

**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** |  | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** | **123** | **年级专业班级** |  |
| **小组成员** | **无** | | | **实验日期** | **2019年4 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

1. 编译环境 java

## 实验要求

### 1.3.1加油问题

* 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**

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。

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“No Solution”。

**测试数据**

2

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

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

### 1.3.2黑白点的匹配

**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。

**具体要求**

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。

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。

**测试数据**

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

# 2. 实验记录

## 2.1 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）

## 2.2 该类算法设计与实现的要点

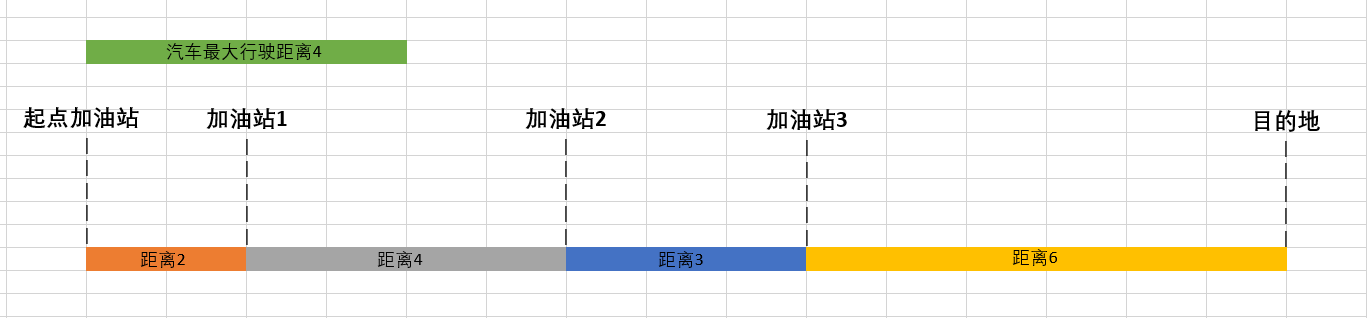
贪心算法往往效率高，一般时间复杂性为多项式阶。贪心算法一般较简单，其关键和难点在于贪心选择策略的确定，以及证明相应的贪心算法确实可求出最优解。

## 2.3 实验过程

### 2.3.1 实验思路

#### 2.3.1.1 加油问题

在考虑怎样省钱之前先考虑汽车能否到达目的地：



上图中，汽车在不加油的情况下最多行驶距离为4，但加油站3与目的地之间距离为6，就算加满油也无法到达。所以在输入加油站距离时，要注意间隔距离不能大于汽车最大行驶距离。

开始解决省钱的问题：由于贪心算法讲究局部最优，所以我只比较当前加油站和下一站加油站的油费，汽车每到达一个加油站时做如下判断：

1. 若够油去下一个加油站，则不加油。
2. 若是最后一站加油站，则加到刚好能到达目的地的油量。
3. 若不够油则比较当前加油站和下一站的油价，若当前油价便宜，则直接加满，反之加到刚好够行驶到下一站加油站的油量。

下面举一个例子来演示整个过程：



设 油箱容量：4 每升行驶距离：1

则在每个加油站做如下工作：

起点加油站：油箱空，且加油站1 的油更便宜，所以加油2，累计花费2；

加油站1：油箱空，且加油站2 的油更便宜，加油4，累计花费16；

加油站2：油箱空，且加油站3 的油更便宜，加油3，累计花费25；

加油站3: 油箱空且最后一站，加油2.24，最后共花费31.272；

#### 2.3.1.2 黑白点的匹配

一看完题目，一开始的思路是先将黑白点分别存入两个数组中，再对两个数组分别进行对x和对y的排序，在实际实验过程中，发现排序完后数组的下标与点不好对应，这样就不容易确定一个点是否已经匹配过。

经过了解查阅后发现了最大匹配问题的算法，和本题类似，而且递归的操作复杂度远小于多次对数组的排序。而且过多的排序也造成了算法思路难以理清。决定先学习掌握最大匹配度算法再考虑本题…

在查阅了最大匹配度问题的思想后，发现这是一种递归形式的方法，算法需要基于对二分图的遍历算法，这就需要学习DFS或者BFS，所以又去复习了一下这两个算法，在彻底掌握了之后终于可以步入正题了…

（dfs和bfs的执行动态图）

<http://5b0988e595225.cdn.sohucs.com/images/20171101/f1f45fe9ca37425ba200180be89624b2.gif>

<http://5b0988e595225.cdn.sohucs.com/images/20171101/a85c0716fcc847f1915dddfcfd019c01.gif>

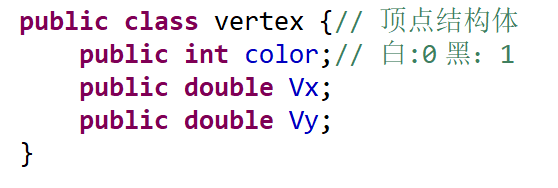
理解了最大匹配算法后，发现只要在图遍历的基础上，多借助一个matching数组，用来储存各匹配点之间的联系，通过一些剪枝和判断就可以实现。

我选择了DFS进行最大匹配算法的基础算法，DFS是对图做出处理，在空间上需要借助一张邻接矩阵，我的想法是将黑白点问题化作图，再根据题目的要求做出对应的邻接矩阵，这样再通过最大匹配就可以求解出来。

下面主要针对这两个问题讨论并通过具体例子演示最大匹配核心思想。

**1、如何将黑白点化作图：**

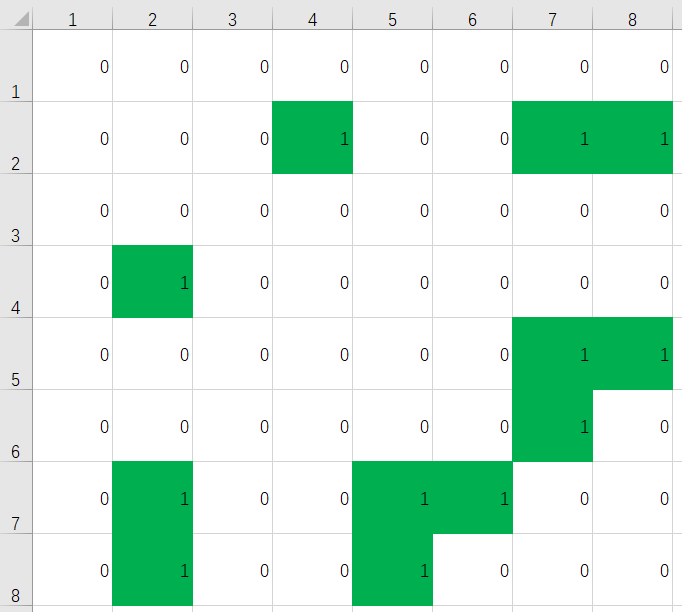
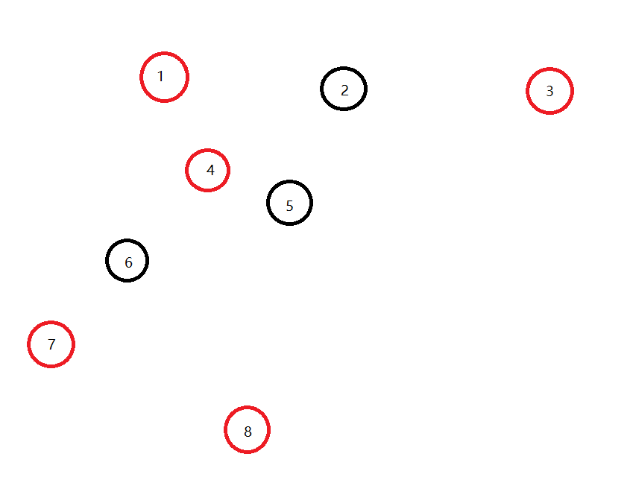
创建一个结构体



黑白点都看作顶点，只通过color进行区别

**2、如何求对应邻接矩阵：**

对储存所有顶点的结构体数组做两次循环，若满足题目中黑点xy坐标大于白点，即将邻接矩阵该位置置为1。

**3、具体流程演示：  
**

通过邻接表可以知道，2能控制4,7,8三点

一开始2就控制了4，跳过2点

接着5控制了7，跳过5点

6控制了7,但是7已经被5控制，这时回到5，

5控制了8，跳过5

这时7没人控制，6控制7，流程结束，匹配度为3。

### 2.3.2 程序代码

#### 2.3.2.1 加油问题

**import** java**.**util**.**Scanner**;**

public class car **{**// 贪心算法 只考虑当前油站和下一个油站的油价 得出的结论不一定最优

public static void main**(**String**[]** args**)** **{**

car c **=** **new** car**();**

c**.**init**();**

**}**

public void init**()** **{**

// 初始并输入所有数据

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

float cost **=** 0**;**

System**.**out**.**println**(**"int dis/c/d:"**);**

float dis **=** sc**.**nextFloat**(),** c **=** sc**.**nextFloat**(),** d **=** sc**.**nextFloat**();**

System**.**out**.**println**(**"int n:"**);**

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

float das\_dis**[]** **=** **new** float**[**n**];**// 油站i离出发点的距离

float p**[]** **=** **new** float**[**n**];**// 及该站每升汽油的价格

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

System**.**out**.**println**(**"int d[" **+** i **+** "] and p[" **+** i **+** "]"**);**

das\_dis**[**i**]** **=** sc**.**nextFloat**();**

p**[**i**]** **=** sc**.**nextFloat**();**

**}**

**if** **(!**isSolution**(**d **\*** c**,** das\_dis**))** **{**// 判断合法性

**return;**

**}**

System**.**out**.**println**(**"cost: " **+** drive**(**cost**,** dis**,** c**,** d**,** das\_dis**,** p**));**

**}**

public boolean isSolution**(**float maxD**,** float**[]** dd**)** **{**// 判断加油站距离是否合理

boolean b **=** **true;**

**for** **(**int i **=** 0**;** i **<** dd**.**length **-** 1**;** i**++)** **{**

**if** **(**maxD **<** dd**[**i **+** 1**]** **-** dd**[**i**])** **{**

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

b **=** **false;**

**}**

**}**

**return** b**;**

**}**

public float drive**(**float cost**,** float dis**,** float c**,** float d**,** float**[]** dd**,** float**[]** p**)** **{**

float now\_oil **=** 0**;**// 当前油量

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

// 加油方案

**if** **(**i **<** dd**.**length **-** 1**)** **{**// 不到最后一站

**if** **(**p**[**i**]** **<** p**[**i **+** 1**])** **{**// 比下一站便宜直接加满

cost **+=** **(**c **-** now\_oil**)** **\*** p**[**i**];**

now\_oil **=** c**;**// 加满

**}** **else** **{**// 比下一站贵则够用就行

**if** **((**now\_oil **\*** d**)** **<** **(**dd**[**i **+** 1**]** **-** dd**[**i**]))** **{**// 不够用加到刚好

cost **+=** **(((**dd**[**i **+** 1**]** **-** dd**[**i**])** **/** d**)** **-** now\_oil**)** **\*** p**[**i**];**

now\_oil **=** **(**dd**[**i **+** 1**]** **-** dd**[**i**])** **/** d**;**

**}**

**}**

**}**

**if** **(**i **==** dd**.**length **-** 1**)** **{**// 最后一站够用就行

**if** **((**dis **-** dd**[**i**])** **>** now\_oil **\*** d**)** **{**// 不够用加到刚好

cost **+=** **(((**dis **-** dd**[**i**])** **/** d**)** **-** now\_oil**)** **\*** p**[**i**];**

**}**

**}**

// 开车耗油

**if** **(**i **!=** dd**.**length **-** 1**)** **{**

now\_oil **-=** **(**dd**[**i **+** 1**]** **-** dd**[**i**])** **/** d**;**

**}** **else** **{**

now\_oil **-=** **(**dis **-** dd**[**i**])** **/** d**;**

**}**

**}**

**return** cost**;**

**}**

**}**

#### 2.3.2.2 黑白点的匹配

**import** java**.**lang**.**reflect**.**Array**;**

**import** java**.**util**.**Arrays**;**

**import** java**.**util**.**Scanner**;**

**import** Unit2**.**student**;**

public class MaxMatching **{**// 基于DFS

static int graph**[][];** // 邻接表 默认全为0

static int n**;** // 节点数

static int visit**[];** // 是否访问

static int matched**[];** // 是否已经匹配，对应的匹配点

static vertex V**[];**//// 结构体数组储存所有黑白

public class vertex **{**// 顶点结构体

public int color**;**// 白:0 黑：1

public double Vx**;**

public double Vy**;**

**}**

public void Init**()** **{**

System**.**out**.**println**(**"输入的黑白点总数为："**);**

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

n **=** sc**.**nextInt**();**

graph **=** **new** int**[**n**][**n**];** // 邻接表 默认全为0

visit **=** **new** int**[**n**];** // 是否访问

matched **=** **new** int**[**n**];** // 是否已经匹配，对应的匹配点

V **=** **new** vertex**[**n**];**

InitGraph**();**// 初始邻接矩阵

**}**

private void InitGraph**()** **{**

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

**for** **(**int i **=** 0**;** i **<** n**;** i**++)** **{**// 输入黑白点

V**[**i**]** **=** **new** vertex**();**

System**.**out**.**println**(**"please int color/x/y"**);**

V**[**i**].**color **=** sc**.**nextInt**();**

V**[**i**].**Vx **=** sc**.**nextDouble**();**

V**[**i**].**Vy **=** sc**.**nextDouble**();**

**}**

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

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

**if** **(**i **!=** j **&&** **(**V**[**i**].**color **==** 1**)** **&&** **(**V**[**j**].**color **==** 0**)** **&&** **(**V**[**i**].**Vx **>** V**[**j**].**Vx**)** **&&** **(**V**[**i**].**Vy **>** V**[**j**].**Vy**))** **{**

graph**[**i**][**j**]** **=** 1**;**

**}**

**}**

**}**

**}**

// 显示匹配结果

public void show**()** **{**

Arrays**.**fill**(**visit**,** 0**);**

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

**if** **(**visit**[**i**]** **==** 0**)** **{**

**if** **(**matched**[**i**]** **!=** **-**1**)** **{**

System**.**out**.**println**(**"(" **+** V**[**i**].**Vx **+** "," **+** V**[**i**].**Vy **+** ")与" **+** "(" **+** V**[**matched**[**i**]].**Vx **+** ","

**+** V**[**matched**[**i**]].**Vy **+** ")" **+** "匹配"**);**

visit**[**i**]** **=** 1**;**

visit**[**matched**[**i**]]** **=** 1**;**

**}**

**}**

**}**

**}**

/\*

\* dfs实现, params: x:起始的未匹配点 return: 1:找到增广路 0:未找到

\*/

public int dfs\_solve**(**int x**)** **{**

// 找到一个和节点存在边的点，并且该点在本轮中没有被访问过

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

**if** **(**graph**[**x**][**i**]** **!=** 0 **&&** visit**[**i**]** **==** 0**)** **{**

visit**[**i**]** **=** 1**;** // 标记为匹配过

// 按照交替路的模式找增广路，增广路相对于交替路的特性是就是，第一个节点和最后一个节点都是未匹配过的节点

**if** **(**matched**[**i**]** **==** **-**1 **||** dfs\_solve**(**matched**[**i**])** **==** 1**)** **{** // 直接跳到matched[i]能够保证匹配边和未匹配边交替

// 说明找到了一个未匹配节点，将所有匹配边变为未匹配边，将所有未匹配边变为匹配边，这样匹配边数会加1,这个交换过程通过回溯实现

matched**[**x**]** **=** i**;**

matched**[**i**]** **=** x**;**

System**.**out

**.**println**(**"(" **+** V**[**x**].**Vx **+** "," **+** V**[**x**].**Vy **+** ") 与 " **+** "(" **+** V**[**i**].**Vx **+** "," **+** V**[**i**].**Vy **+** ")" **+** "匹配"**);**

**return** 1**;**

**}**

**}**

**}**

**return** 0**;**

**}**

public void hungarian1**()** **{**

Arrays**.**fill**(**matched**,** **-**1**);**

int sum **=** 0**;**

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

**if** **(**matched**[**i**]** **==** **-**1**)** **{**

System**.**out**.**println**(**"从 " **+** "(" **+** V**[**i**].**Vx **+** "," **+** V**[**i**].**Vy **+** ")" **+** " 开始匹配"**);**

Arrays**.**fill**(**visit**,** 0**);**// 重置浏览数组，用来浏览邻接矩阵当前列

sum **+=** dfs\_solve**(**i**);**

**}**

**}**

System**.**out**.**println**(**"\n"**+**"最后共有 " **+** sum **+** " 匹配项"**);**

show**();**

**}**

public static void main**(**String**[]** args**)** **{**

MaxMatching mm **=** **new** MaxMatching**();**

mm**.**Init**();**

mm**.**hungarian1**();**

**}**

**}**

### 2.3.3 复杂度分析及代码优化

**加油问题复杂度**：**O（n）**

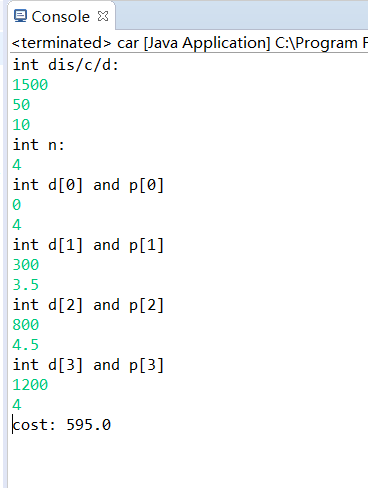
通过一次遍历即可完成整个过程，n为加油站数量

**黑白点的匹配复杂度：O（V+E）**

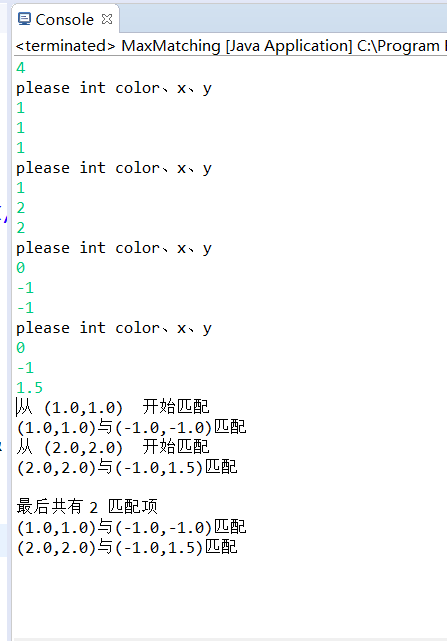
需要借助一个递归工作栈，故它的空问复杂度为O(V）。 遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所采用结构。 邻接表表示时，查找所有顶点的邻接点所需时间为O(E)，访问顶点的邻接点所花时间为O（V）,此时，总的时间复杂度为O(V+E)。

### 2.3.4 实现结果

#### 2.3.4.1 加油问题



#### 2.3.4.2 黑白点的匹配



# 3、实验总结

通过一次实验，稍微将问题细化扩展，就能发现很多问题，逐一去学习归纳对比，能学到很多东西，并在实验中加以运用。

对DFS和BFS的总结：

1、两者实现借助的数据结构不同：

DFS借助了邻接表 BFS借助的是队列，但两者为了部分重复查看一个点都使用了布尔类型数组。

2、思路不同：

DFS是一口气走到底，以递归的方式，直到不能再走为止

BFS像是往水中扔石子，水波荡开。

1. 相同点：

两种方法若是从一点出发，有时候并不能遍历整个图，因为图像可能是不连通图。解决方法是做出一个循环，每个点都出发一次（条件语句将跳过已经查看的点）

写在结尾：

看完同学的代码后发现他没使用最大匹配算法，但是又不出错，我就纳闷了，找了多个反例未果后重新仔细思考了问题，发现此题中黑白点的数目是相同的，所以具有一定的局限性，不需要最大匹配也能完成。在某种程度上我反而错了，但是学习重要的是过程，当黑白点数目不同时最大匹配算法仍然可以解决，在很多实际问题中也需要最大匹配，所以花时间学习还是很值得。