加到生成树中 T , 必然产生一个包含 (u, v) 的回路。在这条回路上, 应至少有三个结点 (两条边), 设 (u', v') 是不同于 (u, v) 的边, 这里, $u' \in U$, $v' \in V-U$ 。现在, 将边 (u', v') 删去, 则得到的图 T' 仍然是生成树 (连通性和无回路性没有因为 (u, v) 的加入和 (u', v') 的删除而遭破坏)。但是, 由已知条件知, $\text{cost}(u, v) < \text{cost}(u', v')$, 故 $\text{cost}(T') < \text{cost}(T)$, 则与 T 是最小生成树的假设矛盾。

该性质表明, 如果将一个连通无向图分割为互不相交的两部分, 则从两部分的割边中选出的权最小的边, 必定属于最小生成树。有这个性质的指导, 我们就可以设计出最小生成树的生成算法了。下面我们介绍基于这个性质的 Prim 算法和 Kruskal 算法。它们都是关于最小生成树的著名算法——前者于 1956 年由 Kruskal 发表, 后者于 1957 年由 Prim 发表。

(三) Prime 算法基本思想

贪心法的 Prim 算法是个逐步生成最小生成树的过程。它从空集合出发, 逐步生成一棵最小生成树, 它的每步选择都满足:

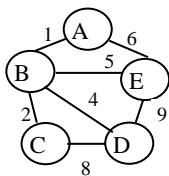
- (a) 当前生成的图是连通的 (连通原则)
- (b) 当前生成的图不含回路 (无回路原则)
- (c) 当前生成的图是最小生成树的一部分 (部分原则)

我们称该三条为三项原则。显然, 按此原则生成的图, 一定是最小生成树, 该原则也可做为用于证明此类算法的正确性的一个公理。Prim 算法就是遵循此原则设计的。设 $G=(V, E)$ 是无向图, 求它的最小生成树 $T=(U, E')$ 的 Prim 算法描述为:

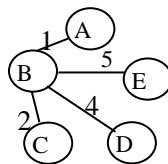
- [1] 从 V 中任取一结点放入 U ;
- [2] 在所有的端点分别在 $(V-U)$ 和 U 中的边 (割边) 中, 选一条权最小的加入 E' ;

- [3] 将所选边在 $(V-U)$ 中的结点（称为**图点**）放入 U （ U 中的结点称为**树点**）；
 [4] 重复步骤[2]~[3]，直到 U 中包含了所有结点为止。

这个算法的贪心性就体现在步骤 2 上。图 12-0 给出了一个最小生成树的 Prim 算法运行例子。在图中，步骤 i 表示在步骤 $(i-1)$ 的基础上从连结 $(V-U)$ 与 U 的边中选一条权最小者，并将其在 $(V-U)$ 中端点取出加入 U 后的结果，步骤 0 表示重复前（循环开始前）状态。



(a) 无向图



(b) 无向图(a)的一棵最小生成树

步骤	V-U(图点)	U(树点)	连接(V-U)与 U 的边(割边)	T
0	BCDE	A	AB, AE	空
1	CDE	AB	AE, BE, BD, BC	AB
2	DE	ABC	AE, BE, BD, CD	AB, BC
3	E	ABCD	AE, BE, DE	AB, BC, BD
4	空	ABCDE	空	AB, BC, BD, BE

(c) 无向图(a)的最小生成树生成过程

图 12-8 Prime 最小生成树算法示例(结点 a 为初始选择结点)

(四) Prim 算法正确性

前面指出，某算法只要满足最小生成树的三项原则，则可认为是正确的。这里我们就证明 Prim 算法满足前面提出的三项原则。

(a) 连通原则：这可用归纳法证明。E'初始之只含一个结点，第一步是从连接 $(V-U)$ 与 U 中的边中选一条边加入 E'，由于该边的一个端点肯定在 U （即 E'）中，故第一步后 E'中含一条边，是连通的。现假定在第 k 步前 E'是连通的，在第 $k+1$ 步也是从连接 $(V-U)$ 与 U 中的边中选一条边加入 E'，由于该边的一个端点肯定在 U (即 E')中显然，故加入所选边后 E'仍连通。

(b) 无回路原则：每次构成的部分树也是无回路的。因为若要构成回路，必要条件是某次新加入的边的两个端点均在 E'中，而我们每次是从连接 $(V-U)$ 与 U 的边中选边的，故构成回路的必要条件不成立。

(c) 部分原则：下面要证明 Prim 算法每步生成的图，均属最小生成树。Prim 算法每步都是从两个集合 U 和 $V-U$ 的割边中选最小边，这恰是我们前面给出的 MST 性质。

(五) Prim 算法的实现----基本流程

由上面的介绍可知，Prim 最小生成树的生成过程，是个逐步扩大 U 的过程：每次从连接 U 与 $(V-U)$ 的边中选择一条最小者，将其在 $(V-U)$ 中的结点加入到 U 中。因此，在算法实现中，必须记下当前的 U 与 $(V-U)$ 之间的边（割边）的信息，以备选择，我们称这种信息为候选集。这个过程可进一步地描述为：

```
从 V 中任选一结点加到 U 中；
设置初始候选集（割边集）；
while（U 中尚未包含全部结点）
{
    从当前候选集中选一条最小边；
    将所选最小边加入到 E' 中；
    将所选最小边的在 V-U 中结点加入到 U；
    调整候选集；
}
```

将从割边中选出的最小边的在 $V-U$ 中结点放入 U 后，与该点关联的那些边就变为新的割边，同时，该边也不是割边了。这就是说，候选集要改变（减与增），这种改变过程，我们称为调整候选集。

(六) Prim 算法实现----数据结构设计

显然，算法的实现，到了必须考虑数据结构设计的时候了。主要数据结构除了做为输入输出的图和生成树外，还有中间数据结构候选集。

1. 图

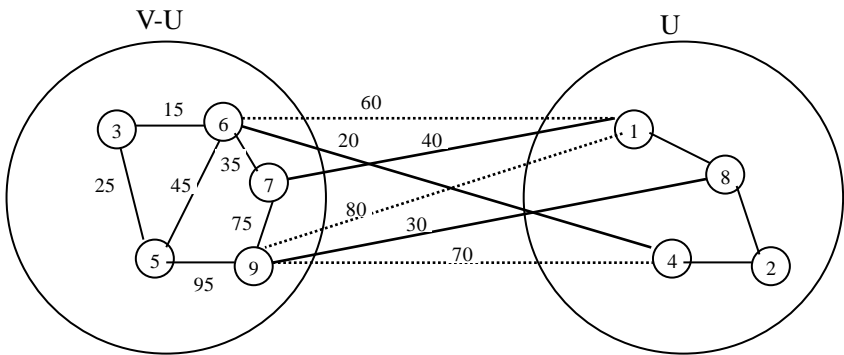
首先，要考虑输入量----图的表示。为了方便，我们这里用邻接矩阵（二维数组） $g[i][j]$ 表示图。当 (i, j) 是边时，令 $g[i][j]$ 等于边 (i, j) 的权，否则令 $g[i][j]$ 等于 ∞ （可用某个比所有边权都大的数表示）。

2. 最小生成树

最小生成树为生成的结果（输出数据），它是一种无根树。这里，为了方便，用边集表示生成树。边集本身用一个一维数组表示。令树的每条边对应于一个数组元素，数组元素为如下形式的三元组：

（边始点，边终点，边权）

这种用边集表示树的方式（集合本身用数组表示），虽然表达了树的全部信息，但父子关系是间接表达的，不易直接识别。不过，我们这里关心的是结点间的连通关系，所以这种表示法也是可以的。当然，我们也可以使用一般树的表示法。



(a) 最小生成树候选集图示

(1) 上图对应的候选集

结点 v	1	2	3	4	5	6	7	8	9
Close[v].vex	*	*	*	*	*	4	1	*	8
Close[v].lowCost	0	0	∞	0	∞	20	40	0	30

(2) 在选择一条割边，调整候选集后

Close[v].vex	*	*	6	*	6	*	6	*	8
Close[v].lowCost	0	0	15	0	45	0	35	0	30

(b) Close 示例 (*表示无关)

图 12-9 最小生成树候选集

3. 候选集 **Close**（注：C 是大写还是小写？ ==答：大写）

候选集，即割边信息集合，是一个中间数据结构，用于实现贪心选择（从割边中选择最小者）。为了使用方便，用一维数组表示。数组的每个元素代表一条割边，它的在 V-U 中的结点（图点）的编号为它在数组中的下标，而它的在 U 中结点（树点）的编号及权值则记录在对应的数组元素中。若与图点关联的割边不止一条，则 Close 中记录其中权值最小的那条。

图 12-0 是 Prim 算法循环了 3 次后的情况的一个例子。图中用虚线表示的割边是不需要保存的割边。图 12-0 中的图点 9，与它关联的割边有(9,1)、(9,8)与(9,4)。对于这种情况，由于我们是在割边集中选最小者，所以只需记录其中最小者（其它不考虑）。例如，对图点 9，只需记录(9,8)，而其他舍弃不记录。这样处理的主要目的是为了设计数据的

方便和提高效率。

close 数组元素的结构为:

(vex, lowCost)

其中的 vex 与 lowCost 分别表示割边的树点编号与割边的权值。Close 的具体定义为:

a) 对每个 $v \in V-U$ (图点)

close[v].lowCost = MIN{cost(v, u) | $u \in U$ } //cost(v, u)表示边(v,u)的权

close[v].vex = u //u 为满足上式的树点, 若 close[v].lowCostwe= ∞ , 则 u 无关

b) 对每个 $u \in U$ (树点)

close[u].lowCost=0

close[u].vex=无关

由于我们已定义: 当某两结点 v 与 u 之间无边时, 邻接矩阵对应元素为 ∞ , 故若某图点 v 无关联的割边时, close[v].lowCose= ∞ 。这种规定方便于求最小值。对于树点 u, 它的 close 是无意义的(因为我们以图点为中心表达割边的), 故 close[u].lowCost 置为 0 作为树点标志。之所以以 0 为标志, 是为了在算法实现中调整候选集的方便

(七) Prim 算法实现----程序设计

有了上面的讨论, Prim 算法的程序实现就容易了。首先, 我们定义最小生成树的元素(树边)类型:

```
struct TTreeEdge //最小生成树的边类型
{
    long v1, v2; //边起点与终点编号
    float weight; //边权
};
```

为了在函数参数中给出二维数组的列数, 定义常量:

```
int const CNST_NumGrphNodes=5;
```

该量的值在实际应用中应等于或大于实际结点个数(做为 MinSpanningTree 的中参数)。

下面是具体的最小生成树程序。这里假定图结点用 0 起的连续自然数编号。

```
long MinSpanningTree(int g[][CNST_NumGrphNodes], long n, TTreeEdge *minTree)
{//求具有 n 个结点的图 g(邻接矩阵)的最小生成树, 结果存于边数组 minTree
    //返回最小生成树的边数。返回值不等于 n-1 时, 表示出错
```

```
struct TCloseRec //候选集元素类型
{
    long vex;
    float lowCost;
};
```

```

long i, j, k;
float minW;
TCloseRec *close;

i=0;
close = new TCloseRec[n];

for (i=1; i<n; i++)
{
    close[i].vex = 0;   close[i].lowCost = g[i][0];
} //初始化候选集，将结点 0 作为始点加入 U
close[0].lowCost = 0; //为结点 0 做树点点标志

for (i=0; i<n-1; i++) //不断挑选最小割边，并调整候选集，每次为 minTree 生成一个元素
{
    for (k=1; k<n; k++) //在 close 中找第一个图点
        if (close[k].lowCost!=0) break;
    for (j=k+1; j<n; j++) //找最小割边
        if (close[j].lowCost!=0)
            if (close[j].lowCost < close[k].lowCost )
                k=j;

    //下面将所找出的最小割边 (k, close[k].vex) 加入树 minTree
    minTree[i].v1 = k;
    minTree[i].v2 = close[k].vex;
    minTree[i].weight = close[k].lowCost;
    close[k].lowCost=0; //表示 k 已成为树点

    //下面调整候选集
    for (j=1; j<n; j++) //检查每个邻接于 k 的(图)边
        if (close[j].lowCost > g[j][k] )
        {
            close[j].lowCost = g[j][k];
            close[j].vex = k;
        }
} //for (i=0; ...

```

```

delete [] close;
return i;
};

```

上面的程序中, 比较关键的是调整候选集。当某图点 k 变为树点时, 与它连接的边也要变为割边了。所以, 调整候选集主要是将这些新割边加入候选集。一个边变为割边后, 可能被保留在候选集, 也可能不保留。设结点 j 是与结点 k 关联的一个图点, 则当 k 新加入到 U 中时, 边 (k, j) 变为割边。若 j 原来没有关联割边时, 新割边 (k, j) 被保留。否则, 要看 (k, j) 是否小于其他关联于 j 的割边, 若小于, 则将 (j, k) 作为候选集中的割边, 否则舍弃不用。上面的调整候选集程序就是基于这种思想的。

程序中检查每个与 k 相连的图点 j , 比较边 (k, j) 权是否小于 j 的原割边权 (即 $close[j].lowCost$), 若是, 则将 (k, j) 作为 j 的新的割边。由于树点的 $lowCost$ 已为 0, 故语句 “if ($close[j].lowCost > g[j][k]$)” 相当于不考虑树点。

显然, 该算法的时间复杂度为 $O(n^2)$, 与边数无关。这里 n 为结点数。

§ 12.3.4 Kruskal 最小生成树算法

(一) 基本方法

Prim 算法是一种 “步步为营” 的算法, 每一步生成的结果均为最终结果的一部分。它虽然是每次从割边中选最小边, 但所选出的并不一定为所有的尚未选出的属于最小生成树的边中最小者, 换言之, Prim 算法不是按边权不减的次序生成最小生成树的。

Kruskal 算法是一种与 Prim 算法不同的算法, 它按边权不减的次序生成最小生成树, 即若某边是最小生成树中第 i 小的边, 则它在第 1~第 $(i-1)$ 小的边全部选出后才加入到部分结果中。这就是 Kruskal 算法的基本原则。显然, Kruskal 算法并不保证每步生成的结果是连通的 (从而部分结果可能不是树)。

设 $G=(V, E)$ 是一个有向图, T 是它的最小生成树的边集, 则由 G 生成 T 的 Kruskal 算法描述如下 (n 为 G 中结点个数):

```

置 T 为空集;
while (E 不空 && T 中边数<n)
{
    从 E 中挑选一条最小边(v, u);
    从 E 中删去(v,u);
    if ((v, u)加入 T 中后不产生回路)
        将(v, u)加入 T 中;
    else 舍弃(v, u);
}

```

这个算法的“贪心”所在是每次都从图的边集的剩余边中挑选权最小者。该算法的具体例子见图 12-0.

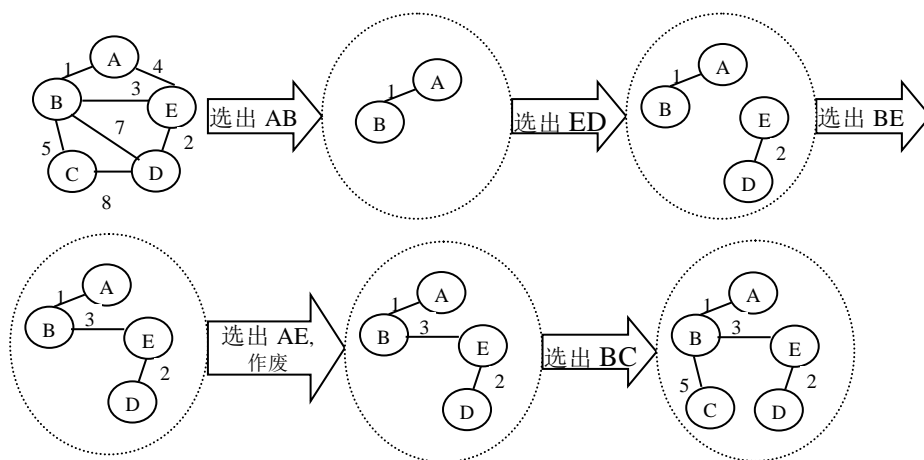


图 12-10 Kruskal 最小生成树算法示例

(二) Kruskal 算法正确性*

直观上讲，如果 Kruskal 算法最后生成的图是相通的，则其为最小生成树。下面我们证明 Kruskal 算法的正确性，为此，先给出几个关于生成树的定理。

定理 12-8 设 $G=(V, E)$ 是一个连通的无向图， $S=(V, T)$ 是关于 G 的一棵生成树，则有下列两点：(a) 对于 V 中的任意两个结点 v 和 w ，在生成树 S 中， v 和 w 之间只有一条唯一的路径；(b) 如果把 $E-T$ 的任何一条边加入 T ，则 S 中将产生一条回路。

证：对于(a)，若 S 中 v 和 w 间存在两条不同的路径，则必构成一个回路，这与 S 是生成树的假设矛盾。对于(b)，若将 E 中不属于 T 的边（记为 (a, b) ）加入 T ，则由于 S 是连通的，故 S 中 a 到 b 存在两条不同的路径，这与刚刚证明的(a)矛盾。

定义[生成森林]：若 $V_i \cap V_j = \Phi$ ， $i \neq j$ ， $1 \leq i, j \leq k$ ， $V_1 \cup V_2 \cup \dots \cup V_k = V$ ，且 (V_i, T_i) 都是一棵关于 G 的子生成树，这里， $T_i \subseteq E$ ，则称 $\{(V_1, T_1), (V_2, T_2), \dots, (V_k, T_k)\}$ 为关于 G 的一个生成森林。

定理 12-9 设 $G=(V, E)$ 是一个连通无向图， $F=\{(V_1, T_1), (V_2, T_2), \dots, (V_k, T_k)\}$ 是 G 的任一生成森林。令 $T = \bigcup_{i=1}^k T_i$ ， $k > 1$ ，并设 $e=(v, w)$ 是集合 $E-T$ 中的一条权最小的边，且存在 s ($1 \leq s \leq n$)，有 $v \in V_s$ ， $w \notin V_s$ ，那么，存在一棵包含 $T \cup \{e\}$ 的关于 G 的生成树，且其代价不大于任何包含 T 的关于 G 的生成树的代价。

证：首先，我们证明包含 $T \cup \{e\}$ 的生成树总是存在的。对 G 的任一不含 e 的元素个数大于 2 的生成森林 F ，我们总可以通过下面的方法将其连通为一个包含 e 的生成树：

➤ 在 $E-T$ 中找到边 $e=(v, w)$ ，将 e 加入到 v 和 w 所属的两个子树中（由已知条件，这两种子树是存在的），则这两棵子树就并为一棵子树，从而生成森林 F 的子树个数就减少 1。

➤ 如果 F 中只含一棵子树，则表明已连通为一个生成树，否则，在 F 中任找一个子树 (V_r, T_r) ，在 $E-T-\{e\}$ 中找一条某个端点在 V_r 中的边，设该边另一端点在 V_p 中，用其将 (V_r, T_r) 和 (V_p, T_p) 连通。这样的边是肯定存在的，否则， G 就不是连通的了。这样， F 中的子树的个数减少了 1，如果未减少到 1，则对剩余的子树同样处理，因此，总有一次， F 中子树个数变为 1，表示已将原所有子树连为一个生成树。

接着证明包含 $T \cup \{e\}$ 的关于 G 的生成树的代价，不大于任何包含 T 的关于 G 的其他生成树的代价。用反证法。设 $S'=(V, T')$ 是关于 G 的一棵最小生成树，它包含 T ，即 $T \subset T'$ ，但不包含 e ，即 $e \notin T'$ 。这里， T 和 $e=(v, w)$ 都是定理条件中提到的集合与边。根据定理 12-8，在 S' 中，结点 v 与 w 之间存在一条唯一的路径（注意， S' 包含 T ，而 T 是 G 的生成森林的边集，它包含了 G 的所有结点，故 v 与 w 也在 S' 中），其上必有一条不同于 e 的边 $e'=(v', w')$ ，使得 $v' \in V_s$ ， $w' \notin V_s$ ，（ V_s 为 v 所属的集合），从而， $e' \notin T$ ，即 $e \in E-T$ ，这表明， $\text{cost}(e) < \text{cost}(e')$ 。

现在，在 S' 中，将 $e=(v, w)$ 加入 T' ，并将 e' 从 T' 中删除（即删除 v' 和 w' ，及它们的各关联边），则构造一新图 $S=(V, T' \cup \{e\} - \{e'\})$ ，显然，由 e 和 e' 的关系知， S 是生成树，但由于 $\text{cost}(e) < \text{cost}(e')$ ，故 $\text{cost}(S) < \text{cost}(S')$ ，这与 S' 是最小生成树的假设矛盾。

该定理表明，对一个连通无向图 G ，如果已经产生了 G 的一个生成森林 F ， e 是 G 中的但不属于森林 F 的任何子树的边中的最小边，则包含 F 的最小生成树中，必包含 e 。

定理 12-10 Kruskal 算法产生的图是最小生成树。

证：在 Kruskal 算法中，在开始的时候，可以认为将所有的图结点加入了森林 T （或说 T 对应的森林），每个结点是一棵子树。在算法每步中，都是从 E 中选择一条最小边，将其加入到森林，使某两棵子树并在一起，同时将其从 E 中逻辑地删除，这样就保证每次从 E 中选择出的边，都属于当前森林。这样，Kruskal 算法的每步都满足定理 12-9 的条件，因此，该定理成立。

至于 Kruskal 算法的时间复杂度，由于每条边都要被选择一次，所以共要有 m 次的选择（Kruskal 选择循环）。若假定从 E 中选择最小边时，采用一般的搜索方法，则其复杂度为 $O(m)$ ，若忽略 Kruskal 选择循环中的其他动作，则 Kruskal 算法的时间复杂度为 $O(m^2)$ 。这里 m 为图的边数。如果我们在进入 Kruskal 选择循环前就将边按升序排列，则 Kruskal 选择循环内的时间花费为常量，因此总的时间复杂度为排序时间复杂度加 $O(m)$ ，若使用某种高级排序算法，则排序的时间复杂度为 $O(n \log_2 n)$ ，因此，总的时间复杂度为 $O(n \log_2 n)$ 。这个结果与图的结点数目无关，这是与 Prim 算法的不同之处。

(三) 算法实现

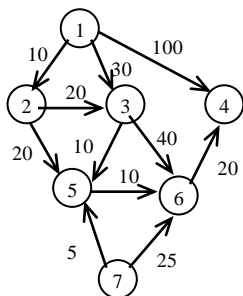
Kruskal 算法的实现，首先要涉及到生成森林，主要的操作有合并子树和检查是否构成回路。其中，检查某边(u,w)的加入是否构成回路，可以检查 u 和 w 是否在一个结点集（子树）中，将它们抽象到集合操作，就是集合的并 Union 和查找 Find（或成员操作 Member）。因此，使用集合类处理较为方便。当然，也可以直接用边的集合表示生成森林，而边集合用一维数组表示，数组的每个元素对应一条边，但这种表示不利于判别边的加入是否构成回路，速度慢。具体的程序实现，留作练习。

§ 12.3.5 单源最短路径

求最短路径是图的一种基本操作，其中，著名的单源最短路径算法是一种典型的贪心法，所以我们在这里介绍。

(一) 基本方法

求最短路径是一种重要的路径问题，它是许多具体问题的抽象，实践中有不少问题可归纳为最短路径问题。例如，若结点代表城市，边代表城市间交通费用，则选择一条从城市 A 到城市 B 的最佳路线就是个路径问题。再如，在基于 TCP/IP 的网络中，不同子网之间的 IP 主机的寻址问题（路由器中的路由问题），也一般是个最短路径问题（常常使用 Dijkstra 单源最短路径算法）。



(a) 一个有向图

始点	终点	最短路径	最短路径长度	长度次序
1	2	(1,2)	10	1
	3	(1,3)	30	2
	4	(1,2,5,6,4)	60	5
	5	(1,2,5)	30	3
	6	(1,2,5,6)	40	4
	7	无	∞	6

(b) 有向图(a)中 1 到其他各点的最短路径

图 12-11 有向图的单源最短路径

关于最短路径的第一个成熟的算法由荷兰籍计算机科学家 Dijkstra 给出，该算法用于求任一给定结点到其余结点的最短路径，是一种典型的贪心法，我们称之为 Dijkstra 单源最短路径算法。另有一个算法由 Floyd 提出，用于求每对结点间最短路径，属于动态规划法。我们这里介绍 Dijkstra 单源最短路径算法。

在连通图中, 从某点出发, 到达其他各点都分别有一条最短路径 (如果有多条, 则它们的长度都相等, 这里只算作一条)。设图共有 n 个不同的结点, 则从某一点出发达到其他各点的最短路径有 $(n-1)$ 条, 这 $(n-1)$ 条最短路径之间当然也存在大小关系。Dijkstra 单源最短路径的基本策略是, 按长度不减次序构造这 $(n-1)$ 条最短路径, 即先求出长度最小的一条最短路径, 然后求出长度第二小的最短路径, 最后求出长度最大的最短路径。图 12-0 给出了一个例子。对图 12-0 所示有向图, 若求结点 1 到其它各结点的最短路径, 则求得的次序为: 1-2, 1-3, 1-5, 1-6, 1-4, 1-7。

另外, 设立两种数据结构:

- **集合 S** : 存放当前已找出的各最短路径的终点。即若 $v \in S$, 则表示始点到 v 的最短路径已被求出。
- **路径数组 $dist[]$** : 它的含意为 (为了说话方便, 我们称 v 到 u 的中间只经过 S 中结点的路径为 v 到 u 的**特殊路径**):
 - 若 $i \in S$, 则 $dist[i]$ 等于始点 v_0 到 i 的距离
 - 若 i 不属于 S , 则 $dist[i]$ 等于始点 v_0 到 i 的最短特殊路径的长度。若 v_0 到 i 无特殊路径, 则视 $dist[i]$ 为 ∞ 。

这两个数据结构是 Dijkstra 单源最短路径算法的关键。其中 $dist$ 充当候选集。

Dijkstra 单源最短路径算法的基本思想是, 初始时, 将始点 v_0 放入 S , 并对每个非始点 i , 令 $dist[i]$ 为边 $\langle v_0, i \rangle$ 的权, 边 $\langle v_0, i \rangle$ 不存在时, 令 $dist[i]$ 为 ∞ 。然后, 进入一个多步 (循环) 过程, 每步都是从 $dist$ 中的未被选择过的元素中 (这些元素对应的结点不属于 S) 选一个值最小的元素, 比如 $dist[k]$, 将 k 加入 S , 然后调整候选集 $dist$ 。这个过程描述如下 (设图为 $G=(V, E)$):

```
置  $S$  为空;
将始点  $v_0$  加入  $S$ ;
初始化  $dist$ , 即对每个非源点  $i$ , 令  $dist[i]$  为边  $\langle v_0, i \rangle$  的权。边  $\langle v_0, i \rangle$  不存在时, 令  $dist[i]$  为  $\infty$ ;
while ( $S$  中没有包含全部结点)
{
  选择  $V-S$  结点  $u$ , 使  $dist[u] = \text{MIN}\{ dist[v] \mid v \in V-S \}$ ;
  将  $u$  加入  $S$ ;
  //下面调整候选集  $dist$ , 使它重新满足定义
  for ( $V-S$  中每个结点  $v$ )
     $dist[v] = \text{MIN}\{ dist[v], dist[u] + \text{cost}(u, v) \}$ 
}
```

其中, 调整候选集 $dist$ 是个重要问题。由于 $dist[i]$ 等于始点 v_0 到 i 的中间只经过 S 中元素且长度最小的路径的长度, 所以, 当 S 中增加了新元素时, $dist$ 中某些元素可能

发生变化，所以应该及时调整。对调整候选集的理解，请参阅图 12-0。图中假定某步选择了 S 外的结点 u ，那么， u 要被拿到 S 中，这样，对从 u 射出的边 $\langle u, w1 \rangle$ 和 $\langle u, w2 \rangle$ 对应的结点 $w1$ 和 $w2$ ， v_0 到它们的特殊路径就分别多了一种选择。此时，应如下调整 dist ：

对 $w1$ ，如果 $\text{dist}[w1] > \text{dist}[u] + \text{cost}(u, w1)$ ，则应令 $\text{dist}[w1] = \text{dist}[u] + \text{cost}(u, w1)$

对 $w2$ ，如果 $\text{dist}[w2] > \text{dist}[u] + \text{cost}(u, w2)$ ，则应令 $\text{dist}[w2] = \text{dist}[u] + \text{cost}(u, w2)$

这正是上面的算法的做法。

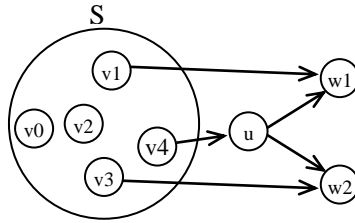


图 12-12 Dijkstra 算法调整候选集

该算法的贪心性体现在每步都从 dist 中选择最小者。

(二) Dijkstra 算法的正确性

先考虑下面的定理。

定理 12-11 设 S 和 dist 如前定义，则有以下两点：(a) 若对当前的 S 有 $\text{dist}[u] = \min\{\text{dist}[v] \mid v \in V - S\}$ ，则 $\text{dist}[u]$ 就是始点 v_0 出发的到 u 的最短路径。(b) 算法每步所确定的各 $\text{dist}[i]$ 都满足 dist 的定义，即算法每步所确定的 $\text{dist}[i]$ ，确实是当前从始出发，中间只经过 S 中结点而到达 i 的最短路径的长度。

证：先证(a)，用反证法。若 v_0 到 u 的路径中，存在一条比 v_0 到 u 的特殊路径 $\text{dist}[u]$ 更短的路径 $p(v_0, u)$ 。由 $\text{dist}[u]$ 的定义知， $p(v_0, u)$ 上至少有某点 w 不在 S 中（否则，由于 $\text{dist}[u]$ 是所有的 v_0 到 u 中间只经过 S 中结点的路径中最小者，所以更短的 $p(v_0, u)$ 就不存在了），那么 $p(v_0, u)$ 上包含了一条从 v_0 到 w 的路径 $p(v_0, w)$ ，有 $p(v_0, u) = p(v_0, w) + p(w, u)$ ，又假设了 $p(v_0, u) < \text{dist}[u]$ ，所以， $p(v_0, w) + p(w, u) < \text{dist}[u]$ ，但总有 $p(w, u) \geq 0$ ，所以有 $p(v_0, w) < \text{dist}[u]$ ，又 $\text{dist}[w] < p(v_0, w)$ ，故 $\text{dist}[w] < \text{dist}[u]$ ，这与 u 是当前最小的特殊路径的假设矛盾。

再证(b)。对 S 的基数（元素个数） m 进行归纳。当 $m=1$ 时， S 中只有始点 v_0 ，所以，对任何非始点结点 v ，若 $\langle v_0, v \rangle$ 存在，从 v_0 到 v 的中间只经过 S 中结点的路径就是边 $\langle v_0, v \rangle$ ，不存在其他这样的路径；若 $\langle v_0, v \rangle$ 不存在，则也不存在这种路径。算法初始时($m=1$ 时)，对每个非始点 i ，令 $\text{dist}[i]$ 为边 $\langle v_0, i \rangle$ 的权，边 $\langle v_0, i \rangle$ 不存在时，令 $\text{dist}[i]$

为 ∞ ，这显然满足 dist 的定义。

现设 $m=k$ 时结论成立，考察 $m=k+1$ 的情况，此时， S 中增加了一个新结点 u 。对任一 S 外结点 v ，源点 v_0 到 v 的特殊路径中，由于 u 的加入，可能会增加。新增加的特殊路径只可能有下列两种：

① u 到 v 路段直接由边 $\langle u, v \rangle$ 构成

② u 到 v 路段中含有 S 中结点

下面说明，特殊路径②肯定大于原 $\text{dist}[v]$ 。设 S 中结点 x 连接结点 v ，则 u 加入 S 后，特殊路径②的长度为 $\text{dist}_2[v] = \text{dist}[u] + p(u, x) + p(x, v)$ 。由于 u 是最新加入 S 的，故有 $\text{dist}[x] < \text{dist}[u]$ ，从而

$$\text{dist}_2[v] > \text{dist}[x] + p(u, x) + p(x, v) > \text{dist}[x] + p(x, v) \geq \text{dist}[v]$$

这里，有 $\text{dist}[x] + p(x, v) \geq \text{dist}[v]$ ，是因为已假设 $\text{dist}[v]$ 是 v_0 到 v 的最短的特殊路径，应该有 $\text{dist}[v] = \text{MIN}\{\text{dist}[x] + p(x, v) \mid \text{对所有的 } \langle x, v \rangle\}$ 。参见图 12-0。

特殊路径②肯定大于原 $\text{dist}[v]$ ，所以，在调整 dist 时可以不考虑路径②，而只考虑路径①。在算法中，对每个不属于 S 的结点 v ，都置 $\text{dist}[v] = \text{MIN}\{\text{dist}[v], \text{dist}[u] + \text{cost}(u, v)\}$ ，这显然考虑了路径①的影响。

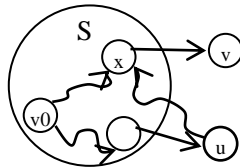


图 12-13 路径①和②对 dist 的影响

现在，(a)和(b)两点已得证，而 Dijkstra 算法完全符合(a)（当然也符合(b)），所以它是正确的。

(三) Dijkstra 算法的实现

为使 Dijkstra 算法化为具体的程序，需确定所涉及到的数据结构。这里分述于下。

(1) 图

输入量，为待求最短路径的图。根据 Dijkstra 算法的特点，采用邻接矩阵，设其为 $g[i][j]$ 。当 i 到 j 有边时，令 $g[i][j]$ 为边 $\langle i, j \rangle$ 的权，否则令 $g[i][j]$ 为 ∞ 。

(2) 路径与候选集 dist

dist 是该算法的关键数据结构，在算法进行中，充当候选集，在算法结束后存放各最短路径的长度。为了能同时得到最短路径的构成（即路径上的结点序列），这里对 dist 作适当的扩充，使得能从它得到各最短路径的结点序列。为此，我们先考查一下最短路的构成规律。事实上，源点 v_0 到任一点 u 的最短路径构成形式为下列两种情形之一：

(v_0, u) 或

$(v_0, u_1, \dots, u_k, u) \quad (k \geq 1)$

其中，路径 (v_0, u_1, \dots, u_m) 为 v_0 到 u_m 的最短路径 $(1 \leq m \leq k)$ ，亦即，最短路径上结点序列的任一最左子序列均为某结点的最短路径结点序列（否则该路径就不是最短路径了）。

这个性质是最短路径的最优子结构性质。为说话方便，我们称 (v_0, u_1, \dots, u_k) 或 (v_0) 为 v_0 到 u 的最短路径的**前趋路径**。最短路径的此性质启发我们，对任一结点 u ，只需设立它的前趋路径的指示器就可推出源点到它的最短路径结点序列。据此，我们为 dist 的元素增设一个 pre 字段，令它存放对应结点的前趋路径的终点编号，即 $\text{dist}[i].\text{pre}$ 的值为源点到 i 的最短路径的前趋路径的终点编号。该字段相当于一个链指针，顺着它搜索的结果是最短路径的结点的逆序。对于图 12-0 所示的最短路径情况，其 dist 如图 12-1 所示。

结点 i	1	2	3	4	5	6	7
$\text{dist}[i].\text{len}$	0	10	30	60	30	40	∞
$\text{dist}[i].\text{pre}$	-1	1	1	6	2	5	-1

图 12-14 路径数组 dist 示例

(3) 集合 S

集合 S 为中间量，用一维数组表示，其元素含意为：

若 $i \notin S$: $S[i]=0$

若 $i \in S$: $S[i]=1$

如果图 g 邻接矩阵的对角线元素不使用，则也可用对角线元素 $g[i][i]$ 充当 $S[i]$ 。

算法的具体的程序实现如下：

```

struct TPath //定义路径数组的元素类型
{
    float len;
    long pre;
};
const long CNST_NumGrphNodes2=10;
long DijkstraPath(float g[][CNST_NumGrphNodes2], long n, long v0,
                  float maxWeight, TPath *dist)
{ //Dijkstra单源最短路径算法
    //g[][]--输入量，表示图的邻接矩阵，无边时置为最大权maxWeight
    //n--输入量，表示图结点个数，其可以小于CNST_NumGrphNodes2
    //v0--源点编号
    //maxWeight--输入量，表示最大边权（比所有边权都大）
    //dist--输出量，一维数组，dist[i]存放v0到i的最短路径的长度及前趋路径的终点编号
    //dist[i].len也用作中间量，表示候选集
    //返回值：dist中元素个数

    char *S;

```

```
long i, j, k;
float minW;

S = new char[n]; //为集合S分配空间

for (i=0; i<n; i++) //初始化
{
    S[i] = 0; //初始化S为空
    dist[i].len = g[v0][i];
    if (dist[i].len !=maxWeight) dist[i].pre = v0;
    else dist[i].pre = -1; //表示不存在
}

S[v0] = 1; //将源点v0放入s

for (i=0; i<n-1; i++) //每次求出一条最短路径
{
    for (k=0; k<n; k++) //在S中找第一个元素不属于S的元素,找到后其编号在k中
        if (!S[k]) break;
    for (j=k+1; j<n; j++) //求下条最短路径
        if (!S[j] )
            if (dist[j].len < dist[k].len) k=j;
    S[k] = 1; //将k加入S

    for (j=1; j<n; j++) //调整候选集
        if (!S[j])
            if (dist[k].len + g[k][j] < dist[j].len )
            {
                dist[j].len = dist[k].len+g[k][j];
                dist[j].pre = k; //将k的前驱路径改为k
            }
} //for (i=1; ...)

delete [] S;
return n-1;
}; // DijkstraPath
```

此算法的时间主要花在两个嵌套的for循环上，它们的循环次数均可认为是 n （结点

个数)，故算法的时间复杂度为 $O(n^2)$ ，另外，此算法对单源多终点与单源单终点有着同样的时间复杂度，但对后者，常数因子会小一些。

表 12-2给出Dijkstra算法对图 12-0所示图求结点1到其余各点的最短路径时，候选集dist的变化。

表 12-2 Dijkstra 算法对图 12-0 的执行过程

结点 i 步骤		1	2	3	4	5	6	7
0	s[i]	1	0	0	0	0	0	0
	dist[i].len	0	10	30	100	∞	∞	∞
	dis[i].pre	-1	1	1	1	-1	-1	-1
1	s[i]	1	1	0	0	0	0	0
	dist[i].len	0	10	30	100	30	∞	∞
	dist[i].pre	-1	1	1	1	2	-1	-1
2	s[i]	1	1	1	0	0	0	0
	dist[i].len	0	10	30	100	30	70	∞
	dist[i].pre	-1	1	1	1	2	3	-1
3	s[i]	1	1	1	0	1	0	0
	dist[i].len	0	10	30	100	30	40	∞
	dist[i].pre	-1	1	1	1	2	5	-1
4	s[i]	1	1	1	0	1	1	0
	dist[i].len	0	10	30	60	30	40	∞
	dist[i].pre	-1	1	1	6	2	5	-1
5	s[i]	1	1	1	1	1	1	0
	dist[i].len	0	10	30	60	30	40	∞
	dist[i].pre	-1	1	1	6	2	5	-1
6	s[i]	1	1	1	1	1	1	1
	dist[i].len	0	10	30	60	30	40	∞
	dist[i].pre	-1	1	1	6	2	5	-1

注：标有删除线者和蓝色/粗体者，表示已进入 s 集合，其中蓝色/粗体者当前最小者

§ 12.3.6 贪心法要素总结

贪心法是从初态出发，逐步递增扩大解，最后扩大为完整解。每次扩大时，都要在

若干方案中选择一定的扩大方式，选择的依据是当前状态下某种意义的最优选择。这种选择与其他步骤（其他子解）无关，这也是与多态规划法的重要区别！这是一种只顾当前“利益”的方法，即保证当前是最优的，这就是“贪心”叫法的来历。显然，这种只顾当前利益的做法，不一定总能获得最好全局利益。因此，使用贪心法要特别注意。

与动态规划法相比，贪心法的最大方便之处是不需要找出最优子结构（不需要递推关系），而只需制定贪心策略。下面对贪心法的使用要点进行总结。

a) 明确目标函数和约束条件。目标函数和约束条件一般是由问题本身的性质决定的，但为了方便算法设计，要表述为适当的形式。

b) 制定部分解结构（分步决策方案）。首先要确定如何将待解问题分解为若干步骤进行，也就是确定部分解的结构。例如，在 Prim 最小生成树中，部分解是最终最小生成树的子树。在 Kruskal 最小生成树算法中，部分解是森林。在背包问题中，部分解是实数序列 (x_1, x_2, \dots, x_k) ，它对应的物品编号序列为 (i_1, i_2, \dots, i_k) ，使得物品的利润与重量比值序

列 $(\frac{p_{i_1}}{w_{i_1}}, \frac{p_{i_2}}{w_{i_2}}, \dots, \frac{p_{i_k}}{w_{i_k}})$ 为不升序列。

c) 确定贪心策略。所谓贪心策略（贪心选择），是指扩大部分解的方法，一般涉及极值选择。其一般做法是，按解的结构，从可选集合（候选集）中选择适当成分加到当前解上，构成新的当前部分解。这里的选择目标是，使新构成的部分解，在当前范围内达到极值（极大/小，在目标函数的意义）。例如，在背包问题中，所用贪心策略是按物品的利润和重量之比 p_i/w_i 的非递减次序选择物品。

d) 确定候选集。为了方便或加速贪心选择，一般需要明确制定一个选择范围，使它包括可能被选择的成分（尽量不包括不可能被选择的成分），而且方便于极值选择，这个范围就叫候选集。候选集一般为了适合贪心策略，也要满足一定的条件。例如，Prim 最小生成树算法中，Close 为候选集，它是割边的集合，即它中的边连接两个集合，一个是当前部分生成树结点集合，另一个是其余结点的集合。在算法实现时，候选集的数据结构设计也是很关键的。

e) 调整候选集。当从候选集中选择一个成份，将其拿出加到当前部分解上后，候选集一般要发生变化，可能有新的候选成分出现（或原成份发生变化），要及时加到候选集中，这种操作称为调整候选集。

f) 正确性证明。一些看上去很可能取得最优解的贪心策略，不一定真正能取得最优解。前面已分析过背包问题的其他几种贪心选择，也分析过钱币问题的贪心选择，它们都表明了这种情况。因此，对所制定的贪心策略，一般要进行严格的证明，才能确定它是否可用。

本章小结

本章介绍了四类算法设计方法----回溯法（含限界剪枝法）、动态规划法、贪心法。

结合前面已介绍的分治法、递归法、穷举法（试探法）、递推法、迭代法、逐步求精法等方法和策略，构成了一个较为完善的针对算法（求解问题的过程）设计的方法与策略体系。

在这几种算法设计方法中，用到几个概念：约束条件/函数、目标函数。在针对具体问题下，解或子解应该满足的条件称为约束条件，约束条件的表达式称为约束函数；解的“好坏”衡量标准的表达式就称为目标函数。

回溯法（含限界剪枝法）、动态规划法、贪心法都基于分步决策的思想，即将一个问题的解决分若干步骤逐步进行，每个步骤都产生一个部分结果（子解）。随着步骤的进行，子解被逐步扩大，最后形成完整解。

回溯法在扩大子解中，采用“试探-纠错”的策略，即扩大子解中，并不保证所作的扩大对最终解是正确的。在任一步的扩大中，若发现不能继续扩大（由于前面的选择导致当前不能继续扩大），则作废上步所作的扩大（回溯），尝试新的选择（新的扩大）。若无新选择可用，则继续回溯。回溯法在扩大解的时候检查新扩大后的子解是否满足问题的条件（约束条件），只有满足约束条件的子解才被保留，然后转向下一步骤。所以，回溯法在中间步骤上产生的子解是可能解，它在当前情况下满足条件，但在解进一步扩大后是否满足，还不一定，若在后来的（子解的）扩大中发现不满足最终条件，则舍弃并重新选择。在某一步骤上扩大子解时，一般是在各种可行的方案中穷举，所以，回溯法一般是适合可列（穷）举的情况。当然，如果有某种启发信息能在子解的扩大中起指导作用，使当前的选择尽可能减少后面的回溯，则效率会提高许多，限界剪枝法就是主要从这点出发对回溯法进行改进的。著名的例子有启发式搜索算法 A^* 。

回溯法一般需要解的约束条件具有完备性，即若任一子解不满足约束条件，则所有包含它的解也都不满足约束条件。显然，如果约束条件不满足完备性，则有可能丢弃正确解。

贪心法其实与回溯法很类似，它们的主要差别在于对子解的扩大的方式上。对回溯法，扩大子解时，只要扩大后是可行的子解（满足约束条件的子解），就接受其，转到下一步骤进行新的扩大，而不论这种扩大对以后是否有不良影响，即有可能在以后发现这种扩大是不正确的，因此可能需要撤销重来（回溯）。对贪心法，子解的扩大，是按精心设计的方式进行的，每次的扩大都是正确的，因而不需要回溯。此外，贪心法关于子解的扩大，是朝当前最优的方向进行，所以称为贪心法。但是，一般而言，当前最优，不一定是全局最优，所以，对具体的贪心策略（子解的扩大方法），需要对其有效性进行证明。

动态规划法与贪心法有些类似。它们的重要差别也是在子解的扩大方式上。对动态规划法，子解的扩大一般是根据前面得出的子解进行---递推。所以，一般需要找出递推式或递推方式。既然是递推，当前子解的得出与后面的子解（扩大了子解）无关，也与前面子解的得出过程无关，而只与结果有关。相比起来，贪心法不需要递推，也不需要根据前面的子解进行扩大。

如何找出有效的递推方法是动态规划的关键。并不是每个问题都存在有效的子解递推法。如果问题满足最优性原理（存在最优子结构），则可以肯定存在有效的递推法。

习 题

1. 编写迷宫问题的求全解程序。
2. 修改迷宫程序（求单解），使其能逼真输出老鼠走迷宫的过程（轨迹）。
3. 编写一个迷宫生成程序，使操作员能在屏幕上象画画儿一样，用鼠标器画迷宫路径。
4. 用递归法实现迷宫问题。
5. 在稳定婚姻问题中，若是求单解，则程序要如何改动？
6. 在稳定婚姻问题中，如何评价解的优劣？如何求出平均最优解？如何求出针对任一指定男子的最优解？
7. 实现 n 皇后算法，分别用递归和非递归方法。
8. 编写程序，实现从字母表{1,2,3}中选取字符，生成一个有 n 个字符的序列，使得其中没有任何相邻的两个子序列相等。例如，若 $n=5$ ，则“12321”是合格的序列，而“12323”和“12123”都不合格。
9. 设 $X="28136"$, $Y="18218832"$ ，请按自底上的递推方式，求它们的最长公共子序列的长度,要求写出每步的计算结果（即给出 $lcsL[][]$ 的各个元素的值）。
10. 用递归方法实现求最长公共子序列的长度 $LCSLen()$ 。
11. 用非递归方法实现求最长公共子序列的算法 $LCSOut()$ 。
12. 写一个在算法中只保留上一步和当前步的计算结果的求最长公共子序列（及长度）的程序。
13. 改造流水线调度算法，为其增加计算最佳作业完工时间的功能。
14. 编写程序，实现 Kruskal 最小生成树算法。
15. 编写程序，实现 0-1 背包问题。