北京邮电大学软件学院
2017-2018 学年第一学期实验报告

课程名称： 算法分析与设计

项目名称： 实验五：贪心及动态规划法

项目完成人：

姓名： 刘禾子 **学号：**2017526019

指导教师： 李朝晖

日　　期： 2017 年 12 月 25 日

一、 实验目的

1、 深刻理解并掌握贪心及动态规划法的设计思想；

2、 提高应用贪心及动态规划法设计算法的技能；

二、 实验内容

基本题 1： 汽车加油问题（贪心法）

假定你开车去香格里拉。出发前你的油箱是满的，可以行驶路程 d。路上一共有 n 个加油站， 加油站之间的距离由数组 A[1…n]给出， 这里 A[i]表示从第 i-1 个加油站到第 i 个加油站之间的距离。最后一个加油站正好在旅程的终点—香格里拉。 你希望沿途停车加油站的次数越少越好。 请设计一个算法，计算沿途需要停车加油的地方，并计算最少加油次数。

基本题 2： 0-1 背包问题（动态规划法）

给定 n 种物品和一背包。物品 i 的重量是 wi，其价值为 vi，背包的容量为 C。问应如何选择装入背包的物品，使得装入背包中物品的总价值最大?

提高题 1：广告牌选取问题

假设你正在管理一条公路的广告牌建设，这条路从西到东 M 英里。广告牌可能的地点假设为 x1,x2,x3…xn， 处于[0,M]中。 若在 xi 放一块广告牌， 可以得到 ri>0 的收益。

国家公路局规定，两块广告牌相对不能小于或等于 5 英里之内。如何找一组地点使你的总收益达到最大?

提高题 3：汽车加油行驶问题

给定一个 N*N 的方形网格，设其左上角为起点，坐标(1， 1)， X 轴向右为正， Y 向下为正，每个方格边长为 1。一辆汽车从起点出发驶向右下脚终点，其坐标为 (N,N)。在若干个网格交叉点处，设置了油库，可供汽车在行驶途中加油。汽车在行驶过程中应该遵守如下规则：

(1) 汽车只能沿网格边行驶，装满油后能行驶 K 条网格边。出发时汽车已装满油，起点与终点处不设置油库；

(2) 当汽车行驶经过一条网格边的时候，若其 X 坐标或者 Y 坐标减小，则应付费 B ，否则免付费用；

(3) 汽车行驶过程中遇油库应该加油并付加油费用 A；

(4) 在需要时可在网格点处增设油库，并付增设费用 C(不含加油费用 A)

(5) 上述(1)~(4)中的各数都是正整数

Input

输入的第一行是 N， K， A， B， C 的值， 2 ≤ N ≤ 100， 2 ≤ K ≤ 10。第二行起是一个 N*N 的 0-1 方阵，每行 N 个值，至 N+1 行结束。方阵的第 i 行第 j 列处的值为 1 表示在网格交叉点（i， j）处设置了一个油库，为 0 时表示未设油库。各行相邻的 2 个数以空格分隔。

Output

程序运行结束时，将找到的最优行驶路线所需的费用，即最小费用输出

Sample Input

9 3 2 3 6

0 0 0 0 1 0 0 0 0

0 0 0 1 0 1 1 0 0

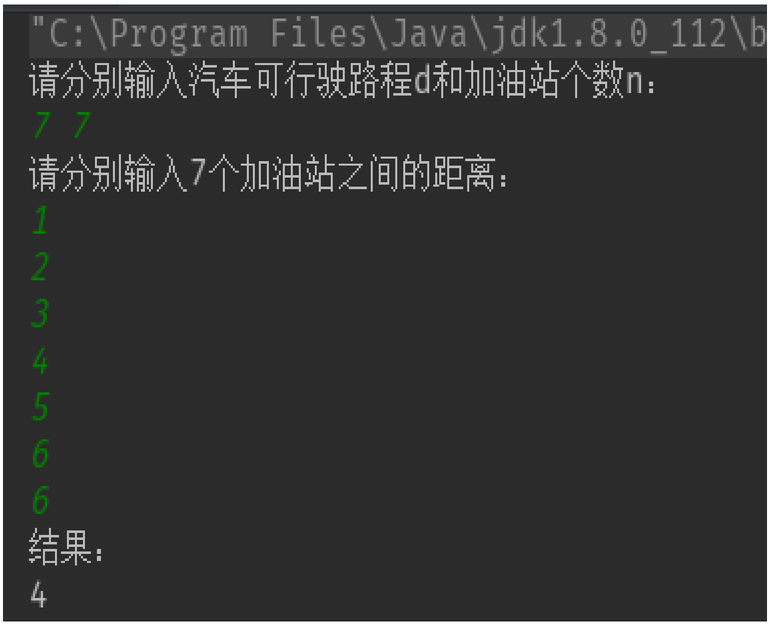
1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
1 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 1
1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0
Sample Output
12

三、 实验环境

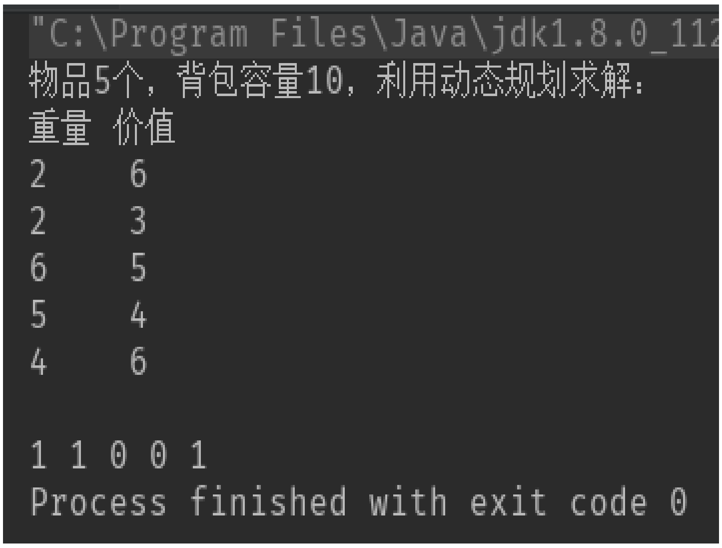
IntelliJ IDEA Community Edition 2017.2.4

四、 实验结果

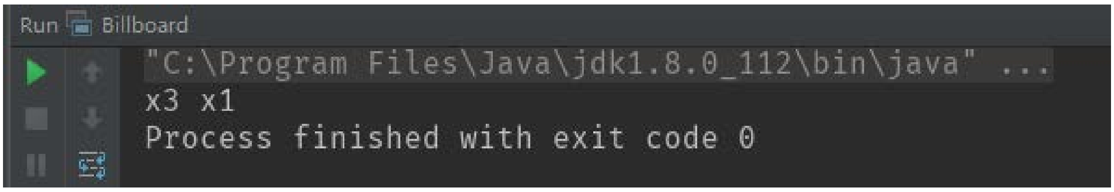
1. 汽车加油问题（贪心法）



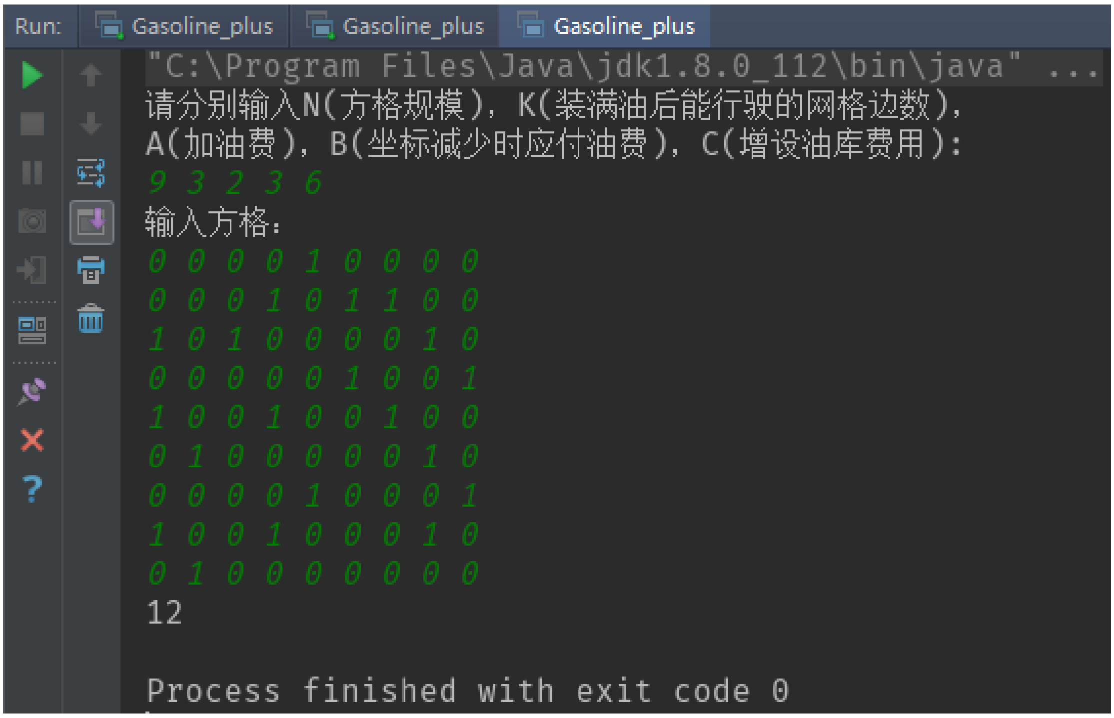
2. 0-1 背包问题(动态规划法)



3. 广告牌选取问题



4. 汽车加油行驶问题



五、 附录

1. 汽车加油问题（贪心法）

（1） 问题分析

给定汽车加满油能行驶的路程 d,以及存在数组 A[n]中的加油站之间的距离，求最少的加油次数。

（2） 设计方案

按照贪心法的算法思想，每次尽量行驶较多的距离直到这个加油站到下个加油站的距离超过当前可行驶路程，再进行加油，若两个加油站之间的距离超过 d 则目的地不可达。

（3） 算法分析

该算法实际上就是遍历了 A[]距离数组，总的时间复杂度为 O（n)。

（4） 源代码

```
import java.util.Scanner;

public class Gasoline {
    private static void greedy(int A[],int d,int n){
        int num=0;
        for (int i=0;i<n;i++){
            if (A[i]>d){
                System.out.println("目的地不可达");
                return;
```

```
            }
        }
        for (int i=0,s=0;i<n;i++){
            s+=A[i];
            if (s>n){
                num++;
                s=A[i];
            }
        }
        System.out.println(num);
    }
    public static void main(String args[]){
        int d,n;
        System.out.println("请分别输入汽车可行驶路程d和加油站个数n：");
        Scanner input=new Scanner(System.in);
        d=input.nextInt();
        n=input.nextInt();
        int A[]=new int[n];
        System.out.println("请分别输入"+n+"个加油站之间的距离：");
        for (int i=0;i<n;i++){
            A[i]=input.nextInt();
        }
        input.close();
        System.out.println("结果：");
        greedy(A,d,n);
    }
}
```

2. 0-1 背包问题（动态规划法）

（1） 问题分析

在 0/1 背包问题中，物品 i 或者被装入背包，或者不被装入背包，设 xi 表示物品 i 装入背包的情况，则当 xi=0 时，表示物品 i 没有被装入背包，xi=1 时，表示物品 i 被装入背包。根据问题的要求，有如下约束条件和目标函数：

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \le C \\ x_i \in \{0,1\} \quad (1 \le i \le n) \end{cases} \qquad \text{（式9）}$$

$$\max \sum_{i=1}^{n} v_i x_i \qquad \text{（式10）}$$

将问题归结为寻找一个满足约束条件式 9，使得目标函数式达到最大的解向量 X=(x1,x2…,xn)。

（2） 设计方案

证明 0-1 背包问题满足最优性原理：

设(x1, x2, …, xn)是所给 0/1 背包问题的一个最优解，则(x2, …, xn)是下面一个子问题的最优解：

$$\begin{cases} \sum_{i=2}^{n} w_i x_i \le C - w_1 x_1 \\ x_i \in \{0,1\} \quad (2 \le i \le n) \end{cases} \qquad \max \sum_{i=2}^{n} v_i x_i$$

如不然，设（y2,…,yn）是上述子问题的一个最优解，则

$$\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i \quad w_1 x_1 + \sum_{i=2}^{n} w_i y_i \le C$$

因此

$$v_1 x_1 + \sum_{i=2}^{n} v_i y_i > v_1 x_1 + \sum_{i=2}^{n} v_i x_i = \sum_{i=1}^{n} v_i x_i$$

这说明(x1, y2, …, yn)是所给 0/1 背包问题比(x1,x2,…,xn)更优的解从而导致矛盾。0/1 背包问题可以看作是决策一个序列(x1,x2,…,xn)，对任一变量 xi 的决策是决定 xi=1 还是 xi=0。在对 xi-1 决策后，已确定了(x1,…,xi-1)，在决策 xi 时，问题处于下列两种状态之一：

（1） 背包容量不足以装入物品 i，则 xi=0，背包不增加价值；

（2） 背包容量可以装入物品 i，则 xi=1，背包的价值增加了 vi。

这两种情况下背包价值的最大者应该是对 xi 决策后的背包价值。令 V(i,j)表示在前 i(1≤i≤n)个物品中能够装入容量为 j（1≤j≤C）的背包中的物品的最大值，则可以得到如下动态规划函数：

$$V(i,0)=V(0,j)=0 \qquad ($$

$$V(i,j)=\begin{cases} V(i-1,j) & j<w_i \\ \max\{V(i-1,j),\ V(i-1,j-w_i)+v_i\} & j>w_i \end{cases}$$

（3） 算法分析

在上述算法中，第一个 for 循环的时间性能是 O(n)，第二个 for 循环的时间性能是 O(C)，第三个循环是两层嵌套的 for 循环，其时间性能是 O(n×C)，第四个 for 循环的时间性能是 O(n)，所以，算法的时间复杂性为 O(n×C)。

（4） 源代码

```
import static java.lang.Integer.max;
public class Knapsack {
    private static int n=5;//物品数量
    private static int c=10;//背包容量
    private static int w[]={2,2,6,5,4};//重量
    private static int v[]={6,3,5,4,6};//价值
    private static int x[]={0,0,0,0,0};//解集合，放入为 1，反之为 0
    public static void main(String args[]){
        System.out.println("物品"+n+"个，背包容量 10，利用动态规划求解：");
        System.out.println("重量 价值");
        for (int i=0;i<n;i++){
            System.out.println(w[i]+"    "+v[i]);
        }
        BP_solution();
    }

    private static void BP_solution() {
        int V[][]=new int[n+1][c+1];
        int i,j;
        //初始化条件
        for (i=0;i<=n;i++)//前 n 个物品放入容量为 0 的背包
            V[i][0]=0;
        for (j=0;j<=c;j++)//前 0 个物品放入容量为 c 的背包
```

```
            V[0][j]=0;
        for (i=1;i<=n;i++)
            for (j=1;j<=c;j++) {
                if (j < w[i-1])
                    V[i][j] = V[i - 1][j];//小于容量则不装入价值和上一层相同
                else
                    V[i][j]=max(V[i-1][j],V[i-1][j-w[i-1]]+v[i-1]);
            }
         //向回求解放入的物品
        for (j=c,i=n;i>0;i--){
            if (V[i][j]>V[i-1][j]){
                x[i-1]=1;j=j-w[i-1];
            }else
                x[i-1]=0;
        }
        System.out.println();
        for (i=0;i<n;i++){
            System.out.print(x[i]+" ");
        }
    }
}
```

3. 广告牌选取问题

（1） 问题分析

问题的最优子结构性质：{x1,x2,…,xn}的最优解一定包含{x1,x2,…,xn-1}的最优解，该结论可以用反证法证明。最优解为：n个地点放置广告牌的方案，同时使总收益最大。

（2） 设计方案

M=20,n=4,{x1,x2,x3,x4}={6,7,12,14},{r1,r2,r3,r4}={5,6,5,1}, 考虑对于给定输入实例的一个最优解，在地点 xn 或者放广告牌，或者不放。

如果不放，那么在地点 x1,x2,…,xn 上的最优解实际上与地点 x1,x2,…,xn-1 上的最优解一样。如果放，那么应该去掉 xn 以及与它在 5 英里之内的所有其他地点，并且找在剩下地点上的最优解。

当考查仅前 j 个地点 x1,x2,…,xj 所定义的问题时，同样的推理也适用：在最优解中包含或不包含 xj,具有同样的结果。

令 e(j)表示编号比 j 小且距 xj 大于 5 英里的最东边的地点

• 令 OPT(j)表示从 x1,…,xj 中地点的最优子集得到的收益

• OPT(j)=max(rj+ OPT(e(j)), OPT(j-1))

地点编号	0	1	2	3	4
位置 x	0	6	7	12	14
收益 r	0	5	6	5	1
e(j)	0	0	0	1	2
OPT(j)	0	5	6	10	10
最优解位置	0	5		12	

（3） 算法分析

第一个 for 循环是两层嵌套循环,算法平均复杂度为 O(j*i/2)约为 O(n^2/2),后面两个 for 循环复杂度均为 O（n）,总的复杂度为 O(n^2)

（4） 源代码

```
import static java.lang.Integer.max;
public class Billboard {
    private static int[] e={0,0,0,0,0};//e[j]表示编号比 j 小且距 xj 大于 5 英里的最东边的地点
    private static int[] opt={0,0,0,0,0};//opt[j]表示从 x1,...,xj 中地点的最优子集得到的收益
    private static int[] pre={-1,-1,-1,-1,-1};//用于记录与当前放置广告牌的前一个可放置广告牌

    public static void main(String args[]){
        int M=20,n=4;//距离 20，4 个广告牌
        int[] x={0,6,7,12,14};//广告牌坐标
        int[] r={0,5,6,5,1};  //广告牌价值
        select(n,x,r);
    }

    private static void select(int n, int[] x, int[] r) {
        for (int j = 2; j <= n; j++) {
            for (int i = 1; i < j; i++) {
                int distance = 5;
                if (x[j] - x[i] > distance){
                    e[j]++;
                    pre[j]=i;
                }
            }
        }
        for (int j=1;j<=n;j++){
            opt[j]=max(r[j]+opt[e[j]],opt[j-1]);
        }
        for (int j=n;j>0;j--){
            if (opt[j]>opt[j-1]&&pre[j]>0){
                System.out.print("x"+j+" x"+pre[j]);
            }
        }
    }
}
```

4. 汽车加油行驶问题

（1）问题分析

本问题的限制条件较多，汽车加满油一次能行驶 d 条边，碰到加油站比必须加油并付费用 A，若油耗尽且没有油库则构建油库并加油支付费用 A+C，汽车经过一条网格边，X 坐标或者 Y 坐标减小，则付费用 B，否则免费，求从 a[1][1]到 a[N][N]最少费用。

（2）设计方案

f[i][j][0]表示汽车从网格点(1,1)行驶至网格点(i,j)所需的最少费用

f[i][j][1]表示汽车行驶至网格点(i,j)还能行驶的网格边数

s[i][0]表示 x 轴方向,s[i][1]表示 y 轴方向,s[i][2]表示行驶费用

f[i][j][0] = min{f[i+s[k][0]][j+s[k][1]][0] + s[k][2]}

f[i][j][1] = f[i+s[k][0]][j+s[k][1]][1] - 1;

f[i][j][0] = f[i][j][0] + A , f[i][j][1] = K   如果(i, j)是油库

f[i][j][0] = f[i][j][0] + C + A, f[i][j][1] = K (i, j)不是油库且 f[i][j][1] = 0

（3）算法分析

本算法的关键部分主要在遍历各个点的四个方向分别计算其费用上，两层 N 次 for 循环,内部还嵌有对四个方向判断的 for 循环,总的复杂度为 O(N*N*4),约为 O（N^2）。

（4）源代码

```java
import java.util.Scanner;

public class Gasoline_plus {
    private static int N,A,C,B,K;
    private static int[][][] f=new int[50][50][2];
    private static int[][] s={{-1,0,0},{0,-1,0},{1,0,B},{0,1,B}};
    private static int[][] a=new int[50][50];//输入的方形网络
    private static int INF=10000;
    private static int sol(){
        int i,j,k;
        //初始化参数
        for (i=1; i<=N; i++){
            for (j=1;j<=N;j++){
                f[i][j][0]= INF;
                f[i][j][1]=K;
            }
        }
        f[1][1][0]=0;
        f[1][1][1]=K;
        int count=1;
        int tx,ty;
        while (count>0){
            count=0;
            for (i=1;i<=N;i++){
                for (j=1;j<=N;j++){
                    if (i==1&&j==1)
                        continue;
                    int minStep= INF;
                    int minDstep=0,step,dstep;
                    for (k=0;k<4;k++){//可行驶的四个方向
                        tx=i+s[k][0];
                        ty=j+s[k][1];
                        if (tx<1||ty<1||tx>N||ty>N)//出界
                            continue;
                        step=f[tx][ty][0]+s[k][2];
                        dstep=f[tx][ty][1]-1;
                        if (a[i][j]==1){//若是油库
                            step+=A;
                            dstep=K;
                        }
                        if (a[i][j]==0&&dstep==0&&(i!=N||j!=N)){//若不是油库，且油已耗光
                            step+=A+C;
                            dstep=K;
                        }
                        if (step<minStep){
                            minStep=step;
                            minDstep=dstep;
                        }
                    }
                    if (f[i][j][0]>minStep){
                        count++;
                        f[i][j][0]=minStep;
                        f[i][j][1]=minDstep;
                    }
                }
            }
        }
        return f[N][N][0];
```

```
    }
    public static void main(String args[]){
        System.out.println("请分别输入N(方格规模),K(装满油后能行驶的网格边数),\nA(加油费),
B(坐标减少时应付油费),C(增设油库费用):");
        Scanner input=new Scanner(System.in);

N=input.nextInt();K=input.nextInt();A=input.nextInt();B=input.nextInt();C=input.nextInt();
        s[2][2]=s[3][2]=B;
        System.out.println("输入方格:");
        for (int i=1;i<=N;i++){
            for (int j=1;j<=N;j++){
                a[i][j]=input.nextInt();
            }
        }/*输入为:
9 3 2 3 6

0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 1 0 0
1 0 1 0 0 0 0 1 0
0 0 0 0 0 1 0 0 1
1 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0 1
1 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0*/
        input.close();
        System.out.println(sol());
    }
}
```

5. 调试心得

经过本次实验，我深刻地理解了动态规划以及贪心法的核心思想，求解动态规划问题要分析问题得到其最优解结构，求出最优解的求解公式，依次计算最后得到最优解。