

**人工智能实验报告**

题 目 2018秋人工智能实验（1-3）

专 业 软件工程

学　　 号 1163710228

学 生 刘思琦

指 导 教 师 李钦策

同 组 人 员 刘鹤，谢昱涛，刘奕兵

目 录

[一. 前 言 4](#_Toc529123215)

[1.1 实验目标概述 4](#_Toc529123216)

[1.1.1 实验一：知识表示 4](#_Toc529123217)

[1.1.2 实验二：搜索策略 4](#_Toc529123218)

[1.1.3 实验三： 4](#_Toc529123219)

[1.2 实验环境说明 4](#_Toc529123220)

[1.2.1 windows下编码环境和测试环境 4](#_Toc529123221)

[1.2.2 Linux下测试环境 5](#_Toc529123222)

[二. 实验一：知识表示 6](#_Toc529123223)

[2.1 简介/问题描述 6](#_Toc529123224)

[2.1.1 待解决问题的解释 6](#_Toc529123225)

[2.1.2 问题的形式化描述 6](#_Toc529123226)

[2.1.3 解决方案介绍（原理） 6](#_Toc529123227)

[2.2 算法介绍 8](#_Toc529123228)

[2.2.1 所用方法的一般介绍 8](#_Toc529123229)

[2.2.2 算法伪代码 8](#_Toc529123230)

[2.3 算法实现 10](#_Toc529123231)

[2.3.1 实验环境与问题规模 10](#_Toc529123232)

[2.3.2 数据结构 10](#_Toc529123233)

[2.3.3 实验结果 11](#_Toc529123234)

[2.3.4 系统中间及最终输出结果（要求有屏幕显示） 12](#_Toc529123235)

[2.4总结及讨论 13](#_Toc529123236)

[三．实验二：搜索策略 14](#_Toc529123237)

[3.1 简介/问题描述 14](#_Toc529123238)

[3.1.1 待解决问题的解释 14](#_Toc529123239)

[3.1.2 问题的形式化描述 14](#_Toc529123240)

[3.1.3 解决方案介绍（原理） 15](#_Toc529123241)

[3.2 算法介绍及算法实现 15](#_Toc529123242)

[3.2.1 问题1：应用深度优先算法找到一个特定的位置的豆 16](#_Toc529123243)

[3.2.2 问题2：宽度优先算法 16](#_Toc529123244)

[3.2.3 问题3：代价一致算法 17](#_Toc529123245)

[3.2.4 问题4：A\*算法 17](#_Toc529123246)

[3.2.5 问题5：找到所有的角落 18](#_Toc529123247)

[3.2.6 问题6：角落问题（启发式） 19](#_Toc529123248)

[3.2.7 问题7：吃掉所有的豆子 21](#_Toc529123249)

[3.2.8 问题8：次最优搜索 22](#_Toc529123250)

[3.3 实验结果 22](#_Toc529123251)

[3.4 总结和讨论 29](#_Toc529123252)

[四．实验三：不确定性推理 30](#_Toc529123253)

[4.1 简介/问题描述 30](#_Toc529123254)

[4.1.1 待解决问题的解释 30](#_Toc529123255)

[4.1.2 问题的形式化描述 30](#_Toc529123256)

[4.1.3 解决方案介绍（原理） 31](#_Toc529123257)

[4.2 算法介绍 32](#_Toc529123258)

[4.2.1 所用方法的一般介绍 32](#_Toc529123259)

[4.2.2算法伪代码 32](#_Toc529123260)

[4.3 算法实现 33](#_Toc529123261)

[4.3.1 实验环境与问题规模 33](#_Toc529123262)

[4.3.2 数据结构 33](#_Toc529123263)

[4.3.3 实验结果 33](#_Toc529123264)

[4.4 总结及讨论 34](#_Toc529123265)

[五．参考文献 35](#_Toc529123266)

# 一. 前 言

## 实验目标概述

此实验报告分为三个模块，分别对应2018秋人工智能课程的三个实验。

### 实验一：知识表示

用学过的知识表示方法来解决猴子摘香蕉问题，具体的问题描述在下文实验一模块。

### 实验二：搜索策略

用各种搜索算法编写一系列吃豆人程序解决问题1 – 8，包括到达指定位置以及有效的吃豆等，具体的问题描述在下文实验二模块。

### 实验三：

利用贝叶斯网络知识解决不确定性搜索问题，具体的问题描述参见下文实验三模块。

## 实验环境说明

包含windows10和linux两部分的环境说明。

### 1.2.1 windows下编码环境和测试环境

|  |  |
| --- | --- |
| 实验环境 | 参数值 |
| 操作系统 | Windows10专业版 64-bit |
| 处理器 | Intel(R) Core(TM) i7-8640U CPU @ 4.20GHz 4.20 GHz |
| 内存 | 16.0GB RAM |
| Python版本 | 实验1/2：python 2.7.15  实验3：python3.7.1 |
| IDE | JetBrains PyCharm Professional 2018.2.4 x64 |

### 1.2.2 Linux下测试环境

|  |  |
| --- | --- |
| 实验环境 | 参数值 |
| 操作系统 | Vmware Workstation Pro  & Ubuntu 18.04.1 |
| 处理器 | Intel(R) Core(TM) i7-8640U CPU @ 4.20GHz 4.20 GHz |
| 内存 | 4.0GB RAM |
| Python版本 | 实验1/2：python 2.7.15  实验3：python3.7.1 |
| 编辑器 | Vim |

# 二. 实验一：知识表示

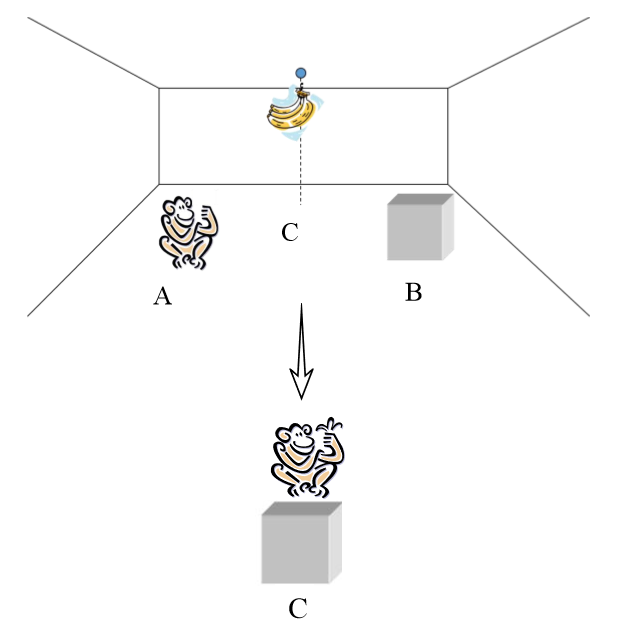
## 2.1 简介/问题描述

### 2.1.1 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，箱子位置为B，香蕉位置在C），如何行动可摘取到香蕉。

### 2.1.2 问题的形式化描述

如图所示：



初始状态：猴子在A处，箱子在B处，香蕉悬挂在C处。

目标状态：猴子和箱子都在C处，且猴子站在箱子上拿到香蕉。

### 2.1.3 解决方案介绍（原理）

1. 定义描述状态的谓词：

AT(x, y)：表示x在y处。

ONBOX：表示猴子在箱子上面。

HB：猴子得到香蕉。

其中，x的个体域是{monkey, box, banana}，y的个体域是{a, b, c}。

2. 初始状态和目标状态：

问题的初始状态是：

AT(monkey, a)；AT(box, b)；┐ONBOX；┐HB

问题的目标状态是：

AT(monkey, c)；AT(box, c)；ONBOX; HB

3. 定义操作：

猴子的动作将会改变问题的状态，其目的也是要把问题的初始状态转换为目标状态，与课本上给出的机器人移盒子问题类似，猴子的每一个操作也都可以分为动作和状态两个部分。在本问题中，猴子需要执行的四个操作可以用谓词描述如下：

GOTO(u, v)：表示猴子从u处走到v处。

PUSHBOX(v, w)：表示猴子推着箱子从v处移到w处。

CLIMBBOX：表示猴子爬上箱子。

GRASP：表示猴子摘取香蕉。

这些操作所对应的先决条件及动作如下：

GOTO(u, v)：

条件：┐ONBOX，AT(monkey, u)

动作：删除表：AT(monkey, u)

添加表：AT(monkey, v)

PUSHBOX(v, w)：

条件：┐ONBOX，AT(monkey, v)，AT(box, v)

动作：删除表：AT(monkey, v)，AT(box, v)

添加表：AT(monkey, w)，AT(box, w)

CLIMBBOX：

条件：┐ONBOX，AT(monkey, w)，AT(box, w)

动作：删除表：┐ONBOX

添加表：ONBOX

GRASP：

条件：ONBOX，AT(box, c)，┐HB

动作：删除表：┐HB

添加表：HB

4. 问题求解：

利用上述谓词和操作，即可以完成猴子摘香蕉问题的求解，求解的过程如下：

S0 = AT(monkey, a) ∧ AT(banana, c) ∧ AT(box, b) ∧ ┐ONBOX ∧ ┐HB

GOTO(a, b)

S1 = AT(monkey, b) ∧ AT(banana, c) ∧ AT(box, b) ∧ ┐ONBOX ∧ ┐HB

PUSHBOX(b, c)

S2 = AT(monkey, c) ∧ AT(banana, c) ∧ AT(box, c) ∧ ┐ONBOX ∧ ┐HB

CLIMBBOX

S3 = AT(monkey, c) ∧ AT(banana, c) ∧ AT(box, c) ∧ ONBOX ∧ ┐HB

GRASP

S4 = AT(monkey, c) ∧ AT(banana, c) ∧ AT(box, c) ∧ ONBOX ∧ HB

## 2.2 算法介绍

### 2.2.1 所用方法的一般介绍

为了使猴子摘香蕉的问题更加一般化，在定义状态的时候我们在程序内部将A处，B处，C处三点抽象化，用数字来定义，即分别用-1,0和1来表示A处，B处和C处。同时为了解决不同情况下的问题，我们设计算法的时候考虑了本问题中很多不同的状态，而不仅仅是猴子在A处，箱子在B处，香蕉在C处的题设状态。在定义输入的时候输入符合逻辑且不矛盾的不同状态，都可以得出最后猴子拿到香蕉的方案；如果输入的时候猴子、箱子、香蕉三者的状态出现矛盾，程序也会相应的给出提醒。经过严格的测试之后，我们的算法可以满足所有状态下的猴子摘香蕉问题。

**程序输入：猴子的位置(A/B/C), 箱子的位置(A/B/C), 香蕉的位置(A/B/C), 猴子是否在箱子上(1/-1)**

**（上述的输入需要四个值，分别是猴子的位置、箱子的位置、香蕉的位置以及猴子与箱子的相对位置（即猴子是否在箱子上）。其中猴子、箱子、香蕉的位置可以取A、B、C当中的任意一个值，猴子与箱子的相对位置可以取-1和1两个值之一，1表示猴子在箱子上，-1表示猴子不在箱子上。**

**当输入不符合实际逻辑的时候，程序会给出提醒。例如，如果相对位置输入1，而猴子和箱子的位置不同的时候，就出现了输入矛盾，程序此时会给出提醒。）**

**程序输出：使猴子可以摘到香蕉的问题最终解决方案。（步骤）**

### 2.2.2 算法伪代码

|  |
| --- |
| **def monkey\_goto(b, a):**  '''  Function monkey\_goto, it makes the monkey goto the other place.  '''  IF a == -1:  “Monkey go to A”  Monkey = -1  ELIF a == 0:  “Monkey go to B”  Monkey = 0  ELIF a == 1:  “Monkey go to C”  Monkey = 1  ELSE  “Error” |
| **def move\_box(b, a):**  '''  Function move\_box, the monkey move the box to the other place.  '''  IF a == -1:  “Monkey move box to A”  Monkey = -1  Box = -1  ELIF a == 0:  “Monkey move box to B”  Monkey = 0  Box = 0  ELIF a == 1:  “Monkey move box to C”  Monkey = 1  Box = 1  ELSE  “Error” |
| **def climb\_onto(i):**  '''  Function climb\_onto, the monkey climb onto the box.  '''  “Monkey climb onto the box”  Monbox = 1 |
| **def climb\_down(i)**  '''  Function climb\_down, the monkey climb down from the box.  ''' |
| **def reach(i)**  '''  Function reach, if the monkey, box, and banana are at the same place, and monkey on box, then the monkey reach banana.  '''  “Monkey reach the banana” |
| **def next\_step(i):**  '''  Perform next step, this is a recursive procedure.  '''  IF isGOAL:  showResult  Return  IF box == banana:  IF monkey == banana:  IF monbox == -1:  climbOntoBOX  reachBanana  nextStep  ELSE:  reachBanana  nextStep  ELSE:  monkeyGotoX  nextStep  ELSE:  IF monkey == box:  IF monbox == -1:  moveBoxToX  nextStep  ELSE:  climbdownFromBox  nextStep  ELSE:  monkeyGotoX  nextStep |

## 2.3 算法实现

### 2.3.1 实验环境与问题规模

**实验环境**：编码环境和测试环境报告的开头已经给出，在此处不做赘述。

**问题规模**：对于此问题来说，算法的时间复杂度是O(1)，因为问题步骤可见，在常数c步骤内一定可以得到问题的解，所以算法的时间复杂度是O(1)。

### 2.3.2 数据结构

用一个list来存储过程状态。

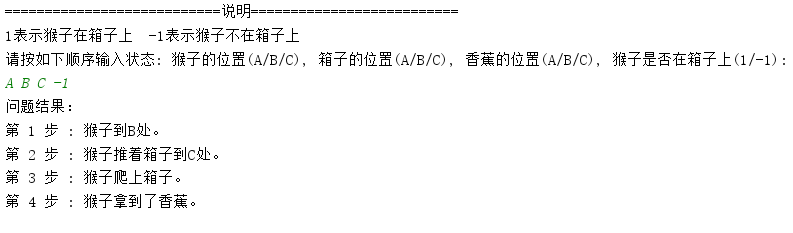
### 2.3.3 实验结果

我们定义了不同状态下的输入来测试程序的处理结果：

**输入1：猴子在A处，箱子在B处，香蕉在C处。**

**解决思路：猴子去B处推箱子到C处，然后爬上箱子拿到香蕉。**

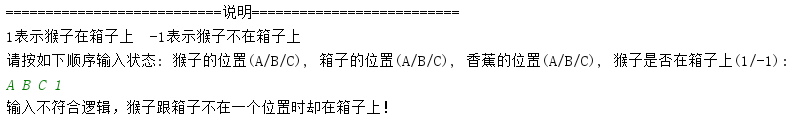
**输出1：**



**输入2：猴子在A处，箱子在B处，香蕉在C处，猴子在箱子上 （1）**

**解决思路：不符合逻辑，猴子和箱子不在同一处的时候猴子在箱子上却为true。**

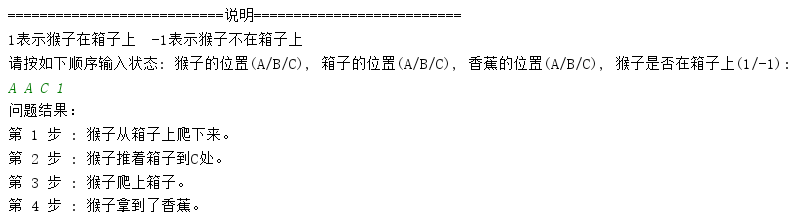
**输出2：**



**输入3：猴子在A处，箱子在A处，香蕉在C处，猴子在箱子上 （1）**

**解决思路：猴子从箱子上爬下来，推箱子到C处，然后爬上箱子拿到香蕉。**

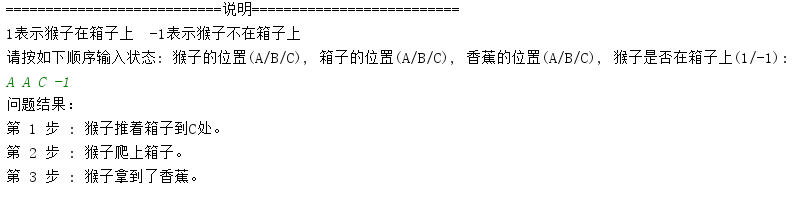
**输出3：**



**输入4：猴子在A处，箱子在A处，香蕉在C处，猴子不在箱子上（-1）**

**解决思路：猴子推箱子到C处，然后爬上箱子拿到香蕉。**

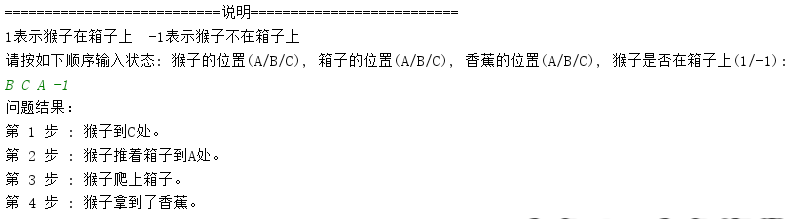
**输出4：**



**输入5：猴子在B处，箱子在C处，香蕉在A处。**

**解决思路：猴子到C处推箱子到A处，爬上箱子拿到香蕉。**

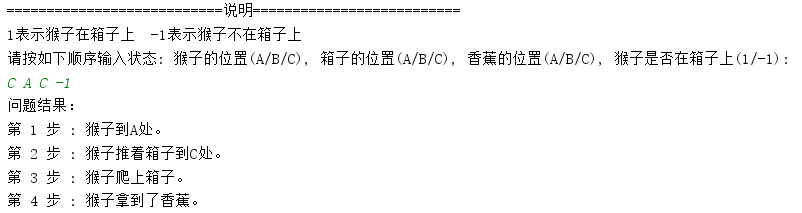
**输出5：**



**输入6：猴子在C处，箱子在A处，香蕉在C处。**

**解决思路：猴子到A处，推箱子回到C处，爬上箱子拿到香蕉。**

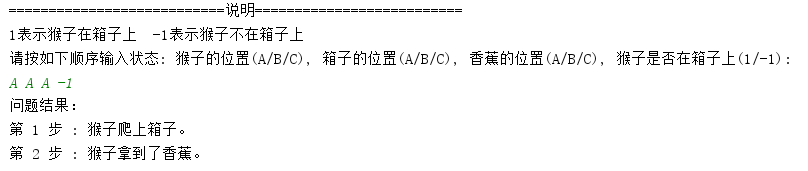
**输出6：**



**输入7：猴子在A处，箱子在A处，香蕉在A处。**

**解决思路：猴子爬上箱子拿到香蕉。**

**输出7：**

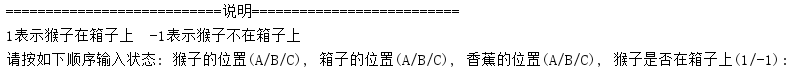


**从上面的实验结果可以看出，我们的程序可以解决任一满足实际情况的初始 状态，同时不满足实际情况的时候，我们的程序也会给出相应的提醒。**

### 2.3.4 系统中间及最终输出结果（要求有屏幕显示）

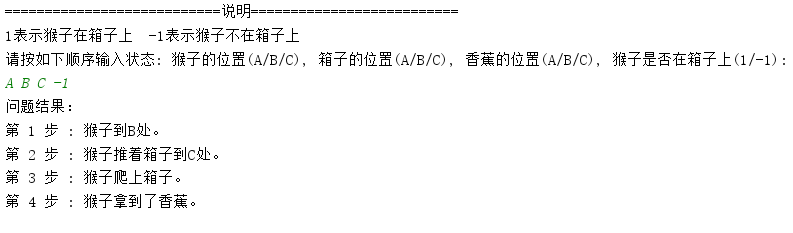
**中间结果：**

运行程序，程序显示如下：



**最终输出结果：**

以猴子在A处，箱子在B处，香蕉在C处为例：



## 2.4总结及讨论

通过本次实验，我对人工智能理论中知识的表示方法有了更深入的理解，尤其是对于一阶谓词表示法和产生式表示法，通过实际编写代码运行知识表示过程，加深了对此类问题的求解能力。同时，通过编写Python代码解决此问题，也回顾了很多python的重要语法和数据结构；通过测试不同状态下程序的运行结果，也熟练了测试程序的方法。

# 三．实验二：搜索策略

## 3.1 简介/问题描述

### 3.1.1 待解决问题的解释

实验二要求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。具体问题如下：

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。

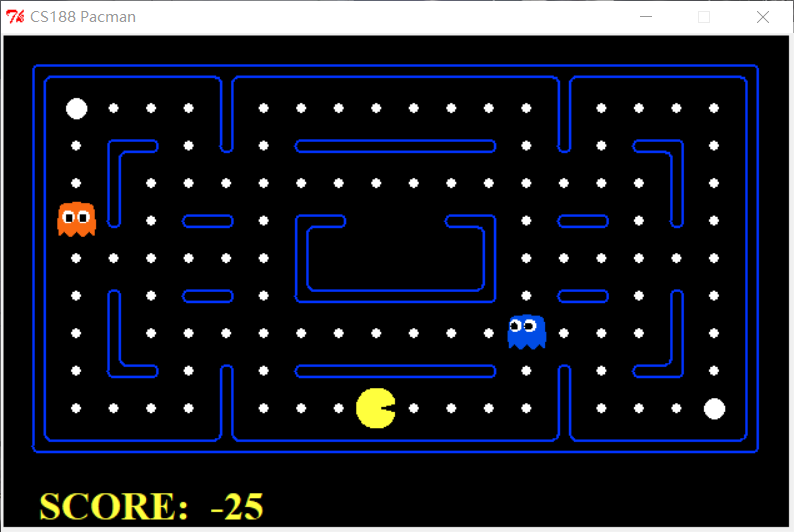
问题6：角落问题（启发式）。

问题7：吃掉所有的豆子。

问题8：次最优搜索。

### 3.1.2 问题的形式化描述

如图所示：



基于如图所示的吃豆人游戏，本次实验要求设计搜索算法来解决不同的搜索问题，解决不同的目标，包括如何到达指定的位置以及如何有效的吃豆等。

### 3.1.3 解决方案介绍（原理）

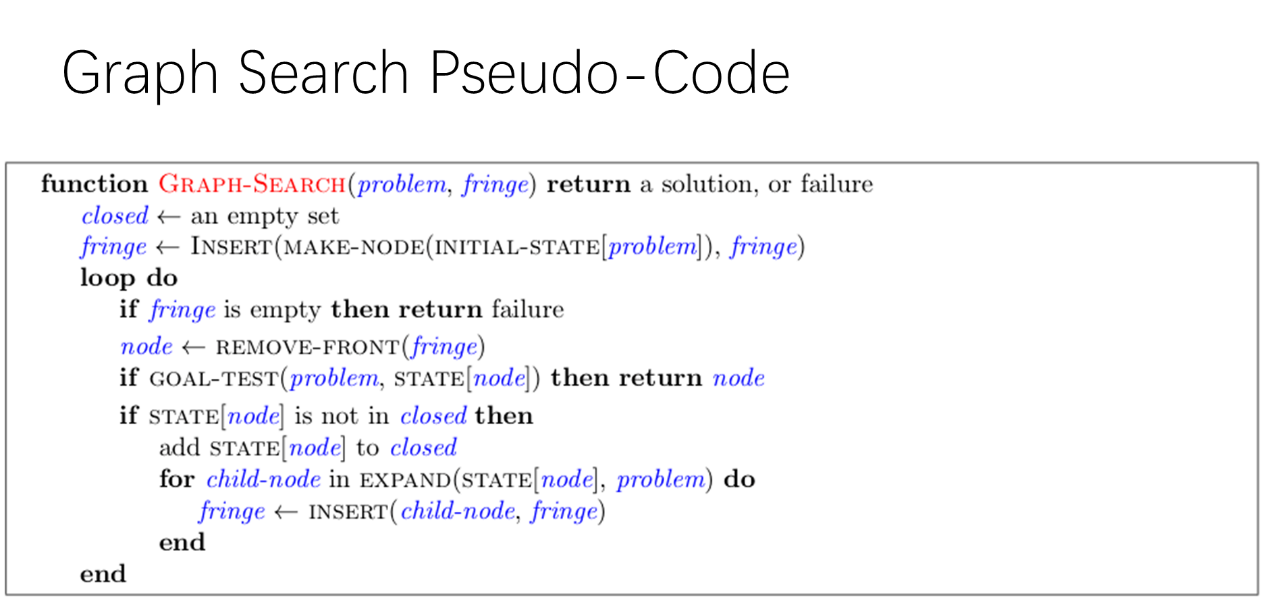
在下文报告中会根据不同的问题来给出不同的解决方案原理介绍。

## 3.2 算法介绍及算法实现

本节将针对实验二给出的八个问题做出简要的介绍，依次分析问题，并且说明解决问题的方法及其算法实现。

首先给出所用方法的一般介绍及伪代码：

根据实验指导书中给出的算法伪代码，我们可以设计出实验二所必要的完整的通用搜索算法。



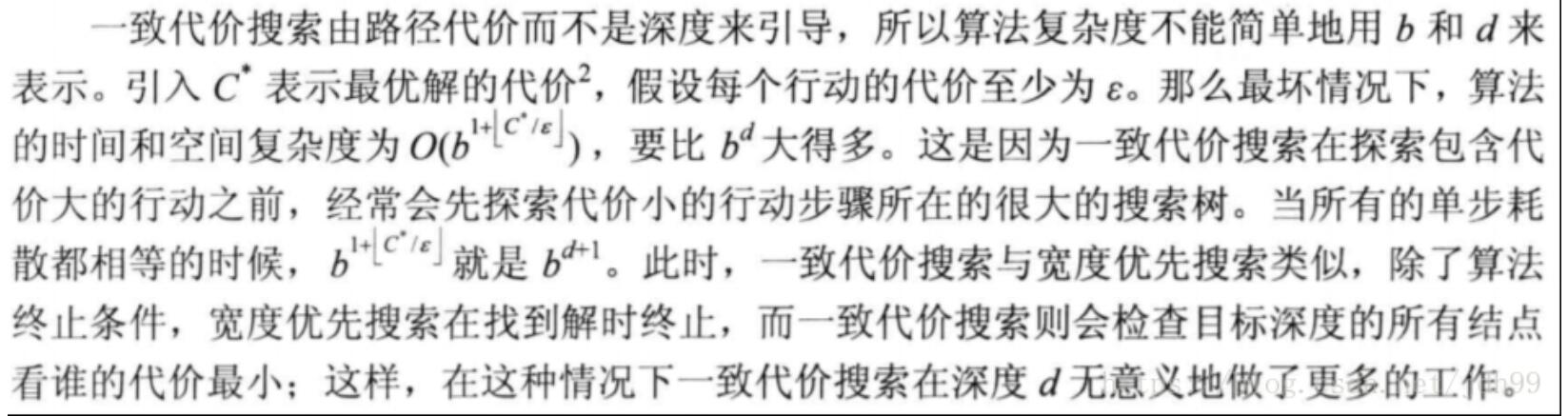
通用的搜索算法与课件中的描述基本一致，唯一的区别是如何保存构建路径的信息。我们使用了(state, actions)的二元组来表示每个节点(node)，其中state为状态，对于问题1到问题4，即为吃豆人所在的坐标(coord)，而对于之后的问题，除坐标之外还含其他参数。而actions为从初始结点到达本状态所要执行的操作序列，actions即为保存构建路径的信息需要的结构，即为路径信息本身。

**通用搜索算法的参数/返回值**等信息如下伪代码所述：

|  |
| --- |
| class GenericSearch:  def\_\_init\_\_(self.problem, data\_struct\_type, usePriorityQueue = False, heuristic=nullHeuristic);  self.problem = problem  self.data\_struct\_type = data\_struct\_type  self.usePriorityQueue = usePriorityQueue  self.heuristic = heuristic  def genericSearch(self):  """  return 值:actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列  """  #下面的代码逻辑与附录中给出的伪代码逻辑相同，在此不再赘述 |

**实验环境**：编码环境和测试环境在报告的开头已经给出，在此不再赘述。

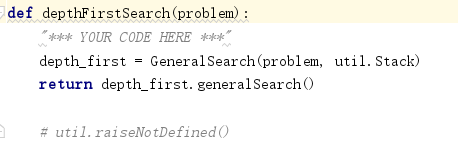
**问题规模**：深度优先搜索算法和广度优先搜索算法的复杂度是O(n²)，A\*算法的复杂度是O(n²\*logn)，代价一致算法的复杂性如下：



### 3.2.1 问题1：应用深度优先算法找到一个特定的位置的豆

**深度优先搜索算法**：在深度优先搜索中，首先扩展最新产生的（即最深的）节点，即为后生成的节点先扩展的策略。在深度优先搜索中，open表是一种栈结构，最先进入的节点排在最后面，最后进入的节点排最前面，即先进后出（FILO）。

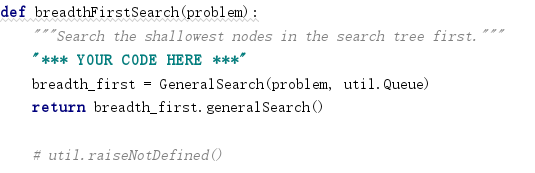
**算法实现**：有了通用的搜索算法，实现深度优先搜索算法只需指定参数data\_struct\_type为实验提供栈结构util.Stack即可：



### 3.2.2 问题2：宽度优先算法

**宽度优先算法**：宽度优先搜索是一种先生成的节点先扩展的策略，基本思想是：从初始节点S0开始，逐层地对节点进行扩展并考察它是否为目标节点，在第n层的节点没有被全部扩展并考察之前，算法不对第n＋1层的节点进行扩展。按照宽度优先搜索的概念，open表中的节点总是按进入的先后顺序排列，先进入的节点排在前面，后进入的排在后面，是一种队列结构，即先进先出（FIFO）。

**算法实现**：有了通用的搜索算法，实现宽度优先搜索算法只需指定参数data\_struct\_type为实验提供队列结构util.Queue即可：



### 3.2.3 问题3：代价一致算法

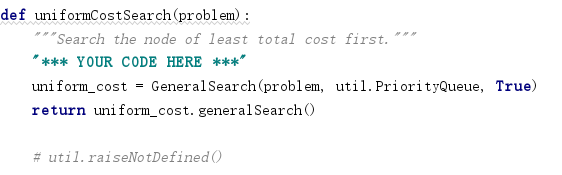
**代价一致算法**：在前面问题中的各种搜索策略中，实际上我们都假设状态空间中各边的代价都相同，且都为一个单位量，从而可以用路径长度来代替路径的代价。但是在实际问题中，这种假设显然是不现实的，它们的状态空间中的各个边的代价不可能完全相同。为此，我们需要在搜索树中给每条边标上其代价。这种边上有代价的树称为代价树。

在代价树中，可以用g(n)表示从初始节点S0到节点n1的代价，用c(n1, n2)表 示从父节点n1到n2的代价。这样，对节点n2的代价有：g(n2) = g(n1) + c(n1, n2) 。

代价一致搜索是宽度优先搜索的一种推广，搜索过程不是沿着等长度路径逐层进行扩展，而是沿着等代价路径逐层进行扩展。

代价一致搜索的基本思想是：在代价一致搜索算法中，把从起始节点S0到任一节点i的路径代价记为g(i)。从初始节点S0开始扩展，若没有得到目标节点，则优先扩展最少代价min(g(i))的节点，一直如此向下搜索，所以我们选择优先队列作为open表的数据结构。

**算法实现**：将通用搜索算法中的data\_struct\_type设置为util.PriorityQueue，并将usePriorityQueue设置为True，表示使用优先级队列：



### 3.2.4 问题4：A\*算法

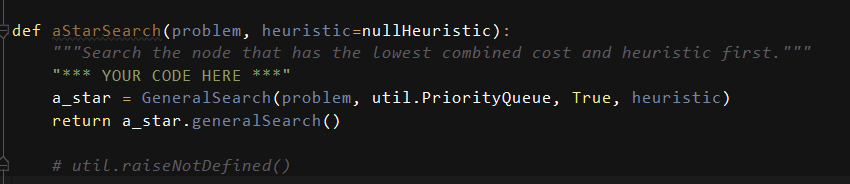
**A\*算法**：A\*搜索算法是一种在图形平面上，对于有多个节点的路径，求出最低通过成本的算法。该算法综合了BFS算法和Dijkstra算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条基于评估函数的最优路径。

在此算法中，如果以g(n)表示从起点到任意顶点n的实际距离，h(n)表示任意顶点n到目标顶点的估算距离（根据所采用的评估函数的不同而变化），那么 A\* 算法的估算函数为：f(n) = g(n) + h(n)。

在A\*搜索问题中，需要优先扩展f(n)值最小的节点，所以选择优先队列作为open表的数据结构。

**算法实现**：将通用搜索算法data\_struct\_type设置为util.PriorityQueue, usePriorityQueue设置为True, heuristic设置为aStarSearch传入的heuristic。

当执行autograder.py时，会将manhattanHeuristic传入aStarSearch的heuristic，即可实现以曼哈顿距离为启发函数的A\*算法。我们也可以通过python pacman.py -l bigMaze -z .5 -p SearchAgent-afn = astar, heuristic = manhattanHeuristic命令来测试我们的A\*程序。

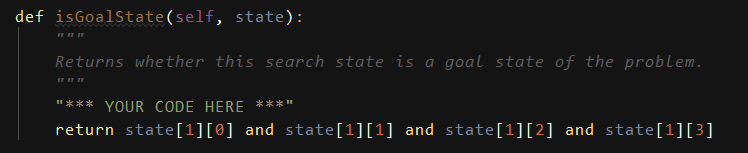


### 3.2.5 问题5：找到所有的角落

**问题描述**：寻找角落问题要求我们实现CornersProblem，即把使吃豆人以最短的路径找到迷宫的四个角落为算法的目标。

**算法实现**：在实验指导中已经指出，新的状态只包含吃豆人的位置和角落的状态。所以在解决问题5的时候，我们把状态state表示为一个形如(coord, foods\_BooleanState)的二元组，其中coord表示的是吃豆人所在的位置坐标(x, y)，foods\_BooleanState是四个角落的食物的状态列表，foods\_BooleanState的初始值是{False, False, False, False}，foods\_BooleanState状态列表对应的角落位置与self.corners中的(1, 1), (1, top), (right, 1), (right, top)一一对应。如果角落上的食物没有被吃掉，那么在foods\_BooleanState中的对应项就保持False状态，如果食物被吃掉了，那么对应的状态就从False更换为True。

函数如下图所示：



此外，生成后继函数getSuccessors也是此问题的一个核心，由于在状态表示中增加了对角落状态的表示，所以在生成后继时也要考虑这部分是否会变化。

考虑后继节点，如果后继节点是还未到达的角落位置，则在此后继的foods\_BooleanState中，要把状态设置成True，表示下一步已经到达此后继角落并吃掉了食物。

生成后继函数getSuccessors的伪代码如下：

|  |
| --- |
| def getSuccessors(self, state):  successors = []  x, y ← state[0] # 获得当前吃豆人坐标  foodBooleanState ← state[1] # 获得接下来的角落的食物状态  foodXYList=[]  CONVERSATION foodBooleanState To foodXYList  FOR action IN [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:  dx, dy ← Actions.directionToVector(action) # 将action解析为坐标增量  nextx, nexty ← int(x + dx), int (y + dy) #获得执行action之后的坐标  # hitsWall即为walls网格中(nextx, nexty)处的布尔值  hitsWall ← self.walls[nextx][nexty]  IF NOT hits Wall:  # hitsWall为true表示有墙，为false表示无墙  nextState ← (nextx, nexty)  leftFoodBooleanState ← ShallowCopy(foodBooleanState)  # 判断(nextx, nexty)是否在剩余的食物坐标列表当中  IF nextState IN foodXYList:  leftFoodBooleanState [self.corners.index((nextx, nexty))] ← True  # 若是，将其从列表中删去，表示该角落已到达  successors.append(((nextState, leftFoodBooleanState), action, 1))  # 将下一状态加入successors  self.\_expanded += 1  return successors |

### 3.2.6 问题6：角落问题（启发式）

**问题描述：**

本题要求实现 CornersProblem中的启发式函数，用这个启发式函数执行A\*算法，其中函数要求有非平凡且一致的性质。

**算法实现：**

首先让吃豆人在当前节点找一个最小代价的角落，然后把吃豆人移动到那个角落再代入函数又找到下一个代价最小的角落，以此循环来完成角落问题。

具体实现的伪代码如下：

|  |
| --- |
| Def cornersHeuristic(state, problem):  Current\_positon ← state[0]  Corners\_bool ← state[1]  Current\_corners\_not\_visited ← set(corners)  Current\_corners\_visited ← set(corners)  FOR i IN len(Corners\_bool):  IF corners\_bool[i] == TRUE:  Current\_corners\_not\_visited REMOVE corners[i] # mark visited  IF corners\_bool[i] == FALSE:  Current\_corners\_visited REMOVE corners[i]  Heuristic\_cost ← 0  WHILE len(current\_corners\_not\_visited) > 0:  Distance, corner ← min\_distance\_point\_targets(current\_positon, current\_corners\_not\_visited)  Heuristic\_cost += distance  current\_positon ← corner  current\_corners\_not\_visited ← tuple(set(current\_corners\_not\_visited) – set(corner))  return heuristic\_cost |

**启发式函数可纳性(admissible)和一致性(consistent)的证明：**

如果证明了其一致性，则可纳性必然成立，因而在此略去其可纳性的证明，仅就其**一致性**给出简要的证明。

|  |
| --- |
| 证明：  证明一致性，即是要证明如下内容：  如果启发函数满足以下两个条件**:**  **(1) h(Sg )=0;**  **(2)** 对任意节点**ni** 及其任一子节点**nj** ，都有  **0**≤**h(ni )-h(nj )**≤**c(ni , nj )**  其中**c(ni , nj )**是**ni** 到其子节点**nj** 的边代价，则称**h(n)**满足一致性。  使用上述形式化定义中的所有符号。  可以看到，每次启发式函数给出的启发代价是：吃豆人前往各个没被吃掉的节点之和的最短长度。  如果吃豆人的移动方向为靠近最近的那个角落，则h(**ni**) = c(**ni**, **nj**) + h(nj)；否则，则h(**ni**) < h(**nj**)，这两种情况均满足h(**ni**) ≤ c(**ni**, **nj**) + h(**nj**)  因此，我们选择的启发式函数是一致的，也显然是可纳的。 |

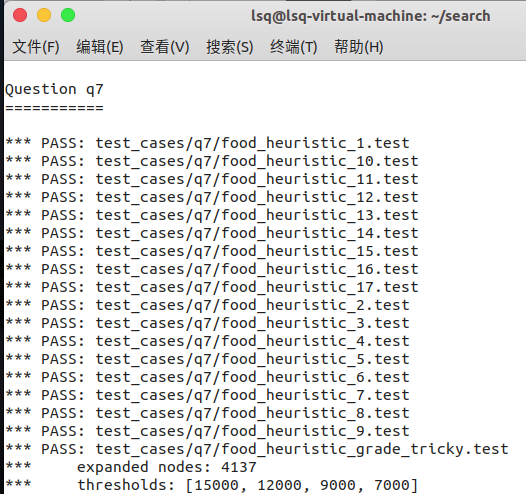
### 3.2.7 问题7：吃掉所有的豆子

**问题描述：**

在本问题中，多个食物不仅仅局限在角落出现，它们可能出现在任何地方，所以本问题需要一个良好的启发式函数，用尽可能少的步数，也就是扩展较少的节点来找到所有的食物。

**算法实现：**

我们小组注意到在课程已经给出的代码中，给出了一个mazeDistance(point1, point2, gameState)函数，可用了计算在迷宫中从一个点到达另一个点所需要的真实最小代价。使用此距离函数可比使用曼哈顿距离函数产生一个更紧的下界（下确界），使用mazeDistance的结果如下：



扩展了4137个节点，效果好了许多，并且拿到了**附加分**。

**启发式函数可纳性(admissible)和一致性(consistent)的证明：**

对于该问题，错误的启发式函数扩展的节点数目同样也会很少。给出一个具有较好结果的启发式函数是不够的，还要证明其可纳性和一致性。我们小组所使用的启发式函数的可纳性显然成立，而且如果证明了其一致性，则可纳性必然成立，因而在此略去其可纳性的证明，仅就其**一致性**给出简要的证明。

|  |
| --- |
| 证明：  证明一致性，即是要证明如下内容：  如果启发函数满足以下两个条件**:**  **(1) h(Sg )=0;**  **(2)** 对任意节点**ni** 及其任一子节点**nj** ，都有  **0**≤**h(ni )-h(nj )**≤**c(ni , nj )**  其中**c(ni , nj )**是**ni** 到其子节点**nj** 的边代价，则称**h(n)**满足一致性。  使用上述形式化定义中的所有符号。  可根据吃豆人执行一个操作后离吃豆人最远的食物是否变化分成两种情况讨论：  1.距离吃豆人最远的食物不变  如果吃豆人的移动方向为靠近最远食物，则h(N) = c(N, P) + h(P)；如果吃豆人的方向为远离最远食物，则h(N) < h(P)，这两种情况均满足h(N) ≤ c(N, P) + h(P)  2.距离吃豆人最远的食物改变  假设食物没有改变，设最远食物到后继P的在迷宫中行走的距离为g(P)，则根据1可以得到h(N) ≤ c(N, P) + g(P)，但是由于在P点的最远食物改变了，因此h(P) > g(P)，从而h(N) ≤ c(N, P) + h(P)  综上所述，我们选择的启发式函数是一致的，也显然是可纳的。 |

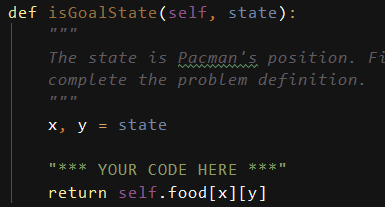
### 3.2.8 问题8：次最优搜索

**问题描述：**

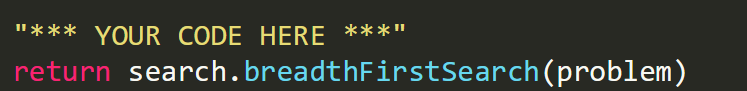
本问题要求实现一个优先吃最近的食物的路径规划算法，需要实现 AnyFoodSearchProblem中 的isGoalState函数和ClosestDotSearchAgent中的 findPathToClosestDot函数。

**算法实现：**

在isGoalState函数中，要定义将吃掉一个食物作为目标状态，以使 findPathToClosestDot 函数在吃到最近的食物后结束并返回action列表，isGoalState的代码如下：



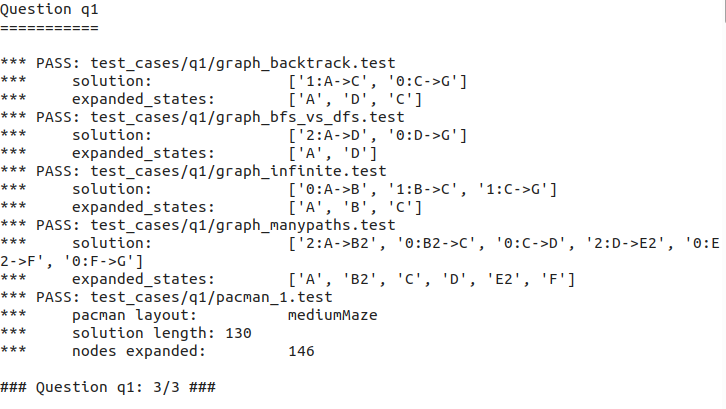
ClosestDotSearchAgent中的 findPathToClosestDot函数，要求找到去往最近的食物的路径，在每走一格代价一致的情况下，可以选用广度优先搜索和代价一致搜索找到路径，其扩展节点数目是一样的。



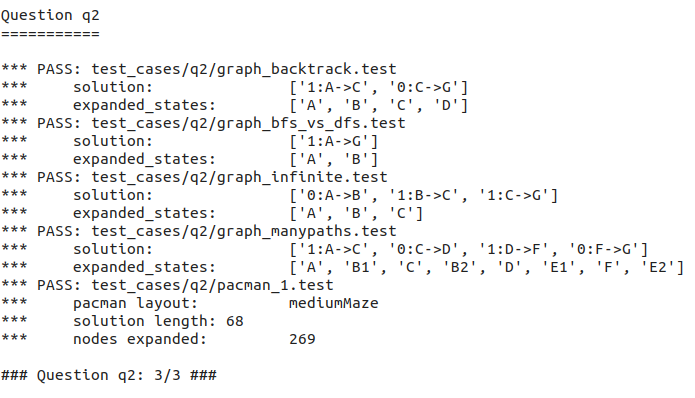
## 3.3 实验结果

本章给出运行 autograder.py 后对于每个问题、每个测试用例的评分结果。

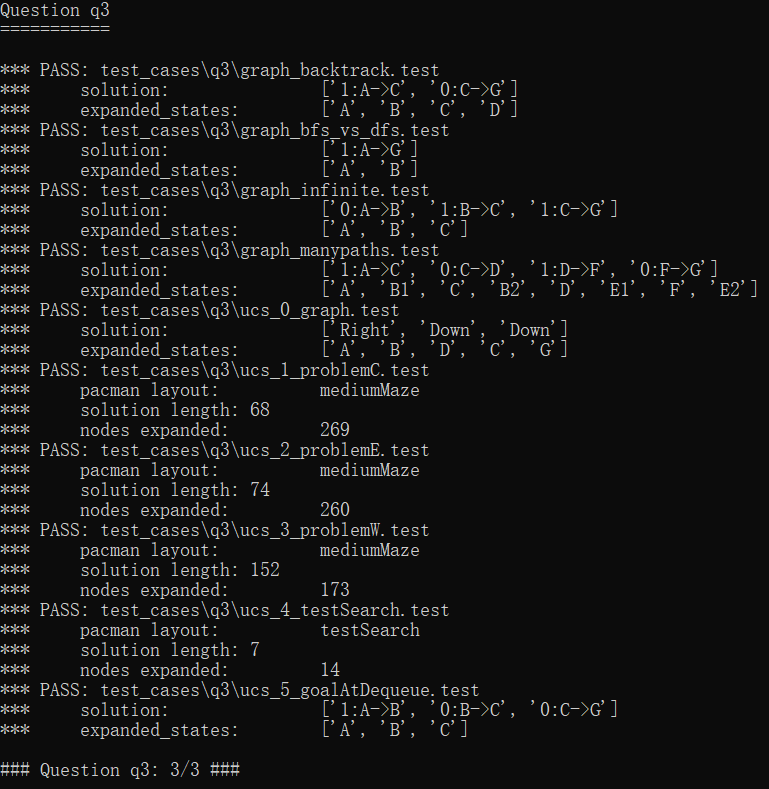
**第一题：**



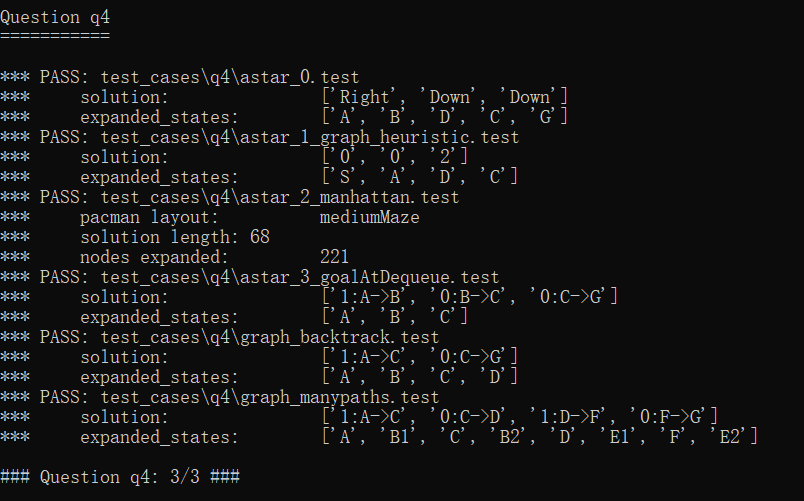
**第二题：**



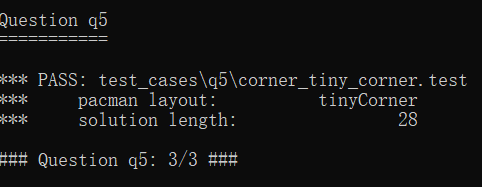
**第三题：**



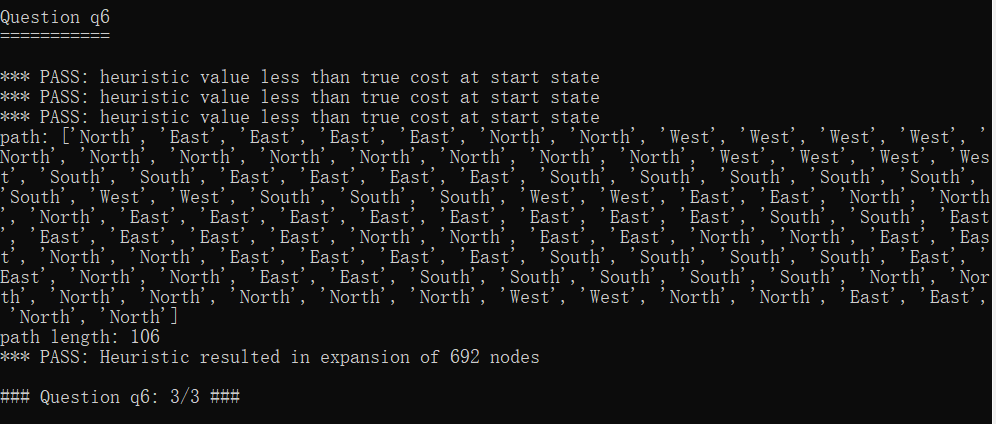
**第四题：**



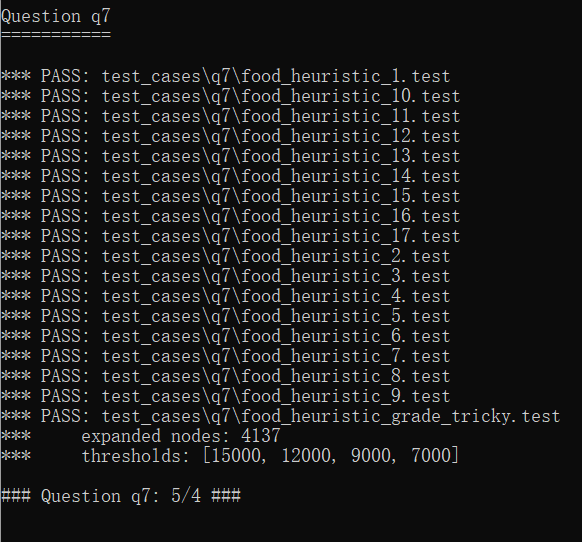
**第五题：**



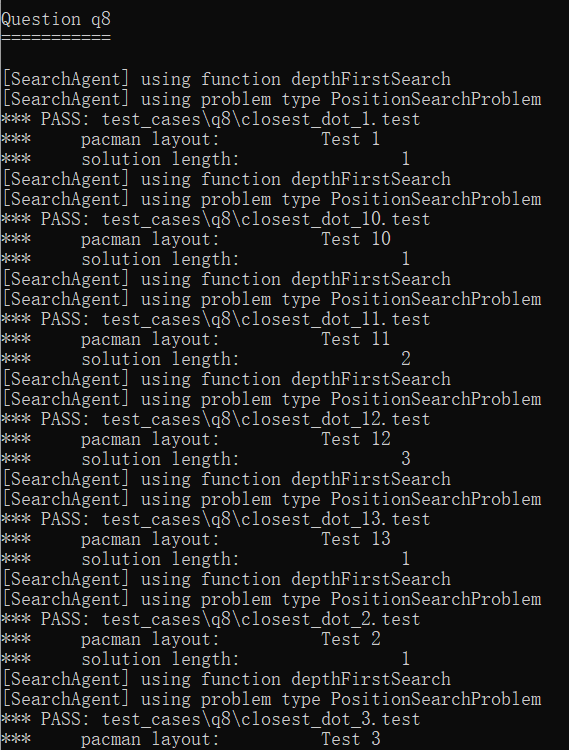
**第六题：**

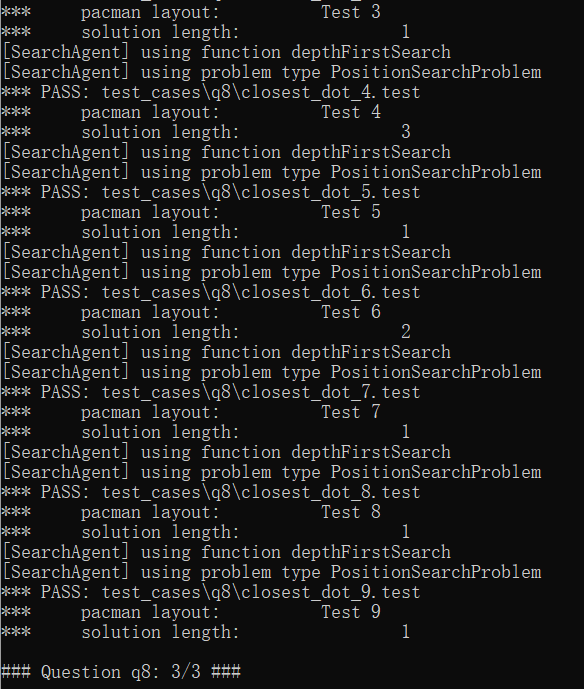


**第七题：**

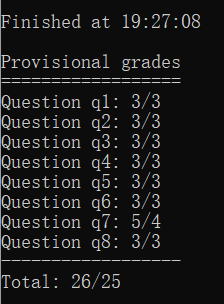


**第八题：**





**总分：**



## 3.4 总结和讨论

在本次实验中，我们实现了DFS、BFS、UCS、A\*等搜索算法，同时也学习了启发式函数的选取要求和方法以及可用性、一致性的证明。通过本次实验，我对上述的搜索算法和启发式函数相关知识有了进一步的理解。同时我也锻炼了阅读代码、理解代码的能力，也强化了python程序设计、数据结构与算法分析的技术。

# 四．实验三：不确定性推理

## 4.1 简介/问题描述

### 4.1.1 待解决问题的解释

不确定性推理实验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。

### 4.1.2 问题的形式化描述

首先，给定的输入文件格式为：

N

rv0 rv1 ... rvN-1

0 0 1 ... 0

1 0 0 ... 1

...

0 1 1 ... 0

mat0

mat1

...

matN-1

在这里:

• N 是贝叶斯网络中随机事件的数目

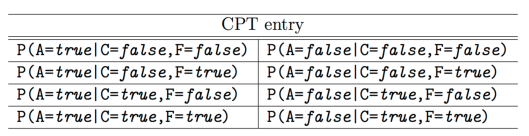
• rv 是随机事件的名字（字符串形式表示）

• mat是一个二维数组，分别表示从他的父亲到其本身的可能性概率。第一个元素表示发生的概率，第二个元素表示不发生的概率，显然两个元素相加为1

在上述中mat即为CPT表（Conditional Probability Table），其被设计为如下格式：

对于每个节点，如果他有N个父节点，则其CPT表中有2^N列，我们记为标号0 - （2N-1），其行序号的定义方法如下，利用二进制分别表示对应的父亲为是否发生，1为发生，0位不发生，将得到的二进制数转化为十进制代表其对应的行号。举例如下:

A有两个父节点C，F，则CPT表如下表所示:



其次，编写程序对应的查询格式为：P(rvQ | rvE1=val, rvE2=val, ...)

rvQ表示查询的条件的名字，即在rvE1=val, rvE2=val, ..发生的条件下，rvQ发生的概率。

RvEx表示条件的名称，而后面的val为true/false，分别表示发生和不发生。

最后，输出格式为两个数据分别表示P(QueryVar=true|...) 和P(QueryVar=false|...))。例如：

0.872 0.128

### 4.1.3 解决方案介绍（原理）

解决本次实验的原理思想是贝叶斯方法，根据贝叶斯公式求出目标概率，编写程序实现基于

贝叶斯网络的推理。

我们可以把burglarnetwork文件中的数组表示成有向无环的概率分布图。概率分布的图表示是概率论与图论的结合，在概率图模型中，每个节点表示一个随机变量（或一组随机变量），边表示变量间的概率关系。

在引出贝叶斯定理之前，先学习几个定义：

**① 条件概率**就是事件A在另外一个事件B已经发生条件下的发生概率。条件概率表示为P(A|B)，读作“在B条件下A的概率”。

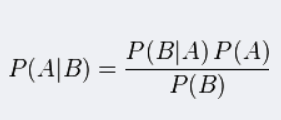
**② 联合概率**表示两个事件共同发生的概率。A与B的联合概率表示为http://img.my.csdn.net/uploads/201212/17/1355742600_1762.png或者http://img.my.csdn.net/uploads/201212/17/1355742607_7553.png。

**③ 边缘概率**（又称先验概率）是某个事件发生的概率。边缘概率是这样得到的：在联合概率中，把最终结果中那些不需要的事件通过合并成它们的全概率，而消去它们（对离散随机变量用求和得全概率，对连续随机变量用积分得全概率），这称为边缘化（marginalization），比如A的边缘概率表示为P(A)，B的边缘概率表示为P(B)。

接着，考虑一个问题：P(A|B)是在B发生的情况下A发生的可能性。

首先，事件B发生之前，我们对事件A的发生有一个基本的概率判断，称为**A的先验概率**，用**P(A)**表示；其次，事件B发生之后，我们对事件A的发生概率重新评估，称为**A的后验概率**，用**P(A|B)**表示；类似的，事件A发生之前，我们对事件B的发生有一个基本的概率判断，称为**B的先验概率**，用**P(B)**表示；同样，事件A发生之后，我们对事件B的发生概率重新评估，称为**B的后验概率**，用**P(B|A)**表示；

**贝叶斯定理**便是基于下述**贝叶斯公式**：

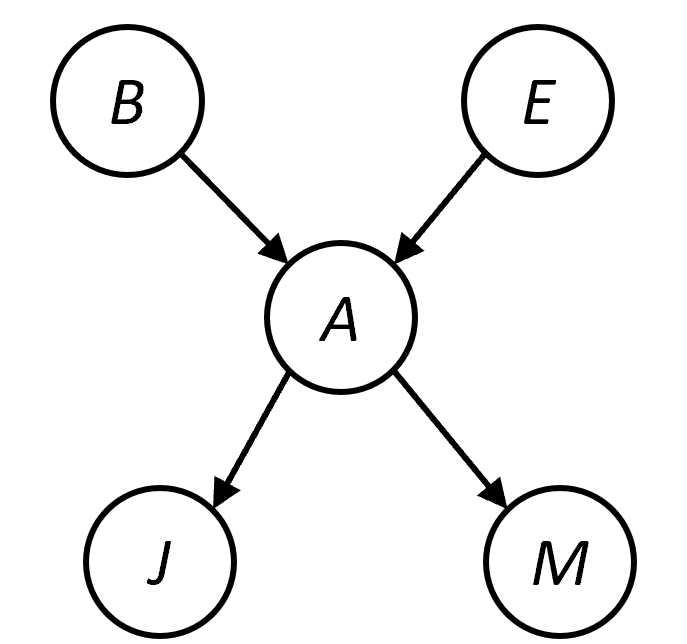


基于贝叶斯网络的定义，我们可以将给定的文件构造为贝叶斯网络（有向无环图），每一条边赋值以概率，最后根据构造出的图求出burglarqueries文件中的五个概率即可。

## 4.2 算法介绍

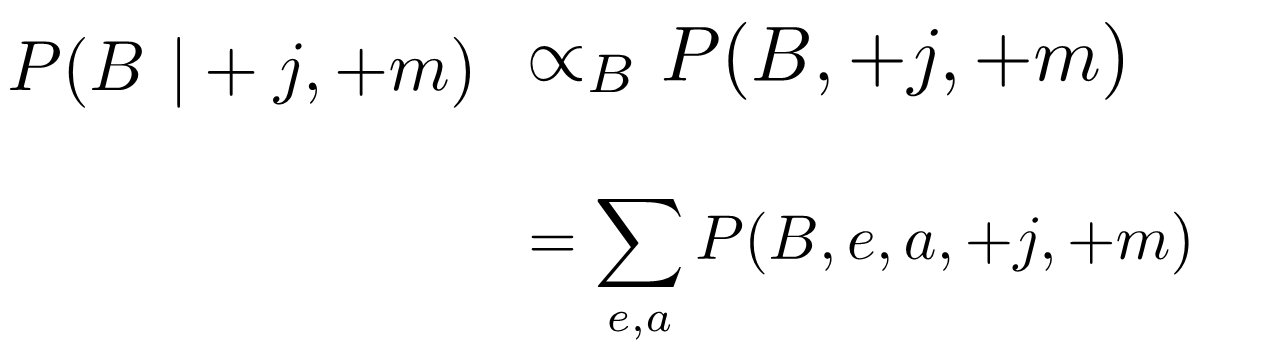
### 4.2.1 所用方法的一般介绍

我们使用的方法是基于贝叶斯网络的枚举推断，以下面的网络为例：



比如，计算latex-image-1.pdf

可以转化为计算全概率再做归一化：



然后对e，a两个变量做枚举，可以得到4种情况，这4种情况都可以通过查表找到相关概率，再做乘积就是解。把4种情况加起来就是全概率的结果，对B=+b和-b两个全概率做归一化就是要求的条件概率。

latex-image-1.pdf

latex-image-1.pdf

### 4.2.2算法伪代码

|  |
| --- |
| **def find\_pre\_count(var):**  count ← 0  FOR ab IN Graph.predecessors(var):  count ++  return count; |
| **def get\_value(values, list\_value):**  calc\_value = []  length ← len(list\_value)  FOR i IN range(2length):  Ints = []  count ← 0  FOR j IN range(length):  IF j == 0:  ints.append(i % 2)  count += i % 2  ELSE  ints.append(((i - count) % 2(j + 1)) >> j)  count ← (i - count) % 2(j + 1)  vals ← dict(values)  FOR j IN range(length):  vals[list\_value[j]] ← ints[j]  calc\_value.append(vals)  return calc\_value |

## 4.3 算法实现

### 4.3.1 实验环境与问题规模

**实验环境**：编码环境和测试环境报告的开头已经给出，在此处不做赘述。

**问题规模：**此问题的规模是由构造出的贝叶斯网络的结点数和边数来决定的。

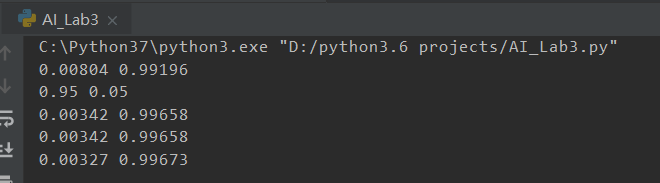
### 4.3.2 数据结构

利用python复杂网络分析库networkx建立有向无环图来存储节点和边（概率）。

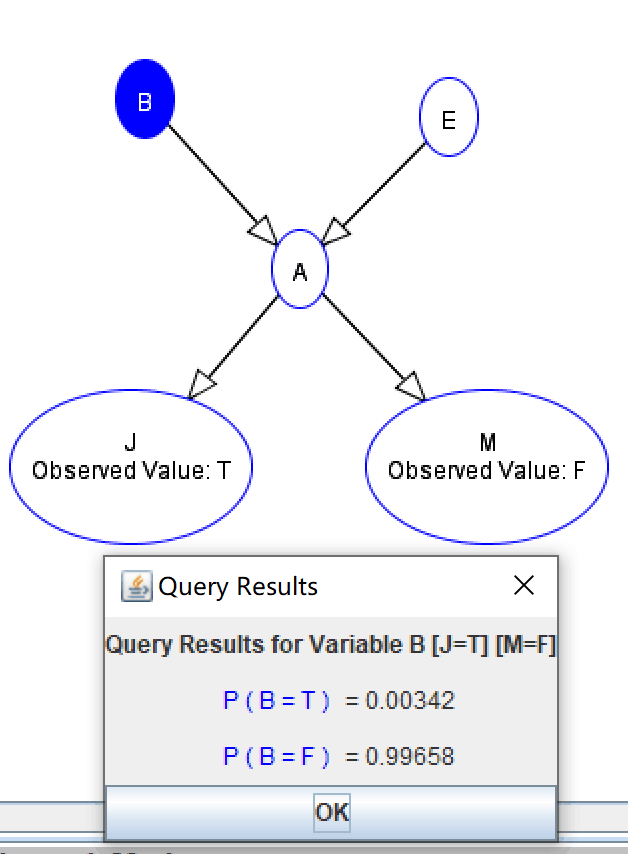
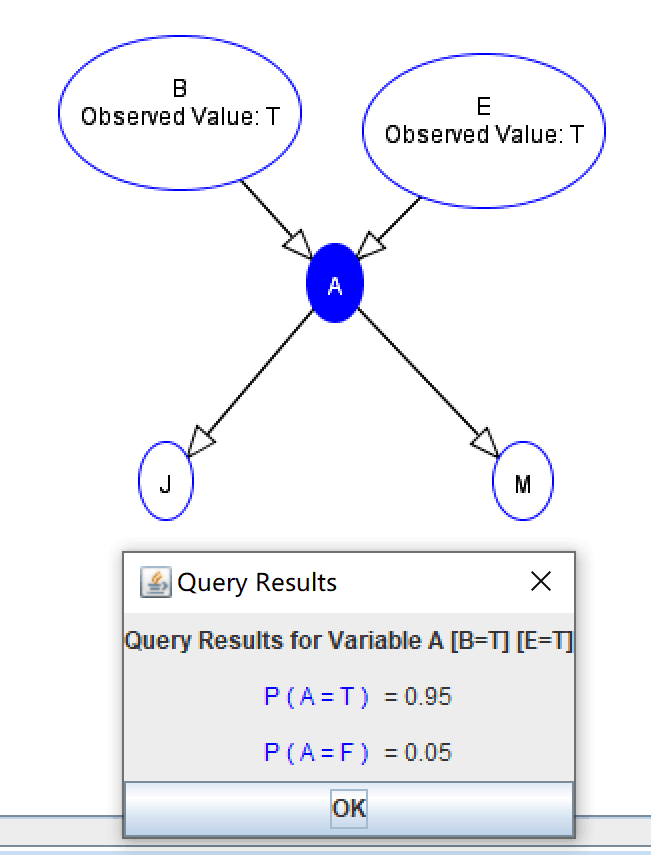
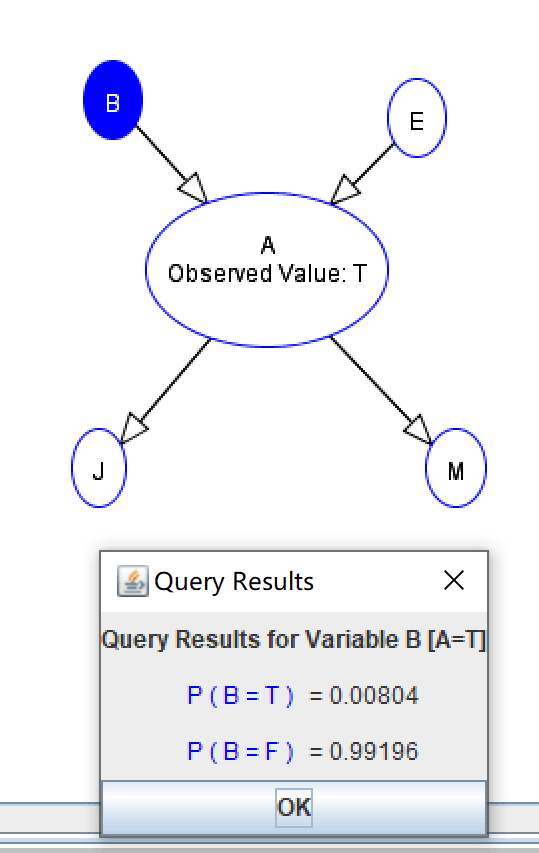
用list来存储节点集合以及计算出的概率。

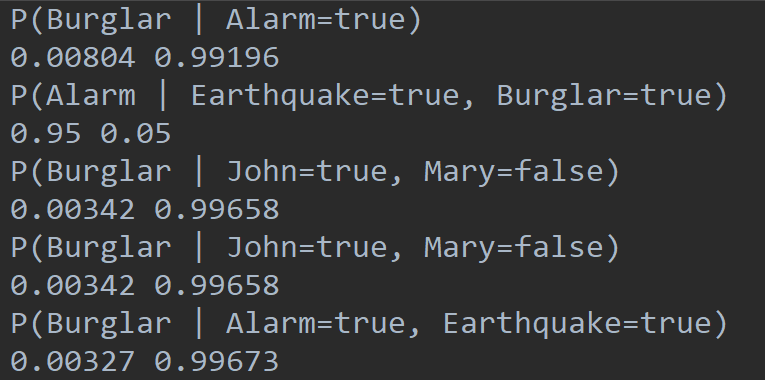
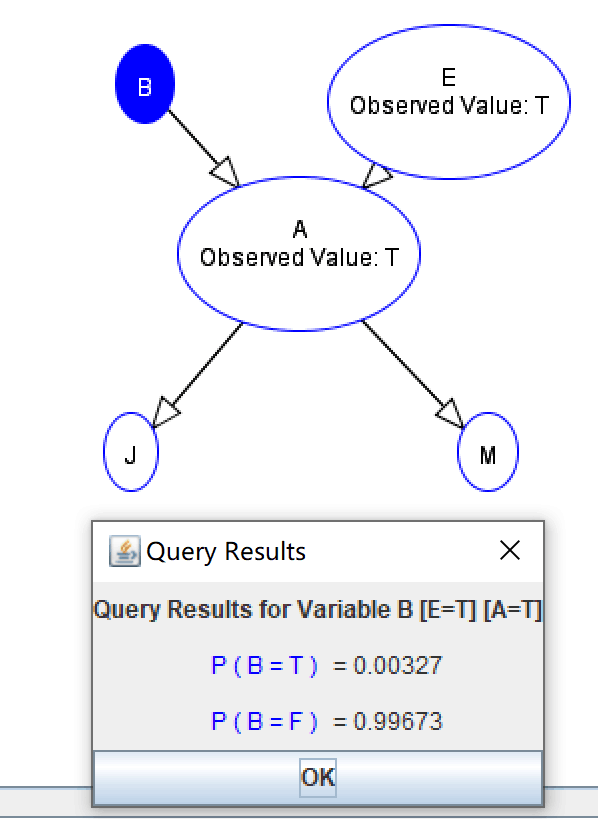
### 4.3.3 实验结果

程序的运行结果如下：



运行bayes.exe所得到的结果：





**可以看到，计算结果和验证结果是完全一致的！**

## 4.4 总结及讨论

通过本次不确定性推理的实验，我对贝叶斯定理和贝叶斯网络的相关知识有了较为深入地了解，通过编写代码构造贝叶斯网络的数据结构来进行概率计算，我基本掌握了贝叶斯网络推理之后验概率问题的计算方法，总的来说，本次实验对于熟悉巩固贝叶斯网络相关知识是非常有意义的。

# 五．参考文献

[1] 王万森.人工智能原理及其应用[M].北京：电子工业出版社，2012.09.01