哈爾濱工業大學

人工智能导论

实验报告

姓　名　　张兴华

学　号　　1150310320

班　级　　1503103

小组成员	学号
张兴华	1150310320
魏博文	1150310305
宋键	1150310303

目 录

前 言

0.1 实验环境说明

实验 1 和实验 2 的实验环境如下：

实验环境	参数值
操作系统	Windows 8.1 中文版 64-bit
处理器	Intel(R) Core(TM) i7-5500HQ CPU @ 2.40GHz 2.40GHz
内存	8.00GB RAM
Python 版本	Python 2.7
IDE	PyCharm PROFESSIONAL 2017.2

注：所得的所有实验结果，均是在此实验环境下获得。

0.2 其他事项

我同时提交了 doc 版本和 pdf 版本的实验报告，为了避免 Microsoft Word/WPS 的语法检查影响阅读报告和附录代码，建议阅读 pdf 版本的实验报告。

虽然时间紧迫，但经过我们小组的不懈努力，仍完成了全部的实验内容。实验 2 中的问题 8 虽然未做要求，我们小组出于兴趣还是写了。

总体感觉实验 2 的吃豆人还是非常有意思的，这也是我们很快做完整个实验的动力。

如有需要，可联系 Tel : 18804652328

第一篇　实验 1 知识表示

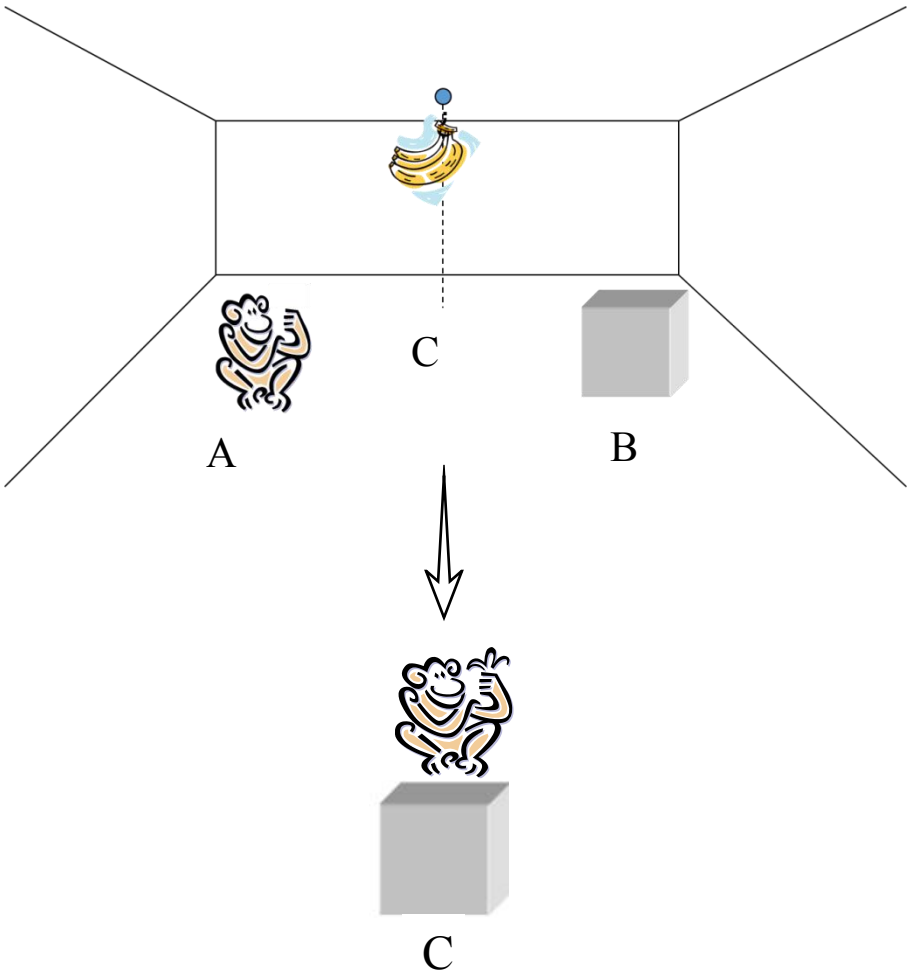
第一章 简　介

本篇实验参照课程第二部分讲授的知识表示方法完成，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，具体问题描述请参照 1.1 节。

1.1 问题描述

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。

1.2 问题形式化描述



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：猴子和箱子同处 C 处，且猴子站在箱子 B 上摘到香蕉

1.3 解决方案介绍

1. 定义描述状态的谓词：
   ON(x, y): x 在 y 处；HANG(w,y):w 悬挂在 y 处
   MONBOX (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：
$S_0 = \mathrm{ON}(monkey, A) \wedge HANG(banana, \mathrm{C}) \wedge \mathrm{ON}(box, \mathrm{B}) \wedge \neg MONBOX(monkey) \wedge \neg HOLDS(monkey)$
目标状态：
$S_g = \mathrm{ON}(monkey, \mathrm{C}) \wedge \neg HANG(banana, \mathrm{C}) \wedge ON(box, \mathrm{C}) \wedge MONBOX(monkey) \wedge HOLDS(monkey)$

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
   reach：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：$\neg\mathrm{MONBOX}(monkey)$、$\mathrm{ON}(monkey, u)$
动作：删除表：$\mathrm{ON}(monkey, \mathrm{u})$　　添加表：$\mathrm{ON}(monkey, \mathrm{v})$
movebox(v,w)
条件：$\neg\mathrm{MONBOX}(monkey)$、$\mathrm{ON}(monkey, \mathrm{v})$、$\mathrm{ON}(\mathrm{box}, \mathrm{v})$
动作：删除表：$\mathrm{ON}(monkey, \mathrm{v})$、$\mathrm{ON}(\mathrm{box}, \mathrm{v})$
　　添加表：$\mathrm{ON}(monkey, \mathrm{w})$、$\mathrm{ON}(monkey, \mathrm{w})$
climbonto
条件：$\neg\mathrm{MONBOX}(monkey)$、$\mathrm{ON}(monkey, \mathrm{B})$、$\mathrm{ON}(\mathrm{box}, \mathrm{B})$
动作：删除表：$\neg\mathrm{MONBOX}(monkey)$
　　添加表：$\mathrm{MONBOX}(monkey)$
reach
条件：$\mathrm{MONBOX}(monkey)$、$\mathrm{ON}(\mathrm{box}, \mathrm{B})$、$\mathrm{HANG}(banana, B)$、$\neg\mathrm{HOLDS}(monkey)$
动作：删除表：$\neg\mathrm{HOLDS}(monkey)$、$\mathrm{HANG}(banana, B)$

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

4. 求解：

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

monkeygoto(A,B)

$$S_1 = \mathrm{ON}(monkey, \mathrm{B}) \wedge HANG(banana, \mathrm{C}) \wedge \mathrm{ON}(box, \mathrm{B}) \wedge \neg MONBOX(monkey) \wedge \neg HOLDS(monkey)$$

movebox(B,C)

$$S_2 = \mathrm{ON}(monkey, \mathrm{C}) \wedge HANG(banana, \mathrm{C}) \wedge \mathrm{ON}(box, \mathrm{C}) \wedge \neg MONBOX(monkey) \wedge \neg HOLDS(monkey)$$

climbonto

$$S_3 = \mathrm{ON}(monkey, \mathrm{C}) \wedge HANG(banana, \mathrm{C}) \wedge \mathrm{ON}(box, \mathrm{C}) \wedge MONBOX(monkey) \wedge \neg HOLDS(monkey)$$

reach

$$S_{4(g)} = \mathrm{ON}(monkey, \mathrm{C}) \wedge \neg HANG(banana, \mathrm{C}) \wedge \mathrm{ON}(box, \mathrm{C}) \wedge MONBOX(monkey) \wedge HOLDS(monkey)$$

第二章 算法介绍

2.1 所用方法一般介绍

为了使问题更具一般化，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，那么如果问题的初始状态为猴子在箱子上，箱子在 B 处，香蕉在 C 处，那这个问题又如何求解呢？不用担心，经过我们小组成员的努力，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)、monbox(-1/1)
（上述输入需要四个值，包括猴子、箱子、香蕉的位置以及猴子与箱子的相对位置，其中，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，monbox 可取-1、1 中的任意一个值，-1 代表

猴子没有站在箱子上、1 代表猴子站在箱子上，但要符合实际情况：例如，若 monbox = 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：问题的解决方案，使猴子最后能够摘到香蕉

2.2 算法伪代码

```
def monkeygoto(b,a):
    '''
    function monkeygoto,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 movebox(b,a):
    '''
    function movebox,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 climbonto(i):
    '''
    function climbonto,the monkey climb onto the box
    '''
    "Monkey climb onto the box"
    Monbox = 1
```

```
def climbdown(i):
    '''
    function climbdown,monkey climb down from the box
    '''
    "Monkey climb down from the box"
    Monbox = -1
```

```
def reach(i):
    '''
    function reach,if the monkey,box,and banana are at the same place,and
        monkey on box,the monkey reach banana
    '''
   "Monkey reach the banana"
```

```
def nextStep(i):
    '''
    perform next step, 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
```

第三章 算法实现

3.1 实验环境与问题规模

实验环境已经在前面指出，在此不再赘述。
对于问题规模，我们算法的时间复杂度为：O(1)（对于该问题来说，由于问题步骤可见，肯定能在常数 c 步内得到解，故为 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, monbox=-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.monbox = monbox
        # -1: monkey not on the box    1: monkey on the box
        默认初始状态为猴子在 A 处，盒子在 B 处，香蕉悬挂在 C 处，猴子没有
        站在箱子上。
```

存储过程状态的列表：

```
Routesave = [None]*num
```

3.3 实验结果

对于我们在前面提出的问题，如果初始状态不是题目假设的初始状态（即，猴子在 A 处，箱子在 B 处，香蕉在 C 处），而是满足实际情况的任一初始状态，比如：猴子在箱子上，箱子在 B 处，香蕉在 C 处，那么接下来就看一下我们的程序处理结果：

- **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：**

```
D:\Python2.7\python.exe D:/desktop/MpickBanana.py
please input state: monkey, box, banana, ifMonkeyIsOnBox:
-1 0 1 -1
Result to problem:
Step 1 : Monkey go to B

Step 2 : Monkey move box to C

Step 3 : Monkey climb onto the box

Step 4 : Monkey reach the 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：**

```
D:\Python2.7\python.exe D:/desktop/MpickBanana.py
please input state: monkey, box, banana, ifMonkeyIsOnBox:
0 0 1 1
Result to problem:
Step 1 : Monkey climb down from the box

Step 2 : Monkey move box to C

Step 3 : Monkey climb onto the box

Step 4 : Monkey reach the 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：**

```
D:\Python2.7\python.exe D:/desktop/MpickBanana.py
please input state: monkey, box, banana, ifMonkeyIsOnBox:
1 1 1 -1
Result to problem:
Step 1 : Monkey climb onto the box

Step 2 : Monkey reach the banana
```

注：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。

第四章 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。由于第一个实验较简单，我们小组采用两种语言实现（Python、C++）。

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]，那这是为什么呢？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(对象)，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，所以在对其中一个修改后，另外一个也会发生相应改变。明白了问题所在之后，我重新定义了 copyState 函数，用于状态复制，其具体实现如下：

```
def copyState(state_i):
    state_iplus1 = State()
    state_iplus1.monkey = state_i.monkey
    state_iplus1.box = state_i.box
    state_iplus1.banana = state_i.banana
    state_iplus1.monbox = state_i.monbox
    return state_iplus1
```

第二篇　实验 2 搜索策略

第一章 简　介

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。

第二章 算法介绍及实现

在本章，我将针对实验 2 吃豆人中的八个问题做简单介绍，分析各个问题并说明解决问题的方法及算法实现。

2.1 问题 1 应用深度优先算法找到一个特定的位置的豆

2.1.1 问题描述

本问题要求使用深度优先搜索给出吃豆人的路径，但是本问题还要求写出一个完整的通用搜索算法。我们首先来看一下通用的搜索算法。

2.1.2 通用的搜索算法

根据实验要求附录中所给出的算法伪代码，其实我们可以比较容易地写出实验要求的完整的通用搜索算法

Graph Search Pseudo-Code

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

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。经过一番尝试（如，打印程序的中间结果），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，其即是路径信息本身。

本通用搜索算法的参数，返回值信息等，如下伪代码所述：

```
class GenericSearch:
    """
    通用的搜索算法类,通过配置不同的参数可以实现 DFS,BFS,UCS,A*搜索
    param1: problem (搜索算法要解决的问题对象)
    param2: data_struct_type (搜索算法中的 open 表采用的数据结构)
    param3: usePriorityQueue (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)
    """

    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):
        """
        节点(node):
```

```
        对于 DFS 和 BFS,open 表中的每个节点(node)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(node)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*)

        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
```

2.1.3 深度优先搜索问题

在深度优先搜索中，首先扩展最新产生的(即最深的)节点，是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）。

2.1.4 解决问题的方法

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。

```
def depthFirstSearch(problem):
    """
    深度优先搜索算法
    实例化类 GenericSearch 为 dfs,使用 util.py 中定义的
    数据结构栈 Stack 来作为 open 表的数据结构
    param: problem  搜索算法要解决的问题对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    dfs = GenericSearch(problem, util.Stack)
    return dfs.genericSearch()
```

2.2 问题 2 宽度优先算法

2.2.1 宽度优先搜索问题

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）。

2.2.2 解决问题的方法

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

```
def breadthFirstSearch(problem):
    """
     宽度优先搜索算法
     实例化类 GenericSearch 为 bfs,使用 util.py 中定义的
     数据结构队列 Queue 来作为 open 表的数据结构
     param: problem  搜索算法要解决的问题对象
     return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
     """
     bfs = GenericSearch(problem, util.Queue)
     return bfs.genericSearch()
```

2.3 问题 3 代价一致算法

2.3.1 代价一致搜索问题

前面的各种搜索策略中，实际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。但实际问题中，这种假设不现实，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 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。

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，所以我们选择优先队列作为 open 表的数据结构。

2.3.2 解决问题的方法

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。

```
def uniformCostSearch(problem):
    """
    代价一致搜索算法
    实例化类 GenericSearch 为 ucs,使用 util.py 中定义的
```

```
    数据结构优先队列 PriorityQueue 来作为 open 表的数据结构
    param: problem  搜索算法要解决的问题对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    ucs = GenericSearch(problem, util.PriorityQueue, True)
    return ucs.genericSearch()
```

2.4 问题 4 A* 算法

2.4.1 A* 搜索问题

A*搜索算法是一种在图形平面上，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(Breadth First Search)和 Dijkstra 算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径（基于评估函数）。

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。

2.4.2 解决问题的方法

将通用搜索算法的 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 -a fn=astar,heuristic=manhattanHeuristic 命令来测试我们的 A*程序。

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """
    A*算法
    实例化类 GenericSearch 为 astar,使用 util.py 中定义的
    数据结构优先队列 PriorityQueue 来作为 open 表的数据结构
    同时指定启发式函数 heuristic,使用该启发式函数来估算当前结点到目标结点的代价
    param1: problem  搜索算法要解决的问题对象
    param2: heuristic  启发式函数对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    astar = GenericSearch(problem, util.PriorityQueue, True, heuristic)
    return astar.genericSearch()
```

2.5 问题 5 找到所有的角落

2.5.1 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。

2.5.2 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，并且参考了官网（http://ai.berkeley.edu/search.html）英文中的实验要求，看到了下面这句话：

> To receive full credit, you need to define an abstract state representation that *does not* encode irrelevant information (like the position of ghosts, where extra food is etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

实验要求不能包含不相关信息，包括额外的食物的位置，即食物的坐标(x,y)

因此我把状态 state 表示为一个形如(coord, foods_statebool)的二元组，其中 coord 为吃豆人所在的位置坐标(x,y)。foods_statebool 为四个角落食物的状态列表，初始值为 [False, False, False, False]，foods_statebool 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 foods_statebool 中对应项为 False,若已经被吃则 foods_statebool 中对应项为 True。

根据上述思路，isGoalState 函数如下所示：

```
def isGoalState(self, state):
    """
        Returns whether this search state is a goal state of the problem.
    """
    # 根据状态表示方法,state[1]即为 foods_statebool,若其中每一项均为 True,
    # 则说明角落的食物均已被吃掉,目标达成
    return state[1][0] and state[1][1] and state[1][2] and state[1][3]
```

另外，getSuccessors 函数应该是本部分的一个核心，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。

```
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）

2.6.1 问题描述

虽然在实验要求中，没有说明启发式函数是怎样的，但是在官网上有下面这句话：

Implement a non-trivial, consistent heuristic for the CornersProblem in cornersHeuristic.

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 给出一个非平凡且一致的（non-trivial, consistent）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。

2.6.2 非平凡且一致的（non-trivial, consistent）启发式函数

只要启发式函数有非零的返回值，其就是非平凡的(non-trivial)，而对于一致的(consistent)启发式函数，其在维基百科上的形式化定义如下：

Formally, for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost

of reaching the goal from P. That is:

$h(N) \leq c(N,P) + h(P)$ and

$h(G) = 0$.

where

h is the consistent heuristic function

N is any node in the graph

P is any descendant of N

G is any goal node

c(N,P) is the cost of reaching node P from N

A consistent heuristic is also admissible, i.e. it never overestimates the cost of reaching the goal (the opposite however is not always true!).

定义中提到的 admissible 即可用的启发式函数即是 A*算法相对于 A 算法所增加的性质，其在维基百科上的形式化定义如下

n is a node

h is a heuristic

$h(n)$ is cost indicated by h to reach a goal from n

$h^*(n)$ is the actual cost to reach a goal from n

$h(n)$ is admissible if

$$\forall n, h(n) \leq h^*(n)$$

一个一致的启发式函数必然是可用的。而在下面对本问题的启发式函数的讨论中，将对可用性和一致性分别给出简要的证明。

2.6.3 启发式函数的选取及可用性的证明

在实验课程序检查时，已现场给助教进行启发式函数的证明，现将证明过程简要描述如下。

启发式函数值应是真实代价的下界。而对于迷宫而言，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。而在无墙的迷宫中从某位置到达目标状态的最小代价即为我们所选择的启发式函数，记为 h(n)，其中 n 为某节点。

而无墙的迷宫中从某位置到达目标状态的最小代价该如何求得呢？该代价是否是从当前节点一直寻找最近的角落直到达到目标状态呢？答案是否定的，这很容易找出一个反例，我在这里就不再赘述了。

为了得到在无墙条件下的最小代价，我们决定根据剩余角落的个数分类讨论。

当剩余 0 个食物时，说明已到达目标状态，这时应返回 0。

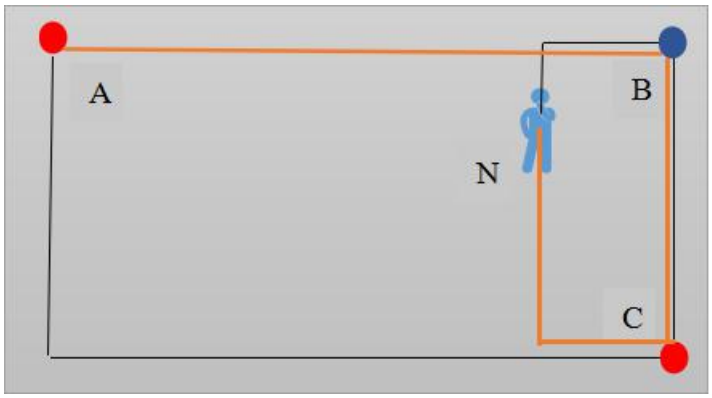
当剩余 1 个食物时，最小代价即为当前坐标与剩余角落的曼哈顿距离。

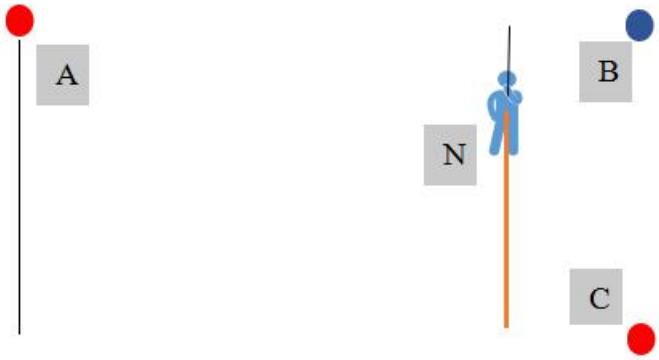
当剩余 2 个食物时，在到达一个剩余角落之前无论怎样走，到达一个剩余角落之后要花费的代价是固定的，即是这两个角落之间的曼哈顿距离，因而应先到达距离当前结点较近的剩余角落（曼哈顿距离意义上的较近，下同），然后从这一角落到达另一剩余的角落。启发式函数值记为这两段路程的曼哈顿距离之和。

当剩余食物个数为 3 时，这种情况需要我们仔细考虑一下，首先我们在下图中表示出剩余的 3 个食物 A、B、C，A 和 C 是边缘的两个食物，B 为中间的食物。接下来我们可以分两种情况来考虑：

当吃豆人的位置 N 离 A 或离 C 最近时，那么我们的最短路径为 N 到 A 或 C 中最近的那个点的曼哈顿距离加上一倍的长和一倍的宽。

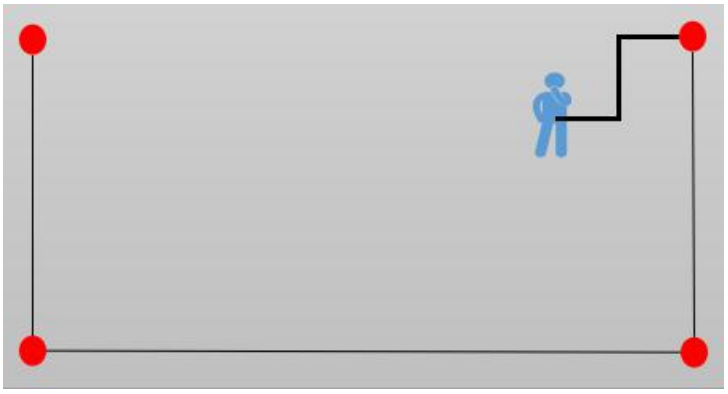
当吃豆人的位置 N 离 B 最近时，最短路径是否是 N 到 B 的曼哈顿距离在加上一个长和宽的固定值呢？下面我们详细证明一下，如下图：黑色细线代表的路径 N->B->C->A 是先选择离当前位置 N 最近的 B 位置，橘黄色粗线代表的路径 N->C->B->A 是先选择离当前位置 N 最近的边缘点 C 位置。那么比较这两段的曼哈顿距离，我们可以从下面的第二个图中得出结论：在互相抵消掉等长的路径之后，剩余的黑色细线路径显然大于橘黄色粗线路径，因此最短路径应为 N 到最近的边缘结点 C 的曼哈顿距离加上一倍的长和一倍的宽。





所以当剩余 3 个食物时，我们的启发式函数值取当前位置到较近的边缘点的曼哈顿距离加上一倍的迷宫长和宽。

当剩余 4 个食物时，当吃豆人到达任意一个角落后，找到剩余角落的最小代价为两倍的迷宫宽与一倍的迷宫长之和（如下图所示）



因而若想使代价最小，需让吃豆人到达第一个角落的代价最小，故吃豆人应首先到达距离其最近的角落，而启发函数值也就等于吃豆人到距离其最近的角落的曼哈顿距离加两倍的迷宫宽和一倍的迷宫长。

至此，启发式函数的所有情况均讨论完成，下面是对应的代码实现：

```
def cornersHeuristic(state, problem):
    corners ⟵ problem.corners    # These are the corner coordinates
    walls ⟵ problem.walls
    # These are the walls of the maze, as a Grid (game.py)
    coord, food_state ⟵ state    # coord 即为吃豆人的当前位置坐标
    COVERSATION food_state TO food_lst #  获得剩余的角落坐标列表
    top = problem.top
    right = problem.right
    length ⟵ GET(Length)
    width ⟵ GET(Width)
    food_num ⟵ GET Food Num    #  剩余角落食物数量
    IF food_num == 0:    #  剩余 0 个,说明达成目标,返回 0
        return 0
    ELIF food_num == 1:
        #  剩余 1 个,返回当前位置 coord 到剩余的这个角落的曼哈顿距离
        return ManhattanDistanceOf CurrentLocation and LeftfoodLocation
    ELIF food_num == 2:
        #  剩余 2 个,返回距离当前位置较近的角落到 coord 的曼哈顿距离与剩余的两个角落之间的曼哈顿距离的和
        mindist ⟵ min(ManhattanDistanceOf CurrentLocation
                                                and LeftfoodLocations)
        return mindist Plus ManhattanDistanceOf LeftfoodLocations
    ELIF food_num == 3:    #  剩余 3 个的处理原理见上面证明所述
        leftedgefood ⟵ GETtwoEdgesfoodLocation
        #  得到非中间位置的角落坐标列表
        man_dist_lst ⟵ min(ManhattanDistanceOf CurrentLocation
                                            and LeftEdgesfoodLocations)
        return man_dist_lst Plus MazeWidth Sub 2 Plus MazeLengthth Sub 2
    ELIF food_num == 4:
```

```
        man_dist_lst <—— min(ManhattanDistanceOf CurrentLocation
                                                    and LeftfoodLocations)
        return man_dist_lst Plus 2*( MazeWidth Sub 2) Plus MazeLengthth Sub 2
```

2.6.4 启发式函数一致性的简要证明

实验需要启发式函数满足更强的条件——一致性（Consistency）。下面简单的证明一下我们所选择的启发式函数的一致性。

欲证明一致性，就是要证明启发式函数满足一致性的形式化定义：

Formally, for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P. That is:

$h(N) \leq c(N, P) + h(P)$ and

$h(G) = 0$.

where

h is the consistent heuristic function

N is any node in the graph

P is any descendant of N

G is any goal node

c(N,P) is the cost of reaching node P from N

A consistent heuristic is also admissible, i.e. it never overestimates the cost of reaching the goal (the opposite however is not always true!).

证明：

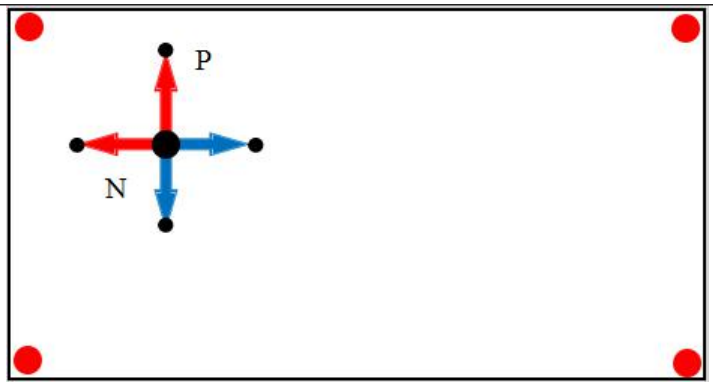
证明中使用在上面的一致性形式化定义中定义的符号

当吃豆人从 N 节点执行了一个 action 到达 P 时，其在 P 处按当前定义的启发式函数规则搜索所要找到的第一个角落可能与 N 相同，也可能不同。为了与上面的可用性对应，下面我们将按照剩余的食物个数，对这两种情况分别讨论：

- **剩余 4 个食物**
  - ✓ **相同时**

如果 N 走向 P 是沿着 N 与 N 会到达的第一个角落的曼哈顿距离变小的方向(下图中的红色箭头方向，也就是左上角)，则显然 $h(N) = c(N,P) + h(P)$。

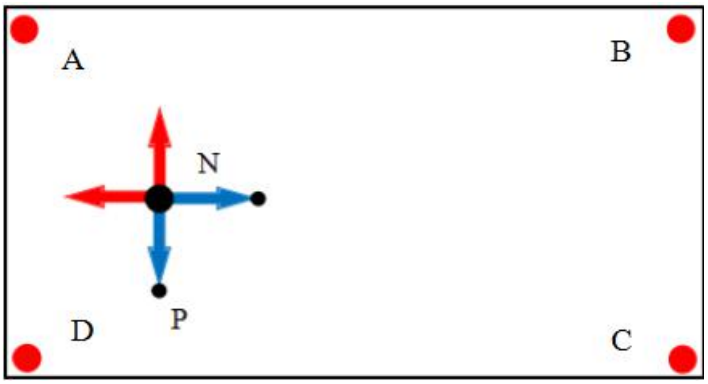
如果 N 走向 P 是沿着 N 与 N 会到达的第一个角落的曼哈顿距离变大的方向(下图中的蓝色箭头方向，也就是右下角）则显然 $h(N) < h(P)$，因而 $h(N) \leq c(N,P) + h(P)$。



✓ **不同时**

当 P 点按当前定义的启发式函数规则搜索所要找到的第一个角落与 N 不相同时，如下图所示，N 找到的第一个角落食物为 A，而 P 可能为 D（由 N 向下），那么 P 点的启发式函数值为 h(P) = Manhattan(P,D) + 2*Width + Length,而 h(N) = Manhattan(N,A) + 2*Width + Length。

运用反证法，如果 h(P) + c(N,P) < h(N)，那么 N 点按当前定义的启发式函数规则搜索所要找到的第一个角落就为 D 点，与实际相反，故 h(P) + c(N,P) ≥ h(N)。



➢ **剩余 3、2、1 个食物**

如果剩余 1 或 2 或 3 个食物，原理与剩余 4 个食物时相同，在此就不再赘述。

综上所述，可以证明所选择的启发式函数是一致的。

使用我们小组提出的启发式函数，扩展了 741 个节点完成了搜索任务，小于 1200 个，如下图所示：

```
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', 'West', 'West',
path length: 106
*** PASS: Heuristic resulted in expansion of 741 nodes

### Question q6: 3/3 ###
```

2.7 问题 7 吃掉所有的豆子

2.7.1 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。

2.7.2 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。因而在问题 7 中对走遍所有食物的总路径寻找下界是一件比较困难的事。因而，我们小组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。

为何要选择最远的食物呢？首先，因为 A*算法总是选择 f 值（$f(n) = g(n) + h(n)$，f 值即为 f(n)的值，可参见 2.4.1 节 A*算法的简要描述）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，这种启发式函数值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（估计）距离，也就是可以看做把最远的豆子作为目标点，这是合理的，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。

在确定了具有代表性的食物位置之后，还要确定使用何种距离。若使用曼哈顿距离，则测试结果如下：

```
Question q7
===========

*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** FAIL: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 9551
***     thresholds: [15000, 12000, 9000, 7000]

### Question q7: 3/4 ###
```

扩展了 9551 个节点，结果不太理想，没有达到满分。

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：

```
Question q7
===========

*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

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

扩展了 4137 个节点，效果好了许多，超额完成，并且拿到了附加分。

2.7.3 启发式函数可用性(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，还要证明其可用性和一致性。

我们小组所使用的启发式函数的可用性显然成立，而且如果证明了其一致性，则可用性必然成立，因而在此略去其可用性的证明，仅就其一致性给出简要的证明。

证明选择的启发式函数的一致性
证明：
证明一致性，即是要证明如下内容
Formally, for every node N and each successor P of N, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P. That is:

$h(N) \leq c(N, P) + h(P)$and

$h(G) = 0.$

where

h is the consistent heuristic function

N is any node in the graph
P is any descendant of N
G is any goal node
c(N,P) is the cost of reaching node P from N

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

与 2.6.4 节中证明的 1 情况思路基本一致，如果吃豆人的移动方向为靠近最远食物，则 $h(N) = c(N,P) + h(P)$，若吃豆人的移动方向为远离最远食物，则 $h(N) < h(P)$，这两种情况均满足 $h(N) \leq c(N,P) + h(P)$。

2.距离吃豆人最远的食物改变

我们可以假设最远食物未变，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 $h(N) \leq c(N,P) + g(P)$，而由于在 P 点的最远食物改变了，因而 $h(P) > g(P)$，从而 $h(N) \leq c(N,P) + h(P)$。

综上所述，我们选择的启发式函数是一致的(consistent)，同时肯定是可用的。

2.8 问题 8 次最优搜索

2.8.1 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

2.8.2 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。本函数同时还要把已经到达的食物标记为已经吃掉了的状态，具体为在 food 对应的 Grid 对象中将相应的位置置为 False。findPathToClosestDot 函数的代码如下：

```
def findPathToClosestDot(self, gameState):
```

```
    position = gameState.getPacmanPosition()   # 当前吃豆人的位置坐标
    food = gameState.getFood()
    x, y = position
    IF food[x][y]:
        gameState.getFood()[x][y] ⟵ False   # (x,y)处食物被吃掉
    IF gameState.getFood().count() == 0:
        return []   #已被吃完，无路可走，结束 return
    ELSE:
        return search.bfs(AnyFoodSearchProblem(gameState))
```

其实，findPathToClosestDot 函数还有另外一种解法，就是不使用 AnyFoodSearchProblem，而是用已知的 PositionSearchProblem，其具体实现如下：

```
def findPathToClosestDot(self, gameState):
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()

    mindis ⟵ 10000
    xindex ⟵ 0
    yindex ⟵ 0
    For x IN RANGE(food.width):
        For y IN RANGE(food.height):
            IF food.data[x][y] == True:
                dis ⟵ manhattan(startPosition,(x,y))
                if dis < mindis:
                    mindis = dis
                    xindex = x
                    yindex = y

    assert not walls[xindex][yindex], 'point [' + xindex + '][' + yindex + '] is a wall.'
    prob = PositionSearchProblem(gameState, start=startPosition, goal=(xindex,
                                                    yindex), warn=False, visualize=False)
    ans ⟵ search.bfs(prob)
    return ans
```

第三章 实验结果

本章给出运行 autograder.py 后对于每个问题的评分结果。

问题 1

```
Question q1
===========

*** PASS: test_cases\q1\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:       ['2:A->D', '0:D->G']
***     expanded_states:    ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:       ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:    ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 130
***     nodes expanded:     146

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

问题 2

```
Question q2
===========

*** PASS: test_cases\q2\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269

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

问题 3

```
Question q3
===========

*** PASS: test_cases\q3\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_bfs_vs_dfs.test
***     solution:       ['1:A->G']
***     expanded_states:    ['A', 'B']
*** PASS: test_cases\q3\graph_infinite.test
***     solution:       ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:       ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:    ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q3\ucs_0_graph.test
***     solution:       ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\ucs_1_problemC.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     269
*** PASS: test_cases\q3\ucs_2_problemE.test
***     pacman layout:      mediumMaze
***     solution length: 74
***     nodes expanded:     260
*** PASS: test_cases\q3\ucs_3_problemW.test
***     pacman layout:      mediumMaze
***     solution length: 152
***     nodes expanded:     173
*** PASS: test_cases\q3\ucs_4_testSearch.test
***     pacman layout:      testSearch
***     solution length: 7
***     nodes expanded:     14
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:       ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']

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

问题 4

```
Question q4
===========

*** PASS: test_cases\q4\astar_0.test
***     solution:       ['Right', 'Down', 'Down']
***     expanded_states:    ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q4\astar_1_graph_heuristic.test
***     solution:       ['0', '0', '2']
***     expanded_states:    ['S', 'A', 'D', 'C']
*** PASS: test_cases\q4\astar_2_manhattan.test
***     pacman layout:      mediumMaze
***     solution length: 68
***     nodes expanded:     221
*** PASS: test_cases\q4\astar_3_goalAtDequeue.test
***     solution:       ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C']
*** PASS: test_cases\q4\graph_backtrack.test
***     solution:       ['1:A->C', '0:C->G']
***     expanded_states:    ['A', 'B', 'C', 'D']
*** PASS: test_cases\q4\graph_manypaths.test
***     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 ###
```

问题 5

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

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

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

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', 'West', 'West', 'West', 'North', 'North'
path length: 106
*** PASS: Heuristic resulted in expansion of 741 nodes

### Question q6: 3/3 ###
```

问题 7

```
Question q7
===========

*** PASS: test_cases\q7\food_heuristic_1.test
*** PASS: test_cases\q7\food_heuristic_10.test
*** PASS: test_cases\q7\food_heuristic_11.test
*** PASS: test_cases\q7\food_heuristic_12.test
*** PASS: test_cases\q7\food_heuristic_13.test
*** PASS: test_cases\q7\food_heuristic_14.test
*** PASS: test_cases\q7\food_heuristic_15.test
*** PASS: test_cases\q7\food_heuristic_16.test
*** PASS: test_cases\q7\food_heuristic_17.test
*** PASS: test_cases\q7\food_heuristic_2.test
*** PASS: test_cases\q7\food_heuristic_3.test
*** PASS: test_cases\q7\food_heuristic_4.test
*** PASS: test_cases\q7\food_heuristic_5.test
*** PASS: test_cases\q7\food_heuristic_6.test
*** PASS: test_cases\q7\food_heuristic_7.test
*** PASS: test_cases\q7\food_heuristic_8.test
*** PASS: test_cases\q7\food_heuristic_9.test
*** PASS: test_cases\q7\food_heuristic_grade_tricky.test
***     expanded nodes: 4137
***     thresholds: [15000, 12000, 9000, 7000]

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

问题 8

问题 8,虽然没做要求，但是我们小组出于兴趣，还是完成了该问题。

```
Question q8
===========

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_1.test
***     pacman layout:      Test 1
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_10.test
***     pacman layout:      Test 10
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_11.test
***     pacman layout:      Test 11
***     solution length:        2
```

```
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_12.test
***     pacman layout:      Test 12
***     solution length:        3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_13.test
***     pacman layout:      Test 13
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_2.test
***     pacman layout:      Test 2
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_3.test
***     pacman layout:      Test 3
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_4.test
***     pacman layout:      Test 4
***     solution length:        3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_5.test
***     pacman layout:      Test 5
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_6.test
***     pacman layout:      Test 6
***     solution length:        2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_7.test
***     pacman layout:      Test 7
***     solution length:        1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q8\closest_dot_8.test
***     pacman layout:      Test 8
***     solution length:        1
[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 ###
```

总成绩

```
Finished at 0:50:49

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

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

第四章 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以及可用性、一致性的证明。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。老师课件中的通用搜索算法描述如下：

(1) 把初始节点 S_0 放入 Open 表，并建立目前仅包 S_0 的图 G，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) 扩展节点 n,生成一组子节点，把这些子节点中不是其父节点的那部分子节点计入集合 M，并把这些子节点作为节点 n 的子节点加 G 中
(6) 针对 M 中子节点的不同情况，分别作如下处理：
    1) 对那些没有在 G 中出现过的 M 成员设置一个指向其父节点（即节点 n）的

指针，并将他它放入 Open 表

2) 对那些原来已经在 G 中出现过，但没有被扩展过的 M 成员，确定是否需要修改它指向父节点的指针
3) 对那些原来已经在 G 中出现过，并已经被扩展过的 M 成员，确定是否需要修改其后继节点指向父节点的指针（注意：在这种情况中也要考虑 2）中的情况，及考虑修改它指向父节点的指针）

(7) 按某种策略对 Open 表中的节点进行排序
(8) 转第（2）步

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，而老师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。老师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。

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

附 录

Ⅰ. MpickBanana.py

```
class State:
    def __init__(self, monkey=-1, box=0,banana=1, monbox=-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.monbox = monbox    # -1: monkey not on the box    1: monkey on the box

def copyState(source):
    state = State()
    state.monkey = source.monkey
    state.box = source.box
    state.banana = source.banana
    state.monbox = source.monbox
    return state
'''
function monkeygoto,it makes the monkey goto the other place
'''
def monkeygoto(b,i):
    a=b

    if (a==-1):
        routesave.insert(i,"Monkey go to A")
        state.monkey = States[i].monkey
        States[i+1]=copyState(States[i])
        States[i+1].monkey=-1
    elif(a==0):
        routesave.insert(i,"Monkey go to B")
        States[i+1]=copyState(States[i])
        States[i+1].monkey=0
    elif(a==1):
        routesave.insert(i,"Monkey go to C")
        States[i+1]=copyState(States[i])
        States[i+1].monkey=1
    else:
        print("parameter is wrong")
'''
end function monkeyygoto
'''

'''
function movebox,the monkey move the box to the other place
'''
```

```
def movebox(a,i):
    B=a
    if(B==-1):
        routesave.insert(i,"Monkey move box to A")
        States[i+1]=copyState(States[i])
        States[i+1].monkey=-1
        States[i+1].box=-1
    elif(B==0):
        routesave.insert(i,"Monkey move box to B")
        States[i+1]=copyState(States[i])
        States[i+1].monkey=0
        States[i+1].box=0
    elif(B==1):
        routesave.insert(i,"Monkey move box to C")
        States[i+1]=copyState(States[i])
        States[i+1].monkey=1
        States[i+1].box=1
    else:
        print("parameter is wrong")
'''
end function movebox
'''

'''
function climbonto,the monkey climb onto the box
'''
def climbonto(i):
    routesave.insert(i,"Monkey climb onto the box")
    States[i+1]=copyState(States[i])
    States[i+1].monbox=1
'''
end function climbonto
'''

'''
function climbdown,monkey climb down from the box
'''
def climbdown(i):
    routesave.insert(i,"Monkey climb down from the box")
    States[i+1]=copyState(States[i])
    States[i+1].monbox=-1
'''
end function climbdown
'''

'''
function reach,if the monkey,box,and banana are at the same place,the monkey
```

```
reach banana
'''
def reach(i):
    routesave.insert(i,"Monkey reach the banana")
'''
end function reach
'''

'''
output the solution to the problem
'''
def showSolution(i):
    print ("Result to problem:")
    for c in range(i+1):
        print("Step %d : %s \n"%(c+1,routesave[c]))
    print("\n")
'''
end function showSolution
'''

'''
perform next step
'''
def nextStep(i):
    #print States[i].box
    if(i>=150):
        print("%s    \n", "steplength reached 150,have problem ")

    if(States[i].monbox==1 and States[i].monkey==States[i].banana and
States[i].banana==States[i].banana and States[i].box==States[i].banana):
        showSolution(i)
        exit(0)
    j=i+1

    if(States[i].box==States[i].banana):
        if(States[i].monkey==States[i].banana):
            if(States[i].monbox==-1):
                climbonto(i)
                reach(i+1)
                nextStep(j)
            else:
                reach(i+1)
                nextStep(j)       #climbdown(i)    nextStep(j)
        else:
            monkeygoto(States[i].box,i)
            nextStep(j)
```

```
        else:
            if(States[i].monkey==States[i].box):
                if(States[i].monbox==-1):
                    movebox(States[i].banana, i)
                    nextStep(j)
                else:
                    climbdown(i)
                    nextStep(j)
            else:
                monkeygoto(States[i].box, i)
                nextStep(j)


if __name__ == "__main__":
    s = raw_input("please input state: monkey, box, banana, ifMonkeyIsOnBox:
\n")
    states = s.split(" ")
    state = State(int(states[0]), int(states[1]), int(states[2]), int(states[3]))
    States = [None]*150
    routesave = [None]*150
    States.insert(0,state)
    nextStep(0)
```

II. search.py

```
class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't implement
    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.
    """

    def getStartState(self):
        """
        Returns the start state for the search problem.
        """
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
          state: Search state

        Returns True if and only if the state is a valid goal state.
        """
        util.raiseNotDefined()
```

```
    def getSuccessors(self, state):
        """
           state: Search state

        For a given state, this should return a list of triples, (successor,
        action, stepCost), where 'successor' is a successor to the current
        state, 'action' is the action required to get there, and 'stepCost' is
        the incremental cost of expanding to that successor.
        """
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        """
         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.
        The sequence must be composed of legal moves.
        """
        util.raiseNotDefined()


def tinyMazeSearch(problem):
    """
    Returns a sequence of moves that solves tinyMaze.    For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return    [s, s, w, s, w, w, s, w]


def nullHeuristic(state, problem=None):
    """
    A heuristic function estimates the cost from the current state to the nearest
    goal in the provided SearchProblem.    This heuristic is trivial.
    """
    return 0


class GenericSearch:
    """
    通用的搜索算法类,通过配置不同的参数可以实现 DFS,BFS,UCS,A*搜索
    """

    def __init__(self,problem, data_struct_type, usePriorityQueue=False,
```

```
                                                    heuristic=nullHeuristic):
        """
        通过配置不同的参数(搜索算法要解决的问题对象,采用的数据结构,是否使用优先队列[即,是否每条路径具有不同的代价],启发式函数)
        可实现 DFS(深度搜索),BFS(广度搜索),UCS(代价一致),A*算法(代价一致+启发式函数)搜索
        param1: problem (搜索算法要解决的问题对象)
        param2: data_struct_type (搜索算法中的 open 表采用的数据结构)
        param3: usePriorityQueue (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)
        """
        self.problem = problem
        self.data_struct_type = data_struct_type
        self.usePriorityQueue = usePriorityQueue
        self.heuristic = heuristic

    def genericSearch(self):
        """
        节点(node):
        对于 DFS 和 BFS,open 表中的每个节点(node)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(node)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*)

        return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
        """
        openList = self.data_struct_type()    # data_struct_type 为 util 中的一个数据结构(Stack、Queue、PriorityQueue)类
                                              # 将其实例化,创建 open 表
        if self.usePriorityQueue:    # 判断是否使用优先队列,如果是,则 push 二元组((state, actions),cost)
            openList.push((self.problem.getStartState(), []),
                                                self.heuristic(self.problem.getStartState(),
                                                               self.problem))
            """
            需要说明的是:在进行 UCS 和 A*搜索时,usePriorityQueue 同样为 True,而 UCS 没有启发式函数,而当进行 UCS 搜索时同样会进入该 if
```

```
        判断,那这是为什么呢？因为 heuristic 默认为函数 nullHeuristic(返
        回值为 0),当进行 UCS 搜索时,不传相应的启发式函数参数即可,
        那就会默认调用函数 nullHeuristic,该函数返回值为 0,相当于没有。
        """
    else:
        openList.push((self.problem.getStartState(), []))

    closed = []   # 初始化 closed 表为空表
    while True:
        if openList.isEmpty():   # 如果 fringe 表为空，则搜索问题无解
            return []
        coord, actions = openList.pop()     # 从 open 表中 pop 出一个节点
        """
        注意:当进行 UCS 和 A*搜索时,虽然我们 push 进入的为((state,
         actions),cost)结构 但当 pop 时,只 pop(state, actions),具体可参照
         util.py 文件中的类 PriorityQueue 中 pop 函数的实现方式
        """
        if self.problem.isGoalState(coord):   # 如果当前结点是目标状态,则搜
        索成功,返回操作序列 actions
            return actions
        if coord not in closed:   # 若当前结点不在 closed 表中
            closed.append(coord)   # 将当前结点加入 closed 表,coord=(x,y)
            for successor_coord, action, step_cost in \
                    self.problem.getSuccessors(coord):
                """
                遍历当前结点的后继节点
                for example, problem.getSuccessors(coord) = [((5, 4),
                                                    'South', 1), ((4, 5), 'West', 1)]
                """
                if self.usePriorityQueue:
                    """
                    若使用优先队列，则在 push 时要给出优先级信息,为从
                     起始节点通过 actions 到达当前结点的代价加
                    以当前结点为参数的启发式函数的值,即((state,
                                                        actions),cost)
                    """
                    openList.push((successor_coord, actions + [action]),
                                  self.problem.getCostOfActions(actions +
                                                                [action]) +
                                  self.heuristic(successor_coord,
                                                 self.problem))
                else:
                    openList.push((successor_coord, actions + [action]))
```

```
def depthFirstSearch(problem):
    """
    深度优先搜索算法
    实例化类 GenericSearch 为 dfs,使用 util.py 中定义的
    数据结构栈 Stack 来作为 open 表的数据结构
    param: problem  搜索算法要解决的问题对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    dfs = GenericSearch(problem, util.Stack)
    return dfs.genericSearch()


def breadthFirstSearch(problem):
    """
    广度优先搜索算法
    实例化类 GenericSearch 为 bfs,使用 util.py 中定义的
    数据结构队列 Queue 来作为 open 表的数据结构
    param: problem  搜索算法要解决的问题对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    bfs = GenericSearch(problem, util.Queue)
    return bfs.genericSearch()


def uniformCostSearch(problem):
    """
    代价一致搜索算法
    实例化类 GenericSearch 为 ucs,使用 util.py 中定义的
    数据结构优先队列 PriorityQueue 来作为 open 表的数据结构
    param: problem  搜索算法要解决的问题对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    ucs = GenericSearch(problem, util.PriorityQueue, True)
    return ucs.genericSearch()


def aStarSearch(problem, heuristic=nullHeuristic):
    """
    A*算法
    实例化类 GenericSearch 为 astar,使用 util.py 中定义的
    数据结构优先队列 PriorityQueue 来作为 open 表的数据结构
    同时指定启发式函数 heuristic,使用该启发式函数来估算当前结点到目标结点
    的代价
    param1: problem  搜索算法要解决的问题对象
```

```
    param2: heuristic  启发式函数对象
    return 值: actions 列表,即,吃豆人吃到豆子所执行的一个 action 序列
    """
    astar = GenericSearch(problem, util.PriorityQueue, True, heuristic)
    return astar.genericSearch()


# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
ucs = uniformCostSearch
```

III. searchAgent.py

```
class GoWestAgent(Agent):
    "An agent that goes West until it can't."
    def getAction(self, state):
        "The agent receives a GameState (defined in pacman.py)."
        if Directions.WEST in state.getLegalPacmanActions():
            return Directions.WEST
        else:
            return Directions.STOP

#######################################################
# This portion is written for you, but will only work #
#       after you fill in parts of search.py          #
#######################################################

class SearchAgent(Agent):
    """
    This very general search agent finds a path using a supplied search
    algorithm for a supplied search problem, then returns actions to follow that
    path.

    As a default, this agent runs DFS on a PositionSearchProblem to find
    location (1,1)

    Options for fn include:
      depthFirstSearch or dfs
      breadthFirstSearch or bfs


    Note: You should NOT change any code in SearchAgent
    """
```

```
def __init__(self, fn='depthFirstSearch', prob='PositionSearchProblem',
                                                    heuristic='nullHeuristic'):
    # Warning: some advanced Python magic is employed below to find the right
    functions and problems

    # Get the search function from the name and heuristic
    if fn not in dir(search):
        raise AttributeError, fn + ' is not a search function in search.py.'
    func = getattr(search, fn)
    if 'heuristic' not in func.func_code.co_varnames:
        print('[SearchAgent] using function ' + fn)
        self.searchFunction = func
    else:
        if heuristic in globals().keys():
            heur = globals()[heuristic]
        elif heuristic in dir(search):
            heur = getattr(search, heuristic)
        else:
            raise AttributeError, heuristic + ' is not a function in
                                                searchAgents.py or search.py.'
        print('[SearchAgent] using function %s and heuristic %s' % (fn,
                                                                heuristic))
        # Note: this bit of Python trickery combines the search algorithm and the
         heuristic
        self.searchFunction = lambda x: func(x, heuristic=heur)

    # Get the search problem type from the name
    if prob not in globals().keys() or not prob.endswith('Problem'):
        raise AttributeError, prob + ' is not a search problem type in
                                                        SearchAgents.py.'
    self.searchType = globals()[prob]
    print('[SearchAgent] using problem type ' + prob)

def registerInitialState(self, state):
    """
    This is the first time that the agent sees the layout of the game
    board. Here, we choose a path to the goal. In this phase, the agent
    should compute the path to the goal and store it in a local variable.
    All of the work is done in this method!

    state: a GameState object (pacman.py)
    """
    if self.searchFunction == None: raise Exception, "No search function
                                                    provided for SearchAgent"
    starttime = time.time()
    problem = self.searchType(state) # Makes a new search problem
```

```
        self.actions    = self.searchFunction(problem) # Find a path
        totalCost = problem.getCostOfActions(self.actions)
        print('Path found with total cost of %d in %.1f seconds' % (totalCost,
                                                    time.time() - starttime))
        if '_expanded' in dir(problem): print('Search nodes expanded: %d' %
                                                    problem._expanded)

    def getAction(self, state):
        """
        Returns the next action in the path chosen earlier (in
        registerInitialState).    Return Directions.STOP if there is no further
        action to take.

        state: a GameState object (pacman.py)
        """
        if 'actionIndex' not in dir(self):
            self.actionIndex = 0
        i = self.actionIndex
        self.actionIndex += 1
        if i < len(self.actions):
            return self.actions[i]
        else:
            return Directions.STOP

class PositionSearchProblem(search.SearchProblem):
    """
    A search problem defines the state space, start state, goal test, successor
    function and cost function.    This search problem can be used to find paths
    to a particular point on the pacman board.

    The state space consists of (x,y) positions in a pacman game.

    Note: this search problem is fully specified; you should NOT change it.
    """

    def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None,
                                                    warn=True, visualize=True):
        """
        Stores the start and goal.

        gameState: A GameState object (pacman.py)
        costFn: A function from a search state (tuple) to a non-negative number
        goal: A position in the gameState
        """
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        if start != None:
```

```
            self.startState = start
        self.goal = goal
        self.costFn = costFn
        self.visualize = visualize
        if warn and (gameState.getNumFood() != 1 or not
                                                    gameState.hasFood(*goal)):
            print 'Warning: this does not look like a regular search maze'

        # For display purposes
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = state == self.goal

        # For display purposes only
        if isGoal and self.visualize:
            self._visitedlist.append(state)
            import __main__
            if '_display' in dir(__main__):
                if 'drawExpandedCells' in dir(__main__._display):
                    #@UndefinedVariable
                    __main__._display.drawExpandedCells(self._visitedlist)
                    #@UndefinedVariable

        return isGoal

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
             For a given state, this should return a list of triples,
         (successor, action, stepCost), where 'successor' is a
         successor to the current state, 'action' is the action
         required to get there, and 'stepCost' is the incremental
         cost of expanding to that successor
        """

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
                                                            Directions.WEST]:
            x,y = state
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
```

```
                if not self.walls[nextx][nexty]:
                    nextState = (nextx, nexty)
                    cost = self.costFn(nextState)
                    successors.append( ( nextState, action, cost) )

            # Bookkeeping for display purposes
            self._expanded += 1 # DO NOT CHANGE
            if state not in self._visited:
                self._visited[state] = True
                self._visitedlist.append(state)

            return successors

        def getCostOfActions(self, actions):
            """
            Returns the cost of a particular sequence of actions. If those actions
            include an illegal move, return 999999.
            """
            if actions == None: return 999999
            x,y= self.getStartState()
            cost = 0
            for action in actions:
                # Check figure out the next state and see whether its' legal
                dx, dy = Actions.directionToVector(action)
                x, y = int(x + dx), int(y + dy)
                if self.walls[x][y]: return 999999
                cost += self.costFn((x,y))
            return cost

    class StayEastSearchAgent(SearchAgent):
        """
        An agent for position search with a cost function that penalizes being in
        positions on the West side of the board.

        The cost function for stepping into a position (x,y) is 1/2^x.
        """
        def __init__(self):
            self.searchFunction = search.uniformCostSearch
            costFn = lambda pos: .5 ** pos[0]
            self.searchType = lambda state: PositionSearchProblem(state, costFn, (1, 1),
                                                                  None, False)

    class StayWestSearchAgent(SearchAgent):
        """
        An agent for position search with a cost function that penalizes being in
        positions on the East side of the board.
```

```
    The cost function for stepping into a position (x,y) is 2^x.
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: 2 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn)

def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

#####################################################
# This portion is incomplete.  Time to write code!  #
#####################################################

class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print 'Warning: no food in corner ' + str(corner)
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        self.top = top    # 迷宫的 top，上边界坐标
        self.right = right    # 迷宫的 right，右边界坐标
```

```
        """
        状态的表示方法
        每个状态为一个形如(coord, foods_statebool)的二元组,其中 coord 为吃豆人所在的位置坐标
        foods_statebool 为食物的状态,foods_statebool 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 foods_statebool 中对应项为 False,若已经被吃
        则 foods_statebool 中对应项为 True
        """
        # 初始状态
        self.start_state = (startingGameState.getPacmanPosition(),
                                    [False, False, False, False])

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        return self.start_state

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        # 根据状态表示方法,state[1]即为 foods_statebool,若其中每一项均为 True,
        # 则说明角落的食物均已被吃掉,目标达成
        return state[1][0] and state[1][1] and state[1][2] and state[1][3]

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples, (successor,
            action, stepCost), where 'successor' is a successor to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """
        successors = []
        x, y = state[0]   # 获得坐标
        foodBool = state[1]   # 获得剩余角落食物状态
        foodXYList = []
        num = 0
```

```
        for food in foodBool:
            if not food:
                foodXYList.append(self.corners[num])
            num += 1
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
                                                            Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a
              wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]
            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 = self.walls[nextx][nexty]   # hitsWall 即为 walls 网格中(nextx,
                                                        nexty)处的布尔值
            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)
                leftFoodBool = foodBool[:]   # 需要进行浅拷贝,否则会出错
                if nextState in foodXYList:   # 判断(nextx, nexty)是否在剩余的
                                                食物坐标列表当中
                    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

    def getCostOfActions(self, actions):
        """
        Returns the cost of a particular sequence of actions.  If those actions
        include an illegal move, return 999999.  This is implemented for you.
        """
        if actions == None: return 999999
        x,y= self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)
```

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
                 (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    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).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
    coord, food_state = state   # coord 即为吃豆人的当前位置坐标
    food_lst = []
    count = 0
    for fstate in food_state:
        if not fstate:
            food_lst.append(corners[count])   # 获得剩余的角落坐标列表
        count += 1
    top = problem.top
    right = problem.right
    # 下面 3 行代码对迷宫的长和宽进行比较,找出较长边和较短边
    width_length = [top, right]
    width = min(width_length)
    length = max(width_length)

    food_num = len(food_lst)   # 剩余角落食物数量
    if food_num == 0:   # 剩余 0 个,说明达成目标,返回 0
        return 0
    elif food_num == 1:
      # 剩余 1 个,返回当前位置 coord 到剩余的这个角落的曼哈顿距离
        return util.manhattanDistance(coord, food_lst[0])
      # 剩余 2 个,返回距离当前位置较近的角落到 coord 的曼哈顿距离与剩余的
        两个角落之间的曼哈顿距离的和
    elif food_num == 2:
        dist = []
        dist.append(util.manhattanDistance(coord, food_lst[0]))
        dist.append(util.manhattanDistance(coord, food_lst[1]))
        mindist = min(dist)
        return mindist + util.manhattanDistance(food_lst[0], food_lst[1])
    elif food_num == 3:   # 剩余 3 个的处理方法有些麻烦,请见报告说明
        leftsidefood = []
```

```
            food_lstset = set(food_lst)
            if food_lstset == {(1, 1), (1, top), (right, 1)}or food_lstset == {(right, top), (1,
                                                                    top), (right, 1)}:
                leftsidefood = [(1, top), (right, 1)]
            elif food_lstset == {(1, 1), (1, top), (right, top)} or food_lstset == {(1, 1),
                                                                    (right, top), (right, 1)}:
                leftsidefood = [(1, 1), (right, top)]
                # 得到非中间位置的角落坐标列表
            man_dist_lst = min(util.manhattanDistance(coord, leftsidefood[0]),
                               util.manhattanDistance(coord, leftsidefood[1]))
            return man_dist_lst + top - 1 + right - 1
        elif food_num == 4:
            man_dist_lst = min(util.manhattanDistance(coord,
food_lst[0]),util.manhattanDistance(coord, food_lst[1]),
util.manhattanDistance(coord, food_lst[2]),util.manhattanDistance(coord, food_lst[3]))

            return man_dist_lst + 2 * (width-1) + length - 1
        #return 0 # Default to trivial solution


class AStarCornersAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"

    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob,
                                                              cornersHeuristic)
        self.searchType = CornersProblem


class FoodSearchProblem:
    """
    A search problem associated with finding the a path that collects all of the
    food (dots) in a Pacman game.

    A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
      pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
      foodGrid:       a Grid (see game.py) of either True or False, specifying
remaining food
    """

    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
                                                        startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0    # DO NOT CHANGE
        self.heuristicInfo = {}    # A dictionary for the heuristic to store information
```

```
    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def getSuccessors(self, state):
        "Returns successor states, the actions they require, and a cost of 1."
        successors = []
        self._expanded += 1    # DO NOT CHANGE
        for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
                                                              Directions.WEST]:
            x, y = state[0]
            dx, dy = Actions.directionToVector(direction)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextFood = state[1].copy()
                nextFood[nextx][nexty] = False
                successors.append((((nextx, nexty), nextFood), direction, 1))
        return successors

    def getCostOfActions(self, actions):
        """Returns the cost of a particular sequence of actions.    If those actions
        include an illegal move, return 999999"""
        x, y = self.getStartState()[0]
        cost = 0
        for action in actions:
            # figure out the next state and see whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost


class AStarFoodSearchAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"

    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)
        self.searchType = FoodSearchProblem


def foodHeuristic(state, problem):
    """
```

```
    Your heuristic for the FoodSearchProblem goes here.

    This heuristic must be consistent to ensure correctness.  First, try to come
    up with an admissible heuristic; almost all admissible heuristics will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search finds,
    your heuristic is *not* consistent, and probably not admissible!  On the
    other hand, inadmissible or inconsistent heuristics may find optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
    (see game.py) of either True or False. You can call foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can query the
    problem.  For example, problem.walls gives you a Grid of where the walls
    are.

    If you want to *store* information to be reused in other calls to the
    heuristic, there is a dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls once and store that
    value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state

    ans = 0
    for x in range(foodGrid.width):
        for y in range(foodGrid.height):
            if foodGrid.data[x][y] == True:
                #dis = abs(x - position[0]) + abs(y - position[1])
                dis = mazeDistance((x, y), position, problem.startingGameState)
                if ans < dis:
                    ans = dis
    return ans


class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"

    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while (currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState)
```

```
                                                    # The missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception, 'findPathToClosestDot returned an illegal
                                                    move: %s!\n%s' % t
                currentState = currentState.generateSuccessor(0, action)
        self.actionIndex = 0
        print 'Path found with cost %d.' % len(self.actions)

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """

        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        mindis = 10000
        xindex = 0
        yindex = 0
        for x in range(food.width):
            for y in range(food.height):
                if food.data[x][y] == True:
                    dis = abs(x - startPosition[0]) + abs(y - startPosition[1])
                    if dis < mindis:
                        mindis = dis
                        xindex = x
                        yindex = y

        # assert not walls[xindex][yindex], 'point1 is a wall: ' + str(point1)
        assert not walls[xindex][yindex], 'point [' + xindex + '][' + yindex + '] is a
                                                    wall.'
        prob = PositionSearchProblem(gameState, start=startPosition, goal=(xindex,
                                                yindex), warn=False, visualize=False)
        ans = search.bfs(prob)
        return ans

        '''
        # Here are some useful elements of the startState
        position = gameState.getPacmanPosition()   #  当前吃豆人的位置坐标
```

```
        food = gameState.getFood()
        #walls = gameState.getWalls()
        #problem = AnyFoodSearchProblem(gameState)
        x, y = position
        if food[x][y]:   # 若吃豆人当前位置有食物
            gameState.getFood()[x][y] = False   # 将 food[x][y]置为 False，表示
                                                食物被吃掉了
        if gameState.getFood().count() == 0:   # 若食物数量为 0，表示目标已达
                                                成，返回空列表
            return []
        else:
            # 使用广度优先搜索算法，确保能搜索到最优解，即最靠近当前位
            # 置的食物，返回从当前位置到最近食物位置需执行的 action 列表
            return search.bfs(AnyFoodSearchProblem(gameState))
        '''


class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.    The state space and
    successor function do not need to be changed.

    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.    You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE

    def isGoalState(self, state):
        """
        The state is Pacman's position. Fill this in with a goal test that will
```

```
        complete the problem definition.
        """
        x, y = state
        return self.food[x][y]
        '''
        if self.food.count == 1 and state == self.food[0].position or self.food.count
                                                                        == 0:
            return True;
        else:
            return False
        '''


def mazeDistance(point1, point2, gameState):
    """
    Returns the maze distance between any two points, using the search functions
    you have already built. The gameState can be any game state -- Pacman's
    position in that state is ignored.

    Example usage: mazeDistance( (2,4), (5,6), gameState)

    This might be a useful helper function for your ApproximateSearchAgent.
    """
    x1, y1 = point1
    x2, y2 = point2
    walls = gameState.getWalls()
    assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2, warn=False,
                                                                    visualize=False)
    return len(search.bfs(prob))
```