

**数学与信息学院学生实验报告**

**实验课程名称：** 算法分析与设计基础 **教师：\_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **实验四 动态规划算法设计与应用** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** | **133** | **年级专业班级** | **2017级软件工程** |
| **小组成员** | **无** | | | **实验日期** | **2019年5 月** |

# 1. 实验目的和要求

## 1.1 实验目的

理解贪心算法的基本原理，掌握贪心算法设计的基本方法及其应用；

## 1.2 实验软硬件环境

① 操作系统

Win10系统

② 编译环境

Java

# 2.基本原理

贪心法是一种算法设计技术，通常用于求解最优化问题。

通过一系列选择步骤来构造问题的解，每一步都是对当前部分解的一个扩展，直至获得问题的完整解。所做的每一步选择都必须满足：

1）可行的：必须满足问题的约束。

2）局部最优：当前所有可能的选择中最佳的局部选择。

3）不可取消: 选择一旦做出，在后面的步骤中就无法改变了。

要注意的是，贪心法不能保证总能得到最优解（一系列的局部最优选择不能保证最后得到整体最优解）

# 3.该类算法设计与实现的要点

贪心算法往往效率高，一般时间复杂性为多项式阶。贪心算法一般较简单，其关键和难点在于贪心选择策略的确定，以及证明相应的贪心算法确实可求出最优解。

# 4实验内容

1. 加油问题（Problem Set 1702）：
   1. **问题描述**

一个旅行家想驾驶汽车从城市A到城市B（设出发时油箱是空的）。给定两个城市之间的距离dis、汽车油箱的容量c、每升汽油能行驶的距离d、沿途油站数n、油站i离出发点的距离d[i]以及该站每升汽油的价格p[i],i=1,2,…,n。设d[1]=0<d[2]<…<d[n]。要花最少的油费从城市A到城市B，在每个加油站应加多少油，最少花费为多少？

* 1. **具体要求**

Input

输入的第一行是一个正整数k，表示测试例个数。接下来几行是k个测试例的数据，每个测试例的数据由三行组成，其中第一行含4个正整数，依次为A和B两个城市之间的距离d1、汽车油箱的容量c（以升为单位）、每升汽油能行驶的距离d2、沿途油站数n (1<=n<=200)；第二行含n个实数d1, d2 ,…, dn，表示各油站离出发点的距离（d1=0）；第三行含n个实数p1, p2 ,…, pn，表示各油站每升汽油的价格。同一行的数之间用一个空格隔开。

Output

对于每个测试例输出一行，含一个实数，表示从城市Ａ到城市Ｂ所要花费的最少油费（输出的结果精确到小数点后一位）。若问题无解，则输出“NoSolution”。

* 1. **测试数据**

Sample Input

3

1500 50 10 4

0 300.0 800.0 1200.0

4.0 5.0 4.0 4.5

1000 40 10 3

0 500.0 750.0

4.5 5.0 4.2

1500 50 10 4

0 400 700 1100

1 4 500 1

Sample Output

640.0

No Solution

**4．设计与实现的提示**

1. 注意考虑无解的情况
2. 对终点站可进行特殊处理

**5.扩展内容**

(1) 演示时建议采用可视化界面

(2) The Express Mail(Problem Set 1755)

1. **黑白点的匹配（Problem Set 1714）：**
   1. **问题描述**

设平面上分布着n个白点和n个黑点，每个点用一对坐标（x, y）表示。一个黑点b=（xb,yb）支配一个白点w=(xw,yw)当且仅当xb>=xw和yb>=yw。若黑点b支配白点w，则黑点b和白点w可匹配（可形成一个匹配对）。在一个黑点最多只能与一个白点匹配，一个白点最多只能与一个黑点匹配的前提下，求n个白点和n个黑点的最大匹配对数。

* 1. **具体要求**

Input

输入的第一行是一个正整数k，表示测试例个数。接下来几行是k个测试例的数据，每个测试例的数据由三行组成，其中第一行含1个正整数n(n<16)；第二行含2n个实数xb1, yb1,xb2, yb2,…, xbn, ybn， (xbi, ybi)，i=1, 2, …, n表示n个黑点的坐标；第三行含2n个实数xw1, yw1,xw2, yw2,…, xwn, ywn，(xwi, ywi)，i=1, 2, …,n表示n个白点的坐标。同一行的实数之间用一个空格隔开。

Output

对于每个测试例输出一行，含一个整数，表示n个白点和n个黑点的最大匹配对数。

* 1. **测试数据**

Sample Input

1

3

5.0 3.0 5.0 -1.0 4.0 4.0

2.0 3.5 2.0 2.0 -2.0 -2.0

Sample Output

3

* 1. **扩展内容**

(1) 建议采用可视化界面

# 5.实验记录

## 5.1加油问题（Problem Set 1702）

#### 5.1.1贪心思路

solution1 // 这是我自己想的但是写出来后局部最优解不是全局最优解  
贪心选择主要思路：//我用了递归解决  
如果，当前站（和后面的所有站比）是最便宜的，则加满或者加到刚好到终点  
否则，把油加到刚好开到下一站。

solution2 // 这是借鉴了网上的代码并结合自己代码改进了的

贪心选择主要思路： //用的是迭代

首先，

先定好了最后一站距离是总距离，最后一站油费是0  
如果，后面的加油站有比当前便宜的，则把油加到刚好开到目标站（比当前便宜的）  
否则（当前站最便宜），把油加满或者加到刚好到终点  
  
solution2 是对 solution1 的改进  
主要是在当前站不是最便宜的时候的选择策略进行了改进  
solution1的选择是：  
把油加到刚好开到下一站  
solution2的选择是：  
把油加到刚好开到目标站（比当前便宜的）

#### 5.1.2代码

**import** java.util.Scanner;

**publicclass** Refueling {

**staticfloat** solution1(**float**i\_c, **int**dis, **float**c, **int**d, **int**i, **float**[] i\_dis, **float**[] i\_value) {

// **TODO** Auto-generated method stub

**int**index = i;

**float**min = i\_value[i];

**for** (**int**j = i; j<i\_value.length; j++) {

**if** (i\_value[j] <min) {

min = i\_value[j];

index = j;

}

}

**if** (i == index)// 目前站最便宜(和后面的站比)

{

**if** (c \* d>dis - i\_dis[i])// 可以直接到终点

{

**return** ((dis - i\_dis[i] - i\_c \* d) / d) \* i\_value[i];

} **else**// 加满

{

**for** (**int**j = i; j<i\_value.length; j++)

**if** (i\_dis[j] - (i\_dis[i] + c \* d) > 0) {

**return** (c - i\_c) \* i\_value[i]

+ *solution1*(c - (i\_dis[j - 1] - i\_dis[i]) / d, dis, c, d, j - 1, i\_dis, i\_value);

}

**return** (c - i\_c) \* i\_value[i] + *solution1*(c - (i\_dis[i\_dis.length - 1] - i\_dis[i]) / d, dis, c, d,

i\_dis.length - 1, i\_dis, i\_value);

}

} **else** {

**return** ((i\_dis[i + 1] - i\_dis[i]) / d - i\_c) \* i\_value[i] + *solution1*(0, dis, c, d, i + 1, i\_dis, i\_value);

}

}

**staticvoid** solution2(**float**i\_dis[], **float**pir[], **int**n, **int**d, **float**c)// i\_dis距离起点距离，pir油价

{

**int**i, j, k;

**float**pirce = 0, c1 = 0, x=0, c2=0;

**for** (j = 0; j<n; j++) {

**for** (i = j + 1; i<= n; i++) {

**if** (pir[i] <pir[j]) {

k = i;

c2 = (i\_dis[k] - i\_dis[j]) / d;

**if** (c2>= c) {

x = c - c1;

} **else** {

x = c2 - c1;

**if** (x< 0) {

x = 0;

}

}

pirce = pirce + pir[j] \* x;

**break**;

}

}

c1 = c1 + x - ((i\_dis[j + 1] - i\_dis[j]) / d);

}

System.***out***.println("solution2: "+pirce);

}

}

**staticboolean** check(**int**dis, **float**c, **int**d, **int**i, **float**[] i\_dis) {

**if** (i<i\_dis.length - 1) {

**if** (i\_dis[i + 1] - i\_dis[i] >c \* d)

**returnfalse**;

**else** {

**return***check*(dis, c, d, i + 1, i\_dis);

}

} **else** {

**if** (dis - i\_dis[i] >c \* d)

**returnfalse**;

**returntrue**;

}

}

**publicstaticvoid** main(String[] args) {

Scanner scan = **new** Scanner(System.***in***);

**int**t = scan.nextInt();

System.***out***.println("测试次数：" + t);

**while** (t != 0) {

**int**dis = scan.nextInt();

System.***out***.println("A、B的距离：" + dis);

**float**c = scan.nextFloat();

System.***out***.println("油箱容量：" + c);

**int**d = scan.nextInt();

System.***out***.println("每升汽油能行驶的距离：" + d);

**int**n = scan.nextInt();

System.***out***.println("沿途油站数：" + n);

**float**i\_value[] = **newfloat**[n];

**float**i\_dis[] = **newfloat**[n];

**for** (**int**i = n; i != 0; i--)

i\_dis[i\_dis.length - i] = scan.nextFloat();

**for** (**int**i = n; i != 0; i--)

i\_value[i\_value.length - i] = scan.nextFloat();

**if** (*check*(dis, c, d, 0, i\_dis))

{

System.***out***.println("solution1: "+ *solution1*(0, dis, c, d, 0, i\_dis, i\_value));

**float**[] arr2=java.util.Arrays.*copyOf*(i\_dis,i\_dis.length+1);

arr2[i\_dis.length]=dis;

**float**[] arr1=java.util.Arrays.*copyOf*(i\_value,i\_dis.length+1);

arr1[i\_dis.length]=0;

*solution2*(arr2, arr1, n, d, c);

}

**else**

System.***out***.println("no solution");

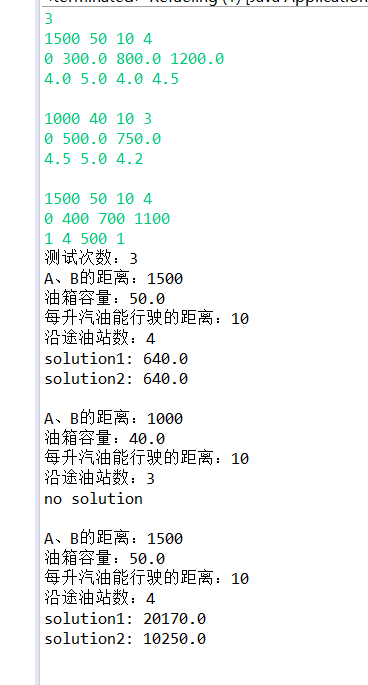
System.***out***.println();

t--;

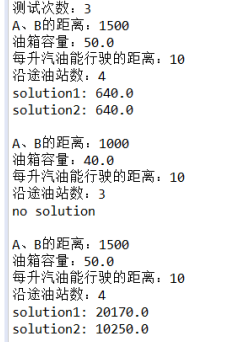
}

}

#### 5.1.3结果



#### 5.1.4结果分析



可以看到solution1 在前两次的测试都得到了正确的答案，但是在第3个测试样例它就出错了，这也说明了贪心算法局部最优解有时候是全局最优解而有时候不是，所以贪心算法需要证明局部最优解一定是全局最优解算法才是可行的。

## 5.2黑白点的匹配（Problem Set 1714）：

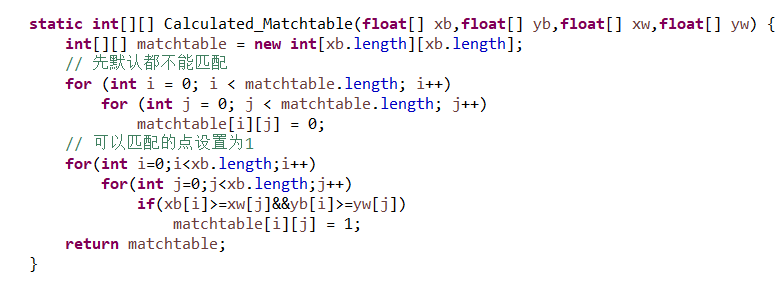
#### 5.2.1贪心思路

**1.**先将黑棋和白棋可匹配和不可匹配关系用matchtable二维数组保存  
可匹配为1，不可匹配为0

计算出如下图b1,b2,b3是黑点 w1,w2,w3是白点 1代表可以匹配，0代表不能

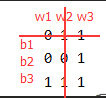


计算代码如下

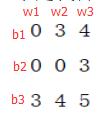


**2.**根据上面的匹配表算出优先匹配的棋子

如果是可匹配那么计算出它的匹配后会影响到多少个棋子

如图b1和w2它们可匹配，如果它们匹配会影响到w3的一个选择和b3的一个选择 横行和竖行有3个1

于是我得到一个这样的表



计算的代码如下：

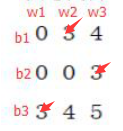
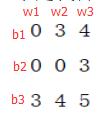


**3.**影响最小则匹配优先级最高（贪心选择）

因为如果先匹配了影响大的可能会造成：

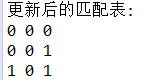
影响小的匹配点被PASS掉了（一个白点只能和一个黑点匹配）

这样匹配数量就变少了



如图影响最小是3 有3个匹配从中随便选一个 //按顺序就选b1,w2

**4.**选择影响最小的匹配完后更新匹配表

 ----->

5.判断是否匹配完了，没有就重复2

#### 5.2.2代码

**import** java.util.Scanner;

**publicclass** ChessMatch {

**staticint**[][] Calculated\_Matchtable(**float**[] xb,**float**[] yb,**float**[] xw,**float**[] yw) {

**int**[][] matchtable = **newint**[xb.length][xb.length];

// 先默认都不能匹配

**for** (**int**i = 0; i<matchtable.length; i++)

**for** (**int**j = 0; j<matchtable.length; j++)

matchtable[i][j] = 0;

// 可以匹配的点设置为1

**for**(**int**i=0;i<xb.length;i++)

**for**(**int**j=0;j<xb.length;j++)

**if**(xb[i]>=xw[j]&&yb[i]>=yw[j])

matchtable[i][j] = 1;

**return**matchtable;

}

**staticvoid** solution1(**int**[][] matchtable) {

**int**n=matchtable.length;

**int**count=0;

//根据匹配表算出优先匹配的棋子（即匹配后影响最小的）

**int**priority[][]=**newint**[n][n];

//优先级表初始化为0

**while**(**true**)

{

**for** (**int**i = 0; i<priority.length; i++)

**for** (**int**j = 0; j<priority.length; j++)

priority[i][j] = 0;

//遍历匹配表如果可以匹配则计算它的优先级

**for** (**int**i = 0; i<n; i++)

**for** (**int**j = 0; j<n; j++)

**if**(matchtable[i][j] == 1)

{

priority[i][j]=1;

**for**(**int**k=0;k<n;k++)

**if**(matchtable[i][k]==1&&k!=j)

priority[i][j]+=1;

**for**(**int**k=0;k<n;k++)

**if**(matchtable[k][j]==1&&k!=i)

priority[i][j]+=1;

}

//显示优先级表

System.***out***.println("优先级:");

**for** (**int**i = 0; i<priority.length; i++)

{

**for** (**int**j = 0; j<priority.length; j++)

System.***out***.print(priority[i][j]+"");

System.***out***.println();

}

//找到优先级最大的

**int**min=n\*n;

**int**indexi=0;

**int**indexj=0;

**for** (**int**i = 0; i<priority.length; i++)

**for** (**int**j = 0; j<priority.length; j++)

**if**(min>priority[i][j]&&priority[i][j]!=0)

{

min=priority[i][j];

indexi=i;

indexj=j;

}

//优先级最大的先匹配，更新匹配表

**for** (**int**i = 0; i<n; i++)

matchtable[i][indexj] = 0;

**for** (**int**j = 0; j<n; j++)

matchtable[indexi][j] = 0;

//显示更新后的匹配表

System.***out***.println("更新后的匹配表:");

**for** (**int**i = 0; i<matchtable.length; i++)

{

**for** (**int**j = 0; j<matchtable.length; j++)

System.***out***.print(matchtable[i][j]+"");

System.***out***.println();

}

count++;

//判断是否匹配完了

**int**matchnum=0;

**for** (**int**i = 0; i<n; i++)

**for** (**int**j = 0; j<n; j++)

**if**(matchtable[i][j] == 1)

matchnum++;

**if**(matchnum==0)

**break**;

}

System.***out***.println("结果："+count);

}

**publicstaticvoid** main(String[] args) {

Scanner scan = **new** Scanner(System.***in***);

**int**t = scan.nextInt();

System.***out***.println("测试次数：" + t);

**int**n ;

**int**k ;

**while** (t != 0) {

n = scan.nextInt();

k=n;

System.***out***.println("输入顶点个数：" + n);

**float**xb[] = **newfloat**[n];

**float**yb[] = **newfloat**[n];

**float**xw[] = **newfloat**[n];

**float**yw[] = **newfloat**[n];

**int**matchtable[][] = **newint**[n][n];

**while** (n != 0) {

xb[xb.length - n] = scan.nextFloat();

yb[yb.length - n] = scan.nextFloat();

n--;

}

**while** (k != 0) {

xw[xw.length - k] = scan.nextFloat();

yw[yw.length - k] = scan.nextFloat();

k--;

}

// 计算每个节点，可匹配为1，不可匹配为0

matchtable = *Calculated\_Matchtable*(xb, yb, xw, yw);

System.***out***.println("匹配表");

**for** (**int**i = 0; i<matchtable.length; i++)

{

**for** (**int**j = 0; j<matchtable.length; j++)

System.***out***.print(matchtable[i][j]+"");

System.***out***.println();

}

*solution1*(matchtable);

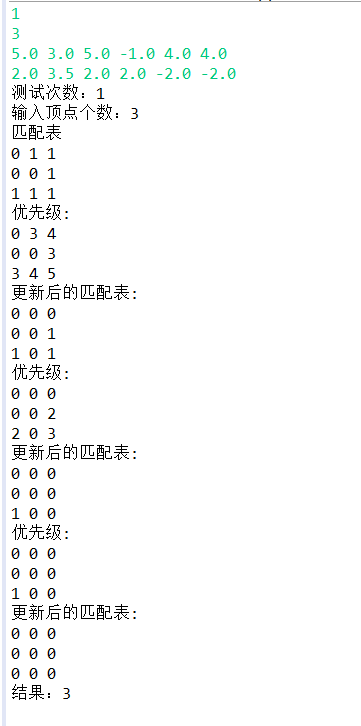
t--;

}

}

}

#### 5.2.3结果



#### 5.2.4结果分析

根据结果来看是得出了正确的结果，但是问题是不知道如何证明这个解是全局最优解，但是值得一提的是我还没有找到一个返例来推翻它。

# 6. 实验总结

在解决加油问题的时候我第一时间想到的是 如果这一站最便宜，那就加满 ，但是这一站不是最便宜的时候我的选择是加到下一站

这满足了基本原理1）可行的：必须满足问题的约束。

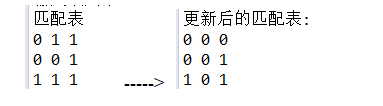
但是没有满足 2）局部最优：当前所有可能的选择中最佳的局部选择。

应该把油加到刚好开到比当前便宜的站

第二个问题的解决有点像动态规划用了表格来表示，但是它应该是贪心算法的。每一步选择影响最小的那个，是贪心策略。每一步都满足：

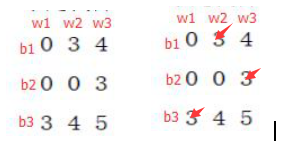
1）可行的：必须满足问题的约束。

//每个白点只能匹配一个黑点

不会出现匹配两次

2）局部最优：当前所有可能的选择中最佳的局部选择。

//影响最小的匹配点先匹配



3）不可取消: 选择一旦做出，在后面的步骤中就无法改变了。

//没有回溯

**不足：**不会证明它们的正确性