**人工智能导论实验报告**

姓名：张茗帅

学号：1140310606

哈尔滨工业大学计算机学院

2017年1月

**一、实验简介**

1、实验题目：

Search in Pacman

汉意：吃豆人找豆豆

2、实验背景：

有一个爱吃豆子的Pacman走进了迷宫的世界，在对应特定的迷宫中的特定位置有着若干的豆子，Pacman当然想要吃到这些豆子，那么我们要帮助他设计一个最优路径来引导他顺利的吃到豆子。

3、实验理解：

实质上可以将此问题抽象为搜索树问题，Pacman初始状态state设置为根结点，Pacman根结点的孩子最多可以有4个，即Pacman在迷宫中向东、南、西、北四个方向前行一步所到达的新的状态state，当然，若Pacman在某一个方向上无法行走，即有墙，那么该状态结点的孩子结点就会少了行走至该方向的新状态，换句话说，每个状态结点的孩子结点必须是可达的（该结点可走向孩子结点而不遇到墙）。

于是，从Pacman的起始位置开始，但凡在对应迷宫中可达的位置均为一个独立的状态，而每个状态都是树中的一个结点，我们按照不同的问题，设定不同的终止结点(例如：包含食物的位置设置为终止结点)，那么我们所需要做的，就是按照不同问题的要求，找到一条从根结点到终止结点的最优路径。

4、实验要求：

本次实验包含8个问题

·问题1：应用深度优先算法找到一个特定的位置的豆

·问题2：应用宽度优先算法找到一个特定的位置的豆

·问题3：应用代价一致搜索方法找到一个特定的位置的豆

·问题4： 应用A\*搜索方法，利用曼哈顿距离作为启发函数寻找一个特定位置的豆

·问题5：在角落迷宫的四个角上面有四个豆。构造算法要求找到一条访问所有四

个角落的最短的路径

·问题6：构建合适的启发函数，找到一条能够访问所有四个角落的最短的路径

·问题7：用尽可能少的步数吃掉所有的豆子（不一定在角落中）。并构建合适的启

发函数优化算法

·问题8：应用次最优搜索策略，定义一个优先吃最近的豆子函数来实现吃掉所有

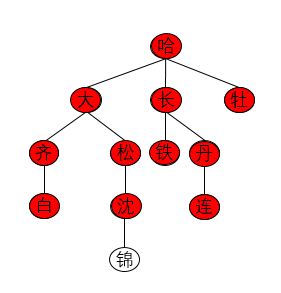
豆子

在实验过程中，我们需要通过编辑search.py和searchAgents.py来设计解决相关问题的算法，运用util.py中的数据结构。

**二、实验方法**

按照不同的问题设计了不同的解决方法

问题1 应用深度优先算法找到一个特定的位置的豆

 先简单介绍下我对深度优先算法的理解，深度优先算法DFS即Depth First Search.其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，若不能继续深入时该结点不是目标结点，则回溯至上该结点的父结点，继续沿着父结点的其它儿子结点（未曾遍历过的）进行搜索，若该父结点不存在未曾遍历过的儿子结点，则继续回溯该父结点的父结点，寻找新的父结点的未曾遍历的儿子结点，以此类推，找到一个未曾遍历的结点后，继续深入到底，若还未找到，则反复回溯，直到找到了目标结点则终止。

图示：

遍历的方向为：

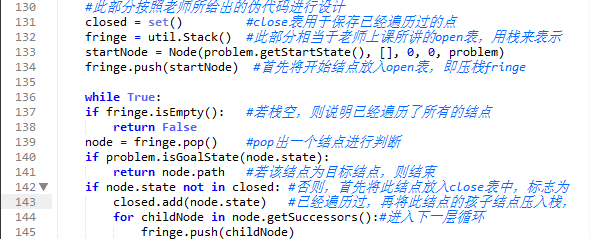
哈->大->齐->白->松

->沈->锦->长->铁->丹->

连->牡

代码设计：(在search.py中)

def depthFirstSearch(problem)



注释如图

我们知道，由于栈的特点是先进后出，后进先出，于是它可以被我们拿来使用与DFS这个算法，就那上个图来说，按照本程序，首先进入栈的元素有：哈（第134行压栈）

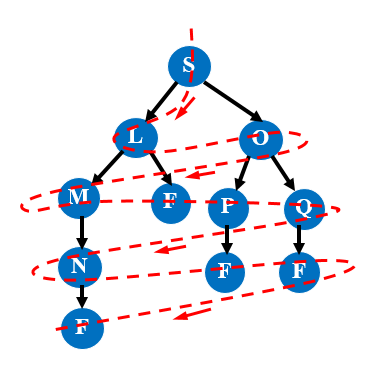
然后第137行进行判断，栈非空，同时在第140行进行判断，该结点非终止结点，继续运行到第142行，该结点未曾遍历过（不在close表中），则将此结点和其孩子结点压栈（孩子为右至左，顺序不影响实质），于是现在栈内（栈底->栈顶）：哈、牡、长、大。

进入下一次循环的时候，取出栈顶元素为“大”，则继续按照上一段的步骤往下走，此次流程完毕后，栈内为（栈底->栈顶）：哈、牡、长、大、松、齐。

再下一次循环时，栈顶元素为“齐”，则继续往深一层遍历……

由于栈的这种特性（先进后出，后进先出），被我们得以利用与DFS算法中十分便利。

问题2 应用宽度(广度)优先算法找到一个特定的位置的豆

该算法与深度优先算法不同的是，按层遍历搜索，将树分层，不妨设根结点为第0层，则根结点的孩子结点为第1层……以此类推。即搜索策略为先搜索跟结点，然后搜索全部的第1层的所有结点，然后继续搜索第2层的所有结点……直到找到目标结点为止。

如图所示：

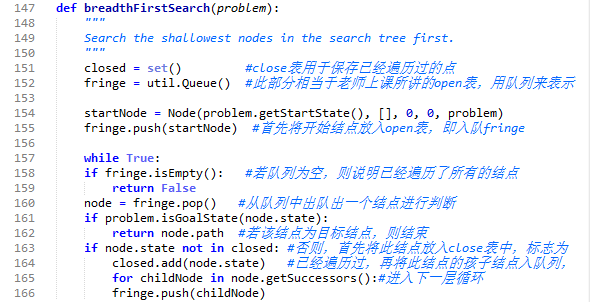
遍历顺序为：

S->L->O->M->F->P->Q->N->F->F->F

代码设计：

如下图

注释如图：

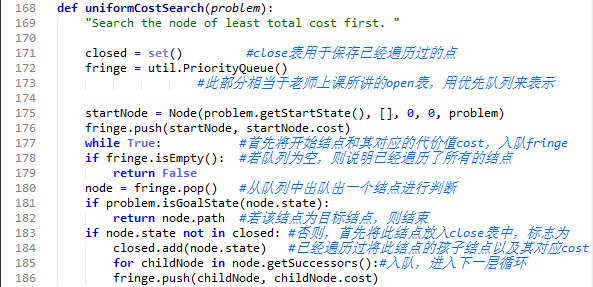


广度优先搜索算法采用了队列Queue作为open表而不是栈，这一点恰恰利用了广度优先算法的特性，队列的特点是先进先出，后进后出，于是按照上个参考图，我们知道当我们首先将根结点S压入队列，然后出队，再把根结点S的孩子结点L、O压入队列，下次出队的是L，此时将L的孩子结点M、F压入队列，但下一次出队的是O（之前先与M和F入队的），再将O的孩子结点P、Q入队，此时这一层的结点遍历完毕，队列中为Q、P、F、M，下一次出队的元素为M，即进行下一层的遍历，直到找到目标结点。

问题3：应用代价一致搜索方法找到一个特定的位置的豆

代价一致搜索树的每一条边都有一个代价值与其对应，然后我们现在要找到一个搜索策略使得花费的代价最小，此时我们仍然运用广度优先算法作为跟，为每一条边添加一个cost代价值，再用一个优先队列来按照cost进行排序

代码设计如图： 注释如下



本设计的核心思想是引入了优先队列PriorityQueue，该队列为util.py中的数据结构，我们将cost值伴随着结点一同入队，然后在队列中按照cost值进行优先级排序，最优的代价优先级为最高，获得了出队的优先权，则下一次首先出队的便是可以当前获得最优代价的那个结点，直到找到终止结点为止。

举个简单的例子

D

A

C

B

如图

2 3

1

首先A入队并且出队后，紧接着B、C、D、

伴随着他们的cost代价值一同入队

进入优先队列后，队列会按照代价值对他们进行

排序，具有最优属性的代价值会排在队列的前端，下一次便会优先出队，不妨设代价值越小越好，则排序过后的队列为（队首->队尾）：C、B、D，以此类推，最终若在此路径上找到终止结点，则花费的代价值最小。

问题4： 应用A\*搜索方法，利用曼哈顿距离作为启发函数寻找一个特定位置的豆

A\*（A-Star)算法是一种静态路网中求解最短路最有效的直接搜索方法，也是许多其他问题的常用启发式算法。

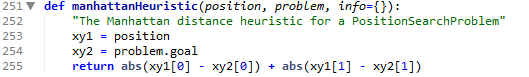
公式表示为： f(n)=g(n)+h(n),

其中 f(n) 是从初始[状态](http://baike.baidu.com/subview/705553/8050644.htm)经由状态n到目标状态的代价估计，

g(n) 是在[状态空间](http://baike.baidu.com/view/3821785.htm)中从初始状态到状态n的实际代价，

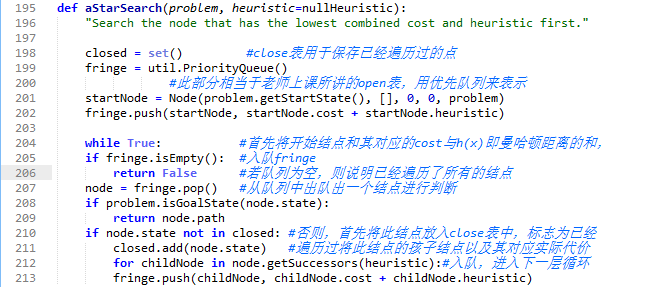
h(n) 是从状态n到目标状态的最佳路径的估计代价。

在本问题中，利用曼哈顿距离作为启发函数，即：



所谓曼哈顿距离，及当前结点和达目标结点这两个点在标准坐标系上的绝对轴距总和。

代码设计：



按照A\*算法的设计思想f(n)=g(n)+h(n)

g(n)为对应结点当前的实际代价cost；

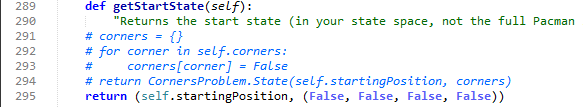
h(n)为当前结点的曼哈顿距离（与目标结点）；

故我们在入队时将g(n)+h(n)作为优先级的判断值，从而继续进行队列中的重新排序，找到下一个应该出队的最优代价结点。

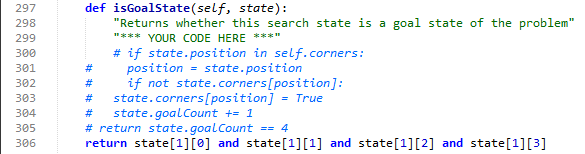
问题5：在角落迷宫的四个角上面有四个豆。构造算法要求找到一条访问所有四个角落的最短的路径

此问题不同于之前的4个问题，此时我们的Pacman不只要吃1个豆子，而是要吃多个豆子，于是，我们需要完成searchAgents.py中的CornersProblem搜索问题，需要重新定义状态，使其能够表示角落是否被访问。此时，我们只需按照老师在注释中给的提示来完成相关的代码即可。修改部分如下：

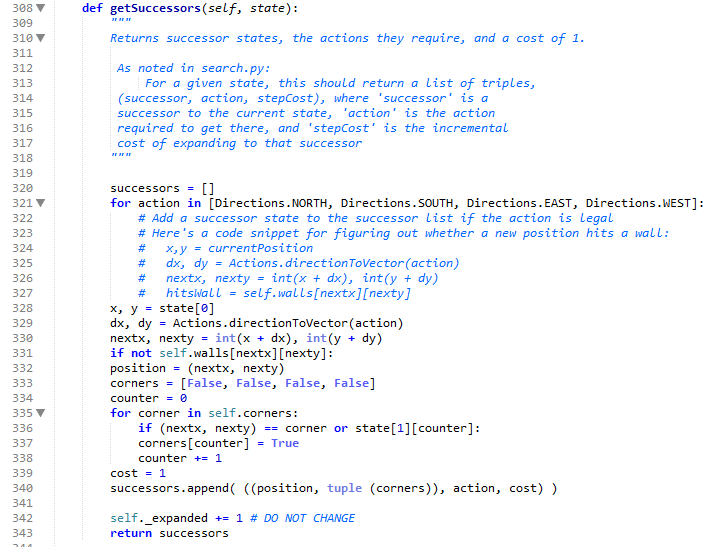
class CornersProblem(search.SearchProblem):



刚开始的StartState返回了开始的位置，以及四个角落中的豆子是否存在的信息（False为豆子还未被吃掉，True为豆子已经被吃掉）



判断是否为终止结点的依据为四个角落的豆子是否都已经被吃掉了（即这四个值是否都为True）



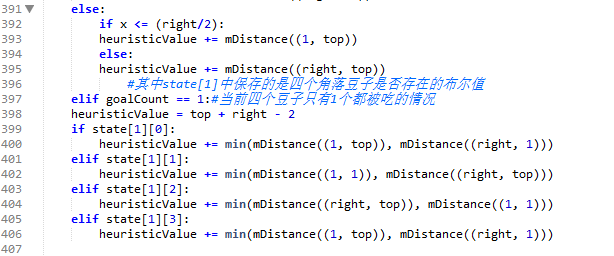
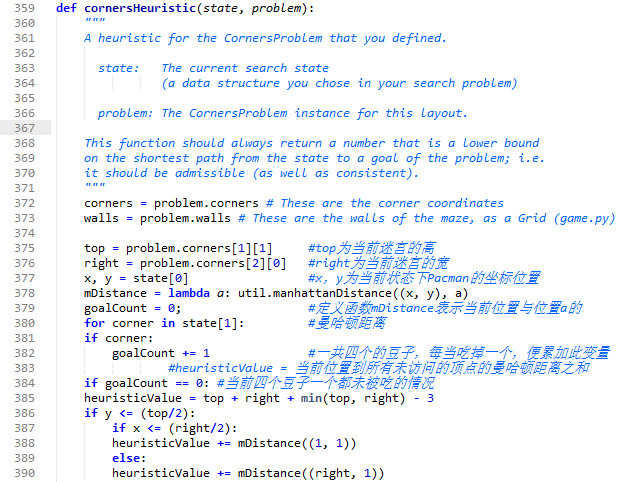
此部分简单，只需按照其对应的伪代码进行书写即可，返回当前状态下一个状态的位置，各个角落豆子是否存在的情况，Pacman的动作行为，以及当前的花费代价的总值。

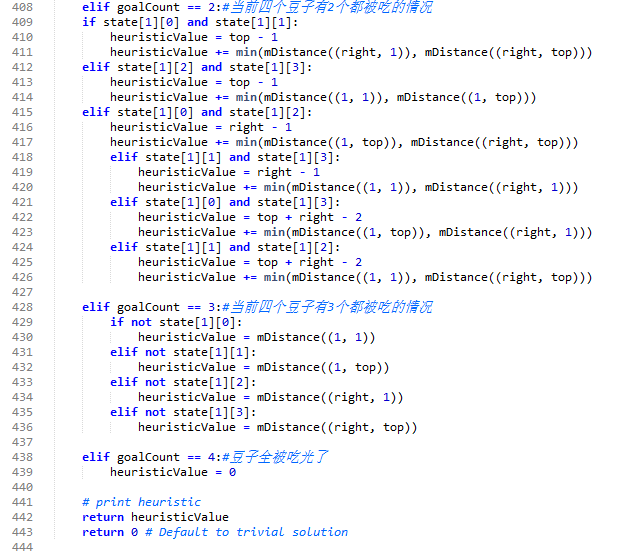
然后我们运用之前的一种搜索策略对此问题进行搜索（题中要求运用bfs）。测试结果请见第三部分，实验结果那部分。

问题6：构建合适的启发函数，找到一条能够访问所有四个角落的最短的路径

本题目为之前启发式搜索策略的拓展，只不过现在的豆子多了，我们需要针对豆子的剩

余情况来分情况讨论决定下一步该怎么走。（请见图片中代码注释）



我们当前的主要任务便是算出启发式函数每个状态下的值，即如代码中的heuristicValue这个值

我们知道 heuristicValue = 当前位置到所有未访问的顶点的曼哈顿距离之和，我们需要分情况讨论，按照当前状态四个角落中豆子的情况分情况讨论，分了五种情况豆子剩余数为：0、1、2、3、4，得到每一种情况下的heuristicValue值，然后利用其搜索策略中的启发式函数，即可找到一条最优路径。

问题7：用尽可能少的步数吃掉所有的豆子（不一定在角落中）。并构建合适的启发函数优化算法

本问题中的豆子又变多了，不只是4个豆子，而且豆子并不一定是在角落中，此时，我们

按照之前设计的A\*算法可以完美解决这个问题，然后，我们要构造一个新的启发式函数来

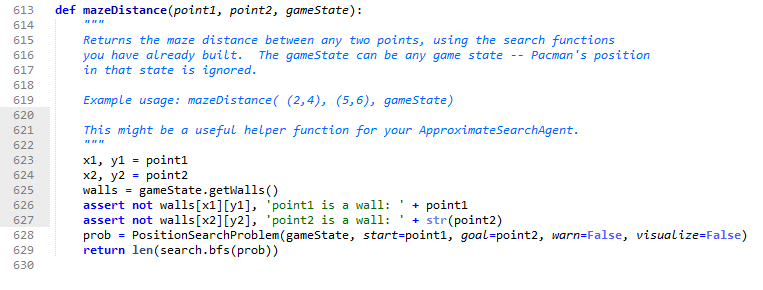
配合我们这个问题进行使用。

首先，我们思考，我们需要用尽可能少的步数来吃掉所有的豆子，那我们这个时候的启发

式函数可以采用当前结点与所剩豆子结点的距离作为线索，发现代码中有可以供我们使的函

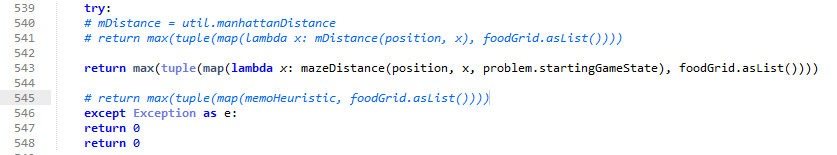
数，即mazeDistance函数。

见下图：



该函数返回两点point1和point2的迷宫内可达的路径长度，指定在特定的gameState下

于是我们可以将这个函数作为启发式函数。



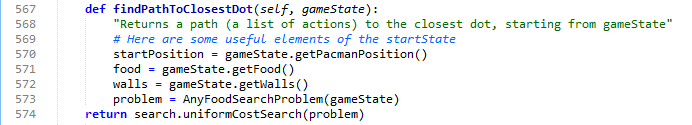
忽略图中的#号后的代码，为编写过程中失败的代码，对于此def foodHeuristic(state, problem):函数，我们返回一个heuristic值作为启发式f（n）的值，我们当前位置与所剩所有食物位置在迷宫距离的长度，最后我们取最大值作为当前的heuristic值。

至此，我们便得到了启发式函数所对应的值，将其作为A\*搜索算法的f()，代入运算即可。

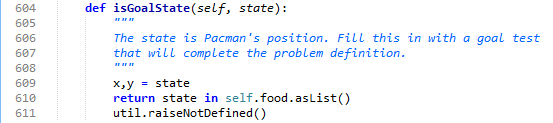
问题8：应用次最优搜索策略，定义一个优先吃最近的豆子函数来实现吃掉所有豆子

基本的代码原程序已为我们调好，我只需增加一小部分即可。

我们主体采用代价一致搜索优化策略



对于判断终止结点，我们采用当前位置是否在剩余食物链food.asList中



**三、实验结果**

问题1：

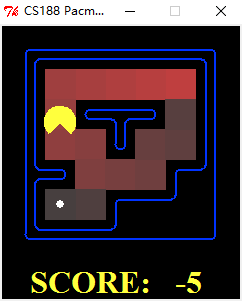
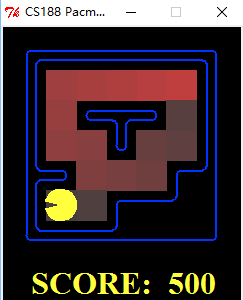
测试指令

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

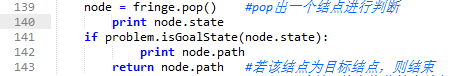
python pacman.py -l bigMaze -z .5 -p SearchAgent

结果截图

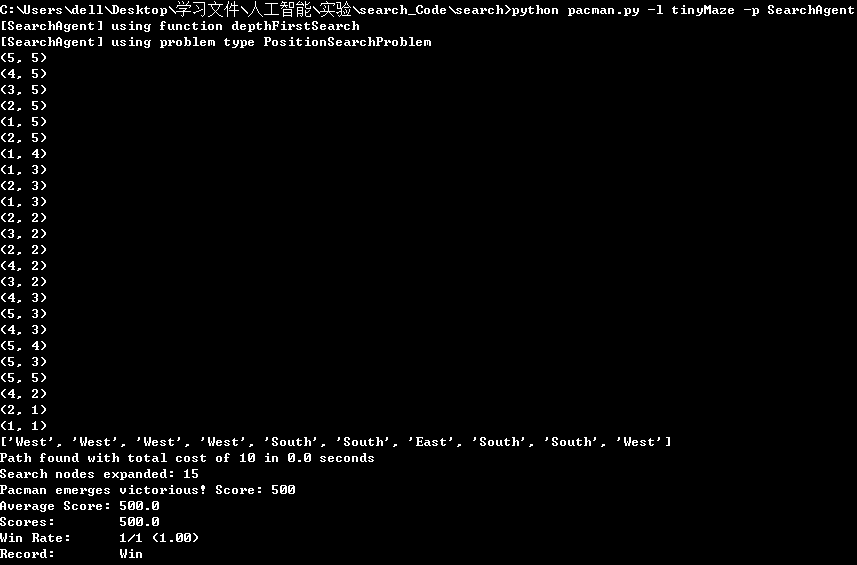
 如下图调试过程中，我添

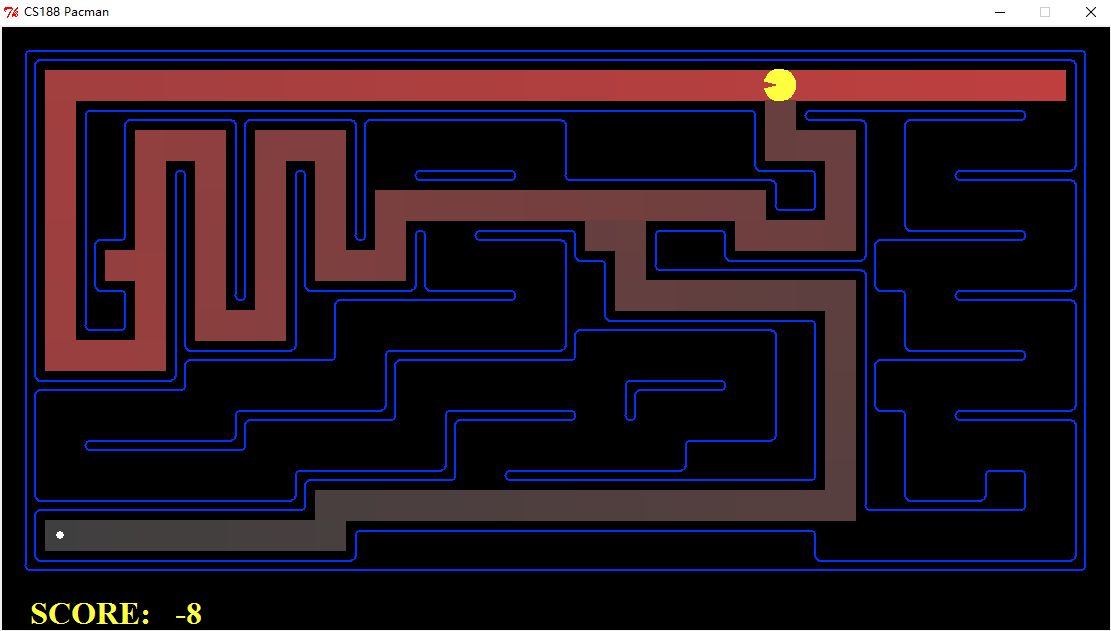
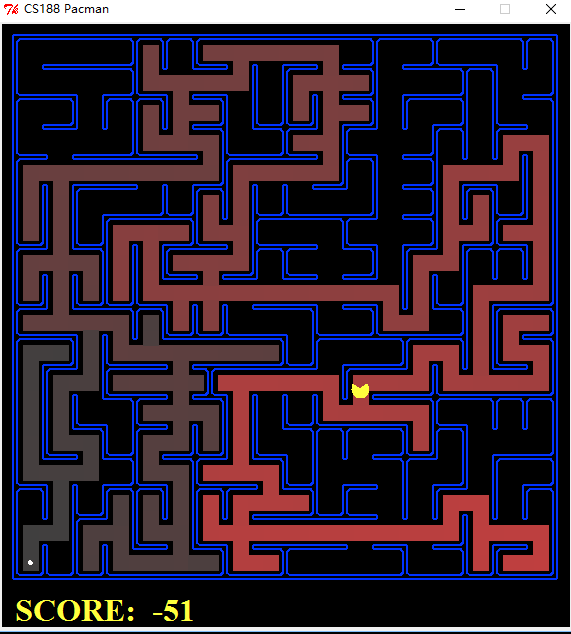
加了两个print来观察吃豆

人寻找豆子（目标结点）时搜寻过程（即DFS过程）我们可以通过最终的打印情况更加清楚地看到该dfs过程，即搜索树的搜索过程。 最终找到目标时，打印吃豆人全部的行进方向，也就是它的路径



最终的调试结果图如下：



左图为medium迷宫 右图为bigsize迷宫





问题2：

测试指令

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

此时仍让采用相同的print来观察调试（如同上个问题）

不再给出GUI界面运行图（跟之前的差不多），因为通过命令行中的结果我们便可以知道是否成功





问题3：

测试指令

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent







问题4：

测试指令：

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

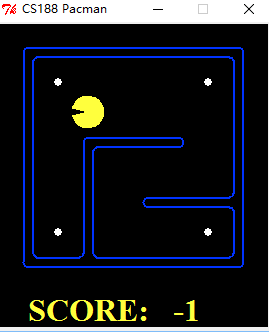


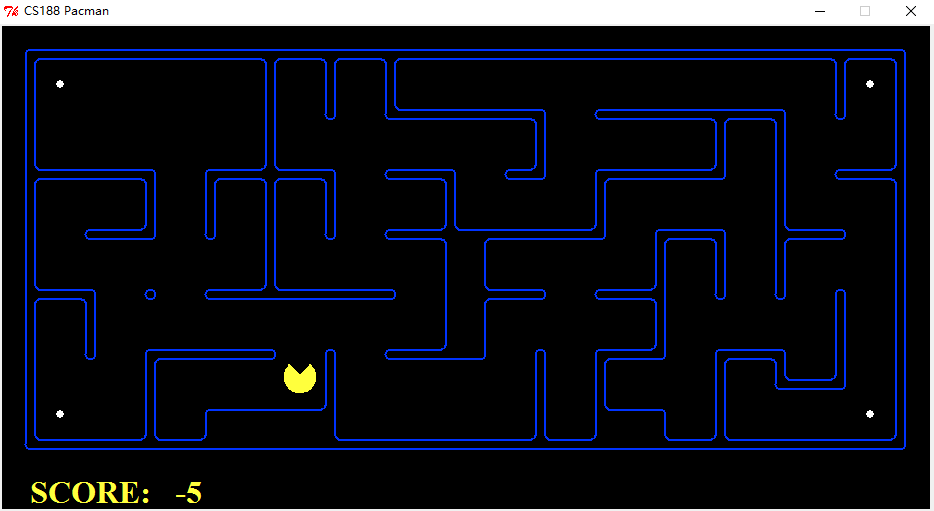
问题5：

测试指令

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

 此时四个角落有四个豆子



调试时，除了打印之前的搜索过程及最终路径，还时时刻刻打印着四个角落豆子的状况





问题6：

测试指令

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

此时调试时候，我们再多打印一个heuristic值，及启发式函数对应的值

调试如下图：

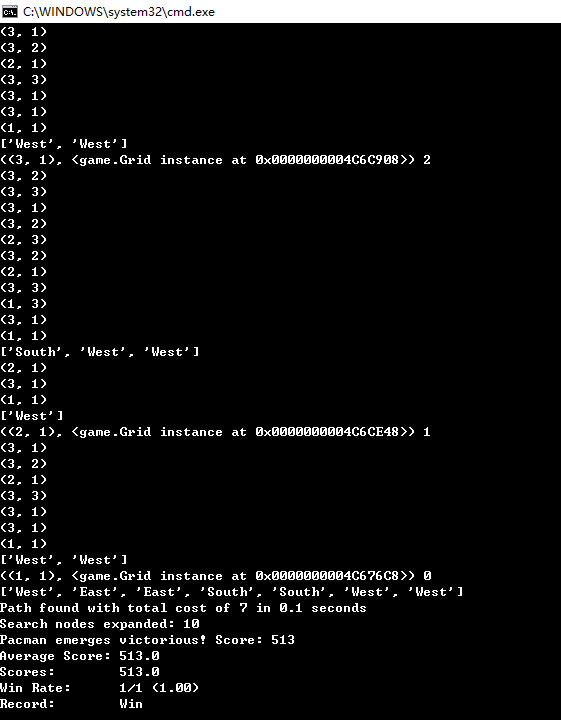


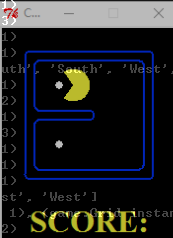
问题7：

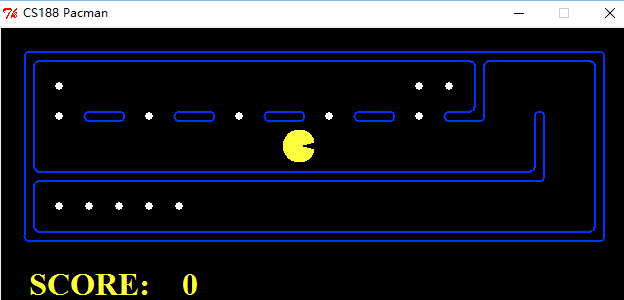
测试指令

python pacman.py -l testSearch -p AStarFoodSearchAgent

python pacman.py -l trickySearch -p AStarFoodSearchAgent

此时豆子更加的多了





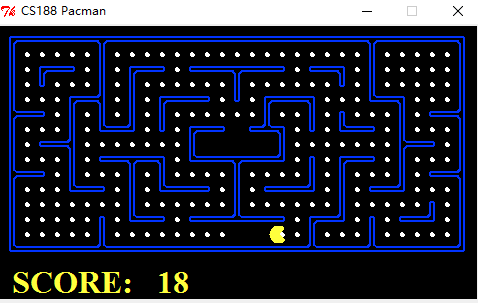


问题8：

测试指令

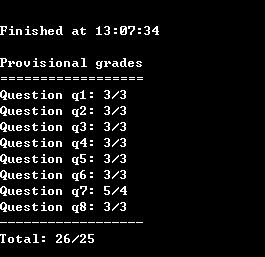
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

此时豆子有很多，我们需要把所有豆子都吃掉



用autograder.py来进行测试

测试结果如下图：



**四、总结讨论**

通过本次实验，我将大一期间学习的python语言又一次捡了起来，复习了python的语法，现在运用起来python更改灵活自如，熟悉掌握了这门编程语言。但最重要的是将多种对树的搜索策略应用到实际问题中来，深入了解了DFS，BFS，A\*等等算法的设计，同时对启发式搜索策略有了自己的思考，关于如何设计效率最高的启发式函数，从而提高我们的搜索效率这种问题有了自己的想法。总之，本次实验提高了我们的逻辑思维和算法设计能力，同时作为人工智能导论课程的一门实验，也加深了我对这门课程的理解。收获颇丰。

**五、附录（本代码含注释）**

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()

class Node():#用于之后各结点的处理

"""

A container storing the current state of a node, the list

of directions that need to be followed from the start state to

get to the current state and the specific problem in which the

node will be used.

"""

def \_\_init\_\_(self, state, path, cost=0, heuristic=0, problem=None):

self.state = state

self.path = path

self.cost = cost

self.heuristic = heuristic

self.problem = problem

def \_\_str\_\_(self):

string = "Current State: "

string += \_\_str\_\_(self.state)

string += "\n"

string == "Path: " + self.path + "\n"

return string

def getSuccessors(self, heuristicFunction=None):

children = []

for successor in self.problem.getSuccessors(self.state):

state = successor[0]

path = list(self.path)

path.append(successor[1])

cost = self.cost + successor[2]

if heuristicFunction:

heuristic = heuristicFunction(state, self.problem)

else:

heuristic = 0

node = Node(state, path, cost, heuristic, self.problem)

children.append(node)

return children

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):

"""

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())

"""

#此部分按照老师所给出的伪代码进行设计

closed = set() #close表用于保存已经遍历过的点

fringe = util.Stack() #此部分相当于老师上课所讲的open表，用栈来表示

startNode = Node(problem.getStartState(), [], 0, 0, problem)

fringe.push(startNode) #首先将开始结点放入open表，即压栈fringe

while True:

if fringe.isEmpty(): #若栈空，则说明已经遍历了所有的结点

return False

node = fringe.pop() #pop出一个结点进行判断

#print node.state

if problem.isGoalState(node.state):

#print node.path

return node.path #若该结点为目标结点，则结束

if node.state not in closed: #否则，首先将此结点放入close表中，标志为

closed.add(node.state) #已经遍历过，再将此结点的孩子结点压入栈，

for childNode in node.getSuccessors():#进入下一层循环

fringe.push(childNode)

def breadthFirstSearch(problem):

"""

Search the shallowest nodes in the search tree first.

"""

closed = set() #close表用于保存已经遍历过的点

fringe = util.Queue() #此部分相当于老师上课所讲的open表，用队列来表示

startNode = Node(problem.getStartState(), [], 0, 0, problem)

fringe.push(startNode) #首先将开始结点放入open表，即入队fringe

while True:

if fringe.isEmpty(): #若队列为空，则说明已经遍历了所有的结点

return False

node = fringe.pop() #从队列中出队出一个结点进行判断

#print node.state

if problem.isGoalState(node.state):

#print node.path

return node.path #若该结点为目标结点，则结束

if node.state not in closed: #否则，首先将此结点放入close表中，标志为

closed.add(node.state) #已经遍历过，再将此结点的孩子结点入队列，

for childNode in node.getSuccessors():#进入下一层循环

fringe.push(childNode)

def uniformCostSearch(problem):

"Search the node of least total cost first. "

closed = set() #close表用于保存已经遍历过的点

fringe = util.PriorityQueue()

#此部分相当于老师上课所讲的open表，用优先队列来表示

startNode = Node(problem.getStartState(), [], 0, 0, problem)

fringe.push(startNode, startNode.cost)

while True: #首先将开始结点和其对应的代价值cost，入队fringe

if fringe.isEmpty(): #若队列为空，则说明已经遍历了所有的结点

return False

node = fringe.pop() #从队列中出队出一个结点进行判断

#print node.state

if problem.isGoalState(node.state):

#print node.path

return node.path #若该结点为目标结点，则结束

if node.state not in closed: #否则，首先将此结点放入close表中，标志为

closed.add(node.state) #已经遍历过将此结点的孩子结点以及其对应cost

for childNode in node.getSuccessors():#入队，进入下一层循环

fringe.push(childNode, childNode.cost)

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."

closed = set() #close表用于保存已经遍历过的点

fringe = util.PriorityQueue()

#此部分相当于老师上课所讲的open表，用优先队列来表示

startNode = Node(problem.getStartState(), [], 0, 0, problem)

fringe.push(startNode, startNode.cost + startNode.heuristic)

while True: #首先将开始结点和其对应的cost与h(x)即曼哈顿距离的和，

if fringe.isEmpty(): #入队fringe

return False #若队列为空，则说明已经遍历了所有的结点

node = fringe.pop() #从队列中出队出一个结点进行判断

#print node.state,node.heuristic

if problem.isGoalState(node.state):

#print node.path

return node.path

if node.state not in closed: #否则，首先将此结点放入close表中，标志为已经

closed.add(node.state) #遍历过将此结点的孩子结点以及其对应实际代价

for childNode in node.getSuccessors(heuristic):#入队，进入下一层循环

fringe.push(childNode, childNode.cost + childNode.heuristic)

# Abbreviations

bfs = breadthFirstSearch

dfs = depthFirstSearch

astar = aStarSearch

ucs = uniformCostSearch

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 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 # 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)"

# corners = {}

# for corner in self.corners:

# corners[corner] = False

# return CornersProblem.State(self.startingPosition, corners)

return (self.startingPosition, (False, False, False, False))

def isGoalState(self, state):

"Returns whether this search state is a goal state of the problem"

"\*\*\* YOUR CODE HERE \*\*\*"

# if state.position in self.corners:

# position = state.position

# if not state.corners[position]:

# state.corners[position] = True

# state.goalCount += 1

# return state.goalCount == 4

return state[1][0] and state[1][1] and state[1][2] and state[1][3]

def getSuccessors(self, state):

"""

Returns successor states, the actions they require, and a cost of 1.

As noted in search.py:

For a given state, this should return a list of triples,

(successor, action, stepCost), where 'successor' is a

successor to the current state, 'action' is the action

required to get there, and 'stepCost' is the incremental

cost of expanding to that successor

"""

successors = []

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]

x, y = state[0]

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:

position = (nextx, nexty)

corners = [False, False, False, False]

counter = 0

for corner in self.corners:

if (nextx, nexty) == corner or state[1][counter]:

corners[counter] = True

counter += 1

cost = 1

successors.append( ((position, tuple (corners)), action, cost) )

self.\_expanded += 1 # DO NOT CHANGE

return successors

def getCostOfActions(self, actions):

"""

Returns the cost of a particular sequence of actions. If those actions

include an illegal move, return 999999. This is implemented for you.

"""

if actions == None: return 999999

x,y= self.startingPosition

for action in actions:

dx, dy = Actions.directionToVector(action)

x, y = int(x + dx), int(y + dy)

if self.walls[x][y]: return 999999

return len(actions)

def cornersHeuristic(state, problem):

"""

A heuristic for the CornersProblem that you defined.

state: The current search state

(a data structure you chose in your search problem)

problem: The CornersProblem instance for this layout.

This function should always return a number that is a lower bound

on the shortest path from the state to a goal of the problem; i.e.

it should be admissible (as well as consistent).

"""

corners = problem.corners # These are the corner coordinates

walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

top = problem.corners[1][1] #top为当前迷宫的高

right = problem.corners[2][0] #right为当前迷宫的宽

x, y = state[0] #x，y为当前状态下Pacman的坐标位置

mDistance = lambda a: util.manhattanDistance((x, y), a)

goalCount = 0; #定义函数mDistance表示当前位置与位置a的

for corner in state[1]: #曼哈顿距离

if corner:

goalCount += 1 #一共四个的豆子，每当吃掉一个，便累加此变量

#heuristicValue = 当前位置到所有未访问的顶点的曼哈顿距离之和

if goalCount == 0: #当前四个豆子一个都未被吃的情况

heuristicValue = top + right + min(top, right) - 3

if y <= (top/2):

if x <= (right/2):

heuristicValue += mDistance((1, 1))

else:

heuristicValue += mDistance((right, 1))

else:

if x <= (right/2):

heuristicValue += mDistance((1, top))

else:

heuristicValue += mDistance((right, top))

#其中state[1]中保存的是四个角落豆子是否存在的布尔值

elif goalCount == 1:#当前四个豆子只有1个都被吃的情况

heuristicValue = top + right - 2

if state[1][0]:

heuristicValue += min(mDistance((1, top)), mDistance((right, 1)))

elif state[1][1]:

heuristicValue += min(mDistance((1, 1)), mDistance((right, top)))

elif state[1][2]:

heuristicValue += min(mDistance((right, top)), mDistance((1, 1)))

elif state[1][3]:

heuristicValue += min(mDistance((1, top)), mDistance((right, 1)))

elif goalCount == 2:#当前四个豆子有2个都被吃的情况

if state[1][0] and state[1][1]:

heuristicValue = top - 1

heuristicValue += min(mDistance((right, 1)), mDistance((right, top)))

elif state[1][2] and state[1][3]:

heuristicValue = top - 1

heuristicValue += min(mDistance((1, 1)), mDistance((1, top)))

elif state[1][0] and state[1][2]:

heuristicValue = right - 1

heuristicValue += min(mDistance((1, top)), mDistance((right, top)))

elif state[1][1] and state[1][3]:

heuristicValue = right - 1

heuristicValue += min(mDistance((1, 1)), mDistance((right, 1)))

elif state[1][0] and state[1][3]:

heuristicValue = top + right - 2

heuristicValue += min(mDistance((1, top)), mDistance((right, 1)))

elif state[1][1] and state[1][2]:

heuristicValue = top + right - 2

heuristicValue += min(mDistance((1, 1)), mDistance((right, top)))

elif goalCount == 3:#当前四个豆子有3个都被吃的情况

if not state[1][0]:

heuristicValue = mDistance((1, 1))

elif not state[1][1]:

heuristicValue = mDistance((1, top))

elif not state[1][2]:

heuristicValue = mDistance((right, 1))

elif not state[1][3]:

heuristicValue = mDistance((right, top))

elif goalCount == 4:#豆子全被吃光了

heuristicValue = 0

# print heuristic

return heuristicValue

return 0 # Default to trivial solution

class AStarCornersAgent(SearchAgent):

"A SearchAgent for FoodSearchProblem using A\* and your foodHeuristic"

def \_\_init\_\_(self):

self.searchFunction = lambda prob: search.aStarSearch(prob, cornersHeuristic)

self.searchType = CornersProblem

class FoodSearchProblem:

"""

A search problem associated with finding the a path that collects all of the

food (dots) in a Pacman game.

A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where

pacmanPosition: a tuple (x,y) of integers specifying Pacman's position

foodGrid: a Grid (see game.py) of either True or False, specifying remaining food

"""

def \_\_init\_\_(self, startingGameState):

self.start = (startingGameState.getPacmanPosition(), startingGameState.getFood())

self.walls = startingGameState.getWalls()

self.startingGameState = startingGameState

self.\_expanded = 0 # DO NOT CHANGE

self.heuristicInfo = {} # A dictionary for the heuristic to store information

def getStartState(self):

return self.start

def isGoalState(self, state):

return state[1].count() == 0

def getSuccessors(self, state):

"Returns successor states, the actions they require, and a cost of 1."

successors = []

self.\_expanded += 1 # DO NOT CHANGE

for direction in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

x,y = state[0]

dx, dy = Actions.directionToVector(direction)

nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:

nextFood = state[1].copy()

nextFood[nextx][nexty] = False

successors.append( ( ((nextx, nexty), nextFood), direction, 1) )

return successors

def getCostOfActions(self, actions):

"""Returns the cost of a particular sequence of actions. If those actions

include an illegal move, return 999999"""

x,y= self.getStartState()[0]

cost = 0

for action in actions:

# figure out the next state and see whether it's legal

dx, dy = Actions.directionToVector(action)

x, y = int(x + dx), int(y + dy)

if self.walls[x][y]:

return 999999

cost += 1

return cost

class AStarFoodSearchAgent(SearchAgent):

"A SearchAgent for FoodSearchProblem using A\* and your foodHeuristic"

def \_\_init\_\_(self):

self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)

self.searchType = FoodSearchProblem

def foodHeuristic(state, problem):

"""

Your heuristic for the FoodSearchProblem goes here.

This heuristic must be consistent to ensure correctness. First, try to come up

with an admissible heuristic; almost all admissible heuristics will be consistent

as well.

If using A\* ever finds a solution that is worse uniform cost search finds,

your heuristic is \*not\* consistent, and probably not admissible! On the other hand,

inadmissible or inconsistent heuristics may find optimal solutions, so be careful.

The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a

Grid (see game.py) of either True or False. You can call foodGrid.asList()

to get a list of food coordinates instead.

If you want access to info like walls, capsules, etc., you can query the problem.

For example, problem.walls gives you a Grid of where the walls are.

If you want to \*store\* information to be reused in other calls to the heuristic,

there is a dictionary called problem.heuristicInfo that you can use. For example,

if you only want to count the walls once and store that value, try:

problem.heuristicInfo['wallCount'] = problem.walls.count()

Subsequent calls to this heuristic can access problem.heuristicInfo['wallCount']

"""

position, foodGrid = state

# return len(foodGrid.asList())

# def memoHeuristic(x):

# if (position, x) not in problem.heuristicInfo:

# problem.heuristicInfo[(position, x)] = mazeDistance(position, x, problem.startingGameState)

# return problem.heuristicInfo[(position, x)]

try:

# mDistance = util.manhattanDistance

# return max(tuple(map(lambda x: mDistance(position, x), foodGrid.asList())))

return max(tuple(map(lambda x: mazeDistance(position, x, problem.startingGameState), foodGrid.asList())))

# return max(tuple(map(memoHeuristic, foodGrid.asList())))

except Exception as e:

return 0

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.uniformCostSearch(problem)

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

return state in self.food.asList()

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: ' + 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))

合作：张茗帅 张韬庚 董浩洋 袁怡轮 李宁 王智 周昊天