

# 吃豆人程序的设计

## 《人工智能导论》实验报告

专 业 名 称： 计算机科学与技术

学 生 姓 名： 马玉坤

学 生 学 号： 1150310618

指 导 教 师： 李钦策

同 组 人 员： 李伟枫, 许浩禹, 张宁

二〇一八年一月

## 摘 要

吃豆人项目是加州大学伯克利分校为了入门级人工智能课程 (CS188) 开发的。在本实验中，我们需要通过修改吃豆人程序的部分程序 (`search.py`, `searchAgents.py`)，从而扩充吃豆人小程序的功能，对各种搜索方法进行学习和实践。

**关键词:** 人工智能导论; 搜索算法; CS188; 吃豆人

# 目 录

摘要	I
1 简介	1
2 算法介绍	3
2.1 编写通用搜索算法	3
2.1.1 待解决问题的解释	3
2.1.2 问题的形式化描述	3
2.1.3 解决方案介绍	3
2.1.4 所用方法的一般介绍	3
2.1.5 算法伪代码	4
2.2 找到所有的角落	5
2.2.1 待解决问题的解释	5
2.2.2 问题的形式化描述	5
2.2.3 解决方案介绍	5
2.2.4 所用方法的一般介绍	5
2.2.5 算法伪代码	6
2.3 角落问题（启发式）	6
2.3.1 待解决问题的解释	6
2.3.2 问题的形式化描述	6
2.3.3 解决方案介绍	7
2.3.4 所用方法的一般介绍	7
2.3.5 算法伪代码	7
2.4 吃掉所有的豆子	7
2.4.1 待解决问题的解释	7
2.4.2 问题的形式化描述	7
2.4.3 解决方案介绍	7
2.4.4 所用方法的一般介绍	8
2.4.5 算法伪代码	8

2.5	次最优搜索 . . . . .	8
2.5.1	待解决问题的解释 . . . . .	8
2.5.2	问题的形式化描述 . . . . .	8
2.5.3	解决方案介绍 . . . . .	9
2.5.4	所用方法的一般介绍 . . . . .	9
2.5.5	算法伪代码 . . . . .	9
<b>3</b>	<b>算法实现</b>	<b>10</b>
3.1	实验环境 . . . . .	10
3.2	数据结构 . . . . .	10
3.3	实验结果 . . . . .	10
3.4	系统中间及最终输出结果 . . . . .	11
<b>4</b>	<b>总结及讨论</b>	<b>12</b>
	<b>参考文献</b>	<b>13</b>
	<b>附录 A 实验代码</b>	<b>14</b>
A.1	通用搜索算法 . . . . .	14
A.2	角落问题 . . . . .	16
A.3	角落问题（启发式） . . . . .	18
A.4	吃掉所有的豆子 . . . . .	19
A.5	次最优搜索 . . . . .	20

# 1 简介

吃豆人项目 [2] 是加州大学伯克利分校为入门级人工智能课程开发的项目。通过一系列的 AI 技术来玩吃豆人。但是吃豆人项目不专注于为视频游戏构建 AI，恰恰相反这些项目教授使用者 AI 基础概念，如知情的状态空间搜索，概率推理和强化学习。这些概念是真实世界的应用领域，如自然语言处理，计算机视觉和机器人。

这些项目允许学生可视化实现的技术的结果。它们还包含代码示例和明确的指示，但不要强迫学生趟过过多的脚手架。最后，吃豆人提供了一个具有挑战性的问题环境，需要创造性的解决方案；真实世界的 AI 问题是具有挑战性的，吃豆人也是。

吃豆人系列项目一共分为五个小项目：Search, Multiagent Search, Reinforcement Learning, Ghostbusters 以及 Classification。其中的项目一——Search 就是我们本次的实验。在这个实验中，吃豆人将通过一系列构建的一系列的搜索算法实现达到特定位置并有效地吃到食物。包括深度优先搜索、广度优先搜索、一致代价搜索、A\* 搜索、启发式搜索等 8 个问题。

这八个问题分别是：

1. 应用深度优先算法找到一个特定的位置的豆
2. 宽度优先算法
3. 代价一致算法
4. A\* 算法
5. 找到所有的角落
6. 角落问题（启发式）
7. 吃掉所有的豆子
8. 次最优搜索

在这个实验的实现过程中需要编辑两个关键文件“search.py”和“searchAgents.py”，“search.py”中需要补充编写四个基本搜索算法，“searchAgents.py”中需要补充编写所有启发式算法的基本内容。除此之外还有三个比较重要的文件：“pacman.py”、“game.py”和“util.py”。“pacman.py”是游戏的主文件，描述了一个 Pacman GameState 类型。“game.py”说明了 Pacman 世界如何工作的逻辑并且描述了几种支持类型，如 AgentState，Agent，Direction 和 Grid。“util.py”是对我们来说非常重要的文件，包括了实现搜索算法可以用到的几种数据结构。

通过这个实验，我们可以更好的理解各种基本搜索算法，并且通过可视化的

方法可以更加有成就感，激发学习兴趣。在人工智能的领域继续研究下去。

## 2 算法介绍

### 2.1 编写通用搜索算法

#### 2.1.1 待解决问题的解释

编写一个通用的搜索算法，来解决“应用深度优先算法找到一个特定的位置的豆”，“应用广度优先搜索算法找到一个特定的位置的豆”，“应用代价一致算法找到一个特定的位置的豆”，“应用 A\* 算法找到一个特定的位置的豆”等四个问题。

#### 2.1.2 问题的形式化描述

**输入：**problem(吃豆人游戏搜索问题),open(遍历所用的数据结构),heuristic(启发式算法)

**输出：**path[](由方向组成的数组，可以指示吃豆人从起始状态走到终止状态)

#### 2.1.3 解决方案介绍

四种算法既有相似性也有相异性。相似性体现在四种算法的框架类似，相异性主要体现在两个方面。

第一个方面便是启发式函数的问题。在四个问题中，最复杂的问题是 A\* 搜索问题，因为 A\* 搜索问题需要启发式函数。换个角度来看，其他三种算法（深度优先搜索，广度优先搜索以及代价一致搜索）可以看做是启发式函数恒为 0。

第二个方面数据结构的不同。

因此四种问题只需分别定义好数据结构以及启发函数，调用通用搜索算法便可以获得解决方案。

#### 2.1.4 所用方法的一般介绍

我们定义了一个函数 unifiedSearch（意为通用搜索），unifiedSearch 需要三个参数，首先是搜索问题，第二个是遍历所用的数据结构对象，最后一个是用到的启发式（估价）函数。

函数执行的第一步便是初始化过程，将初始状态的  $f, g, h(f[state] = g[state] + h[state])$ ，其中  $h$  为启发式函数) 三个函数定义好，然后将初始状态 push 到数据结

构对象中。

接下来便是遍历过程，每次从数据结构对象中 **pop** 元素，设为当前的状态，然后判断当前状态是否为结束状态，如果是结束状态就返回找到的路径，否则就扩展新的状态，放到数据结构对象中。每次扩展子状态都需要使用启发式函数计算状态的  $f[childState]$  函数值。

### 2.1.5 算法伪代码

---

#### 算法 1 通用搜索算法

---

**输入:** *problem* 搜索问题, *open* 遍历所需的数据结构对象, *heuristic* 启发式函数

**输出:** 指示从起点到终点的路径的方向列表

```
1: initialState  $\leftarrow$  problem.getStartstate()
2: 初始化 initialState 的 f, g, h 函数
3: open.push(initialState)
4: while open 非空 do
5:   state  $\leftarrow$  open.pop()
6:   将 state 加入 close 表
7:   if state 为结束状态 then
8:     return getPath(state, initialState)
9:   end if
10:  for childState in state 的子状态 do
11:    更新 childState 的信息
12:    将 childState 加入 open
13:  end for
14: end while
15: function getPath(dest, src)
16:   state  $\leftarrow$  dest
17:   path  $\leftarrow$  []
18:   while state  $\neq$  src do
19:     path  $\leftarrow$  path + state 的转移方向
20:     state  $\leftarrow$  父状态
21:   end while
22:   return path
23: end function
```

---



## 2.2 找到所有的角落

### 2.2.1 待解决问题的解释

需要重新定义问题状态，使状态能够表示四个角落的访问状态，使得该问题使用之前的搜索算法解决时，能够遍历所有的角落（左上、左下、右上、右下四个角落）。

### 2.2.2 问题的形式化描述

修改 `searchAgents.py` 文件中的 `CornersProblem` 类。编写 `__init__()`、`getStartState()`、`isGoalState()` 以及 `getSuccessors()` 方法，使得 `CornersProblem` 能够成为一个自治的问题，且当问题解决时，四个角落都会被访问到。

### 2.2.3 解决方案介绍

其实，定义好这样一个问题，只需要考虑清楚如何将四个角落的访问状态加入到状态中。因此我们在状态设计中新增维数，添加表示角落访问状态的维度，从而来表示状态。

### 2.2.4 所用方法的一般介绍

在之前的问题中，状态的表示为一个二元组  $(x, y)$ 。在该问题中，我们设计了一个新的维度，用来存储四个角落的访问状态。新的状态形式为  $((x, y), corners)$ ，是一个新的二元组，该二元组的第一维是一个坐标二元组，第二维是一个 4 位二进制数 *corners*。

当 *corners* 为 1111 时，表示四个角落都访问过，这也是终止状态的充分必要条件。当 *corners* 为 1000 时，表示仅仅访问过左上角落。而当 *corners* 为 0000 时，表示四个状态都没有访问过。初始状态即为  $(startingPosition, 0000)$ 。

### 2.2.5 算法伪代码

---

#### 算法 2 角落问题

---

```
1: function getStartState(self)
2:   return (self.startingPosition, 0)
3: end function
4: function isGoalState(self, state)
5:   if state[1] = 15 then
6:     return True
7:   else
8:     return False
9:   end if
10: end function
11: function getSuccessors(self, state)
12:   successors  $\leftarrow$  []
13:   for action  $\in$  Directions do
14:     从 action 和当前位置获取新的位置
15:     从新位置和当前四个角落的访问情况获取新的四个角落的访问情况
16:     if 新位置可达 then
17:       successors  $\leftarrow$  successors + 新状态
18:     end if
19:   end for
20:   return successors
21: end function
```

---

## 2.3 角落问题（启发式）

### 2.3.1 待解决问题的解释

编写上一个问题（角落问题）对应的 A\* 算法的启发式函数。

### 2.3.2 问题的形式化描述

输入：当前状态 *state*，当前问题 *problem*

输出：当前状态对应的估价函数值

### 2.3.3 解决方案介绍

枚举访问顺序，通过曼哈顿距离测算最小的代价。

### 2.3.4 所用方法的一般介绍

由于没有访问的角落最多只有四个，所以我们可以枚举四个角落的访问顺序（最多  $4! = 24$  种），找到代价最小的访问顺序，代价的计算是通过曼哈顿距离来测算的，这样效率比较高，拿到了满分。

### 2.3.5 算法伪代码

---

#### 算法 3 角落问题启发式函数

---

**输入：** *state* 当前状态，*problem* 搜索问题

**输出：** 当前状态的启发式函数值

```
1:  $minDist \leftarrow \infty$ 
2: for  $perm \in [0, 1, 2, 3]$  的四种排列 do
3:    $dist \leftarrow$  使用曼哈顿距离计算的按照排列顺序所需的最小代价
4:    $minDist \leftarrow \min(minDist, dist)$ 
5: end for
6: return  $minDist$ 
```

---

## 2.4 吃掉所有的豆子

### 2.4.1 待解决问题的解释

编写 *searchAgents.py* 中的 *foodHeuristic* 函数，能够成为使用 A\* 算法解决 *FoodSearchProblem* 的启发式函数。

### 2.4.2 问题的形式化描述

补充 *searchAgents.py* 中的 *foodHeuristic* 函数，使得其是一个 consistent 且 admissible 的启发函数。

**输入：** 当前状态 *state*，当前吃掉所有豆子的问题 *problem*

**输出：** 一个数字，表示当前状态的估价函数值

### 2.4.3 解决方案介绍

在吃掉所有豆子的问题中，状态的定义为  $((x, y), foodList)$ 。*foodList* 表示所有豆子的位置列表。当豆子列表为空时，显然估计函数值应该为 0。当豆子列表不

为空的时候，我们只需要计算所有豆子中，与当前位置曼哈顿距离最大的豆子，距离当前位置的精确距离（使用 *searchAgents.py* 中的 *mazeDistance*）就能达到非常惊人的效果，获得了 5/4 的分数。

#### 2.4.4 所用方法的一般介绍

当豆子列表为空时，显然估计函数值应该为 0。当豆子列表不为空的时候，我们只需要计算所有豆子中，与当前位置曼哈顿距离最大的豆子，距离当前位置的精确距离（使用 *searchAgents.py* 中的 *mazeDistance*）。

#### 2.4.5 算法伪代码

---

**算法 4** 吃掉所有豆子的启发式函数

---

**输入：** *state* 当前状态，*problem* 搜索问题

**输出：** 当前状态的启发式函数值

```
1: position ← 当前状态的位置
2: foodList ← 当前状态中所有没吃掉的豆子的位置列表
3: if  $|foodList| = 0$  then
4:   return 0
5: else
6:   farestFood ← foodList 中距离 position 最近的豆子的位置
7:   return farestFood 距离 position 的精确位置 (mazeDistance)
8: end if
```

---

### 2.5 次最优搜索

#### 2.5.1 待解决问题的解释

定义一个优先吃最近的豆子函数是提高搜索速度的一个好的办法。补充完成 *searchAgents.py* 文件中的 *AnyFoodSearchProblem* 目标测试函数，并完成 *searchAgents.py* 文件中的 *ClosestDotSearchAgent* 部分，在此 Agent 当中缺少一个关键的函数：找到最近豆子的函数。

#### 2.5.2 问题的形式化描述

补充 *searchAgents.py* 中的 *ClosestDotSearchAgent* 类中的 *findPathToClosestDot* 函数。

**输入：** *gameState* 游戏状态

**输出：** 一条路径，能够从当前位置走到最近的豆子的位置

### 2.5.3 解决方案介绍

使用之前定义的 *bfs* 函数，来解决这个问题即可。

### 2.5.4 所用方法的一般介绍

使用 *search.bfs(problem)* 可以调用之前写好的 *bfs* 函数，返回值即为当前位置到最近的豆子的位置的路径。

### 2.5.5 算法伪代码

---

**算法 5** 找到到最近的豆子的路径

---

**输入:** *gameState* 游戏状态

**输出:** 从当前吃豆人的位置到最近的豆子的路径

- 1: *problem*  $\leftarrow$  AnyFoodSearchProblem(*gameState*)
  - 2: *path*  $\leftarrow$  bfs(*problem*)
  - 3: **return** *path*
-

## 3 算法实现

### 3.1 实验环境

实验环境:

- CPU:E3-1230 v5
- 操作系统: Arch Linux
- 内核版本: Linux version 4.14.11-1-ARCH

### 3.2 数据结构

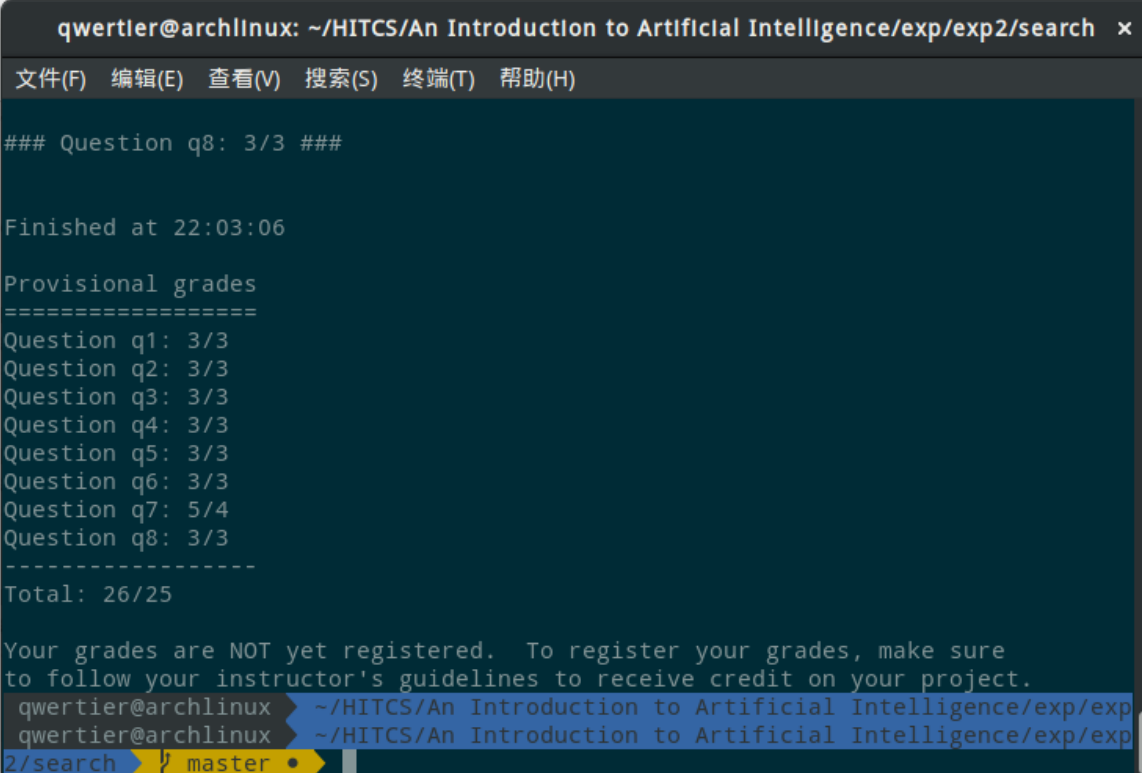
本实验使用了多种数据结构, 包括:

1. 栈
2. 队列
3. 优先队列
4. 字典 (哈希表)
5. 列表 (线性表)

### 3.3 实验结果

实验证明, 最终我们使用的方法非常出色, 得到了 26/25 的总分数 (满分 25, 在问题 7 中得到了 1 分加分)。

### 3.4 系统中间及最终输出结果



```
qwertier@archlinux: ~/HITCS/An Introduction to Artificial Intelligence/exp/exp2/search x
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

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

Finished at 22:03:06

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.
qwertier@archlinux ~/HITCS/An Introduction to Artificial Intelligence/exp/exp
qwertier@archlinux ~/HITCS/An Introduction to Artificial Intelligence/exp/exp
2/search master
```

图 3.1 结果截图

## 4 总结及讨论

在本次实验的 8 个问题中，我们明确分工，积极讨论，认真研读课上讲义，广泛查阅资料 [1]。整个实验的难度并不高，因此我们完成得较为迅速。最终成果令我们十分开心。

令人印象深刻的是在问题 7（评分标准如图 4.1）中，当我们在猜想，实验，推倒，再猜想，再实验，再推倒的过程中，一系列启发式函数要么效果不佳（只能得部分分数），要么不满足可纳性，要么不满足一致性。当我们进行了很多次尝试之后，终于找到了现在这个效果好，且满足两个性质的启发式函数。

Number of nodes expanded	Grade
more than 15000	1/4
at most 15000	2/4
at most 12000	3/4
at most 9000	4/4 (full credit; medium)
at most 7000	5/4 (optional extra credit; hard)

图 4.1 问题 7 的评分标准

完成该实验之后，经过反思，我们首先认为这个实验过程十分有趣，通过补充编写吃豆人这个游戏，来学习人工智能导论的基础知识。加州伯克利分校 [2] 的这种寓教于乐的方式很值得我们学习。其次，我们对人工智能导论的学习也产生了更浓厚的兴趣，在接下来的时间中，我们会更认真学习人工智能导论课程的知识。



## 参考文献

- [1] 人工智能原理及其应用. 高等学校教材. 电子工业出版社, 2000.
- [2] U. Berkeley. Berkeley ai materials. <http://ai.berkeley.edu/search.html>. Accessed December 25, 2017.

## 附录 A 实验代码

### A.1 通用搜索算法

```
def unifiedSearch(problem, open, heuristic):
    closed = set()
    preDirection = dict()
    prePosition = dict()
    f = dict()
    g = dict()
    h = dict()

    def getPath(dest, src):
        # 获得从起点 (src) 到终点 (dest) 的路径, 由 dest 向前遍历得到
        state = dest
        path = []
        while state != src:
            path.append(preDirection[state])
            state = prePosition[state]
        path.reverse()
        return path

    initialState = problem.getStartState()
    g[initialState] = 0
    h[initialState] = heuristic(initialState, problem)
    f[initialState] = g[initialState] + h[initialState]
    open.push((initialState, f[initialState], None, None))

    while not open.isEmpty():
        (state, _, prePositionTmp, preDirectionTmp) = open.pop()
        if state in closed:
            continue
        preDirection[state] = preDirectionTmp # 找到从父状态所转移来的方向
        prePosition[state] = prePositionTmp # 找到父状态
        closed.add(state)
        if problem.isGoalState(state): # 到达终点
            return getPath(state, problem.getStartState())
```

```

    for childState in problem.getSuccessors(state):
        if childState[0] not in g: # 子状态的 g 函数未定义
            g[childState[0]] = g[state] + childState[2]
        else:
            g[childState[0]] = min(g[childState[0]], g[state] + childState[2])
        if childState[0] not in h: # 子状态的 h 函数未定义
            h[childState[0]] = heuristic(childState[0], problem)
        if childState[0] not in f:
            f[childState[0]] = g[childState[0]] + h[childState[0]]
        else:
            f[childState[0]] = min(f[childState[0]], g[childState[0]] +
                                   h[childState[0]])
        open.push((childState[0], f[childState[0]], state, childState[1]))
    return []

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:
    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())
    """
    return unifiedSearch(problem, util.Stack(), nullHeuristic)
    return []

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    return unifiedSearch(problem, util.Queue(), nullHeuristic)

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    return unifiedSearch(problem, util.PriorityQueueWithFunction(lambda x: x[1]),
                          nullHeuristic)

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

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    return unifiedSearch(problem, util.PriorityQueueWithFunction(lambda x: x[1]),
                          heuristic)

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

```

## A.2 角落问题

```

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

```

```

def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    """ YOUR CODE HERE """
    return (self.startingPosition, 0)

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    """ YOUR CODE HERE """
    if state[1] == 15:
        return True
    else:
        return False

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]:
        # 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]

        """ YOUR CODE HERE """
        x,y = state[0]

```

```

dx, dy = Actions.directionToVector(action)
nextx, nexty = int(x + dx), int(y + dy)
hitsWall = self.walls[nextx][nexty]

# 获得新的二进制数，表示四个角落的访问状态
isCorner = 0
for i in xrange(len(self.corners)):
    corner = self.corners[i]
    if (nextx, nexty) == corner:
        isCorner |= 1 << i

if not hitsWall: # 没有撞墙
    successors.append(((nextx,nexty), state[1]|isCorner), action, 1))

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)

```

### A.3 角落问题（启发式）

```

import itertools
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)

*** YOUR CODE HERE ***
def manhattan(p1, p2):
    return abs(p1[0]-p2[0]) + abs(p1[1]-p2[1])
minDist = -1
cornerState = state[1]

# 枚举 4!=24 中排列
for perm in itertools.permutations([0,1,2,3], 4):
    dist = 0
    lastPos = state[0]

    # 计算按顺序遍历这四个角落需要的最少步数
    for i in perm:
        if ((cornerState >> i) & 1) == 0:
            dist += manhattan(corners[i], lastPos)
            lastPos = corners[i]
            #print corners, perm, cornerState, dist
    if minDist == -1:
        minDist = dist
    else:
        minDist = min(minDist, dist)
return max(minDist, 0) # Default to trivial solution

```

## A.4 吃掉所有的豆子

```

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
```

```
*** YOUR CODE HERE ***
```

```
def manhattan(p1, p2):
```

```
    return abs(p1[0]-p2[0])+abs(p1[1]-p2[1])
```

```
foodList = foodGrid.asList()
```

```
if len(foodList) == 0:
```

```
    return 0
```

```
else:
```

```
    # 找到曼哈顿距离最近的
```

```
    farest = max([(manhattan(food, position), food) for food in foodList])
```

```
    # 然后返回准确的最短距离
```

```
    return mazeDistance(position, farest[1], problem.startingGameState)
```

## A.5 次最优搜索

```
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)

    *** YOUR CODE HERE ***
    path = search.bfs(problem)
    return path

```