哈爾濱工業大學

人工智能实验报告

题　　目 人工智能实验 1-2-3 报告

专　　业 计算机科学与技术

学　　号 1160300901

学　　生 孙月晴

指 导 教 师 李钦策

同 组 人 员 姜梦奇　郑南燕

目 录

第一篇　实验 1 知识表示

一． 问题描述

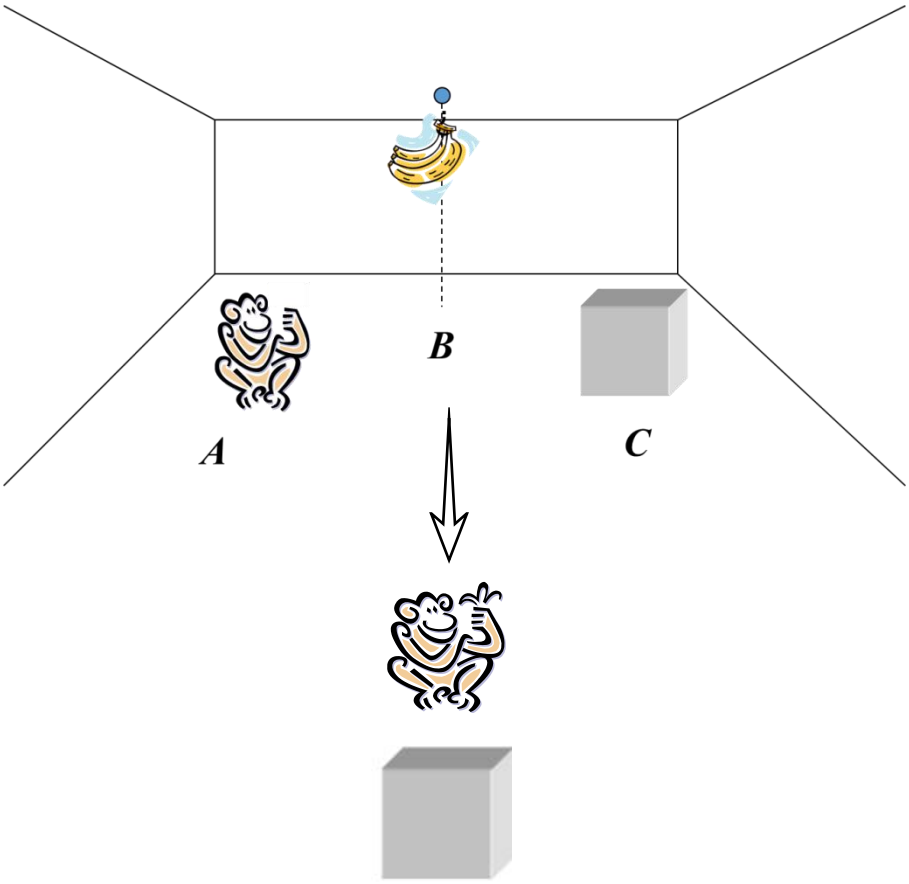
本篇实验选择知识表示方法完成，主要运用一阶谓词逻辑，同时也有一些产生式系统的思想来解决实验问题。

1.1 待解决问题

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如何行动可摘取到香蕉。

1.2 问题的形式化描述

初始状态：猴子处在 A 处，箱子处在 B 处，香蕉悬挂在 C 处
目标状态：猴子和箱子在 C 处，猴子站在箱子上摘到香蕉



1.3 解决方案介绍（原理）

1. 定义描述状态的谓词：
   SITE(x, y): x 在 y 处
   HANG(w,y):w 悬挂在 y 处
   ONBOX (z): z 站在箱子上；
   HOLDS(z): z 手里拿着香蕉
   注：变元的个体域
   x 的个体域是{monkey, box}
   y 的个体域是{A, B, C}
   z 的个体域是{monkey}
   w 的个体域是{banana}

2. 初始状态和目标状态：

初始状态：

$$S_0 = \mathrm{SITE}(monkey, A) \wedge HANG(banana, \mathrm{C}) \wedge \mathrm{SITE}(box, \mathrm{B}) \wedge \neg ONBOX(monkey) \wedge \neg HOLDS(monkey)$$

目标状态：

$$S_g = \mathrm{SITE}(monkey, \mathrm{C}) \wedge \neg HANG(banana, \mathrm{C}) \wedge SITE(box, \mathrm{C}) \wedge ONBOX(monkey) \wedge HOLDS(monkey)$$

3. 定义操作：
   monkeygoto(u,v)：猴子从 u 走到 v 处
   movebox(v,w)：猴子推着箱子从 v 走到 w 处
   climbonto：猴子爬上箱子
   pick：猴子摘到香蕉

注：其实各操作是有条件和动作的

monkeygoto(u,v)

条件：$\neg\mathrm{ONBOX}(monkey)$、$\mathrm{SITE}(monkey, u)$

动作：删除表：$\mathrm{SITE}(monkey, \mathrm{u})$　　添加表：$\mathrm{SITE}(monkey, \mathrm{v})$

movebox(v,w)

条件：$\neg\mathrm{ONBOX}(monkey)$、$\mathrm{SITE}(monkey, \mathrm{v})$、$\mathrm{SITE}(\mathrm{box}, \mathrm{v})$

动作：删除表：$\mathrm{SITE}(monkey, \mathrm{v})$、$\mathrm{SITE}(\mathrm{box}, \mathrm{v})$

添加表：$\mathrm{SITE}(monkey, \mathrm{w})$、$\mathrm{SITE}(monkey, \mathrm{w})$

climbonto

条件：$\neg\mathrm{ONBOX}(monkey)$、$\mathrm{SITE}(monkey, \mathrm{B})$、$\mathrm{SITE}(\mathrm{box}, \mathrm{B})$

动作：删除表：$\neg\mathrm{ONBOX}(monkey)$

添加表： ONBOX(*monkey*)

pick

条件：ONBOX(*monkey*) 、SITE(box, B) 、HANG(*banana*, *B*) 、¬HOLDS(*monkey*)

动作：删除表： ¬HOLDS(*monkey*) 、 HANG(*banana*, *B*)

添加表： HOLDS(*monkey*) 、 ¬HANG(*banana*, *B*)

4. 求解：

$$S_0 = \text{SITE}(monkey, A) \wedge HANG(banana, B) \wedge \text{SITE}(box, C) \wedge \neg ONBOX(monkey) \wedge \neg HOLDS(monkey)$$

$$\Downarrow \textbf{monkeygoto(A,B)}$$

$$S_1 = \text{SITE}(monkey, \text{B}) \wedge HANG(banana, \text{C}) \wedge \text{SITE}(box, \text{B}) \wedge \neg ONBOX(monkey) \wedge \neg HOLDS(monkey)$$

$$\Downarrow \textbf{movebox(B,C)}$$

$$S_2 = \text{SITE}(monkey, \text{C}) \wedge HANG(banana, \text{C}) \wedge \text{SITE}(box, \text{C}) \wedge \neg ONBOX(monkey) \wedge \neg HOLDS(monkey)$$

$$\Downarrow \textbf{climbonto}$$

$$S_3 = \text{SITE}(monkey, \text{C}) \wedge HANG(banana, \text{C}) \wedge \text{SITE}(box, \text{C}) \wedge ONBOX(monkey) \wedge \neg HOLDS(monkey)$$

$$\Downarrow \textbf{pick}$$

$$S_{4(g)} = \text{SITE}(monkey, \text{C}) \wedge \neg HANG(banana, \text{C}) \wedge \text{SITE}(box, \text{C}) \wedge ONBOX(monkey) \wedge HOLDS(monkey)$$

二． 算法介绍

2.1 所用方法的一般介绍

为了简化问题输入，我们将 A、B、C 三点抽象化，分别用-1、0、1 来表示，同时为了使问题更加符合实际，我们在进行编程时考虑了各种情况，而不仅仅使初始状态为：猴子在 A，箱子在 B，香蕉在 C，我们的程序可以解决各种初始状态。

程序输入：monkey(-1/0/1)、box(-1/0/1)、banana(-1/0/1)、monkeybox(-1/1)

注: 1. 输入需要四个值，猴子、箱子、香蕉的位置以及猴子与箱子的相对位置

2. monkey、box、banana 分别可取-1、0、1 中任意一个值，-1 代表在 A 处、0

代表在 B 处、1 代表在 C 处
3. monbox 可取-1、1，-1 代表猴子没有站在箱子上、1 代表猴子站在箱子上
4. 但要符合实际情况：例如，若 moneybox = 1，而 monkey 和 box 取值不同的话，就与实际情况不符，是无效的。
程序输出：解决问题的每个状态步骤，使猴子最后能够摘到香蕉

2.2 算法伪代码

由于函数 MonkeyGoto(at,i), MoveBox(a,i), ClimbOntoBox(i), DownBox(i), PickBanana(i)实现比较简单,这里只给出 nextStep(i)的伪代码:

```
def nextStep(i):
    '''
    perform next step, a recursive procedure
    '''
    IF isGOAL:
        SHOWRESULT
        RETURN
    IF box == banana:
        IF monkey == banana:
            IF monbox == -1:
                ClimbOntoBox
                PickBanana
                nextStep
            ELSE:
                pickBanana
                nextStep
        ELSE:
            MonkeyGotoX
            nextStep
    ELSE:
        IF monkey == box:
            IF monbox == -1:
                MoveBoxToX
                nextStep
            ELSE:
                DownBox
                nextStep
        ELSE:
            MonkeyGotoX
            nextStep
```

三． 算法实现

3.1 实验环境与问题规模

实验环境	参数值
操作系统	Windows 10 中文版 64-bit
处理器	Intel(R) Core(TM) i5-6200U CPU @ 2.8GHz
Python 版本	Python 3.6
IDE	PyCharm PROFESSIONAL 2018.2

对于问题规模，对于该问题，由于问题步骤可见，肯定能在常数 c 步内得到解，故算法的时间复杂度为：O(1)
空间复杂度为：O(1)

3.2 数据结构

状态类：

```
class State:
    def __init__(self, monkey=-1, box=0,banana=1, monkeybox=-1):
        self.monkey = monkey    # -1:monkey at A, 0: monkey at B, 1: monkey at C
        self.box = box          # -1:box at A    0:box at B    1:box at C
        self.banana = banana      # banana at C,banana=1
        self.monkeybox = monkeybox # -1: monkey not on the box, 1: on the box
        # 默认初始状态为猴子在 A，盒子在 B，香蕉悬挂在 C，猴子没
        站在箱子上。
```

存储过程状态的列表：

```
Route = [None]*num
```

3.3 实验结果

我们的程序满足实际情况的任一初始状态，运行示例如下：
输入 1：猴子在 A 处，箱子在 B 处，香蕉在 C 处
（即，monkey=-1，box=0，banana=1，monbox=-1）
输出 1：

```
C:\Python37\python.exe C:/Users/孙月晴/Desktop/2018秋/3人工智能/实验1&2/实验1/Lab_1.py
please input state: monkey(-1/0/1), box(-1/0/1), banana(-1/0/1), monkeybox(-1/1):
-1 0 1 -1
The step sequences is:

Step 1 : Monkey go to B.

Step 2 : Monkey move box to C.

Step 3 : Monkey climb onto box

Step 4 : Monkey picking banana
```

输入 2：猴子在 B 处，箱子在 B 处，猴子站在箱子上，香蕉在 C 处（即，monkey=0，box=0，banana=1，monbox=1）

输出 2：

```
C:\Python37\python.exe C:/Users/孙月晴/Desktop/2018秋/3人工智能/实验1&2/实验1/Lab_1.py
please input state: monkey(-1/0/1), box(-1/0/1), banana(-1/0/1), monkeybox(-1/1):
0 0 1 1
The step sequences is:

Step 1 : Monkey climb down fron box

Step 2 : Monkey move box to C.

Step 3 : Monkey climb onto box

Step 4 : Monkey picking banana
```

输入 3：猴子在 C 处，箱子在 C 处，猴子不在箱子上，香蕉在 C 处（即，monkey=1，box=1，banana=1，monbox=-1）

输出 3：

```
C:\Python37\python.exe C:/Users/孙月晴/Desktop/2018秋/3人工智能/实验1&2/实验1/Lab_1.py
please input state: monkey(-1/0/1), box(-1/0/1), banana(-1/0/1), monkeybox(-1/1):
1 1 1 -1
The step sequences is:

Step 1 : Monkey climb onto box

Step 2 : Monkey picking banana
```

四． 总结及讨论

通过本次实验，我又熟悉了知识的表示方法，尤其是一阶谓词表示和产生式系统，是对作业题的一种实现，加深了对该类问题的印象，提高了解决该类问题的能力。

同时通过本次实验我也学到了 python 的深拷贝和浅拷贝的问题，在程序中，需要把上一状态复制给下一状态，然后在对新状态(下一状态)做相应的修改，最初，我是直接把第 i 个状态赋给了第 i+1 个状态，即 State[i+1] = State[i]，但总是结果出错。经过大量的调试，发现当改变 State[i+1]时，同时也会改变 State[i]，经过查阅相关资料，发现这是一个深浅拷贝的问题；由于我们 State 列表中存放的每一项是一个类的实例，当我们进行 State[i+1] = State[i]赋值操作时，实际上是进行了深拷贝，使两个状态指向了同一地址空间。

第二篇 实验 2 搜索策略

一. 问题描述

实验 2 要求采用且不限于课程第四章内各种搜索算法，编写一系列吃豆人程序解决下面列出的 8 个问题，包括到达指定位置以及有效地吃豆等。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

在本实验中，我们要使用 DFS，BFS，代价一致算法(UCS)，A*算法等搜索算法来为吃豆人规划路径以完成特定的目标。

二. 算法介绍及实现

2.0 通用的搜索算法

因为不同的搜索方法的不同之处仅仅在于 open 表的排序不同，因此我定义了一个通用的搜索算法解决问题 1-4

1. 算法思想:通过配置不同的参数实现 DFS,BFS,UCS,A*搜索

param1: problem (搜索算法要解决的问题对象) param2: open_type (搜索算法中的 open 表采用的数据结构) param3: PriorityQueue (bool 值,默认 False,是否采用优先队列数据结构,用于代价一致搜索和 A*搜索算法) param4: heuristic (启发式函数,默认为 nullHeuristic[返回值为 0,相当于没有该函数],用于 A*算法)
return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列

对于 DFS 和 BFS,open 表中的每个节点均是(state, actions)的二元组,其中 state 为状态,即为吃豆人所在的坐标(coord),actions 为从初始结点到达本状态所要执行的操作序列["South","North",....]

对于 UCS 和 A*,open 表中的每个节点均是((state, actions),cost)的二元组,其中 state 为状态,即为吃豆人所在的坐标(coord)，actions 为从初始结点到达本状态所要执行的操作序列["South","North",....], cost 为从初始结点到当前结点的代价(对

于 UCS)或从初始结点到当前结点的代价+启发式函数值(对于 A*)

2. 代码实现如下:

```
def genericSearchAlgorithm(problem, open_type, PriorityQueue=False,
heuristic=nullHeuristic):
    openTable = open_type()
    if PriorityQueue:
        openTable.push((problem.getStartState(), []),
                       heuristic(problem.getStartState(), problem))
    else:
        openTable.push((problem.getStartState(), []))
    closed = []
    while True:
        if openTable.isEmpty():
            return []
        coord, actions = openTable.pop()
        if problem.isGoalState(coord):
            return actions
        if coord not in closed:
            closed.append(coord)
            for successor_coord, action, step_cost in \
                    problem.getSuccessors(coord):
                if PriorityQueue:
                    openTable.push((successor_coord, actions + [action]),
                                   problem.getCostOfActions(actions +
[action]) +
                                   heuristic(successor_coord, problem))
                else:
                    openTable.push((successor_coord, actions + [action]))
```

2.1 问题 1 应用深度优先算法找到一个特定的位置的豆

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）。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 open_type 为实验提供栈结构 util.Stack 即可。

```
def depthFirstSearch(problem):
    return genericSearchAlgorithm(problem, util.Stack)
```

2.2 问题 2 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 S_0 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

全部扩展并考察之前，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）。

将通用搜索算法中的 open_type 设置为 util.Queue 即可。

```
def breadthFirstSearch(problem):
    return genericSearchAlgorithm(problem, util.Queue)
```

2.3 问题 3 代价一致算法

代价一致搜索是宽度优先搜索的一种推广，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。在代价树中，可以用 g(n)表示从初始节点 S_0 到节点 n_1 的代价，用 $c(n_1, n_2)$表示从父节点 n_1 到 n_2 的代价。这样，对节点 n_2 的代价有：$g(n_2) = g(n_1) + c(n_1, n_2)$

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 S_0 到任一节点 i 的路径代价记为 g(i)。从初始节点 S_0 开始扩展，若没有得到目标节点，则优先扩展最少代价 g(i)的节点，一直如此向下搜索，所以选择优先队列作为 open 表的数据结构。

将通用搜索算法中的 open_type 设置为 util.PriorityQueue,并将 PriorityQueue 设置为 True，表示使用优先级队列。

```
def uniformCostSearch(problem):
    return genericSearchAlgorithm(problem, util.PriorityQueue, True)
```

2.4 问题 4 A* 算法

A*搜索算法在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径（基于评估函数）。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。

将通用搜索算法的 open_type 设置为 util.PriorityQueue,PriorityQueue 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。

```
def aStarSearch(problem, heuristic=nullHeuristic):
    return    genericSearchAlgorithm(problem, util.PriorityQueue, True, heuristic)
```

2.5 问题 5 找到所有的角落

(1) 问题描述

本题要求实现 CornersProblem，从而使吃豆人将以最短的路径找到迷宫的四个角落作为目标。本题的重点即是如何表示吃豆人的状态，显然不能仅仅以吃豆

人的坐标作为状态了。

(2) 解决问题的方法

在实验要求中，在该问题的末尾有一句小提示(Tip)：新的状态只包含吃豆人的位置和角落的状态。所以我定义了 CornerSrate()类,包括:

```
class CornerState():
    def         __init__(self,corners,corners_visited=[False,False,False,
False]):# 初始化
        self.corners = tuple(zip(corners, corners_visited))
    def set_visited(self, position):# 设置角落为被访问
        new_corners = ()
        for corner in self.corners:
            if position == corner[0]:
                new_corners = new_corners + ((position, True),)
            else:
                new_corners = new_corners + (corner,)
        self.corners = new_corners
    def is_corner(self, position):# 判断当前位置是否是角落
        return any(map(lambda x: x[0] == position, self.corners))
    def is_all_visited(self):# 返回所有已被访问的角落
        return all(map(lambda x: x[1], self.corners))
    def unvisited_corners(self):# 返回所有未被访问的角落
        return map(lambda x: x[0], filter(lambda x: not x[1], self.corners))
```

getStartState 函数是返回开始时的状态,设置开始位置的状态为已访问

```
def getStartState(self):
    self.corners_state.set_visited(self.startingPosition)
    return self.startingPosition, self.corners_state
```

isGoalState 函数判断当前状态是否是目标状态。corners_state 与 self.corners((1,1), (1,top), (right, 1), (right, top))位置一一对应，若对应位置的食物未被吃掉，则 corners_state 中对应项为 False,若已经被吃则 corners_state 中对应项为 True。根据上述思路，isGoalState 函数如下所示：

```
def isGoalState(self, state):
    corner_state = state[1]
    return corner_state.is_all_visited()
```

另外，getSuccessors 函数应该是本部分的一个核心，考虑生成的后继节点，若后继节点即是未到达过的角落位置，则在该后继的 foods_statebool 中，要将对应的位置置为 True，表示已经到达过把该角落的食物吃掉了。Successors 是一个三元组列表(successor,action,stepCost),successor 表示下一步的状态,action 表示到达下一步所需的行动,stepCost 表示扩展到下一步的增量成本。伪代码如下:

```
def getSuccessors(self, state):
    successors = []
```

```
        x, y ← state[0]   # 获得坐标
        foodBool←state[1]   # 获得剩余角落食物状态
        foodXYList = []
        CONVERSATION foodBool 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 hitsWall:
            # hitsWall 为 True 表示有墙,为 False 表示无墙,此处需要其无墙
                nextState←(nextx, nexty)
                # 需要进行浅拷贝,否则会出错
                leftFoodBool←ShallowCopy(foodBool)
                # 判断(nextx, nexty)是否在剩余的食物坐标列表当中
                IF nextState IN foodXYList:
                    leftFoodBool[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, leftFoodBool),action, 1))
                  # 将下一状态加入 successors
            self._expanded += 1 # DO NOT CHANGE
            return successors
```

2.6 问题 6 角落问题（启发式）

(1) 问题描述

在实验的源码注释中也有相应注释要求：

```
This function should always return a number that is a lower bound on the
shortest path from the state to a goal of the problem; i.e.  it should be
admissible (as well as consistent).
```

即，本题要求为 CornersProblem 给出一个可纳且一致的启发式函数 cornersHeuristic，以通过扩展尽量少的节点找到 4 个角落，吃掉角落上的食物。

(2) 启发式函数的选取

启发式函数值应是真实代价的下界。对于迷宫而言，如果我们选择在无墙的迷宫中从某位置到达目标状态的最小代价作为启发式函数值，那么这个值一定是真实代价的下界。我们决定选择距离吃豆人当前位置最远的食物。启发式函数值即为吃豆人与距离其最远的剩余食物之间的距离,这里使用曼哈顿距离。

为何要选择最远的食物呢？首先，因为算法总是选择 f 值最小的节点进行扩展，因而这样选择启发式函数值可以使节点更倾向于扩展到剩余食物的中心位置，使节点距离剩余食物中的每一个都不至于很远。其次，这种选取方法还可以防止对某个食物节点的临近位置的过度探索，不会被局限在迷宫的某个局部，而

是全局考虑。最后我们可以定性地考虑一下，永远把离当前位置最远的那个豆子与当前位置之间的距离作为当前位置的启发式距离，也就是可以看做把最远的豆子作为目标点，因为我们肯定是吃完当前位置附近的豆子，最后再吃最远的豆子。

```
def cornersHeuristic(state, problem):
    corners_states = state[1]
    unvisited_corners = corners_states.unvisited_corners()
    if len(unvisited_corners) == 0:
        return 0
    return max(map(lambda x: util.manhattanDistance(state[0], x),
unvisited_corners))
```

2.7 问题 7 吃掉所有的豆子

(1) 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，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以找到所有的食物。

(2) 启发式函数的选取

一个始终走向距离其最近食物的吃豆人行走的路径不一定是最短路径，因而，我们决定考虑在所有剩余食物中选择具有代表性的食物，使用该食物位置与吃豆人当前位置之间的距离来代表吃豆人到达目标所需要的代价下界。对于具有代表性的食物，我们决定选择距离吃豆人当前位置最远的食物。启发式函数值即为吃豆人与距离其最远的剩余食物之间的距离。(这里为何要选择最远的食物已经在问题 6 中给出了解释)

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）：

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    food_coordinates = foodGrid.asList()
    if not food_coordinates:  # no food
        return 0
    farthest_food = max(map(lambda x: mazeDistance(x, position,
problem.startingGameState), food_coordinates))
    return farthest_food
```

2.8 问题 8 次最优搜索

(1) 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

(2) 解决问题的方法

通过阅读 ClosestDotSearchAgent 下的 registerInitialState 方法，发现本问题的运行逻辑是不断的使用 findPathToClosestDot 寻找到最近的食物的路径并将该路径加入到总路径之中，直到所有的食物均被吃完。因而在 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：

```
def isGoalState(self, state):
    x, y = state
    return self.food[x][y]
```

而在 findPathToClosestDot 函数中，我们要给出吃豆人到最近的食物的 actions 列表，因此决定使用一定会产生最优解的广度优先搜索算法（BFS）进行搜索。代码如下：

```
def findPathToClosestDot(self, gameState):
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)
    return search.bfs(problem)
```

三. 实验结果

本节给出运行 autograder.py 后对于每个问题的评分结果,篇幅所限,这里只给出部分扩展节点和得分。

(1) 问题 1

```
***      solution length: 130
***      nodes expanded:     146

### Question q1: 3/3 ###
```

(2) 问题 2

```
***      solution length: 68
***      nodes expanded:     269

### Question q2: 3/3 ###
```

(3) 问题 3

```
***      solution:         ['1:A->B', '0:B->C', '0:C->G']
***      expanded_states:    ['A', 'B', 'C']

### Question q3: 3/3 ###
```

(4) 问题 4

```
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###
```

(5) 问题 5

```
Question q5
===========

*** PASS: test_cases\q5\corner_tiny_corner.test
***     pacman layout:      tinyCorner
***     solution length:        28

### Question q5: 3/3 ###
```

(6) 问题 6

```
Question q6
===========

*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West',
path length: 106
*** PASS: Heuristic resulted in expansion of 1136 nodes
```

(7) 问题 7

```
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 5/4 ###
```

(8) 问题 8

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_9.test
***     pacman layout:      Test 9
***     solution length:        1

### Question q8: 3/3 ###
```

(9) 总成绩

```
Finished at 10:39:41

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3
------------------
Total: 26/25
```

四. 总结及讨论

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。经过本次实验，我对搜索算法有了进一步的理解。

本次实验，并不是从零开始进行，而是在已有的比较成熟的代码上进行补充，完成实验要求。我觉得虽然看起来比较容易，但是实际上是比较困难的。因为在补充的过程中，需要对原有的程序进行阅读，比如数据结构的定义、输入输出等。整体感觉，这种类型的实验非常好，可以说是一举两得，既可以锻炼编程能力也可以锻炼阅读别人代码的能力，对于计算机专业的学生来说，这两者都是缺一不可的。

对于本次实验所用到的通用搜索算法，我认为还可以进行一定程度的优化，利用老师上课中讲到的通用搜索算法会更好一些，会得到更优解。

老师课件中的算法相较于本实验中给出的搜索算法多出了对返回指针的考虑。在本实验中，在搜索中的每个节点可以有多个父节点（可能会走或扩展重复的点），是通过节点中不同的 actions 数组来实现的，而课件中的算法每个节点只能有一个父节点，是用指向父节点的指针实现的。课件中的算法的好处在于，如果扩展到了已经扩展过的节点，可以根据开始节点到当前节点的代价，变更节点指向父节点的返回指针，从而避免重复扩展或遍历某个点的可能。

第三篇　实验 3 不确定性推理

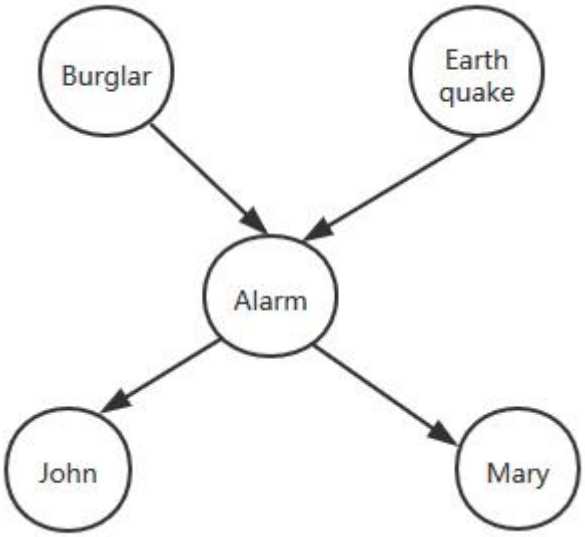
一. 问题描述

1.1 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。

1.2 问题的形式化描述

如下图所示的贝叶斯网络,给定各节点的CPT表,实现基于贝叶斯网络的推理



二. 算法介绍

2.1 所用方法的一般介绍

根据贝叶斯网络的定义,对于子节点变量 X_i,其取值 x_i 的条件概率仅依赖于 X_i 的所有父节点的影响,用 $par(X_i)$表示 X_i 的所有父节点的相应取值,$P(X_i|par(X_i))$ 是节点 X_i 的条件概率分布函数,则对 X 的所有节点有如下联合概率分布:

$$P(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} P(X_i \mid par(X_i))$$

这个公式就是贝叶斯网络的联合概率分布,可知,贝叶斯网络的联合分布比全联合分布简单的多,其计算复杂度比全联合概率分布小得多。

贝叶斯网络精确推理的主要方法包括基于枚举的算法和基于变量消元的算法。我们这里使用基于枚举的算法,使用全联合概率分布去推断查询变量的后验概率: $P(X \mid s) = \alpha P(X, s) = \alpha \sum_{Y} P(X, s, Y)$

各变量的含义：X 表示查询变量；s 表示一个观察到的特定事件；Y 表示隐含变量集$\{y_1, y_2, ..., y_m\}$；α是归一化常数,用于保证相对于 X 的所有取值的后验概率总和等于 1.

为了对贝叶斯网络进行推理,可利用贝叶斯网络的概率分布公式,将 P(X,s,Y)改写成条件概率乘积的形式,这样就可以通过先对 Y 的各枚举值求其条件概率乘积,再对各条件概率求总和的方式计算查询变量的条件概率。

2.2 算法

(1) 解析问题查询文件

根据问题的形式,解析问题

```
P(Burglar | Alarm=true)

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

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

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

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

存入查询列表中,querieslist 中的数据结构如下:

[[('Burglar',), ('Alarm', 'true')],
[('Alarm',), ('Earthquake', 'true'), ('Burglar', 'true')],
[('Burglar',), ('John', 'true'), ('Mary', 'false')],
[('Burglar',), ('John', 'true'), ('Mary', 'false')],
[('Burglar',), ('Alarm', 'true'), ('Earthquake', 'true')]]

第一个元组表示查询变量,第二个元组表示观察到的特定事件及其真值,代码实现比较简单,进行正则匹配和字符串切分即可,不再复述。

(2) 解析贝叶斯网络文件

根据给定的输入文件格式解析, 构造贝叶斯网络如下:

{'Alarm': [('Burglar', 'Earthquake'), ('0.001', '0.999'), ('0.29', '0.71'), ('0.94', '0.06'), ('0.95', '0.05')],
'John': [('Alarm',), ('0.05', '0.95'), ('0.9', '0.1')],
'Burglar': [(None,), ('0.001', '0.999')],
'Earthquake': [(None,), ('0.4', '0.6')],

'Mary': [('Alarm',), ('0.01', '0.99'), ('0.7', '0.3')]}
用 Map 存储,key 表示当前节点,value 列表的第一个元组表示当前节点的父节点,没有则用 None 表示,后面的元组表示 CPT 表中的概率分布。

这里主要说明怎么读每个节点的 CPT 表。首先遍历每一列, 如果所在列的和为 0 说明其没有父节点, 则父节点用 None 表示,否则遍历该列,如果值为 1 则说明此节点为当前节点的父节点,加入元组中,特别注意行数计数器 N 的变化,考虑到代码的通用性,这里不能指定特定的值。具体代码实现如下:

```
# 读每个节点的CPT 表
for i in range(0, N):
    # 遍历每一列,如果所在列的和为0 说明其没有父节点
    for k in range(0, N):
        temp[k] = parentTable[k][column]
        sum += temp[k]
    column += 1
    if sum == 0:
        CPT = []
        CPT.append((None,))
        CPT.append(tuple(lines[row].split(' ')))
        row += 1
        Network[names[i]] = CPT
    else:
        CPT = []
        parents = []
        for col in range(0, N):
            if parentTable[col][i] == 1:
                parents.append(names[col])
        CPT.append(tuple(parents))
        for k in range(row, row + pow(2, sum)):
            CPT.append(tuple(lines[k].split(' ')))
            row += 1
        Network[names[i]] = CPT
    sum = 0
```

(3) 求联合概率分布

根据贝叶斯网络的联合分布函数 $P(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} P(X_i \mid par(X_i))$

对于没有父节点的节点,直接取其真值对应的概率相乘,对于有父节点的节点,遍历其父节点,查看其父节点是真还是假,然后根据当前节点的真假取出 CPT 表中相应的表项相乘。代码实现如下:

```
# 如果没有父节点
if cmp((None,), parents[0]) == 0:
    if value == '0':
        # F 时的概率
        sums *= float(parents[1][1])
```

```
    else:
        # T 时的概率
        sums *= float(parents[1][0])
# 有父节点
else:
    number_of_father = len(parents[0])  # 父节点个数
    chooseposition = 0
    for j in range(0, number_of_father):  # 遍历父节点
        pat = parents[0][j]
        value2 = TFsequence[names.index(pat)]  # 查看父节点是真还是假
        if value2 == '0':
            continue
        else:
            chooseposition += pow(2, number_of_father - j - 1)
    if value == '0':
        sums *= float(parents[chooseposition + 1][1])  # 当前节点为F
    else:
        sums *= float(parents[chooseposition + 1][0])  # 当前节点为T
```

(4) 进行贝叶斯网络的推理,求解查询问题的概率

我们知道,在基于枚举的贝叶斯网络推理中,需要对查询变量的不同真假值分别进行计算,然后求得归一化因子进行归一。由于前面求联合概率设计的比较简洁,所以这里我做了一个转化,将查询变量的后验概率计算形式变为:

$$P(X \mid s) = \frac{\sum_{Y} P(X, s, Y)}{P(s)}$$

这里我将查询问题的各变量的真假值做了一个映射:将问题的条件节点真假值映射为 1 和 0,如果是条件节点且为真则为 1,为假则为 0,否则为 2,然后求联合概率的序列:

```
jointPro = [[0 for i in range(2)] for j in range(0, pow(2, N))]
# 生成联合概率序列,0 表示 F,1 表示 T
for i in range(0, pow(2, N)):
    jointPro[i][0] = bin(i).replace('0b', '').zfill(N)
    temp = jointPro[i][0]
    # 计算联合概率
    jointPro[i][1] = JointDistribution(temp.zfill(N), Network, names)
```

这里关键是计算后验概率的分子和分母,计算分母时,隐含变量分别取不同的真假值然后相加,计算分子先指定查询变量的真值为真,隐含变量也是分别取不同的真假值,计算其联合分布概率。相除后用 1 减即是查询变量为假时的概率。代码实现如下:

```
# 计算分母
for k in range(0, pow(2, N)):
    for j in range(0, N):
        if cmp(seq[j], '2') != 0 and cmp(seq[j], jointPro[k][0][j]) != 0:
            flag = 1
```

```
    if flag == 0:
        denominator += float(jointPro[k][1])
        listresult.append(jointPro[k])
    flag = 0
# 计算分子
for j in range(0, len(listresult)):
    if cmp(listresult[j][0][positions], '1') == 0:
        numerator += float(listresult[j][1])
print [numerator / denominator, 1 - numerator / denominator]
```

三. 算法实现

3.1 实验环境与问题规模

1. 实验环境:

实验环境	参数值
操作系统	Windows 10 中文版 64-bit
处理器	Intel(R) Core(TM) i5-6200U CPU @ 2.8GHz
Python 版本	Python 2.7
IDE	PyCharm PROFESSIONAL 2018.2

2.问题规模: 贝叶斯网络之所以能够大大降低计算复杂度,一个重要原因是它具有的局部化特征。节点仅受该节点的父节点的直接影响,而不受其他节点的直接影响,因此贝叶斯网络是一种线性复杂度的方法。在一个包含 n 个布尔随机变量的贝叶斯网络中,如果每个随机变量最多只受 k 个随机变量的直接影响,k 是某个常数,则贝叶斯网络最多可由 $2^k \times n$ 个数据描述,因此其复杂度是线性的。

3.2 数据结构

1. 查询列表 querieslist 中的数据结构,这里以 burglarqueries.txt 为例:
[[('Burglar',), ('Alarm', 'true')],
[('Alarm',), ('Earthquake', 'true'), ('Burglar', 'true')],
[('Burglar',), ('John', 'true'), ('Mary', 'false')],
[('Burglar',), ('John', 'true'), ('Mary', 'false')],
[('Burglar',), ('Alarm', 'true'), ('Earthquake', 'true')]]
第一个元组表示查询变量,第二个元组表示观察到的特定事件及其真值

2. 贝叶斯网络的数据结构,这里以 burglarnetwork.txt 为例:
{'Alarm': [('Burglar', 'Earthquake'), ('0.001', '0.999'), ('0.29', '0.71'), ('0.94', '0.06'), ('0.95', '0.05')],
'John': [('Alarm',), ('0.05', '0.95'), ('0.9', '0.1')],
'Burglar': [(None,), ('0.001', '0.999')],
'Earthquake': [(None,), ('0.4', '0.6')],
'Mary': [('Alarm',), ('0.01', '0.99'), ('0.7', '0.3')]}
用 Map 存储,key 表示当前节点,value 列表的第一个元组表示当前节点的父节点,没有则用 None 表示,后面的元组表示 CPT 表中的概率分布。

3.3 实验结果

对于 burglarqueries.txt 文件,输出结果为:

```
[0.0080390096348893, 0.9919609903651107]
[0.9500000000000002, 0.04999999999999982]
[0.0034174642570400115, 0.99658253574296]
[0.0034174642570400115, 0.99658253574296]
[0.003268423587696966, 0.996731576412303]
```

对于 carqueries.txt 文件,输出结果为:

```
[0.8946164361643469, 0.10538356383565306]
[0.008345656161823346, 0.9916543438381766]
```

四. 总结及讨论

通过本次实验,我对贝叶斯网络的推理有了更深刻的了解。处理贝叶斯网络推理的一般步骤是:首先确定各相邻节点之间的初始条件概率分布,然后对各证据节点取值,接着选择适当推理算法对各节点的条件概率分布进行更新,最终得到推理结果。贝叶斯网络能实现简化计算的最根本基础是条件独立性,即一个节点与它的祖先节点之间是条件独立的,这也导致贝叶斯网络是一种线性复杂度的算法。

其实在完成实验的过程中,除了精确推理的基于枚举的算法和基于变量消除的算法外,我也了解到了一些贝叶斯网络的近似推理算法,它是在不影响推理正确性的前提下,通过适当降低推理精确度来提高推理效率的一类方法。比如马尔科夫链蒙特卡洛算法通过对前一个世界状态进行随机改变来生成下一个问题状态,通过对某一个隐变量进行随机采样来实现对随机变量的改变。接下来我也打算尝试用近似推理解决这个问题。

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

[2] 林尧瑞，马少平.人工智能导论[M].北京：清华大学出版社，2000

[3] Wesley Chun.Python 核心编程[M].北京：人民邮电出版社，2016.05

[4] Thomas H.Cormen 等.算法导论[M].北京：机械工业出版社，2012.12：第六部分