**华南农业大学**

**《算法分析与设计》  
课程实验**

专业年级： 2014级信息与计算科学3班

学生学号： 201430120308

学生姓名： 何 唯

实验题目： 部落卫队问题(回溯法)

指导老师： 赵 峰

实验时间： 2017/05/07

1. **实验题目**

• [题目描述]

原始部落Byteland中的居民们为了争夺有限的资源，经常发生冲突。几乎每个居民都有他的仇敌。部落酋长为了组织一支保卫部落的队伍，希望从部落的居民中选出最多的居民入伍，并保证队伍中任何两个人都不是仇敌。

• [输入说明]

仇敌关系

• [输出说明]

组成部落卫队的最佳方案.

• [实验要求]

1.至少运行3组不同的输入数据；

2.用回溯法求解.

实验报告应包括

1. 题目；

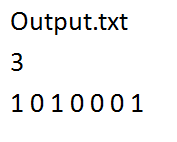
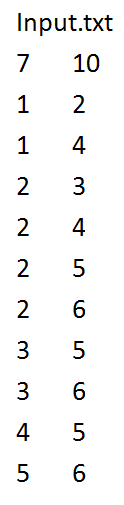
2. 解决思想、方法阐述;

3. 实验结果截图;

4. 算法分析(时间复杂度推导)；

5. 小结和心得；

6. 源程序清单.

****

**数据输入：**

由文件input.txt给出输入数据。

第一行给出居民数n和仇敌关系数m；

下面的m行给出每一个仇敌关系，

仇敌关系u和v表示居民u和v是仇敌。

**结果输出：**

将计算结果输出到output.txt中。

第一行给出部落卫队人数

第二行是卫队组成，1代表对应居民

在卫队中，0代表对应居民不在卫队中。

**实验实现**

**• 1．**[编程语言]:实验用Java完成算法设计和程序设计并调试通过。

**• 2．**[解题思想、方法]

**符号说明**

|  |  |  |
| --- | --- | --- |
| **符号** | **数据类型** | **含义** |
| Intdata | Int[][] | 仇敌关系矩阵 |
| Total | int | 关系总数 |
| Best | int | 问题最优解(卫队人数) |
| Currentbest | int | 当前最优解 |
| Bestplan | Int[] | 最优计划 |
| Currentplan | Int[] | 当前计划 |
| Size | Int | 总居民人数 |
| Count | Int | 特定居民仇敌个数 |

**算法描述**

其实本问题可以通过回溯法解决，与最大团问题的解法相似。回溯法的解题步骤主要包含三个步骤：

1. **定义对应问题的解空间；**
2. **确定易于搜索的解空间结构；**
3. **以深度优先方式搜索解空间，并在搜索过程中用剪枝函数(约束函数、限界函数)避免无效搜索，优化算法。**

回溯法在本题的思想主要为，确定部落卫队解空间的组织结构，此处可以定义为从全集(所有部落成员)中选择子集（部分成员）的问题，问题的解可以表示为子集形式，所以可以表示为组合数。也就是说解空间定义为子集树。每次深度搜索到叶子节点的话，可以输出一个可行解，即一个可行的部落卫队方案。

此外本题类似于无向图，有边的2个点是双向连通，查找方案时，可以从编号小的开始往后查找不冲突的居民，每增加一个居民，保证该居民与这套方案已有的居民都不冲突。然后找到一套方案后，如果总居民数增多，就把这个方案保存下来。最后，输出最大居民数的方案。后面得到的居民仇恨关系矩阵也相当于图中的邻接矩阵，但是本问题还是基于问题本身进行讨论，具体如下：

回溯法在根节点出发**(此处在后面算法优化提出一种优化方法)**，以深度优先的搜索方式历遍整个解空间。当前节点成为活接垫，同时也成为当前的扩展节点。在当前扩展节点出，搜索向纵深方向移至一个新节点(下一个居民)。此新节点成为新的活节点，并成为当前扩展节点。若在当前扩展节点处不能再向纵深方向移动，则当前扩展节点就成为死节点。此时，应往回移动(回溯)至最近的活节点处，并使这个活节点成为当前扩展节点。

**具体方法**

回溯法在本程序中以递归函数的形式实现**(BackTrack()函数)**。首先，对于每次输入的**Input.txt**文件来说，先进行预处理，将输入的数据转化为一个仇敌关系矩阵(n×n),其中n为居民总数。然后，将输入编号为1的居民默认为解空间树的根节点，并从此根节点开始深度遍历，进行子集选取。在搜索子树的时候，以**if**(*currentbest*+*size*-currentdeep<=*best*)作为限界函数，判断子树价值的上界是否能大于当前最优解，若上界能大于当前最优解则继续遍历子树，否则的话将跳出本次递归，也就是剪去该子树。此限界函数与0-1背包问题相似，有如下三个特征：

1. **非搜索得到的，是由估计得到的；**
2. **该限界函数是找对应子树的上界，在于当前最优解best进行比较；**
3. **分支越小越好(深度越小越好)，因为这样可以使得不是最优解的方案马上被剔除。**

其次，以代码块**if**(currentdeep==0&&*Intdata*[currentdeep][i]!=1)

{

flag=**true**;

**break**;

}

**else** **if**(*Intdata*[currentdeep][i]!=1)//节点是否与所有人不为敌

flag=**true**;

**else** **if**(*currentplan*[i]==1&&*Intdata*[currentdeep][i]==1)

{

//与已选节点是否为敌

flag=**false**;

**break**;

}

设置flag作为约束函数，判断是否能继续深度搜索，否则就减去该子树。以currentdeep>*size*-1作为可行解判断函数，判断当前深度是否到达叶子节点，假如到达则输出可行解，否则继续深度遍历。

**• 3．**[实验说明，包括输入、输出、结果截图]

**本实验测试数据，在题目要求的三个测试输出的基础上，添加了两个有代表性的测试用例，具体如下：**

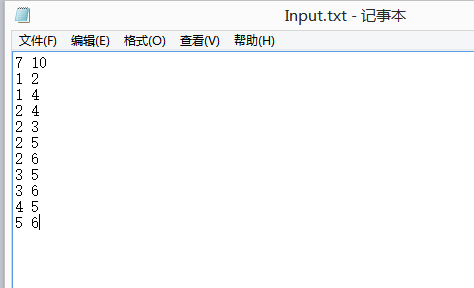
**测试一：**

输入：居民数为7，仇敌关系数为10，以下为仇敌关系:(1,2)(1,4)(2,4)(2,3)

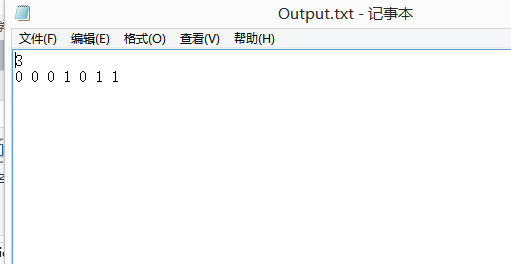
(2,5)(2,6)(3,5)(3,6)(4,5)(5,6)

输出：3

0 0 0 1 0 1 1

****

**图一 测试一输入**

****

**图二 测试一输出**

****

**图三 测试一仇敌关系矩阵**

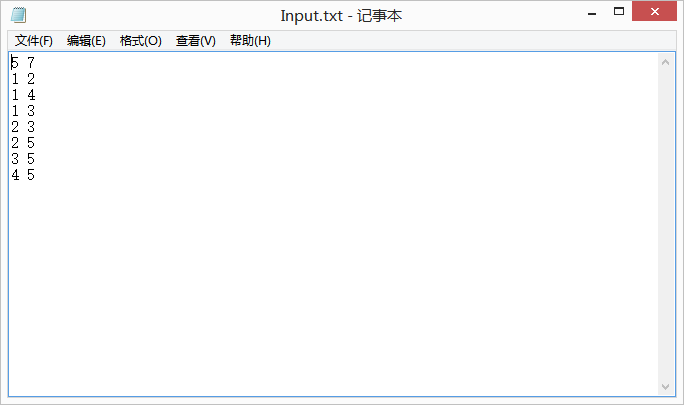
**测试二：**

输入：居民数为5，仇敌关系数为7，以下为仇敌关系:(1,2)(1,4)(1,3)(2,3)

(2,5)(3,5)(4,5)

输出：2

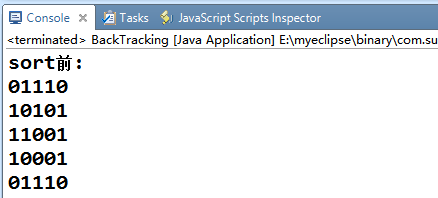
0 0 1 0 1

****

**图四 测试二输入**

****

**图五 测试二输出**

****

**图六 测试二仇敌关系矩阵**

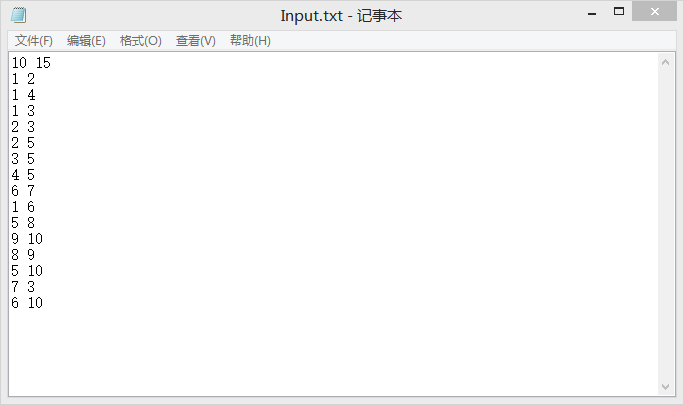
**测试三：**

输入：居民数为5，仇敌关系数为7，以下为仇敌关系:(1,2)(1,4)(1,3)(2,3)

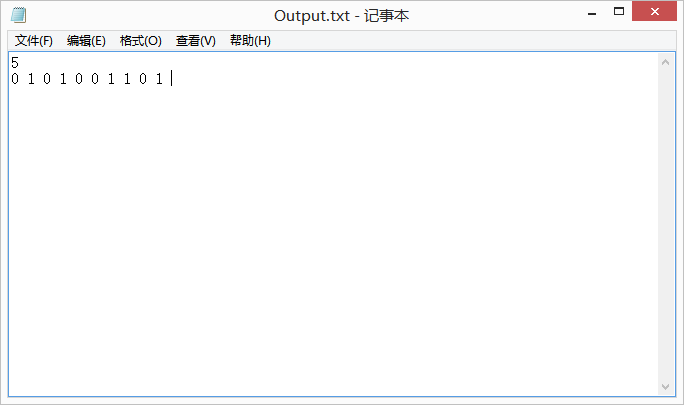
(2,5)(3,5)(4,5)(6,7)(1,6)(5,8)(9,10)(8,9)(5,10)(7,3)(6,10)

输出：5

0 1 0 1 0 0 1 1 0 1



**图七 测试三输入**

****

**图八 测试三输出**

****

**图九 测试三仇敌关系矩阵**

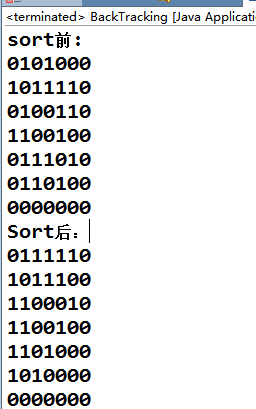
**算法优化：**

下面通过一个sort()函数按照居民仇敌关系数目多少对其进行排序，以测试一的数据为例：

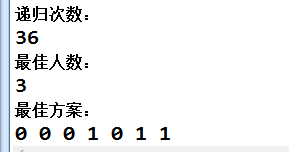
即输入：居民数为7，仇敌关系数为10，以下为仇敌关系:(1,2)(1,4)(2,4)(2,3)(2,5)(2,6)(3,5)(3,6)(4,5)(5,6)

输出：3

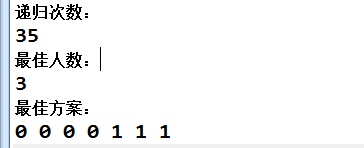
0 0 0 1 0 1 1

****

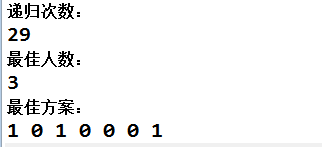
**图十 对比Sort()函数前后仇敌关系矩阵**

****

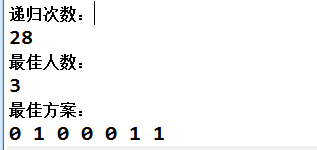
**图十一 没有剪枝函数 没有优化解空间结构**

****

**图十二 没有剪枝函数 有优化解空间结构**

****

**图十三 有剪枝函数 没有优化解空间结构**

****

**图十四 有剪枝函数 有优化解空间结构**

**图十四 对比优化后递归次数**

从上述结果可以看到，决定一个便于搜索的解空间树对于问题求解的递归次数优化有一定的作用，虽然在本测试例中作用效果不够显著，但是我相信对于问题效率的提升还是有一定作用的。同时，我们也可以看到，使用剪枝函数前后的递归次数变化结果显著。因为假如判断了该居民是否有资格成为部落卫队成员，那么后续与他为敌对的居民就不需要进行比较了，可以删去对应的分支。说明剪枝函数对于回溯法的优化是起着决定性作用的。在本问题中，将具有仇敌数目较多的居民放在深度比较小的层比较合适，这样的话能够更快的得到问题一个可行解，甚至最优解。

**• 4．**[算法复杂性计算过程]

**4.1 时间复杂性**

对于本问题部落卫队来说，假如事先输入的居民顺序是没有按照得知的仇恨总数进行排序的话，程序需要调用Sort()方法对居民进行排序，仇敌数目多的将排在考前的位置。该排序算法，这里使用的是冒泡排序时间复杂度为(也可以使用快速排序等)。因为每次交换居民元素的话时间复杂度为，调用swap()函数。加上本身冒泡排序的时间复杂度可以得到整个Sort()函数时间复杂度。

假如不使用sort(),只进行深度历遍的话，部落卫队的最坏情况下时间复杂度与0-1背包问题以及最大团问题一样都是。

综上所述，本问题最坏情况下时间复杂度为

**4.2空间复杂性**

从上面算法可知 ，用到了三个数组 ，分别为仇恨关系矩阵Intdata[n][n]，currentplan[n],bestplan[n]分别存储当前、该问题最优方案。空间复杂性主要由Intdata[n][n]决定，为 0(n×n)，即。因此，该算法的空间复杂性为，可以应用到即使是小内存、低速度的计算机上。因此本算法有广阔的应用前景。

**小结和心得**

[例如：设计中遇到的问题，如何解决，实验心得，对算法改进的设想等]

* 本次算法实验考察的是对回溯法的应用以及设计。设计中遇到的问题首先是问题解空间的构造，现实问题的解空间构造多种多样，最常见的就是子集树以及排列树。一个正确的解空间对问题求解有着至关重要的作用。假如你构造的解空间树并没有包含该问题真正的最优解，那么求解过程将变得没有任何意义的。
* 其次，就是在解空间中设置剪枝函数，通过剪枝函数对一些不可能得到最优解的子树剪去从而减少递归的次数，使得求解的过程得以优化。
* 实验的心得的话，就是在设计剪枝函数的时候，发现其实有必要去使解空间便于搜索，也就是解空间树的结构设置，由根节点开始，应该是仇恨数目较多的居民放在靠前面的位置，因为假如判断了该居民是否有资格成为部落卫队成员，那么后续与他为敌对的居民就不需要进行比较了，可以删去对应的分支。这个想法最终也得以实现，通过上述实验结果的截图也能看到是确实可行的，对于算法优化有着一定意义。
* 最后，说到算法的改进设想的话，首先是对排序算法进行一定的改进，如何选择合理的算法是一方面的解决措施。比如选择快速排序、插入排序等常见排序算法是需要对给定数据分析来选择的，比如数据给出的时候是完全倒序，这样用快速排序是比较麻烦的。当数据量比较大的时候也是可以考虑桶排序等等。
* 另一方面就是对剪枝函数进行下一步优化设计，继续减掉不必要的子树，减少遍历子树的递归次数。
* 总的来说，这次实验学到的东西还是比较多的，已经是算法课的最后一次实验了。对于算法的设计有了自己相应的心得，从实验一到实验四均有本人独立完成，没有过多借鉴博客上他人的成果。最后来看收获也是很明显的，从对编程语言的进一步熟悉，到对算法时间空间复杂度分析，均有比较大的收获。
* 希望往后的学习中，即使实验课程结束，也要通过自学的方式自己实现课本或者一些常用的基础算法，继续增强自己的竞争力。

**源程序清单**

**BackTracking.java**

**package** Algorism;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.io.InputStreamReader;

**import** java.io.Writer;

/\*\*

\*

\* **@Project**：Algorithm

\* **@File**：BackTracking

\* **@Author**：何唯

\* **@Date**：2017年5月7日 下午10:28:12

\* **@Description**：

\*/

**public** **class** BackTracking {

**private** **static** **int**[][] *Intdata*=**null**;//仇敌关系矩阵

**private** **static** **int** *total*;//关系总数

**private** **static** **int** *best*;//问题最优解(卫队人数)

**private** **static** **int** *currentbest*;//当前最优解

**private** **static** **int**[] *bestplan*;//最优计划(每个人是否在卫队)

**private** **static** **int**[] *currentplan*;//当前计划

**private** **static** **int** *size*;//总人数

**private** **static** **int** *count*;//某人仇敌数目

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

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

//首先接收Input.txt文件数据

String[] Stringdata=**null**;

File file=**new** File("F:\\Desktop\\JAVA project eric\\Algorithm\\src\\Input.txt");

**try** {

FileInputStream in=**new** FileInputStream(file);

InputStreamReader read=**new** InputStreamReader(in);

BufferedReader bufferedReader=**new** BufferedReader(read);

String x=**null**;

x=bufferedReader.readLine();

Stringdata =x.split(" ");

*size*=Integer.*parseInt*(Stringdata[0]);

*Intdata*=**new** **int**[*size*][*size*];

*total*=Integer.*parseInt*(Stringdata[1]);

//设置仇敌关系矩阵

**int** x1,x2;

**while**((x=bufferedReader.readLine()) != **null**)

{

Stringdata =x.split(" ");

x1=Integer.*parseInt*(Stringdata[0]);

x2=Integer.*parseInt*(Stringdata[1]);

*Intdata*[x1-1][x2-1]=1;

*Intdata*[x2-1][x1-1]=1;

}

System.*out*.println("sort前:");

**for**(**int**[]i:*Intdata*){

**for**(**int** j:i){

System.*out*.print(j);

}

System.*out*.println();

}

*Sort*();//首先构造易于搜索的解空间结构

System.*out*.println("Sort后：");

**for**(**int**[]i:*Intdata*){

**for**(**int** j:i){

System.*out*.print(j);

}

System.*out*.println();

}

*currentbest*=0;//初始化当前最优

*best*=0;//初始化总体最优

*currentplan*=**new** **int**[*size*];//初始化当前计划

*bestplan*=**new** **int**[*size*]; //初始化最优计划

//开始递归回溯求解

*BackTrack*(0);//根节点算深度为0，开始深度优先搜索树

read.close();

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

//输出结果

FileWriter output;

**try** {

output = **new** FileWriter("F:\\Desktop\\JAVA project eric\\Algorithm\\src\\Output.txt");

BufferedWriter bf = **new** BufferedWriter(output);

bf.write(*best*+" \n");

bf.newLine();

**for**(**int** t:*bestplan*)

bf.write(t+" ");

bf.flush();

} **catch** (IOException e) {

// **TODO** Auto-generated catch blockf

e.printStackTrace();

}

System.*out*.println("最佳人数：");

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

System.*out*.println("最佳方案：");

**for**(**int** t:*bestplan*)

System.*out*.print(t+" ");

}

//对居民按仇敌数目从多到少排序

**private** **static** **void** Sort() {

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

/\*for(int j:Separate)

System.out.println(j);\*/

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

{

**for**(**int** j=i;j<*size*;j++)

**if**(*Separate*(i)<*Separate*(j))

*swap*(i,j);//交换两个居民对应关系

}

}

//计算特定位居民仇敌个数

**private** **static** **int** Separate(**int** index) {

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

**int** c=0;

**for**(**int** temp:*Intdata*[index])

{

**if**(temp==1)

c++;

}

**return** c;

}

//交换两位居民位置

**private** **static** **void** swap(**int** i, **int** j) {

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

**int** temp1,temp2;

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

{

**if**(k!=i&&k!=j)

{

temp1=*Intdata*[i][k];

temp2=*Intdata*[k][i];

*Intdata*[i][k]=*Intdata*[j][k];

*Intdata*[k][i]=*Intdata*[k][j];

*Intdata*[j][k]=temp1;

*Intdata*[k][j]=temp2;

}

}

}

//回溯法求解

**private** **static** **void** BackTrack(**int** currentdeep) {

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

*count*++;

System.*out*.println("当前深度：");

System.*out*.println(currentdeep);

**boolean** flag=**false**;

//到达叶子节点的话输出可行解

**if**(currentdeep>*size*-1)

{

//System.out.println("输出可行解");

*best*=*currentbest*;

**for**(**int** k=0;k<*currentplan*.length;k++)

*bestplan*[k]=*currentplan*[k];

**return**;

}

//限界函数，减去得不到最优解(小于等于当前最优解的)子树

**if**(*currentbest*+*size*-currentdeep<=*best*)

{

//System.out.println("剪枝函数");

**return**;

}

//设置flag进行判断是否进入左子树

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

{

//是根节点的话是否与其他节点不为敌

**if**(currentdeep==0&&*Intdata*[currentdeep][i]!=1)

{

flag=**true**;

**break**;

}

**else** **if**(*Intdata*[currentdeep][i]!=1)//节点是否与所有人不为敌

flag=**true**;

**else** **if**(*currentplan*[i]==1&&*Intdata*[currentdeep][i]==1)

{

//与已选节点是否为敌

flag=**false**;

**break**;

}

}

**if**(flag==**true**)//进入左子树

{

System.*out*.println("左");

*currentplan*[currentdeep]=1;

*currentbest*++;

*BackTrack*(currentdeep+1);//递归深度遍历子树

*currentbest*--;

*currentplan*[currentdeep]=0;

}

//进入右子树

**if**(*currentbest*+*size*-currentdeep>*best*)

{

*currentplan*[currentdeep]=0;

*BackTrack*(currentdeep+1);

}

System.*out*.println("递归次数：");

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

}

}

**五、评语及评分**

|  |
| --- |
| **评语及评分**  评阅人签名： |