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

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：T2 实验台号____

学生专业：计算机类

学生学号：SZ170110320

学生姓名：韩雪婷

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

实验报告至少包括：

一. 简介/问题描述

1.1 待解决问题的解释

1.2 问题的形式化描述

1.3 解决方案介绍（原理）

二. 算法介绍

2.1 所用方法的一般介绍

2.2 算法伪代码

三. 算法实现

3.1 实验环境与问题规模

3.2 数据结构

3.3 实验结果

3.4 系统中间及最终输出结果（要求有屏幕显示）

四. 总结及讨论（对该实验的总结以及任何该实验的启发），

参考文献

附录—源代码及其注释（纸质版不需要打印）

小组成员贡献量：各 25%

1 问题描述

1.1 待解决问题的解释

利用课本第四章内各类搜索算法编写程序解决 8 个吃豆人问题。

[1]. 运用深度优先算法找到一个特定的位置的豆。

[2]. 运用宽度优先算法找到一个特定的位置的豆。

[3]. 运用代价一致算法找到一个特定的位置的豆。

[4]. 运用 A*算法找到一个特定的位置的豆，其中启发式函数为曼哈顿距离。

[5]. 在角落迷宫的四个角上面有四个豆。要求找到一条访问所有四个角落的最短的路径。

[6]. 构建合适的启发函数，利用 A*算法寻找访问迷宫的四个角落的最短路径。

[7]. 用尽可能少的步数吃掉所有的豆子。

[8]. 定义一个优先吃最近的豆子的次最优算法。

1.2 问题的形式化描述

[1]. 初始状态

吃豆人的初始位置(x,y)，和 n 颗豆子们的初始分布位置：$\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}\}$

[2]. 吃豆人的行为

IF $x=x_i$ && $y=y_i$ ($i=1,2,\cdots,n$) THEN Eat（吃豆人吃豆）。

IF $x+1!=Wall$ THEN Right（吃豆人向右移动）

IF $x-1!=Wall$ THEN Left（吃豆人向左移动）

IF $y+1!=Wall$ THEN Up（吃豆人向上移动）

IF $y-1!=Wall$ THEN Down（吃豆人向下移动）

[3]. 行动转移模型

- Eat: $\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$ → $\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_{i-1},y_{i-1}),(x_{i+1},y_{i+1}),\cdots(x_{n-1},y_{n-1})\}\}$
- Right：$\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$ → $\{(x+1,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$
- Left：$\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$ → $\{(x-1,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$
- Up：$\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$ → $\{(y+1,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$
- Down：$\{(x,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$ →

$$\{(y-1,y),\{(x_1,y_1),(x_2,y_2),\cdots,(x_i,y_i),\cdots(x_n,y_n)\}\}$$

[4]. 目标测试

IF n = 0 THEN IsGoalState

如果迷宫内没有剩余豆子，则到达目标状态。

[5]. 路径耗散(Cost)

吃豆人所走过的距离。

问题的解：从初始状态到目标状态的解

最优解：路径耗散最小的解。

1.4 解决方案介绍

吃豆人问题实际上是一个寻路搜索问题。吃豆人要寻找从当前位置到一颗特定豆子的路径，迷宫里的墙限制了吃豆人的行动，是寻路上的障碍物。当在地图中有多颗豆子的时候，我们要寻找一条能遍历各个豆子的最短路径。

因此我们可以采用各种搜索算法实现吃豆人寻豆问题，包括非启发式的深度优先搜索、宽度优先搜索，以及启发式的 A*算法等。

2 算法介绍

2.1 所用方法的一般介绍

[1]. 深度优先搜索

在深度优先搜索中，优先扩展最新产生的（即最深的）节点，是后生成的节点先扩展的策略。

从初始节点开始扩展，若没有得到目标节点，则选择最新产生的子节点进行扩展；若还是不能到达目标节点，则再对刚才最新产生的子节点进行扩展，一直如此向下搜索。当到达某个子节点，且该子节点既不是目标节点又不能继续扩展时，才选择其兄弟节点进行考察。

[2]. 宽度优先搜索

又称为宽度优先搜索，是一种先生成的节点先扩展的策略。

从初始节点开始，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。

[3]. 代价一致搜索

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断层进行扩展，而是沿着等代价路径断层进行扩展。

在代价一致搜索算法中，把从起始节点到任一节点 i 的路径代价记为 g(i)。从初始节点 S0 开始扩展，若没有得到目标节点，则优先扩展最少代价 g(i)的节点，一直如此向下搜索。

[4]. A*搜索

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。两条限制如下：

（1）g(n)是对最小代价 g*(n)的估计，且 g(n)>0；

（2）h(n)是最小代价 h*(n)的下界，即对任意节点 n 均有 h(n) ≤ h*(n)

假设 f*(n)是从初始节点出发，约束经过节点 n 到达目标节点的最小代价，估价函数 f(n)是对 f*(n)的估计值。记 f*(n) = g*(n) + h*(n).

其中，g*(n)是从初始节点出发到达当前节点的最小代价,h*(n)是当前节点到目标节点的最小代价。

需要注意的是，启发式函数 h(n)可以用来控制 A*的行为。

- 一种极端情况，如果 h(n)是 0，则只有 g(n)起作用。此时 A* 算法演变成 Dijkstra 算法，能保证找到最短路径。
- 如果 h(n)总是比从 n 移动到目标的代价小（或相等），那么 A* 保证能找到一条最短路径。h(n)越小，A* 需要扩展的点越多，运行速度越慢。
- 如果 h(n)正好等于从 n 移动到目标的代价，那么 A* 将只遵循最佳路径而不会扩展到其他任何结点，能够运行地很快。
- 如果 h(n)比从 n 移动到目标的代价高，则 A* 不能保证找到一条最短路径，但它可以运行得更快。
- 另一种极端情况，如果 h(n)比 g(n)大很多，则只有 h(n)起作用，同时 A* 算法演变成贪婪最佳优先搜索算法（Greedy Best-First-Search）。

[5]. 问题 5-7 在以上四个算法的基础上稍作修改就可解决，不再赘述。

[6]. 次最优搜索

使用 A*算法，找到吃豆人通过最短路径能到达的一颗豆子，将其视为“最近的豆子”。

吃豆人按照最短路径，将“最近的豆子”吃掉后，再将该点作为新的起始位置，进行 A*算法搜索，找到新的最近豆子吃掉，如此循环往复，直到吃完所有的豆子。

2.2 算法伪代码

[1]. 深度优先搜索

```
Open.Push(So)        #把初始节点 So 放入 Open 表（栈）
Closed = []          #建立一个 CLOSED 表，置为空
#检查 Open 表是否为空表
while Open != NULL
#把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n
    Sn = Open.Pop
```

```
        Closed.append(Sn)
        #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
        If Sn == IsGoalState()
            return True
        #扩展节点 n， 将其子节点放入 Open 表的首部，并为每个子节点设置指向父节点的指针
        successors = GetSuccessors(Sn)
        If (successors)
            Open.Push(successors)
            successors → Sn
    #若 Open 表为空，则问题无解，失败退出
    return False
```

[2]. 宽度优先搜索

```
    Open.Entry(So)      #把初始节点 So 放入 Open 表（队列）
    Closed = []         #建立一个 CLOSED 表，置为空
    #检查 Open 表是否为空表
    while Open != NULL
      #把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n
      Sn = Open.Delete
      Closed.append(Sn)
      #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
      If Sn == IsGoalState()
        return True
      #扩展节点 n， 将其子节点放入 Open 表的尾部，并为每个子节点设置指向父节点的指针
      successors = GetSuccessors(Sn)
      If (successors)
        Open.Entry(successors)
        successors → Sn
    #若 Open 表为空，则问题无解，失败退出
    return False
```

[3]. 代价一致搜索

```
    Open.Entry(So)      #把初始节点 So 放入 Open 表（队列）
    g(So) = Calculateg(So)
    Closed = []         #建立一个 CLOSED 表，置为空
```

```
#检查 Open 表是否为空表
while Open != NULL
#把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n
    Sn = Open.Delete
    Closed.append(Sn)
    #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
    If Sn == IsGoalState()
        return True
    #扩展节点 n， 将其子节点放入 Open 表，并为每个子节点设置指向父节点的
    指针
    successors = GetSuccessors(Sn)
    If (successors)
       Open.Entry(successors)
        successors → Sn
       #计算各个节点的代价 g（ni）
       g(successors) = Calculateg(successors)
       #将 Open 表内的节点按 g（ni）从小到大排序
       Open.Sort(ni, g(ni), SmalltoLarge)
#若 Open 表为空，则问题无解，失败退出
return False
```

[4]. A*算法

```
Open.Entry(So)        #把初始节点 So 放入 Open 表（队列）
f(So) = Calculatef(So)
Closed = []           #建立一个 CLOSED 表，置为空
#检查 Open 表是否为空表
while Open != NULL
#把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n
Sn = Open.Delete
Closed.append(Sn)
#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
If Sn == IsGoalState()
    return True
#扩展节点 n， 将其子节点放入 Open 表，并为每个子节点设置指向父节点的指针
successors = GetSuccessors(Sn)
If (successors)
```

```
Open.Entry(successors)
    successors → Sn
#计算各个节点的代价 f（ni）
f(successors) = Calculatef(successors)
#将 Open 表内的节点按 f（ni）从小到大排序
Open.Sort(ni, f(ni), SmalltoLarge)
#若 Open 表为空，则问题无解，失败退出
return False
```

3 算法实现

3.1 实验环境与问题规模

硬件环境：PC 机

软件环境：python2.7 ; Windows10

问题规模：地图大小 k＝m*n，则有 k 个可能的位置，对吃豆人和 a 颗豆子，共$O(k^{1+a})$种状态。

3.2 数据结构

[1]. 栈

```
class Stack:
    "A container with a last-in-first-out (LIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Push 'item' onto the stack"
        self.list.append(item)

    def pop(self):
        "Pop the most recently pushed item from the stack"
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the stack is empty"
        return len(self.list) == 0
```

[2]. 队列

```
class Queue:
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)

    def pop(self):
        """
          Dequeue the earliest enqueued item still in the queue. This
          operation removes the item from the queue.
        """
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0
```

[3]. 优先队列

```
class PriorityQueue:
    """
      Implements a priority queue data structure. Each inserted item
      has a priority associated with it and the client is usually interested
      in quick retrieval of the lowest-priority item in the queue. This
      data structure allows O(1) access to the lowest-priority item.
    """
    def   __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1
```

```
    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        # If item already in priority queue with higher priority, update its priority and rebuild the heap.
        # If item already in priority queue with equal or lower priority, do nothing.
        # If item not in priority queue, do the same thing as self.push.
        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)
```

3.3 实验结果

[1]. 深度优先 宽度优先

表 3-1-1

问题	地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
深度优先	tinyMaze	0.0	10	15	500
	mediumMaze	0.0	130	146	380
	bigMaze	0.0	210	390	300
宽度优先	tinyMaze	0.0	8	15	502
	mediumMaze	0.0	68	269	442
	bigMaze	0.0	210	620	300

问题 1、2 的实验结果如表 3-3-1 所示。

深度优先搜索不能保证找到最优解，是不完备的搜索策略。在 tinyMaze 中搜索到的路径的路径代价为 10，大于最优解代价。

广度优先搜索得到的解是搜索树中路径最短的解（最优解），但搜索效率较低，扩展节点较深度优先搜索更多。

[2]. 代价一致 A*

表 3-3-2

问题	地图类型	搜索耗时（s）	路径代价	拓展节点数	得分
代价一致	mediumMaze	0.0	68	269	442
	mediumDottedMaze	0.0	1	186	186
	mediumScaryMaze	0.0	68719479864	108	418
A*	tinyMaze	0.0	8	14	502
	mediumMaze	0.0	68	221	300
	bigMaze	0.0	210	549	300

问题 3、4 的实验结果如表 3-3-2 所示。

代价一致算法找到的是最小代价路径，而最小代价路径不一定是最短路径。即代价一致算法找到的路径一定有最小的 g(h)，但是实际路径代价不一定最小。在 mediumScaryMaze 中代价一致算法找到的道路明显不是最短路径。

A*算法一定能结束在最佳路径上，即其路径代价一定最小。

[3]. 角落问题

表 3-3-3

问题	地图类型	搜索耗时（s）	路径代价	扩展节点数	得分
查找所有角落（BFS）	tinyCorners	0.0	28	252	512
	mediumCorners	0.3	106	1996	434
	bigCorners	5.0	162	7949	378
角落问题（A*）	tinyCorners	0.0	28	154	512
	mediumCorners	0.1	106	692	434
	bigCorners	0.5	162	1725	378

问题 5、6 的实验结果如表 3-3-3 所示。

在角落问题中，广度优先和 A*算法都可以找到最短路径，路径代价分别为 28，106，162。但是 A*算法效率更高，它的扩展节点数普遍少于广度优先算法。

此处 A*算法采用的启发函数是曼哈顿距离。

[4]. 吃掉所有的豆子

问题 7 在问题 4A*算法的基础上稍作修改就可解决，在问题 4 中 A*算法采用的启发式函数是默认的曼哈顿距离。我们对于选择合适的启发函数做了如下探究。

我们考虑四个常用启发式函数如表 3-3-4 所示。

表 3-3-4

启发式函数	计算公式	备注
Manhattan	$\lvert node.x - goal.x\rvert + \lvert node.y - goal.y\rvert$	适用于只能四方向移动的网格地图。
Chebyshev	$\max\{\lvert node.x - goal.x\rvert, \lvert node.y - goal.y\rvert\}$	适用于可以八方向移动的网格地图。
Euclidean	$\sqrt{[(node.x - goal.x)^2 + (node.y - goal.y)^2]}$	适用于可以向任何方向移动的地图。
Octile	$a = \lvert node.x - goal.x\rvert$ $b = \lvert node.y - goal.y\rvert$ $\max\{a,b\} + (\sqrt{2} - 1) \cdot \min\{a,b\}$	欧氏距离的改良，减少开方计算开销。

这四个启发式函数的实验结果如表 3-3-5 所示。

表 3-3-5

问题	启发式函数	搜索耗时（s）	路径代价	扩展节点数	得分
吃掉所有豆子	Manhattan	6.8	60	7553	570
	Chebyshev	8.9	60	8869	570
	Euclidean	8.6	60	8513	570
	Octile	13.6	60	11302	570

由表 3-3-5 可知，实验结果与理论相同，吃豆人属于网格地图中只能四方向移动的寻路问题，此时采用曼哈顿距离作为启发函数效果最好。

引入值相等的决胜法，修改一下估价函数：

$$f^*(n) = g^*(n) + (1 + p) \cdot h^*(n)$$

其中，当 p 越大，估价函数越向 h(n)方向偏移，即此时算法倾向于选择到终点代价更小的路径。

由 2.1[4]部分介绍内容可知，在加上(1+p)之后该启发式函数不符合 A*算法对估价函数的要求，不一定能够保证找到最短路径，但可以减少扩展节点数、使算法运行更快。

按照 autograder.py 的评分要求，该算法要在所有地图中找到最短路径，则需要牺牲一点运算时间，采用最简单的曼哈顿距离作为启发式函数。也可以直接采

用宽度优先方式搜索，根据评估方式，其消耗节点数较少。

针对某一特定地图，可以修改 p 值，更快地找到最短路径，如表 3-3-6 所示。

表 3-3-6

问题	启发式函数	地图	搜索耗时（s）	路径代价	扩展节点数	得分
吃掉所有豆子	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	27	259	573

可知，Breaking Tie 策略可以使得搜索耗时和扩展节点数减少，且减少量与具体 p 值的选择相关。

在豆子更密集的地图 tinySearch 中，Breaking Tie 的影响更大。

[5]. 次最优搜索

表 3-3-7

问题	地图	路径代价	得分
次最优搜索	bigSearch	350	2360
	mediumSearch	171	1409
	tinySearch	31	569
	trickySearch	68	562

问题 8 的实验结果如表 3-3-7 所示。可以看到，次最优搜索不追求全局最优，因此路径代价不一定是最小代价，但搜索速度更快，扩展节点较少。

3.4 系统中间及最终输出结果

问题 1

```
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
```

```
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 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
```

问题 2

```
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
```

问题 3

```
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
```

问题 4

```
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
```

问题 5

```
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
```

问题 6

```
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
```

问题 7

```
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
```

问题 8

```
C:\Users\Lenovo\Documents\Tencent Files\1271600927\FileRecv\search\search>python
 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

4 总结及讨论

伯克利的这个实验非常经典了，语言也使用的是相对旧一些的 python2，github 上有很多人给出解决方案。这种情况下想做出什么多么新的东西可能是比较困难了，只能在过程中多自己亲自动手去探索一下，也算是有所收获吧。

我负责 1,4,7 三题，其中 1 题和 4 题都属于比较基础的部分，对着老师 PPT 上的伪代码写出来了。而第七题为了找合适的启发函数，在网上查了许多资料，其中发现了不少有趣的算法，不过不适用于吃豆人的场景。对于寻路算法，地图中可能有移动的障碍物（吃豆人中的幽灵），可能有不同的移动方式等等，在不同场景下有不同的适用的算法。

一开始加了 Breaking Tie 算法，autograder 第 7 题直接判我零分，不得不找到伯克利大学这个实验的页面看评分标准。在之前也不知道评判标准是什么，是算法的用时还是拓展节点的多少还是得分，不同的算法各有所长。

探索的过程还比较有趣，也算有所收获。

参考文献

[1]. 邱磊. 基于 A*算法的游戏地图寻路实现及性能比较[J]. 陕西科技大学学报：自然科学版, 2011.

附录

代码

[1]. searchAgents.py

```
# -*- coding: utf-8 -*-
# searchAgents.py
# ---------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
This file contains all of the agents that can be selected to control Pacman.  To
select an agent, use the '-p' option when running pacman.py.  Arguments can be
passed to your agent using '-a'.  For example, to load a SearchAgent that uses
depth first search (dfs), run the following command:

> python pacman.py -p SearchAgent -a fn=depthFirstSearch

Commands to invoke other search strategies can be found in the project
description.

Please only change the parts of the file you are asked to.  Look for the lines
that say

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

The parts you fill in start about 3/4 of the way down. Follow the project description for details.

Good luck and happy searching!
"""

```
from game import Directions
from game import Agent
from game import Actions
import util
import time
import math
import search

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 # Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.right = right
        self.top = top
```

```
def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        #初始节点（开始位置，角落情况）
        allCorners = (False, False, False, False)
        start = (self.startingPosition, allCorners)
        return start
        util.raiseNotDefined()

def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        #目标测试：四个角落都访问过
        corners = state[1]
        bool = corners[0] and corners[1] and corners[2] and corners[3]
        return bool      #返回布尔值
        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 = []
        #遍历能够做的后续动作
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
```

```
Directions.WEST]:
                    # Add a successor state to the successor list if the action is legal
                    "*** YOUR CODE HERE ***"
                    x,y = state[0]
                    holdCorners = state[1]
                    dx, dy = Actions.directionToVector(action)
                    nextx, nexty = int(x + dx), int(y + dy)
                    hitsWall = self.walls[nextx][nexty]
                    newCorners = ()
                    nextState = (nextx, nexty)
                    #不碰墙
                    if not hitsWall:
                        #能到达角落，四种情况判断
                        if nextState in self.corners:
                            if nextState == (self.right, 1):
                                newCorners = [True, holdCorners[1], holdCorners[2], holdCorners[3]]
                            elif nextState == (self.right, self.top):
                                newCorners = [holdCorners[0], True, holdCorners[2], holdCorners[3]]
                            elif nextState == (1, self.top):
                                newCorners = [holdCorners[0], holdCorners[1], True, holdCorners[3]]
                            elif nextState == (1,1):
                                newCorners = [holdCorners[0], holdCorners[1], holdCorners[2], True]
                            successor = ((nextState, newCorners), action,  1)
                        #去角落的途中
                        else:
                            successor = ((nextState, holdCorners), action, 1)
                        successors.append(successor)
                self._expanded += 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.  This is implemented for you.
        """
        if actions == None: return 999999          #找不到路径
        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          #如果撞墙
        return len(actions)        #以路径长度评估代价


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 ***"
    position = state[0]
    stateCorners = state[1]
    corners = problem.corners          #获取角落位置
    top = problem.walls.height-2#最远能到达的高度
    right = problem.walls.width-2     #最远能到达的宽度
    node = []
   #将角落放入待遍历的 list
    for c in corners:
```

```
            if c == (1,1):
                if not stateCorners[3]:
                    node.append(c)
            if c == (1, top):
                if not stateCorners[2]:
                    node.append(c)
            if c == (right, top):
                if not stateCorners[1]:
                    node.append(c)
            if c == (right, 1):
                if not stateCorners[0]:
                    node.append(c)
        cost = 0
        currPosition = position
        while len(node) > 0:
            distArr= []
            for i in range(0, len(node)):
                dist = util.manhattanDistance(currPosition, node[i])
                distArr.append(dist)
            mindist = min(distArr)        #minmanhattan
            cost += mindist
            minDistI= distArr.index(mindist)
            currPosition = node[minDistI]
            del node[minDistI]       #访问后从 List 中删除节点
        return cost



    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
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


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

        #util.raiseNotDefined()

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
        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()

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))
```

[2]. search.py

```
# -*- coding: utf-8 -*-
# search.py
# ---------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
In search.py, you will implement generic search algorithms which are called by
Pacman agents (in searchAgents.py).
"""

import util

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 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()

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

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()

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."""
    "*** 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()

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

[3]. util.py 中部分增加内容

```
    def manhattanDistance( xy1, xy2 ):
        "Returns the Manhattan distance between points xy1 and xy2"
        return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )

    def ChebyshevDistance(xy1,xy2):
        return max(abs( xy1[0] - xy2[0] ) ,abs( xy1[1] - xy2[1] ))
```

```
def EuclideanDistance(xy1,xy2):
    a = xy1[0] - xy2[0]
    b = xy1[1] - xy2[1]
    return math.sqrt(a*a+b*b)

def OctileDistance(xy1,xy2):
    a = xy1[0] - xy2[0]
    b = xy1[1] - xy2[1]
    k = math.sqrt(2)-1
    return max(a,b)+k*min(a,b)

def CrossDistance(start,current,goal):
    dx1 = current[0] - goal[0]
    dy1 = current[1] - goal[1]
    dx2 = start[0] - goal[0]
    dy2 = start[1] - goal[1]
    cross = abs(dx1*dy2 - dx2*dy1)
    return cross
```