

**人工智能实验报告**

题 目 人工智能实验

专 业 计算机科学与技术

学　　 号 1160300426

学 生 李国建

指 导 教 师 李钦策

同 组 人 员 安宏展、王宇楠、姚远、李忠旭、鲁东祐

**实验1知识表示**

1. **简介**
   1. 待解决问题的解释

一个房间里，天花板上挂有一串香蕉，有一只猴子可在房间里任意活动（到处走动，推移箱子，攀登箱子等）。设房间里还有一只可被猴子移动的箱子，且猴子登上箱子时才能摘到香蕉，问猴子在某一状态下（设猴子位置为A，香蕉位置在B，箱子位置为C），如何行动可摘取到香蕉。

* 1. 问题的形式化描述

该问题的本质就是利用一种知识表示的方法来解决猴子摘香蕉的问题。如果把猴子、香蕉和箱子的位置用{-1.0.1}里的数字表示（对应为A,B,C），那么就可以用一个三元组来表示问题的状态，任意输入一个合法的状态，即可输出一个从初始状态到目标状态的过程。实际实现需要使用四元组，还需要记录猴子与箱子的相对位置。

* 1. 解决方案（原理）

按照题目要求，选择产生式表示法。

首先定义一个综合数据库，即包含所有状态的状态空间（monkey, box, banana, monbox），分别表示猴子、香蕉和箱子的位置，用{-1.0.1}表示。

定义猴子的行动规则，即规则数据库：走到X位置处；跳到箱子上；从箱子上跳下来；将箱子移到另一个位置；如果站在箱子上且有香蕉，则可以摘香蕉。

根据问题，确定终止状态是猴子摘到香蕉{（x,x,x,0）}（x 属于 {0，-1, 1}）。

输入一个初始状态（a, b, c, d），即可输出一个从初始状态到目标状态的解决方案。

1. **算法介绍**

2.1 所用方法的一般介绍

产生式表示法，首先定义一个综合数据库；再确定初始状态和目标状态；然后定义规则库；根据初始状态和规则库中的规则，不断地产生新的状态，直到到达目标状态为止。

2.2 算法伪代码

输入：猴子、香蕉、箱子的位置以及猴子是否站在箱子上（a, b, c, d）

输出：猴子摘香蕉的过程

if state.monkey == state.box and state.box == state.banana and state.monbox==1:

print ("步数" + str(step) + ":猴子将香蕉摘下来")

os.\_exit(0)

if state.monbox==1: #猴子在箱子上面站着

if state.box!=state.banana: #箱子和香蕉不在一起

state = MonkeyUp(MoveBox(MonkeyDown(state,step),step+1),step+2)

step = step + 3

if state.monbox==-1: #猴子不在箱子上面，此时猴子和箱子的位置不一定在一起

if state.box!=state.banana: #箱子和香蕉不在一起

if state.monkey!=state.box: #猴子和箱子不在一起

MonkeyUp(MoveBox(MoveMonkey(state,step),step+1),step+2)

step = step + 3

else: #猴子和箱子在一起

MonkeyUp(MoveBox(state,step),step+1)

step = step + 2

else: #箱子和香蕉在一起

if state.monkey!=state.box:

MonkeyUp(MoveMonkey(state,step),step + 1)

step = step + 2

else:

MonkeyUp(state,step)

step = step + 1

print ("步数" + str(step) + ":猴子将香蕉摘下来")

1. **算法实现**

3.1 实验环境与问题规模

实验环境：Windows10，Python3，VSCode；

问题规模：最多四步

3.2 数据结构

定义状态类：

class State:

def \_\_init\_\_(state, monkey=-1,box=0,banana=1,monbox=-1):

state.monkey = monkey

state.box = box

state.banana = banana

state.monbox = monbox

3.3 实验结果

输入0 1 -1 -1

输出：

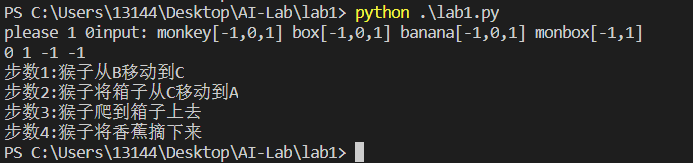
步数1:猴子从B移动到C

步数2:猴子将箱子从C移动到A

步数3:猴子爬到箱子上去

步数4:猴子将香蕉摘下来

3.4 系统中间及最终输出结果（要求有屏幕显示）



1. **总结及讨论**

该实验让我们了解了利用知识表示方法解决实际问题。在综合了各个表示方法的优缺点后，我们最终选择了产生式的表示方法。构建了产生式的组成（综合数据库、规则库以及控制系统），并定义了初始状态和目标状态。最终产生了一个从初始状态到目标状态的一条路径。

**实验2搜索策略**

1. **简介**
   1. 待解决问题的解释

采用且不限于课程第四章内各种搜索算法此编写一系列吃豆人程序解决以下列出的问题1-8，包括到达指定位置以及有效的吃豆等。在这些已经设计好的框架中需要做的就是实现不同的图的搜索算法。并通过策略选择以最少的步数，在搜索最少节点的情况下达成问题目标。

问题1：应用深度优先算法找到一个特定的位置的豆

问题2：宽度优先算法

问题3：代价一致算法

问题4：A\*算法

问题5：找到所有的角落

问题6：角落问题（启发式）

问题7：吃掉所有的豆子

问题8：次最优搜索

* 1. 问题的形式化描述

问题1-4，分别利用一个搜索算法让吃豆人选择一条合适的路径进行吃豆。分别是深度优先搜素、宽度优先搜索、代价一致算法和A\*算法。然后利用这四个搜索算法去解决各种搜索问题。问题5，要求找到一条访问所有四个角落的最短路径；问题6，构建启发函数，进行角落搜索问题；问题7，用尽可能少的步数吃掉所有的豆子；问题8，定义一个优先吃最近的豆子函数是提高搜索速度的一个好的办法。

* 1. 解决方案（原理）

问题一：

本问题要求编写以深度优先遍历搜索迷宫的算法。在题目描述中提示了在util.py中已经实现了可能用到的栈，队列，优先队列。根据栈后进先出的特点只有顺着一条路径到达无法再扩展的节点才会遍历上一层的兄弟节点，选择栈作为数据结构编写deepfirstSearch函数。

分别用序列记录当前已经过的路径（movements），遍历过的节点（visited）。开始时首先将问题的初始节点与空的movements作为元组压入栈中。之后进入循环，弹出当前栈顶元素，将其中节点部分加入到visited序列中。如果当前节点为目标节点返回路径movements。否则，保存路径部分。之后遍历刚才弹出节点的所有后继节点，如果其还没有被遍历过（即还没有被加入visited中），那么将该节点与相应更改的movements压入栈中。本轮循环结束。若直到栈空仍没有到达目标节点则返回一个空的序列。

问题二：

与问题一类似。该问题用队列实现广度优先遍历函数。算法与问题一基本相同，利用了队列的先进先出的特点，在上述算法的结构下只有当上一层的节点全部遍历后，才会遍历下一层节点。

这里选择使用 util 的 Queue 队列数据结构实现这个算法，利用它先进先出 的性质。首先将根节点加入队列，然后扩展根节点，依此将它的子节点（如果没 有在 visitedStates 里面出现过的话）加入队列，每次取队首扩展，队列是由元 组（state,directin）组成的，保留了每个状态，和到这个状态的行为序列。

问题三：

与上述两个问题的实现算法有所不同。问题三用优先队列实现uniformCostSearch。代价一致搜索不关注路径经过的点，只关注最终所消耗的代价，每次选择代价最小的点扩展。

在算法中使用优先队列来存储状态。状态为一个三元组，其中第三部分为从开始节点到当前节点的代价。将其作为优先列的优先级。首先将初始状态插入队列。进入循环，每次从队列中弹出一个元素，若该状态为目标状态则循环结束。否则，判断当前节点是否遍历过，若还未遍历过则将其加入visited序列。若已经遍历过则跳过此次循环，否则将其所有还未遍历过的节点的后继节点以及代价加入队列，并记录这些后继节点的前驱为当前节点。当循环结束后可从目标节点开始利用记录的前驱，向前回溯整条路径。将路径翻转返回即可。

问题四：

本问题与前两个类似，利用优先队列，优先级为从当前节点到某一后继节点的代价与这个后继节点到目标节点的启发性函数的值。A\* 搜索是最佳优先搜索的一种形式，它对节点的评估是 f(n) = g(n) + h(n) .g(n)是到达此节点花费的代价，和 h(n):从该节点到目标节点需要花费 的代价(的估计值)。所以 f(n)是进过节点 n 的最小代价解的估计代价。所最先扩展 f(n) 最小的节点，事实上，这样当 h(n) 满足可采纳性和一致性的时候，A\* 算法是完备和最优的。

问题五：

这个问题要实现搜索所有角落的问题。

首先定义问题的状态为一个二元组，分别为当前的位置与所遍历过的角落。

GetStarState函数返回这个二元组。

isGoalState函数返回一个布尔值，判断已经遍历过的顶点个数是否为4.

getSuccessor函数从当前为位置开始朝四个方向移动，如果该位置不是墙且没有遍历过那么将其加入返回的序列中。

问题六：

本问题要求实现一致的启发函数，为上一问题中的cornerProblem在执行中可以扩展尽量少的节点。

启发性函数的启发策略为从当前位置依次向四个角落中还未被遍历过的且曼哈顿距离最近的距离值开始。并以此为起点继续上述动作，直至所有角落被遍历。将这些曼哈顿距离之和作为返回值。

问题七：

本问题目标实现解决FoodSearchProblem，即吃掉所有的迷宫中的食物。

问题中的启发性函数的启发策略为在当前状态下吃到下一个食物的最短距离。由于在阅读代码时发现在这个问题后有一个实现了的求在迷宫中的两点的距离的函数。于是对这个函数加以利用，启发函数的返回值为当前节点到所有剩余食物中距离最近的值。

问题八：

在这个问题中我按照题目提示的最简单的方式实现了这个问题。

首先补充完AnyFoodSearchProblem类中对于目标节点的测试部分。对当前剩余的所有食物进行遍历，找到距离当前状态最近的食物的曼哈顿距离，如果这个距离为0，那么该节点即为目标节点。

之后选取适当的搜索方式得到路径即可，这里我选用的是A\*算法。

1. **算法介绍**

2.1 所用方法的一般介绍

搜索算法的过程如下：

问题1-4：深度优先搜素的一般算法

1. 把初始节点放入open表中，建立一个close表，置为空；
2. 检查open表是否为空表，若为空，则问题无解，失败退出；
3. 把open表的第一个节点取出放入closed表，并记该节点为n；
4. 考察节点n是否为目标节点，若是则得到问题的解成功退出；
5. 若结点n不可扩展，则转第二步；
6. 扩展节点n，将其子节点放入open表的首部，并为每个子节点设置指 向父节点的指针，转向第二步。

宽度优先搜索的一般算法

（1）把初始节点放入open表中，建立一个close表，置为空；

（2）检查open表是否为空表，若为空，则问题无解，失败退出；

（3）把open表的第一个节点取出放入closed表，并记该节点为n；

（4）考察节点n是否为目标节点，若是则得到问题的解成功退出；

（5）若结点n不可扩展，则转第二步；

（6）扩展节点n，将其子节点放入open表的尾部，并为每个子节点设置指 向父节点的指针，转向第二步。

代价一致搜索的一般算法

1. 把初始节点放入open表中设g(s0)=0,建立一个closed表，置为空；
2. 检查open表是否为空表，若为空，则问题无解，失败退出；
3. 把open表的第一个节点取出放入closed表，并纪该节点为n;
4. 考察节点n是否为目标节点，若是则得到问题的解成功退出；
5. 若节点n步可扩展，则转第二步；
6. 扩展n节点，生成子节点ni(i=1,2,...),将其子节点放入open表，并为每个子节点设置指向父节点的指针，计算各个节点的代价g(ni),将open表内的节点按g(ni）从小到大排序，转向第二步。

全局择优搜索A\*算法描述

1. 把初始节点放入open表中，f(S0)=g(S0)+h(S0);
2. 如果open表为空，则问题无解，失败退出；
3. 把open表的第一个节点取出放入closed表，并记该节点为n；
4. 考察节点n是否为目标节点。若是，则找到了问题的解，成功退出；
5. 若节点n不可扩展，则转第二步；
6. 扩展节点n，生成其子节点ni(i=1,2,..),计算每一个节点的估计值f(ni)(i=1,2,..),并为每一个子节点设置指向父节点的指针，然后将这些子节点放入open表中；
7. 根据各节点的估价函数值，对open表中的全部节点按从小到大的顺序重新进行排序；
8. 转第二步。

问题5-8实现思想见上面。

2.2 算法伪代码

def generalSearch(problem, openTable, needCost=False, heuristic=None) :

if not needCost :

path = []

visited = []

openTable.push((problem.getStartState(), path))

while not openTable.isEmpty() :

state, path = openTable.pop()

if state in visited :

continue

visited.append(state)

if problem.isGoalState(state) :

return path

for successor, direction, cost in problem.getSuccessors(state) :

if not successor in visited :

openTable.push((successor, path + [direction]))

return []

else :

visited = []

path = []

parent = {}

startState = problem.getStartState()

if not heuristic == None :

openTable.push((startState, path), 0)

while not openTable.isEmpty():

state = openTable.pop()

if problem.isGoalState(state[0]):

return state[1]

if state[0] not in visited:

visited.append(state[0])

successors = problem.getSuccessors(state[0])

for successor, action, cost in successors:

tempActions = state[1] + [action]

if successor not in visited:

openTable.push((successor,tempActions), problem.getCostOfActions(tempActions) + heuristic(successor,problem))

return []

else :

openTable.push((startState, None, 0))

parent[(startState, None)] = None

while not openTable.isEmpty() :

state = openTable.pop()

if problem.isGoalState(state[0]) :

break

else :

if state[0] in visited :

continue

visited.append(state[0])

for successor, direction, cost in problem.getSuccessors(state[0]) :

if successor not in visited :

openTable.push((successor, direction, cost + state[2]))

parent[(successor, direction)] = state

child = state

while (child != None):

path.append(child[1])

if child[0] != problem.getStartState():

child = parent[(child[0],child[1])]

else:

child = None

path.reverse()

return path[1:]

1. **算法实现**

3.1 实验环境与问题规模

实验环境：Windows10，Python2，VSCode

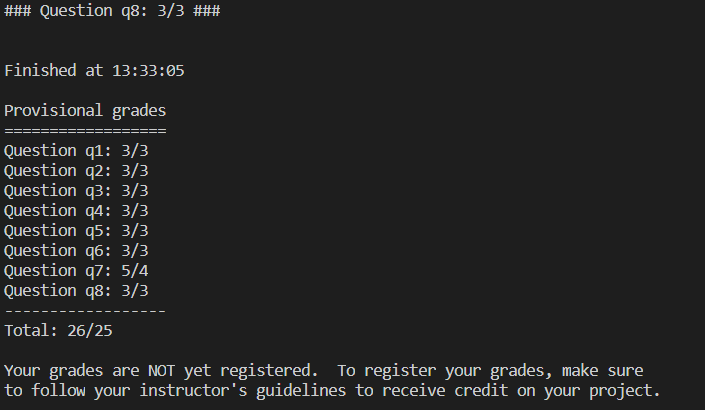
问题规模：规模为整个地图

3.2 数据结构

包括栈，队列，优先队列等

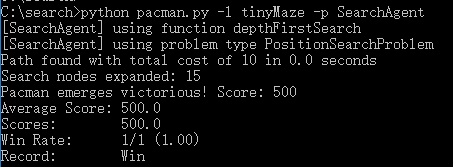
3.3 实验结果

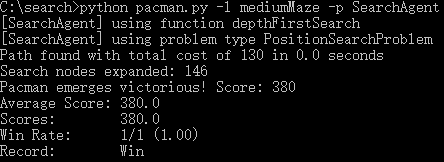
利用测试代码进行打分，结果如下：

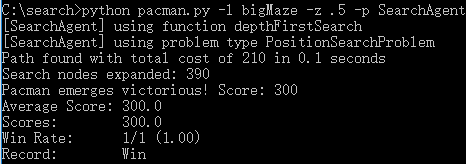


3.4 系统中间及最终输出结果（要求有屏幕显示）

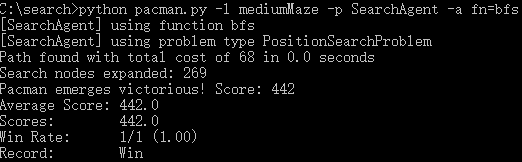
问题一结果：

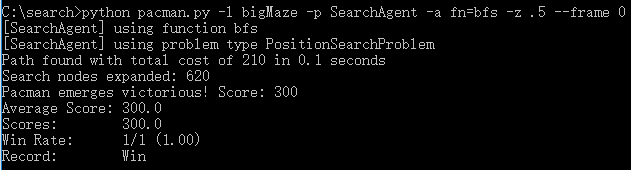




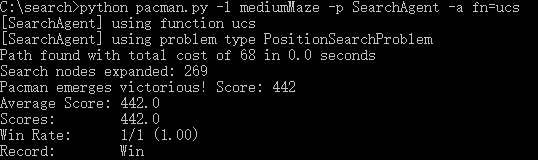


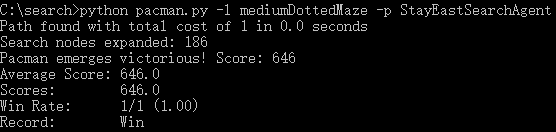
问题二结果：

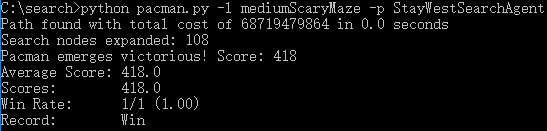




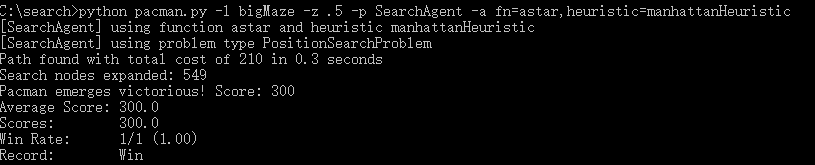
问题三结果：



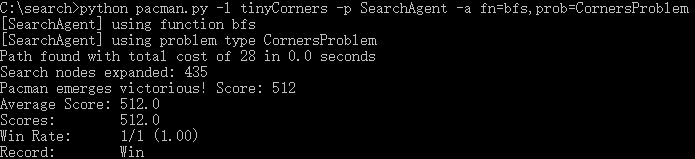


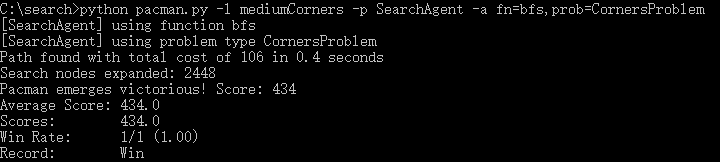


问题四结果：

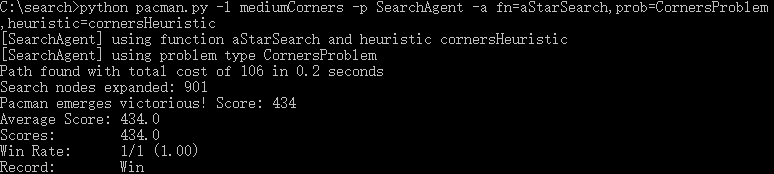


问题五结果：

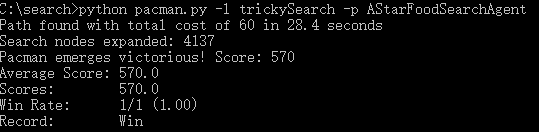




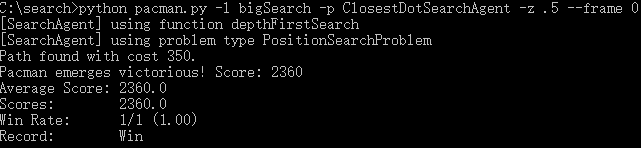
问题六结果：



问题七结果：



问题八结果：



1. **总结及讨论**

本次实验可以说是一次很有意思的实验了，看到吃豆人按照自己写的算法来搜索吃豆的路径，感觉很有成就感。本次四个搜索算法，由于仅仅是Open表中的排序方式和代价的计算有区别外，其他大部分结构都是一样的，因此将四个算法写在了一起，组合成了一个函数。后面四个问题是在前四个搜索算法已经写好的条件下计算的，根据不同的要求，填写代码不同的部分并选择合适的搜索算法。最后实现一个满足条件的算法。

**实验3不确定性推理**

1. **简介/问题描述**
   1. 待解决问题的解释

参照课程第五部分讲授的贝叶斯网络完成，给定事件和事件之间的关系，并且给出每个事件的CPT图，根据贝叶斯公式根据上述条件求出目标概率，编写程序实现基于贝叶斯网络的推理。在这里用到的贝叶斯算法是建立在有向无环图和CPT表的技术上实现的。

* 1. 问题的形式化描述

根据输入文件给的概率矩阵，构建贝叶斯网络。然后读取问题文件，利用构建的贝叶斯网络，计算出一个CPT表，然后读取解决问题文件中的概率问题，给出概率值。

* 1. 解决方案介绍（原理）

首先，利用读取文件的函数读取文件，然后构建一个贝叶斯网络图（使用geteverycpt函数，信息存储在储存在count,strings,q,p中），计算出一个CPT表。利用数据结构，计算并存储联合概率。然后读取问题文件（存储在arrayList2中），根据贝叶斯网络的公式，计算相应的联合概率，然后进行计算。将所有的全概率算出，利用穷举法，将各种情况相加，相除得到结果。

这里的重点是要算出所有的全概率。使用的枚举的方式，用两层循环，从第一个点开始计算，遍历每种情况。将全概率储存在pro中。一共要计算2^N次。

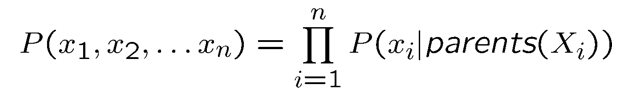
知道全概率后就可以求得具体问题的答案了。程序中使用getPT函数，在函数中调用comute函数，可以具体算出问题的概率，每个问题会有两个结果，一个是T，一个是F。在计算过程中，使用的是二进制数，T是1，F是0

1. **算法介绍**

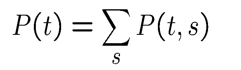
2.1 所用方法的一般介绍

利用输入的数据，构建CPT表，然后利用表中的数据进行计算，主要利用下面两个计算公式：

贝叶斯网络的联合概率公式：



计算概率：



2.2 算法伪代码

Note geteverycpt（file）函数:

{ Note xinxi； //一个新的数据结构，存储输入文本中的有效信息

xinxi.count=readline（）；//读入文本第一行的事件个数

String[] strings = file.readLine().split(" "); //读入所有随机事件名称

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

xinxi.arrayList.add(strings[i]);

}

xinxi.q = new int[count];//存父节点个数

xinxi.p = new int [count][count];//存节点关系

for(int i=0;i<count;i++){ //给p和q具体赋值

HashMap<String, Integer> map=new HashMap<>();

String[] tstrings= file.readLine().split(" ");

for(int j=0;j<tstrings.length;j++){

xinxi.p[i][j]=Integer.valueOf(tstrings[j]);//转化字符串为int类型

if(xinxi.p[i][j]==1){

xinxi.q[j]++;

if(!xinxi.arrayList.isEmpty()&&xinxi.arrayList.size()>j&&xinxi.arrayList.get(j)!=null){

xinxi.arrayList.get(j).put(arrayList.get(i),xinxi.q[j]);

}

else{

map.put(arrayList.get(i), xinxi.q[j]);

xinxi.arrayList.add(j,map);

}

}

}

}

xinxi.t = new double[(int)Math.pow(2,( count-1))][2][count]; //存cpt 0列为假 1列为真

for(int i=0;i<count;i++){ //给t赋值

for(int j=0;j<=((int)Math.pow(2,( xinxi.q[i]))-1);j++){

str4 = file.readLine();

String str3[]=str4.split(" ");

xinxi.t[j][0][i]=Double.valueOf(str3[0]);

xinxi.t[j][1][i]=Double.valueOf(str3[1]);

}

}

return xinxi; //返回储存完的信息

ArrayList<String> getquestion（file）函数：

{ ArrayList<String> arrayList；//一个字符串集合存储查询文件中的问题

while((string=file.readLine())!=null){

String[] strings=string.split("P\\(");//去掉“P（”

if (strings.length == 0) {

continue;

}

for(int i=1;i<strings.length;i++){

arrayList.add(strings[i].substring(0,(strings[i].length()-1)));//去掉“）”

}

}

return arrayList;

double[] allpro（Note note）函数：//第一个节点在二进制数中为最高位

{ int sum= int sum = Math.pow(2, note.count); //事件数为n时全概率个数为

for(int i = 0;i < sum;i++){

string = Integer.toBinaryString(i);//将i转换成2进制

int shu[] = string.toCharArray();//将i转化成2进制后存在数组中

for(int j = 0;j < note.count;j++){ //分别计算每个全概率

if(j == 0){//计算条件概率事件1发生的概率时需要用加法

if(note.q[j]==0){ //该事件没有父亲节点

pro[i] = note.t[0][shu[cpt.count-1]][j];

}

else{

pro[i] = note.t[pos（note，shu，j）][shu[cpt.count-1]][j];//根据pos函数得到该事件发生应选择其cpt表的第几行

}

}

else{//以后的使用乘法

if(note.q[j]==0){

pro[i]=pro[i]\*note.t[0][shu[cpt.count-1-j]][j];

}

else{

pro[i]=pro[i]\*note.t[pos(note,shu,j)]

[shu[cpt.count-1-j]][j];

}

}

}

}

double P getPT（Note note，String question，double [] allpros，ArrayList<String> arrayList）函数//arrayList为事件集合：

String[] strings=question.split(" \\| ");//将欲求事件和已知的条件事件分开

double[] answer=new double[2];// 0项表示查询问题为真概率，1项表示为假概率

int count=arrayList.indexOf(strings[0]);//欲求事件的Note序号

if(note.q[count]==0){

answer[0]=note.t[0][0][count];

answer[1]=note.t[0][1][count];

}

else{

String[] strings2=strings[1].split(", ");//将条件事件分开

String[] strings3; int sum=0;

int[] count2=new int[strings2.length];//条件Note序号

int[] TF=new int[strings2.length];//存储每个条件是t还是f

for(int i=0;i<strings2.length;i++){

strings3=strings2[i].split("=");

count2[i]=arrayList.indexOf(strings3[0]);

if(strings3[1].contains("true")){

TF[i]=1;

}

else{

TF[0]=0;

}

}

//compute函数直接算出概率

double[] p1=compute(pro, count2, TF, count,cpt);//查询问题的概率（T，F）

answer[0]=p1[0]/p1[0]+p1[1];

answer[1]=p1[1]/p1[0]+p1[1];

}

return answer;

}

1. **算法实现**

3.1 实验环境与问题规模

实验环境：Windows10，Java，Eclipse

问题规模：假设事件数量为n，函数geteverycpt规模最大是O（）；getquestion规模最大是O（）；allpro规模是O（）；getPT规模是O（）。所以总的问题规模是O（）。

3.2 数据结构

**public** **class** Note {

**int** count;//有的多少个变量

**double**[][][] t;//所有mat表

**int**[][] p;//点的关系表

**int** [] q;//各个节点父节点个数

ArrayList<HashMap<String, Integer>> arrayList=**new** ArrayList<>();

}

3.3 实验结果

第一个测试文件结果：

P(Alarm | Earthquake=true, Burglar=true) =

: 0.95 : 0.05

P(Burglar | John=true, Mary=false) =

: 0.001 : 0.999

P(Burglar | John=true, Mary=false) =

: 0.001 : 0.999

P(Burglar | Alarm=true, Earthquake=true) =

: 0.001 : 0.999

第二个测试文件结果：

P(CarWontStart | Dipstick=false, Lights=true, OilLight=true, FuelGauge=true) =

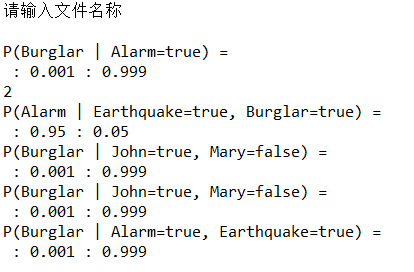
: 0.9834050864967842 : 0.01659491350321645

P(AlternatorBroken | CarWontStart=true, Dipstick=true, Lights=false, BatteryMeter=false) =

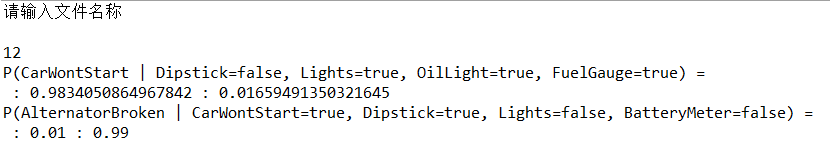
: 0.01 : 0.99

3.4 系统中间及最终输出结果（要求有屏幕显示）

第一个测试文件输出：



第二个测试文件输出：



1. **总结及讨论**

本次实验是利用贝叶斯网络计算各种概率的问题。该实验首先利用输入的文件，构建了一个贝叶斯网络，输出一个CPT表。然后读取问题文件中的概率问题，利用 CPT 表进行计算。由于计算过程中浮点数有一些误差，导致实验结果并不是一直很稳定，但还好，在可以接受的范围内。完成本次实验也是比较曲折的，经历了很多 bug ，在团队的协作下，最终还是实现出来了。