第5章

马踏棋盘与道路规划

本章将使用贪心算法解决一些问题，如马踏棋盘和道路规划的问题。马踏棋盘问题（又称骑士周游或骑士漫游问题）是算法设计的经典问题之一。国际象棋的棋盘为8×8的方格棋盘，现将“马”放在任意指定的方格中，按照“马”走棋的规则将“马”进行移动。要求每个方格只能进入一次，最终使得“马”走遍棋盘64个方格。程序输出一个8×8的矩阵，并用数字1～64来标注马的移动。

道路规划问题是指有一张城市地图，图中的顶点为城市，无向边代表两个城市间的连通关系，边上的权为在这两个城市之间修建高速公路的造价，研究后发现，这个地图有一个特点，即任一对城市都是连通的。现在的问题是，要修建若干高速公路把所有城市联系起来，问如何设计可使得工程的总造价最少？

5.1 贪心算法

什么是贪心算法？我们以世界各地成千上万的收银员所要面对的找零问题作为本章的开始：用当地面额为$d_1>d_2>\cdots>d_m$的最少数量的硬币找出金额为n的零钱。例如，在美国广泛使用的硬币的面额是：$d_1=25$（二角五分硬币）、$d_2=10$（一角硬币）、$d_3=5$（五分硬币）和$d_4=1$（一分硬币），我们如何用这些种面额的硬币给出48美分的找零？如果我们给出的答案是1个二角五分硬币、2个一角硬币和3个一分硬币，就遵循了一种从当前几种可能的选择中确定一个最佳选择序列的逻辑策略。

的确，在第一步中，可以给出4种面额中的任意一个硬币。贪心的想法导致我们给出1个二角五分硬币，因为它把剩余金额降到最低，也就是23美分。在第二步中，还有同样面额的硬币，但不能再给出1个二角五分硬币，因为超出了需要的金额总数，所以在这步中的最佳选择是1个一角硬币，把余额降到了13美分。再给出1个一角硬币，还差3美分就用3个一分硬币给掉。

对于找零问题的这个实例，这个解是不是最优的呢？它的确是最优的。实际上，可以证明，就这些硬币的面额来说，对于所有的正整数金额，贪心算法都会输出一个最优解。与此同时，也可以给出一个“怪异”的硬币面额的例子，例如，$d_1=7$，$d_2=5$，$d_3=1$，这样的面额对于某些金额来说，贪心算法无法给出一个最优解。

本章开头段落中对找零问题应用的方法称为贪心算法。尽管实际上这个方法只能用于最优问题，但计算机科学家把它当作一种通用的设计技术。贪心算法通过一系列步骤来构

造问题的解，每一步对目前构造的部分解做一个扩展，直到获得问题的完整解为止。这个技术的核心是所做的每一步选择都须满足以下条件。

- 可行的：即它必须满足问题的约束。
- 局部最优：它是当前步中所有可行选择中最佳的局部选择。
- 不可取消：即选择一旦做出，在算法的后面步骤就无法改变了。

这些要求对这种技术的名称做出了解释：在每一步中，它要求“贪心”地选择最佳操作，并希望通过一系列局部的最优选择，能够产生一个整个问题的（全局的）最优解。从算法的角度来看，这个问题应该是贪心算法是否是有效的。就像我们将会看到的，的确存在某类型问题，一系列局部的最优选择对于它们的每一个实例都能够产生一个最优解。然而，还有一些问题并不是这种情况。对于这样的问题，如果我们关心的是，或者说我们能够满足于一个近似解，贪心算法仍然是有价值的。

作为一种规则，贪心算法看上去既诱人又简单。尽管看上去该算法并不复杂，但在这种技术背后有着相当复杂的理论，它是基于一种称为“拟阵”的抽象组合结构。有兴趣的读者可以查看相关的资料。我们先用贪心算法解决一个简单的问题作为热身。

5.2 活动安排问题

【例 5-1】 设有 n 个活动的集合 $E=\{1,2,\cdots,n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i，且 $s_i<f_i$。如果选择了活动 i，则它在半开时间区间 $[s_i,f_i)$ 内占用资源。若区间 $[s_i,f_i)$ 与区间 $[s_j,f_j)$ 不相交，则称活动 i 与活动 j 是相容的。也就是说，当 $s_i\geqslant f_j$ 或 $s_j\geqslant f_i$ 时，活动 i 与活动 j 相容。活动安排问题就是要在所给的活动集合中选出最大的相容活动子集合。

求解思路：将活动按照结束时间进行从小到大排序。然后用 i 代表第 i 个活动，$s[i]$ 代表第 i 个活动开始时间，$f[i]$ 代表第 i 个活动的结束时间。按照从小到大排序，挑选出结束时间尽量早的活动，并且满足后一个活动的起始时间晚于前一个活动的结束时间，全部找出这些活动就是最大的相容活动子集合。事实上，系统一次检查活动 i 是否与当前已选择的所有活动相容。若相容，活动 i 加入已选择活动的集合中；否则，不选择活动 i，而继续计算下一活动与集合 A 中活动的相容性。若活动 i 与之相容，则 i 成为最近加入集合 A 的活动，并取代活动 j 的位置。

下面给出求解活动安排问题的贪心算法，各活动的起始时间和结束时间存储于数组 startTime 和 endTime 中，且按结束时间的非减序排列。具体代码如下。

```
#include <iostream>
using namespace std;

void GreedyChoose(int len, int * startTime, int * endTime, bool * mark);

int main()
{
    //每个活动开始的时间
```

```
    int startTime[11] = { 1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12 };
    //每个活动结束的时间,并假设结束时间已经排序
    //如果没有,应该在算法开始时按升序排序
    int endTime[11] = { 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 };
    //选择标志,如果选择某个活动,则数组 mark 对应的位置为 true;否则为 false
    bool mark[11] = { false};

    //求解问题
    GreedyChoose(11, startTime, endTime, mark);

    //输出结果
    cout << "NO.\t" << "sTime\t" << "eTime\t" << endl;
    for(int i = 0; i < 11; i++)
    {
        if(mark[i])
            cout << i+1 << "\t" << startTime[i] << "\t" << endTime[i] << endl;
    }
    return 0;
}

void GreedyChoose(int len, int * startTime, int * endTime, bool * mark)
{
    //第一个活动是一定可以安排的
    mark[0] = true;

    int j = 0;
    for(int i = 1; i < len; ++i)
    {
        //从第二个活动开始,寻找开始时间大于前一个结束时间的
        if(startTime[i] >= endTime[j])
        {
            mark[i] = true;          //此活动可以安排
            j = i;                   //记住最后一个安排的活动
        }
    }
}
```

测试数据如表 5-1 所示。

表 5-1　测试数据

活动序号	1	2	3	4	5	6	7	8	9	10	11
开始时间	1	3	0	5	3	5	6	8	8	2	12
结束时间	4	5	6	7	8	9	10	11	12	13	14

【运行结果】

```
NO.     sTime   eTime
1       1       4
4       5       7
8       8       11
11      12      14
```

由于输入的活动以其完成时间的非减序排列，所以贪心算法每次总是选择具有最早完成时间的相容活动加入集合中。直观上，按这种方法选择相容活动为未安排活动留下尽可能多的时间。也就是说，该算法的贪心选择的意义是使剩余的可安排时间段极大化，以便安排尽可能多的相容活动。

若被检查的活动 i 的开始时间 S_i 小于最近选择的活动 j 的结束时间 f_i，则不选择活动 i，否则选择活动 i 加入集合中。贪心算法并不总能求得问题的整体最优解。但对于活动安排问题，贪心算法却总能求得整体最优解，即它最终所确定的相容活动集合的规模最大。这个结论可以用数学归纳法证明。

证明如下：设 $E=\{0,1,2,\cdots,n-1\}$ 为所给的活动集合。由于 E 中活动安排按结束时间的非减序排列，所以活动 0 具有最早完成时间。首先证明活动安排问题有一个最优解以贪心选择开始，即该最优解中包含活动 0。设 a 是所给的活动安排问题的一个最优解，且 a 中活动也按结束时间非减序排列，a 中的第一个活动是活动 k。如 $k=0$，则 a 就是一个以贪心选择开始的最优解。若 $k>0$，则设 $b=a-\{k\}\cup\{0\}$。由于 $\text{end}[0]\leqslant\text{end}[k]$，且 a 中活动是互为相容的，故 b 中的活动也是互为相容的。又由于 b 中的活动个数与 a 中活动个数相同，且 a 是最优的，故 b 也是最优的。也就是说，b 是一个以贪心选择活动 0 开始的最优活动安排。因此，证明了总存在一个以贪心选择开始的最优活动安排方案，也就是算法具有贪心选择性质。

通过上面的问题，可以看到贪心算法总是做出在当前看来最好的选择。也就是说，贪心算法并不从整体最优考虑，它所做出的选择只是在某种意义上的局部最优选择。当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解，但对许多问题它能产生整体最优解，如单源最短路径问题、最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

贪心算法具有以下基本要素。

- 贪心选择性质。所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式做出相继的贪心选择，每做一次贪心选择就将所求问题简化为规模更小的子问题。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最终导致问题的整体最优解。
- 当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。

贪心算法的基本思路是从问题的某一个初始解出发逐步逼近给定的目标，以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，算法停止。该算法存在以下问题。

- 不能保证求得的最后解是最佳的。
- 不能用来求最大或最小解问题。
- 只能求满足某些约束条件的可行解的范围。

【例 5-2】 数字组合问题：设有 N 个正整数，现在需要设计一个程序，使它们连接在一

起成为最大的数字。例如,3 个整数为 12,456,342,很明显连接为 45634212 为最大;4 个整数 342,45,7,98 连接为 98745342 最大。

程序要求:输入整数 N,接下来一行输入 N 个数字,最后一行输出最大的那个数字。

题目解析:题目不难,就是寻找哪个开头最大,然后连在一起。难在如果 N 比较大,假如几千几万,好像就不是那么回事了,要解答这个题目需要选对合适的贪心策略,并不是把数字由大排到小那么简单,在此,我们对冒泡排序法稍做改进来完成这个任务。冒泡排序的基本算法如下。

```
for(i=0; i<=9; ++i)
  for(int j=0;j<10-1-i;j++)
    if(array[j] > array[j+1])
     {
         temp = array[j];
         array[j] = array[j+1];
         array[j+1] = temp;
     }
```

程序中最核心的比较规则是:

```
if(array[j] > array[j+1])
```

以数字大小作为比较规则来返回 true 或者是 false,那么我们完全可以改变一下这个排序规则,如 23,123 这两个数字,在这个题中它可以组成两个数字 23123 和 12323,分明是前者大些,所以可以说 23 排在 123 前面,也就是 23 的优先级比 123 大,123 的优先级比 23 小,所以不妨写个函数,传递参数 a 和 b,如果 ab 比 ba 大,则返回 true,反之返回 false,函数原型如下。

```
bool compare(int Num1,int Num2);
```

全部代码如下。

```
#include <iostream>
#include <cmath>
using namespace std;

bool compare(int Num1, int Num2);
int main(int argc, char * argv[])
{
    int N;
    cout << "please enter the number n:" << endl;
    cin >> N;
    int * array = new int[N];

    //输入 N 个数
    for(int i = 0; i<N; i++)
        cin >> array[i];

    //按给定的规则排序
    for(int i = 0; i <= N - 1; ++i)
    {
```

```
            for(int j = 0; j < N - i - 1; j++)
            {
                if(compare(array[j], array[j + 1]))
                {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }

        cout << "the max number is:";
        for(int i = N - 1; i >= 0; --i)
            cout << array[i];
        cout << endl;
        delete[] array;

        return 0;
    }

    bool compare(int Num1, int Num2)
    {
        int count1 = 0, count2 = 0;
        int MidNum1 = Num1, MidNum2 = Num2;

        //计算 Num1 有几位
        while(MidNum1)
        {
            ++count1;
            MidNum1 /= 10;
        }
        //计算 Num2 有几位
        while(MidNum2)
        {
            ++count2;
            MidNum2 /= 10;
        }

        int a = Num1 * pow(10, count2) + Num2;
        int b = Num2 * pow(10, count1) + Num1;
        return(a>b) ? true : false;
    }
```

5.3 马踏棋盘问题

现在来求解马踏棋盘问题。棋盘可以看作一个矩阵，当马位于棋盘上某一位置时，它就有一个唯一的坐标，那么根据国际象棋的规则，它有 8 个位置可以跳，这 8 个位置的坐标是和当前马的坐标有联系的，例如，马的坐标是(x,y)，那么它的下一跳的位置可以是$(x-1,$

$y-2$)。当然坐标不能越界，如图 5-1 所示。马所在的当前位置标为 1，它的下一跳的位置标为 2，再下一跳的位置标为 3，以此类推，如果马走完棋盘，那么最后在棋盘上标的位置是 64。

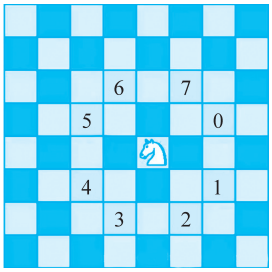

图 5-1　马在当前的位置，根据规则，下一步共有 8 个可选的位置

可以采用回溯法求解，当马在当前位置时，将它下一跳的所有位置保存，然后从中选择一个位置作为当前位置再跳，这样递归下去，如果跳不下去，则回溯。这有点类似图的深度优先搜索。深度优先搜索属于图算法的一种，其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个结点只能访问一次。

代码如下。

```
//采用深度优先的搜索算法解决马踏棋盘的问题
#include<iostream>
using namespace std;

//棋盘的大小
const int ROWS = 8;
const int COLS = 8;

//以当前马的位置为原点，可能移动的 8 个位置的 x 和 y 坐标
const int xMove[] = { -2, -1, 1, 2, 2, 1, -1, -2 };
const int yMove[] = { 1, 2, 2, 1, -1, -2, -2, -1 };

//函数声明
//打印最后的矩阵
void PrintMatrix(int chess[][COLS]);
//寻找下一个位置
//如果存在下一个可以跳的位置，函数返回 true
//位置在 x,y 中，count 取值 0~7，依次测试可能的 8 个位置
bool NextXY(int chess[][COLS], int * x, int * y, int count);
//深度优先的递归算法
bool DeepSearch(int chess[][COLS], int x, int y, int j);

//主函数
void main()
{
    //定义初始矩阵并初始化为 0
    int chess[ROWS][COLS] = { 0 };

    //递归搜索，假设马的初始坐标是(2,0)
    DeepSearch(chess, 2, 0, 1);
    PrintMatrix(chess);
    return;
}
//打印最后的矩阵
void PrintMatrix(int chess[][COLS])
```

```
{
    for(int i = 0; i < ROWS; ++i)
    {
        for(int j = 0; j < COLS; ++j)
        {
            cout.width(2);
            cout.fill('0');
            cout << chess[i][j] << "  ";
        }
        cout << endl;
    }
    cout << endl;
}

//寻找下一个位置
bool NextXY(int chess[][COLS], int * x, int * y, int count)
{
    if( * x + xMove[count] < ROWS &&     * x + xMove[count] >= 0
        && * y + yMove[count]< COLS && * y + yMove[count] >= 0
        && chess[ * x + xMove[count]][ * y + yMove[count]] == 0)
    {
        * x += xMove[count];
        * y += yMove[count];
        return true;
    }
    else/ * 失败 * /
        return false;
}

//深度优先的递归算法
bool DeepSearch(int chess[][COLS], int x, int y, int j)
{
    //保存当前的 x 和 y 的值
    int x1 = x, y1 = y;

    //将新的一步标注到矩阵中
    chess[x][y] = j;

    //判断是否递归结束
    if(j == COLS * ROWS)
    {
        return true;
    }
    / * find the next point in eight directions * /
    int i = 0;
    bool tag = NextXY(chess, &x1, &y1, i);

    //寻找下一个可以跳的位置
    while(!tag && i < 7)
    {
        i++;
        tag = NextXY(chess, &x1, &y1, i);
    }
```

```
    //找到下一个位置
    while(tag)
    {
        //递归调用,继续寻找下一个位置
        if(DeepSearch(chess, x1, y1, j + 1))
            return true;

        //如果失败,换一个可能的位置
        x1 = x; y1 = y;
        i++;
        tag = NextXY(chess, &x1, &y1, i);
        while(!tag && i < 7)
        {
            i++;
            tag = NextXY(chess, &x1, &y1, i);
        }
    }

    //无路可走,回溯
    if(!tag)
        chess[x][y] = 0;
    return false;
}
```

前面已经讲过,贪心算法在对问题求解时,总是做出在当前看来是最好的选择。也就是说,不从整体最优上加以考虑,它所做出的仅是在某种意义上的局部最优解。那么我们在回溯法的基础上,用贪心算法进行优化,在选择下一跳的位置时,总是选择出口少的那个位置,这里出口少是指这个位置的下一跳位置个数少。这是一种局部调整最优的做法,如果优先选择出口多的子结点,那出口少的子结点就会越来越多,很可能出现“死”结点,这样对下面的搜索纯粹是徒劳,会浪费很多无用的时间。反过来,如果每次都优先选择出口少的结点跳,那出口少的结点就会越来越少,这样跳成功的机会就更大一些。

代码如下。

```
//采用贪心算法解决马踏棋盘的问题

#include<iostream>
#include<cstdlib>
using namespace std;

//棋盘的大小
const int ROWS = 8;
const int COLS = 8;

//以当前马的位置为原点,可能移动的 8 个位置的 x 和 y 坐标
const int xMove[] = { -2, -1, 1, 2, 2, 1, -1, -2 };
const int yMove[] = { 1, 2, 2, 1, -1, -2, -2, -1 };

//函数声明
```

```
//打印最后的矩阵
void PrintMatrix(int chess[][COLS]);
//找到数组中最小的非零数的索引位置
int MinIndexInMatrix(int a[], int cols);
//贪心算法的马踏棋盘
void WarnsdorffRole(int matrix[][COLS], int startX, int startY);

//主函数
void main()
{
    int chess[ROWS][COLS] = { 0 };
    WarnsdorffRole(chess, 5, 1);
    PrintMatrix(chess);
}

//打印最后的矩阵
void PrintMatrix(int chess[][COLS])
{
    for(int i = 0; i < ROWS; ++i)
    {
        for(int j = 0; j < COLS; ++j)
        {
            cout.width(2);
            cout.fill('0');
            cout <<chess[i][j] << "  ";
        }
        cout << endl;
    }
    cout << endl;
}
//找到数组中最小的非零数的索引位置
int MinIndexInMatrix(int a[], int cols)
{
    int i = 0, index = 0;
    int min = a[0];
    for(i = 0; i< cols; ++i)
    {
        if(a[i] >0)
        {
            min = a[i];
            index = i;
            break;
        }
    }
    for(i = index + 1; i < cols; ++i)
    {
        if(a[i] > 0 && min > a[i])
        {
            min = a[i];
            index = i;
```