哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：大三上学期

课程名称：人工智能

实验名称：搜索策略

实验性质：设计

实验时间：2019/10 地点：T2109 实验台号 无

学生专业：计算机类

学生学号：SZ170110301

学生姓名：卢茉莉

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2019 年 10 月

一、 实验概述

1. 问题描述

本实验以典型的吃豆人小游戏为例，需要基于各种不同的搜索策略来设计吃豆人程序，使小人能够在一定的时间、空间代价内，在不遵守游戏规则的前提下（不会“撞墙”）沿着搜索算法得到的最优路径到达指定的位置成功吃到豆子。该游戏的评价指标为找到最优路径所需要的时间、拓展的节点数等等，以此为依据可以对算法实现额外的改进。

2. 问题的形式化描述

（1） 状态定义：

$Locate(x, y)$：吃豆人位于(x, y)坐标上

$IsGoal(x, y)$：坐标(x, y)是目标豆子所在的位置

$IsWall(x, y)$：坐标(x, y)处有墙

（2） 动作定义：

West(pacman)：向西一步

条件：$\neg \text{IsWall}(x-1, y)$， $Locate(x, y)$

结果：删除 $Locate(x, y)$，添加 $Locate(x-1, y)$

$East(pacman)$：向东一步

条件：$\neg \text{IsWall}(x+1, y)$， $Locate(x, y)$

结果：删除 $Locate(x, y)$，添加 $Locate(x+1, y)$

$North(pacman)$：向北一步

条件：$\neg \text{IsWall}(x, y+1)$， $Locate(x, y)$

结果：删除 $Locate(x, y)$，添加 $Locate(x, y+1)$

$South(pacman)$：向南一步

条件：$\neg \text{IsWall}(x, y-1)$，$Locate(x, y)$

结果：删除 $Locate(x, y)$，添加 $Locate(x, y-1)$

（3） 目标状态：

$Locate(x, y) \cap IsGoal(x, y)$

3. 实验原理

本次实验主要运用了深度优先、广度优先、代价一致以及A*四种搜索策略，并且针对不同的具体问题对于上述搜索策略进行改进以达到目标。以下为对上述四种算法的原理介绍：

（1）深度优先：

深度优先算法以搜索树为原型，每一次选择深度最大的节点进行拓展，是后生成的节点先拓展的策略。因此，深度优先算法的数据结构采用栈的形式，利用栈后进先出的特性实现深度优先搜索，它往往不能保证找到最优解，并且在最坏的情况下，其代价等同于穷举法。

（2）广度优先：

广度优先算法同样以搜索树为原型，每一次将同样深度的节点按一定顺序逐个进行拓展，是一种先生成的节点先拓展的策略。因此，广度优先算法采用队列的数据结构进行拓展节点的存储，利用队列先进先出

的特性实现广度优先搜索。广度优先算法属于完备的算法，即它一定能找到一条从开始节点到目标节点的最短路径（问题的最优解），缺点是搜索效率较低。

（3）代价一致：

代价一致算法是广度优先算法的推广，不同的是，代价一致算法基于有代价的搜索树，利用优先队列存储拓展的节点，每一次选择队列中代价最小的节点进行拓展。代价一致算法找到的是开始节点到目标节点代价最小的路径，它与广度优先算法找到的最短路径不一定相同。

（4）A*算法：

A*算法是对A算法的估价函数$f(n)=g(n)+h(n)$加上某些限制后得到的一种启发式搜索算法，经过限制后，新的估价函数为$f^*(n)=g^*(n)+h^*(n)$。其中，$g^*(n)$表示从开始节点到节点n的最小代价，$h^*(n)$表示从节点n到目标节点的最小代价，通过对二者进行估计，进而得出最优路径实际代价$f^*(n)$的估计值$f(n)$，结合贪心算法选择具有全局最小$f(n)$的节点进行拓展，直到找到目标节点。A*算法一定能结束在最佳路径上，并且通过调整$h(n)$，增加估价函数所携带的启发性信息，可以提高A*算法的搜索效率。

二、 算法介绍

（一）应用深度优先算法找到特定位置的豆

1. 方法介绍

该算法利用栈的形式来存储拓展到的节点，结合 closed 表来对已经遍历过的位置进行标记。如果栈不为空，则取出栈顶未被遍历过的节点，将其未被遍历过的子节点（含位置、方向内容）进行压栈，一直到找到目标豆子的位置。由于栈具有后进先出的特性，因此每一次都会先拓展搜索树上最深的节点，达到深度优先的目的。

2. 伪代码

（1）将初始状态取出，作为初始节点，压栈
（2）建立标记表 closed
（3）while 栈不空：
（4）{
（5）　出栈一个节点
（6）　If 到达目标节点：
（7）　　返回路径
（8）　If 该节点未被遍历：
（9）　{
（10）　　在 closed 中标记它
（11）　　将子节点中未被遍历的节点入栈
（12）　}
（13）}

（二）应用广度优先算法找到特定位置的豆

1. 方法介绍

该算法与深度优先算法类似，不同的是它利用队列的形式来存储拓展到的节点，由于队列具有先进先出的特性，因此每一次都会逐层拓展节点，达到广度优先的目的。

2. 伪代码

```
（1） 将初始状态取出，作为初始节点，入队列
（2） 建立标记表 closed
（3） While 队列不空：
（4） {
（5）  出列一个节点
（6）  If 到达目标节点：
（7）      返回路径
（8）  If 该节点未被遍历：
（9）  {
（10）     在 closed 中标记它
（11）     将子节点中未被遍历的节点入队列
（12）  }
（13） }
```

（三）代价一致算法

1. 方法介绍

该算法与上述广度优先算法相似，不同之处在于存储节点的数据结构由队列变成优先队列，并且节点中存储的内容除了位置、方向信息还有路径的长度，则当在节点入队列时会按路径长度进行排序，再次出队列时取出的便是路径最短（代价最小）的行走方案。

2. 伪代码

（1） 将初始状态取出，作为初始节点，入优先队列
（2） 建立标记表 closed
（3） While 队列不空：
（4） {
（5） 出列一个节点
（6） if 到达目标节点：
（7） 返回路径
（8） if 该节点未被遍历：
（9） {
（10） 在 closed 中标记它
（11） 将子节点中未被遍历的节点入优先队列
（12） }
（13） }

（四）A*算法

1. 方法介绍

该算法增加了一个启发函数对于节点的代价进行评估，这个代价为起点到节点 n 的距离与 n 到目标节点的曼哈顿距离之和，以此作为标准将其放入优先队列，每次选择的是估计代价最小的路径进行拓展，知道找到目标豆子所在的位置。

2. 伪代码

（1） 将初始状态取出，作为初始节点，入优先队列
（2） 建立标记表 closed
（3） while 队列不空：
（4） {
（5） 出列一个节点
（6） if 到达目标节点：
（7） 返回路径

（8） if 该节点未被遍历：
（9） {
（10） 在 closed 中标记它
（11） for 该节点中未被遍历的子节点：
（12） {
（13） 计算子节点代价=起点到节点距离+节点到目标节点的曼哈顿距离
（14） 将子节点入优先队列
（15） }
（16） }
（17） }

（五）角落问题

1. 方法介绍

角落问题的目标在于找到位于四个角落的豆子，因此在定义状态的时候需要标记的不仅只有位置坐标、路径和代价，还应该标记遍历过的角落状态。它与标记路径的方法相似，通过记录遇到过的节点，避免重复遍历同样的角落，并且可以通过计算记录中节点的个数来判断是否到达目标状态。该问题中仅仅是针对问题要求来设计新的节点结构，结合上述四种搜索算法可以得到代价更优的路径。

2. 伪代码

isGoalState(self, state):
（1）location = state[0]
（2）Corner_state = state[1]
（3）Return len(corner_state) == 4
getSuccessors(self, state):

（1） 建立后继节点表
（2） for 四个方向：
（3） {
（4） 计算下一个位置的坐标
（5） if 下一个位置处没有墙：
（6） {
（7） if 该位置是没有被遍历过的角落：
（8） 将其记录到节点信息中
（9） 将该节点加入后继节点表中
（10） }
（11） }
（12） 返回后继节点表

（六）角落问题（启发式）

1. 方法介绍

该问题是应用启发式算法 A*的角落问题，其中，每个节点的估计代价变为起始节点到该节点的距离与该节点到最近的角落曼哈顿距离之和，因此在角落问题的基础之上需要计算当前点到剩下未被遍历的角落的距离，选择最小的一个作为拓展的下一个节点。

2. 伪代码

cornersHeuristic(state, problem):

（1） 取出节点位置 node=state[0]
（2） 通过 state[1]来得到未被遍历的角落表 un_viscorner
（3） 计算当前节点到 un_viscorner 表中角落的曼哈顿距离最小值 hn=minmanhattan(un_viscorner, state[0])

Minmanhattan(corners, pos):

（1） for 每一个角落：
（2） {

（3） 计算曼哈顿距离，记录到列表 hn 中

（4） }

（5） 返回 hn 中的最小值

（七）吃掉所有的豆子

1. 方法介绍

该问题中需要吃掉地图上所有的豆子，因此需要一个列表来标记豆子是否被遍历过；另外，需要找到一条能够遍历所有的豆子的路径，在此实验中，我们通过利用 A*算法来寻找路径，并且组员韩雪婷同学提出采用值相等时的决胜法，通过对启发函数进行加权，使算法倾向于选择到终点代价更小的路径，使算法的效率有了一定的提高。

2. 伪代码

getSuccessors(self, state):

（1） 创建后继节点列表

（2） for 每个方向：

（3） {

（4） 计算下一个位置的坐标

（5） if 下一个位置处没有墙：

（6） {

（7） 将食物矩阵中该位置标记为 False，表示已遍历过

（8） 记录该节点的信息，将其加入后继节点列表中

（9） }

（10） }

foodHeuristic(state, problem):

（1） 遍历食物矩阵，记录地图上未被遍历过的食物坐标

（2） 计算任意两个食物间的最大值，将这两个点以及最大距离值计入三元组 max_distance(current_food, next_food,distance)中

（3） 计算当前 state 到上述两个点的距离 d1, d2

（4） 计算 h(x)=man_distance[2]+min(d1, d2)作

（5） 对 h(x)进行加权，h(x)=1.75h(x)

（6） Return h(x)

（八）次最优搜索

1. 方法介绍

该方法需要测试是否到达一个目标状态，并且选择距离当前位置最近的豆子路径，循环此过程直到吃掉所有的豆子。在本实验中我们选择的是 A*来找到最短的路径，因为 A*算法总能找到最佳的路径，相对于其他搜索策略来说是更为合适的方案。

2. 伪代码

findPathToClosestDot(self, gameState):

（1）准备工作：得到起始位置和食物、墙的位置

（2）定义问题 AnyFoodSearchProblem(gameState)

（3）返回用 A*算法找到的上述问题的最短路径

isGoalState(self, state):

（1） 取出坐标值(x, y)

（2） 得到食物位置 foodGrid

（3） if 该坐标处有食物或者当前地图中已经没有食物：

（4） Return True

三、 算法实现

1. 实验环境：python2.7

2. 问题规模：本实验总的来说是基于搜索树进行的最优路径搜索问题，每个树节点上最多有三个子节点，对应下一步可能选择的三个方向。假设地图规模是 n*n 的，则最坏情况下树的节点数为$\frac{n^2-1}{2}$，即问题规模为$O(n^2)$

3. 数据结构：

（1）深度优先算法：栈

（2）广度优先算法：队列

（3）代价一致和 A*算法：优先队列

4. 实验结果：以下实验结果中，截图为实验要求测试结果，列表为额外测试得到结果。

（1）应用深度优先算法找到特定位置的豆：

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigMaze -z .5 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
tinyMaze	0.0	10	15	500
mediumMaze	0.0	130	146	380
bigMaze	0.0	210	390	300

（2）广度优先算法：

```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l tinyMaze -p SearchAgent
-a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l mediumMaze -p SearchAgen
t -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
C:\Users\Lenovo\Desktop\flr\search>python2 pacman.py -l bigMaze -p SearchAgent -
a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
tinyMaze	0.0	8	15	502
mediumMaze	0.0	68	269	442
bigMaze	0.0	210	620	300

（3）代价一致：

```
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
        半:
```

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>Python
 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
mediumMaze	0.0	68	269	442
mediumDottedMaze	0.0	1	186	186
mediumScaryMaze	0.0	68719479864	108	418

（4）A*算法：

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
tinyMaze	0.0	8	14	502
mediumMaze	0.0	68	221	300
bigMaze	0.0	210	549	300

（5）角落问题：

```
D:\HIT\AI\lab\search>python2 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.3 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 162 in 4.8 seconds
Search nodes expanded: 7949
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:        378.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
tinyCorners	0.0	28	252	512
mediumCorners	0.3	106	1966	434
bigCorners	4.8	162	7949	378

（6）角落问题（启发式）：

```
D:\HIT\AI\lab\search>python2 pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 692
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win

D:\HIT\AI\lab\search>python2 pacman.py -l tinyCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 154
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
D:\HIT\AI\lab\search>python2 pacman.py -l bigCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 162 in 0.5 seconds
Search nodes expanded: 1725
Pacman emerges victorious! Score: 378
Average Score: 378.0
Scores:        378.0
Win Rate:      1/1 (1.00)
Record:        Win
```

地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
tinyCorners	0.0	28	154	512
mediumCorners	0.1	106	692	434
bigCorners	0.5	162	1725	378

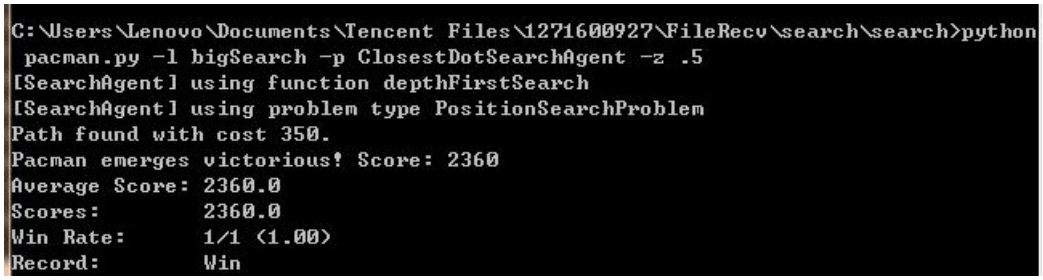
（7）吃掉所有的豆子：

```
D:\HIT\AI\lab\search>python2 pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 5.6 seconds
Search nodes expanded: 7101
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

启发式函数	搜索耗时（s）	路径代价	拓展节点数	得分
Manhattan	6.8	60	7553	570
Chebyshev	8.9	60	8869	570
Euclidean	8.6	60	8513	570
Octile	13.6	60	11302	570

启发式函数	地图类型	搜索耗时（s）	代价	拓展节点数	得分
Manhattan	trickySearch	7.5	60	7553	570
	tinySearch	0.2	27	957	573
Manhattan + Breaking Tie (p = 0.001)	trickySearch	7.1	60	7275	570
	tinySearch	0.1	27	612	573
Manhattan +Breaking Tie (p = 0.75)	trickySearch	6.8	60	7101	570
	tinySearch	0.0	27	259	573

（8）次最优搜索：



地图类型	路径代价	得分
bigSearch	350	2360
mediumSearch	171	1409
tinySearch	31	569
trickySearch	68	562

四、 总结及讨论

通过比较问题一（深度优先）和问题二（广度优先）的结果可以看出，深度优先算法拓展的节点数要多于广度优先算法，但是其分数总和来说没有广度优先算法更高。考虑到二者的策略不同，分析其原因是深度优先算法能在拓展出节点较少的情况下便能先于广度优先算法找到一条到达目标节点的路径，但是往往不会是最短的路径，所以在这一点上，广度优先算法会优于深度优先算法。

对于问题二（广度优先）和问题三（代价一致），通过在 mediumMaze 地图中的结果可以看出，由于此处每一步的代价都为 1，所以二者的实验结果相同，也就是说，当代价都相同且为 1 时，代价一致的结果与广度优先算法是等同的。

问题四（A*算法）和问题二（广度优先）使用的策略相比，明显 A*算法需要拓展的节点更少，因为它使用了启发性函数的策略和贪心的思想，使得寻找了路径的过程中不必对所有遇到的节点进行遍历，也正是因为这个原因，使得 A*算法的速度相较于广度优先算法更高，这可在问题五（使用 bfs 的角落问题）和问题六（使用启发性算法 A*的角落问题）的实验结果比较中得出。但是无论如何，两种方法找到路径的代价都是相同的。由于广度优先算法具有完备性，这充分说明了 A*算法一定能结束在最佳路径上。

问题七（吃掉所有的豆子）中，韩雪婷同学对于该问题进行

了探索性实验，在综合考虑了 Manhattan、Chebyshev、Euclidean、Octile 四种计算距离的方法以后，发现 Manhattan 是花费时间和拓展节点最少的，因为 Manhattan 正是适用于只能四方移动的网格地图的距离计算方法。另外，她考虑了对启发性函数 f(n)=g(n)+h(n)进行加权，使得算法倾向于选择到终点距离更小的路径，加权后的函数为 f(n)=g(n)+(1+p)h(n)，通过微调 h(n)的权值，得到使得耗时和拓展节点数最少的参数 p=0.75，在一定程度上提高了 A*算法的效率。

问题八（次最优搜素）使用独立的地图，因此未设计对比实验，从理论上分析，该搜索算法每次扩展局部最优的节点，并不总能找到最优路径。

五、 组员贡献度

韩雪婷：25% 陈抒语：25% 廖思瑀：25% 卢茉莉：25%

附（源代码截图）：

问题一：深度优先搜索

```
def depthFirstSearch(problem):
    s = problem.getStartState()      #初始节点
    closed = []      #建立一个closed表，置为空
    open = util.Stack()
    open.push((s,[]))    #将初始节点放入open表（栈）
    while not open.isEmpty():    #检查open表是否空
        cnode,action = open.pop()
        if problem.isGoalState(cnode):       #到达目标节点，退出
            return action
        if cnode not in closed:
            closed.append(cnode)
            successor = problem.getSuccessors(cnode)     #将子节点放入open表
            for location,direction,cost in successor:
                if(location not in closed):
                    open.push((location,action+[direction]))
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()
```

问题二：广度优先搜索

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    s = problem.getStartState()      #初始节点
    closed = []      #标记已经遍历过的节点，置为空
    q = util.Queue()     #建立队列来保存拓展到的各节点
    q.push((s,[]))
    while not q.isEmpty():
        state,path = q.pop()
        if problem.isGoalState(state):  #如果是目标状态，则返回当前路径，退出函数
            return path
        if state not in closed:
            closed.append(state)     #标记其为已经遍历过的状态
            for node in problem.getSuccessors(state):
                n_state = node[0]
                direction = node[1]
                if n_state not in closed:   #如果后继状态未被遍历过，将其入队列
                    q.push((n_state, path + [direction]))
    return path
```

问题三：代价一致算法

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    #初始状态
    start = problem.getStartState()
    #标记已经搜索过的状态集合exstates
    exstates = []
    #用优先队列PriorityQueue实现ucs
    states = util.PriorityQueue()
    states.push((start,[]),0)
    while not states.isEmpty():
        state,actions = states.pop()
        #目标测试
        if problem.isGoalState(state):
            return actions
        #检查重复
        if state not in exstates:
            #拓展
            successors = problem.getSuccessors(state)
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    #ucs比bfs的区别在于getCostOfActions决定节点拓展的优先级
                    states.push((coordinate,actions + [direction]),problem.getCostOfActions(newActions))
        exstates.append(state)
    return  actions
    util.raiseNotDefined()
```

问题四：A*算法

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    start = problem.getStartState()     #初始状态
    exstates = []        #是否访问过该节点，初始为空
    states = util.PriorityQueue()
    states.push((start,[]),nullHeuristic(start,problem))        #初始节点入栈
    nCost = 0
    while not states.isEmpty():
        state,actions = states.pop()
        if problem.isGoalState(state):      #到达目标节点，退出
            return actions
        if state not in exstates:
            successors = problem.getSuccessors(state)       #查找子节点
            for node in successors:
                coordinate = node[0]
                direction = node[1]
                if coordinate not in exstates:
                    newActions = actions + [direction]
                    newCost = problem.getCostOfActions(newActions) + heuristic(coordinate,problem)
                    states.push((coordinate,actions + [direction]),newCost)
        exstates.append(state)
    return  actions
    util.raiseNotDefined()
```

问题五：查找所有角落

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

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    location = state[0]
    corner_state = state[1]
    return len(corner_state) == 4        #角落状态表中不含重复状态，包含四个状态时达到目标
    util.raiseNotDefined()
```

```
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 ***"
        corner_state = state[1]
        list_corner_state = list(corner_state)
        x,y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)     #计算下一个状态的位置坐标
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            if ((nextx,nexty) in self.corners) & ((nextx,nexty) not in list_corner_state):      #是一个角落位置并且未被遍历过，将其加入角落状态表中
                list_corner_state.append((nextx, nexty))
            successor = (((nextx,nexty), list_corner_state), action, 1)
            successors.append(successor)
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

问题六：角落问题——启发式

```
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).
    """
    #角落坐标
    corners = problem.corners
    #墙壁坐标
    walls = problem.walls
    #起始源点
    node = state[0]
    un_viscorner = [item for item in corners if item not in state[1]]
    #计算所有未经过的角落的到现在距离的最小曼哈顿值
    hn = minmanhattan(un_viscorner,state[0])
    return hn
    #"*** YOUR CODE HERE ***"
    #return 0 # Default to trivial solution

#遍历所有的corner的豆子，选择最小的曼哈顿距离作为启发函数
def minmanhattan(corners,pos):
    #corners长度为0，返回0
    if len(corners) == 0:
        return 0
    hn = []
    #循环计算每一个点的曼哈顿距离
    for loc in corners:
        dis = abs(loc[0] - pos[0] + abs(loc[1] - pos[1])+minmanhattan([c for c in corners if c != loc],loc))
        hn.append(dis)
    #返回最小的曼哈顿值
    return min(hn)
```

问题七：吃掉所有的“豆”

```
"""
position, foodGrid = state
food_available = []
hvalue = 0
for i in range(0,foodGrid.width):
    for j in range(0,foodGrid.height):
        if(foodGrid[i][j] == True):
            food_location = (i,j)
            food_available.append(food_location)
if(len(food_available) == 0):
    return 0
#初始化距离为0
max_distance=((0,0),(0,0),0)
for current_food in food_available:
    for next_food in food_available:
        if(current_food==next_food):
            pass
        else:
            #使用曼哈顿距离构造启发式函数
            distance = util.manhattanDistance(current_food,next_food)
            if(max_distance[2] < distance):
                max_distance = (current_food,next_food,distance)
#把起点和第一个搜索的食物连接起来
#处理只有一个食物的情况
if(max_distance[0]==(0,0) and max_distance[1]==(0,0)):
    hvalue = util.manhattanDistance(position,food_available[0])
else:
    d1 = util.manhattanDistance(position,max_distance[0])
    d2 = util.manhattanDistance(position,max_distance[1])
    hvalue = max_distance[2] + min(d1,d2)
#Breaking Tie
hvalue *=(1.0+0.75)
return hvalue
"*** YOUR CODE HERE ***"
return 0
```

问题八：次优先搜索

```
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)
    return search.aStarSearch(problem)        #调用A*算法，寻找最短路径
```

```
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
    foodGrid = self.food    #找到食物所在位置
    if(foodGrid[x][y] == True)or(foodGrid.count() == 0):        #判断该点是否有食物
        return True
    # util.raiseNotDefined()
```