

**Harbin Institute of Technology**

**人工智能导论实验**

**实 验 报 告**

班 级： 1403104

姓 名： 万晓珑

学 号： 1140310414

小组成员： 陆鑫、赵世琦、李元昊

司傲然、万晓珑

目录

[第1部分 简 介 2](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160556)

[第2部分 方 法 3](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160557)

[2.1 前四个基本问题 3](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160558)

[2.2 找到所有角落和角落问题的启发式方法 4](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160559)

[2.3 吃掉所有的豆子和次最优搜索 5](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160560)

[第3部分 实验结果 7](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160561)

[3.1 深度优先搜索 7](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160562)

[3.2 宽度优先搜索 8](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160563)

[3.3 一致代价搜索 9](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160564)

[3.4 A\*算法 10](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160565)

[3.5 找到所有的角落 11](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160566)

[3.6 角落问题（启发式） 11](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160567)

[3.7 吃掉所有的豆子 12](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160568)

[3.8 次最优搜索 12](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160569)

[第4部分 总结与讨论 13](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160570)

[第5部分 附 录 14](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160571)

[5.1 Search.py 14](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160572)

[5.2 SerchAgents.py 20](file:///C:\Users\mayuk\Desktop\人工智能导论实验报告v2.docx#_Toc471160573)

# 第1部分 简 介

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

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

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

在这个实验的实现过程中需要编辑两个关键文件“[search.py](https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/docs/search.html)”和“[searchAgents.py](https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/docs/searchAgents.html)”，“[search.py](https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/docs/search.html)”中需要补充填写四个基本搜索算法，“[searchAgents.py](https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/docs/searchAgents.html)”中需要补充填写所有启发式算法的基本内容。除此之外还有三个比较重要的文件：“pacman.py”、“game.py”和“util.py”。“pacman.py”是游戏的主文件，描述了一个Pacman GameState类型。“game.py”说明了Pacman世界如何工作的逻辑并且描述了几种支持类型，如AgentState，Agent，Direction和Grid。“util.py”是对我们来说最重要的文件，包括了实现搜索算法最有用的几种数据结构。

通过这个实验，我们可以更好的理解各种基本搜索算法，并且通过可视化的方法可以更加有成就感，激发学习兴趣。在人工智能的领域继续研究下去。

# 第2部分 方 法

## 前四个基本问题

**深度优先搜索**：是一种用于遍历或搜索[树](https://zh.wikipedia.org/wiki/%E6%A0%91_(%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84))或[图](https://zh.wikipedia.org/w/index.php?title=%E5%9B%BE_(%E6%8A%BD%E8%B1%A1%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B)&action=edit&redlink=1)的[算法](https://zh.wikipedia.org/wiki/%E7%AE%97%E6%B3%95)。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

**宽度优先搜索**：深度优先搜索正好相反，宽度优先搜索一层一层地探索空间。只有给定层上不再存在要探索的状态时算法才转移到下一个更深层次。同样属于盲目搜索。

**一致代价搜索**：一致代价搜索是宽度优先搜索的改进版本，其采用优先队列作为数据结构，抛弃宽度优先算法的普通队列，采用优先队列的方式为每个点存一份最短累计代价，路径上所有点都是最小代价。

**A\*算法**：是一种典型的启发式搜索算法，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" \t "_blank)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" \t "_blank)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。

在刚开始实现这几种算法的时候，难免有些茫然，用来熟悉整个程序的时间比较多，摸清整个程序大体内容，5个重要的py程序之间的关联。在摸清这些内容后，我们很快就完成了深度和广度优先搜索的代码。这是我们在数据结构的课程中实现了很多遍的算法，也是在这几个问题中最简单的两个问题。参考下图1的伪代码，我们做的顺风顺水。

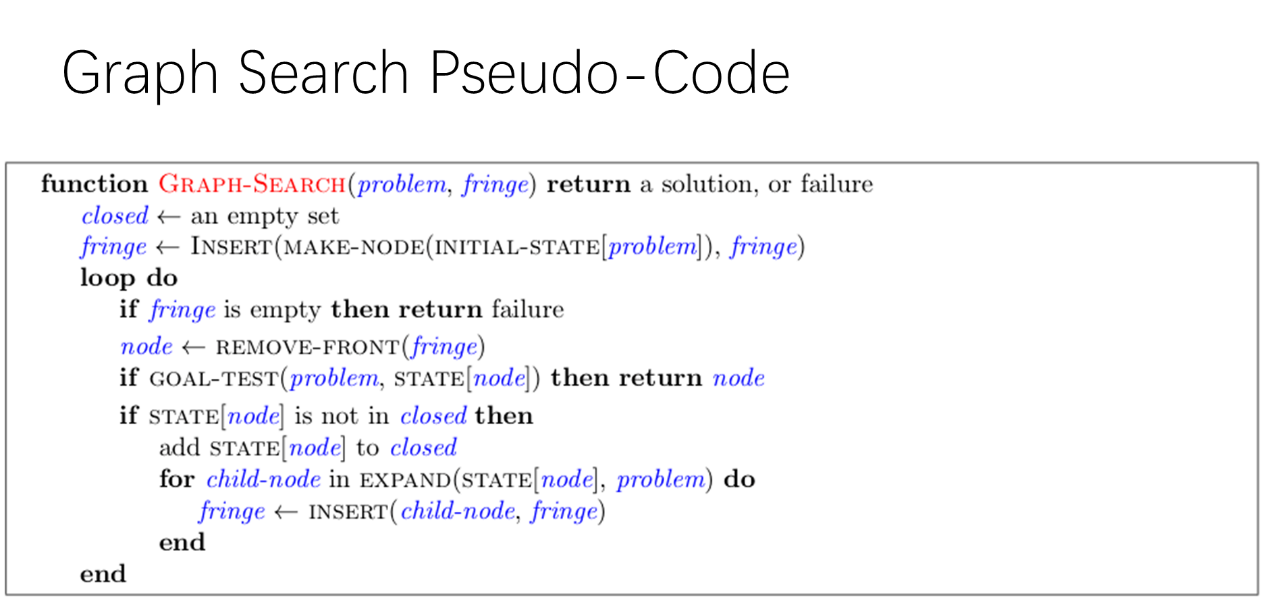


图1 图搜索算法伪代码

在实现一致代价搜索和A\*算法的时候，出了一点小问题。组内成员对于算法的理解有一定差距，并且没有熟悉代码的整体结构，导致在实现算法的过程中出现了一些小插曲。好在后来，我们又重新熟悉重要的py文件，从中找到了思路。

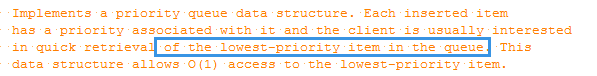


图2 思路来源 util.py 中优先队列注释

最重要的提示就在util.py 中优先队列注释部分，最终实现了一致代价算法A\*算法。不过现在代码实现通用性较差，后期我们会继续优化改进。

## 找到所有角落和角落问题的启发式方法

这个问题一开始的时候我们也有一点摸不到头脑，但是通过阅读代码中给出的提示逐步猜测和尝试，最终找到了问题5的解决办法。

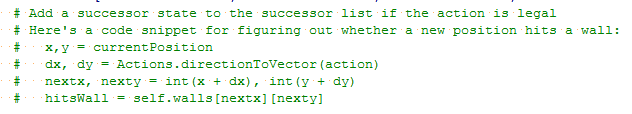


图3 CornersProblem需要实现部分的提示

问题五的关键在于计算出下一个合法的节点位置和角落状态信息。问题六结合了A\*算法，由2.1节可知，我们在这里需要构建一个合理的启发式函数，也就是说要计算一个合理的h(n)函数。结合项目代码中给出的注释提示和对于A\*算法h(n)函数的理解，我们找到了此处的思路。

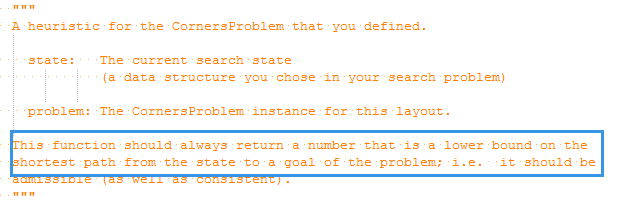


图4 cornersHeuristic提示信息

通过计算当前状态节点与周围节点曼哈顿最大值，进而就可以找到满足h(n)<=h\*(n)的h(n)函数值。

## 吃掉所有的豆子和次最优搜索

解决了问题6，那么问题7的基本思路也算是有了，就是通过寻找当前状态节点和曼哈顿距离最远的食物点的值。当然这样简单的想法是绝对不可能是问题的正确性解，因为这样偏差太大。

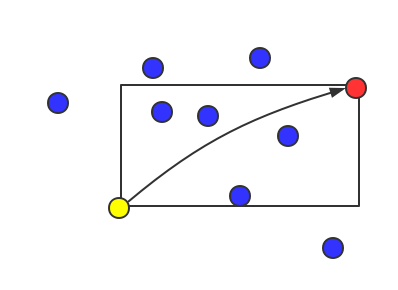


图5 从状态节点到曼哈顿距离最远节点

经过讨论，我们将问题的解可以这样想从状态节点到曼哈顿距离最大节点的矩形范围内的所有节点都是有可能访问到的，但是矩形框外面得节点是访问不到的。所以最大距离应该在加上一个访问外部节点，如果有外部节点，那么至少需要访问一个外部节点这是显而易见的。这样的话就会产生下面的两种情况。

情况一：先访问最远外部节点。那么最小距离至少需要加上外部最远节点和矩形框的曼哈顿距离才可能等于实际距离。

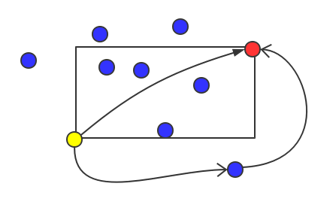


图6 访问节点第一种情况

情况二：先访问曼哈顿距离最远点。这种情况下显然原有最远点的曼哈顿距离加上外部最远节点和矩形框的曼哈顿距离显然小于实际问题的长度。

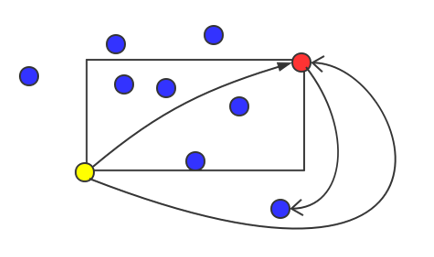


图7 访问节点第二种情况

经过检验，这种思想可以将扩展节点数降到8000以下，对我们来说已经是一个非常好的结果。

第八个问题我们调用了宽度优先算法解决。显然经过上面的测试，宽度优先算法更适合这个问题。

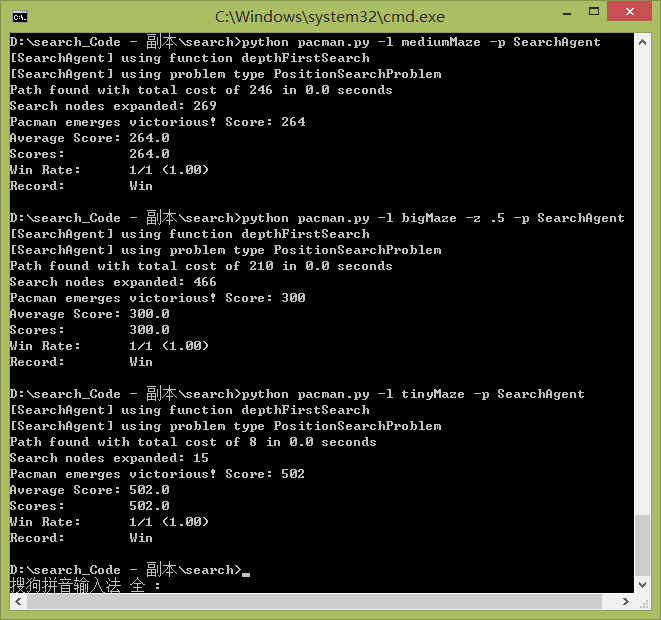
# 第3部分 实验结果

## 深度优先搜索

**python pacman.py -l tinyMaze -p SearchAgent**

**python pacman.py -l mediumMaze -p SearchAgent**

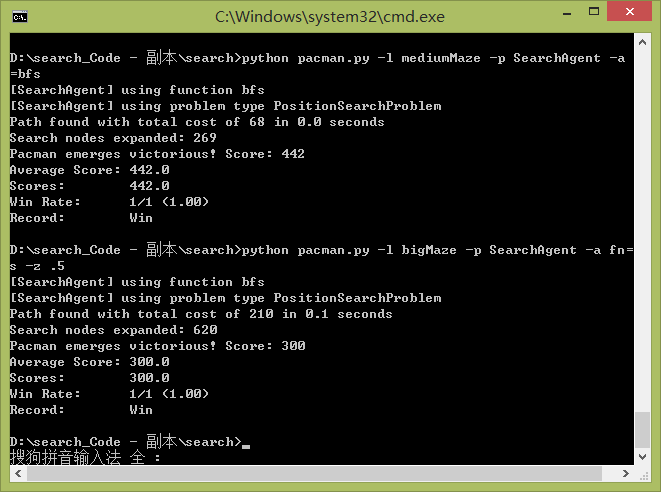
**python pacman.py -l bigMaze -z .5 -p SearchAgent**



## 宽度优先搜索

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

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

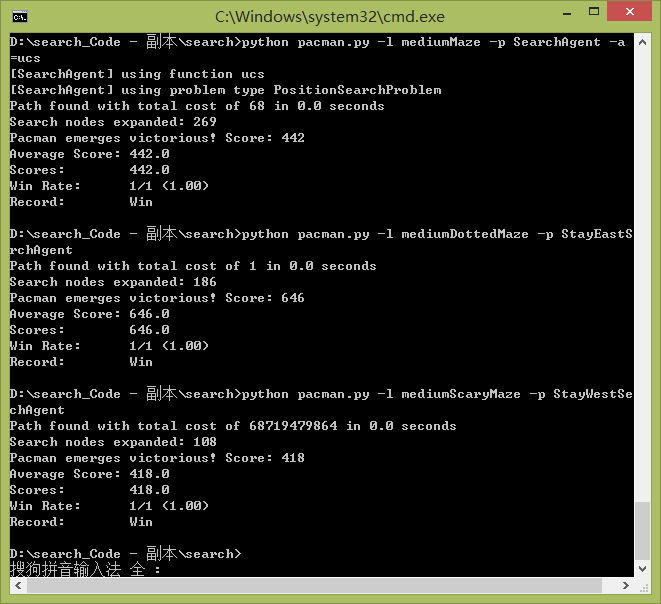


## 一致代价搜索

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

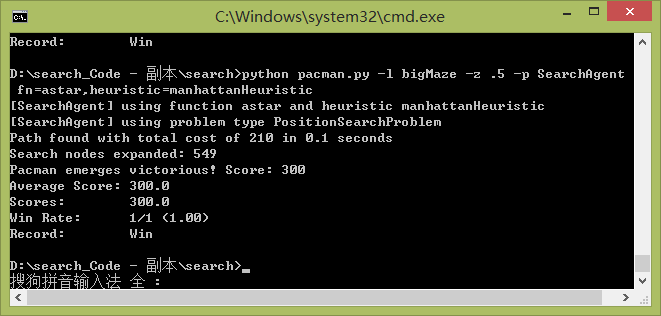
**python pacman.py -l mediumDottedMaze -p StayEastSearchAgent**

**python pacman.py -l mediumScaryMaze -p StayWestSearchAgent**



## A\*算法

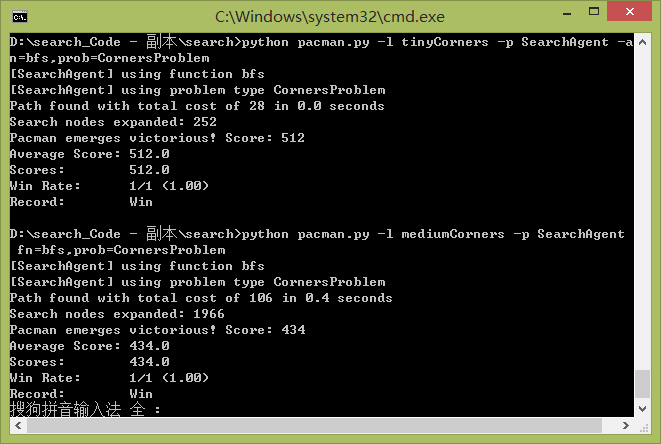
**python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic**



## 找到所有的角落

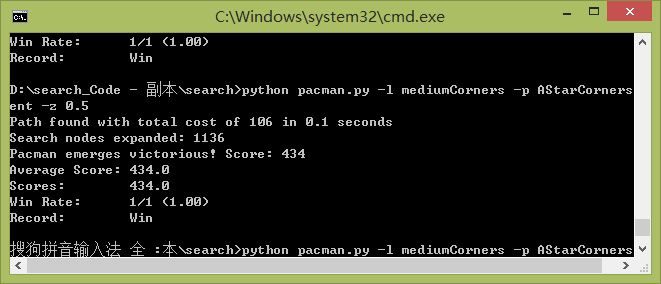
**python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem**

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



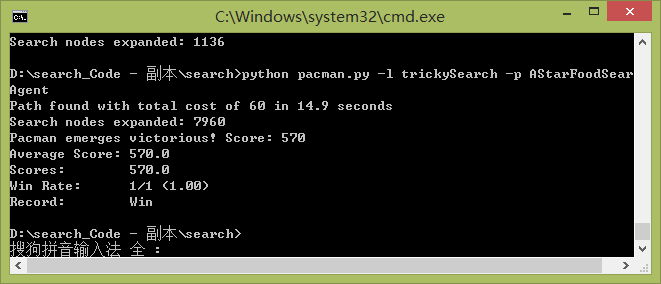
## 角落问题（启发式）

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



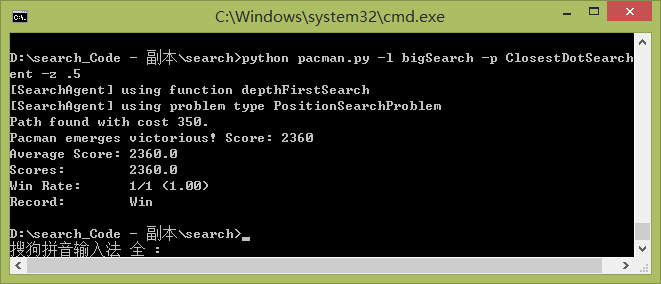
## 吃掉所有的豆子

**python pacman.py -l trickySearch -p AStarFoodSearchAgent**



## 次最优搜索

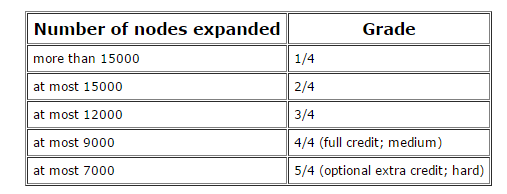
**python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5**



# 第4部分 总结与讨论

在这次实验中我和小组成员一起探究了有关吃豆人项目的8个搜索问题，深入的掌握和理解了包括深度优先、广度优先、一致代价和A\*在内的四个基本搜索算法。并且逐步探索了几个启发式算法，在角落问题的启发式算法中，我们经过了一轮又一轮的讨论与校验，终于得出了一个可行的启发式算法。

最难但是有最有趣的是问题7，根据问题六的思考，其实很容易就可以得到一个问题七的可行解，但是在项目的网站中我们居然看到了下面的评分规则，也就是说还有更优解的存在，于是我们又开始思考，但是很不幸，直到截止时间我们也没有找到一种更好的方式，将扩展的节点压缩至7000个以内，这确实是一件遗憾的事情，不过实验结束之后我们将继续探索这个有意思的游戏。



# 第5部分 附 录

## 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):

#递归函数

def dfsFunction(problem, currentState, dfsPath, dfsVisited):

#判断/设置访问标志

if currentState in dfsVisited:

return False

else:

dfsVisited.append(currentState)

#判断终点

if problem.isGoalState(currentState):

return True

#遍历后继，填/抹动作点

for nextInfo in problem.getSuccessors(currentState):

dfsPath.append(nextInfo[1])

if dfsFunction(problem, nextInfo[0], dfsPath, dfsVisited):

return True

else:

dfsPath.pop()

#拜拜

return False

#深度算法

dfsPath = []

dfsVisited = []

dfsFunction(problem, problem.getStartState(), dfsPath, dfsVisited)

return dfsPath

def breadthFirstSearch(problem):

#引入队列

from util import Queue

queue = Queue()

#初态入队

queue.push(problem.getStartState())

#广度算法

bfsPath = []

bfsRouteMap = {} #(5,5):((4,0),"east") #反向路由关系,键是子节点,值是父节点和动作

bfsVisited = [problem.getStartState()]

while not queue.isEmpty():

#出队一个点

currentState = queue.pop()

#判断目标点

if problem.isGoalState(currentState):

pathState = currentState;

while bfsRouteMap.has\_key(pathState):

bfsPath.append(bfsRouteMap[pathState][1])

pathState = bfsRouteMap[pathState][0]

bfsPath=list(reversed(bfsPath))

break

#遍历后继，入队并填充路由信息

for nextInfo in problem.getSuccessors(currentState):

nextStatus = nextInfo[0];

action = nextInfo[1];

if not nextStatus in bfsVisited:

bfsVisited.append(nextStatus)

bfsRouteMap[nextStatus] = (currentState, action)

queue.push(nextStatus)

return bfsPath

#基本想法: 为每个点存一份最短累计代价，路径上所有点都最短，这是贪心--是Dijkstra算法

#当贪心选到终结点时(出队)，就一定是最小代价了，是贪心限制的

#关于顶点数动态变化：不重要，因为是广度，能保证经由后续点的路径仍比现在长，即使是新节点也是现有节点的子节点

def uniformCostSearch(problem):

#引入优先队列

from util import PriorityQueue

priQueue = PriorityQueue()

#初态入队

priQueue.push(problem.getStartState(), 0)

#广度算法

uniCostPath = []

uniCostRouteMap = {} #(5,5):((4,0),"east") #反向路由关系,键是子节点,值是父节点和动作

uniCostValue = {problem.getStartState():0} #每个点累计代价

uniCostVisited = [problem.getStartState()]

while not priQueue.isEmpty():

#出队一个点

currentState = priQueue.pop()

#判断目标点

if problem.isGoalState(currentState):

pathState = currentState;

while uniCostRouteMap.has\_key(pathState):

uniCostPath.append(uniCostRouteMap[pathState][1])

pathState = uniCostRouteMap[pathState][0]

uniCostPath=list(reversed(uniCostPath))

break

#遍历后继，入队并填充路由信息，以及累计代价

for nextInfo in problem.getSuccessors(currentState):

nextStatus = nextInfo[0];

action = nextInfo[1];

cost = nextInfo[2];

#未放入队列的点

if not nextStatus in uniCostVisited:

uniCostVisited.append(nextStatus)

uniCostRouteMap[nextStatus] = (currentState, action)

uniCostValue[nextStatus] = uniCostValue[currentState] + cost; #保存新节点累计代价

priQueue.push(nextStatus, uniCostValue[nextStatus])

#放入过队列的点

else:

#代价更小，换爹

if uniCostValue[nextStatus]>uniCostValue[currentState]+cost:

uniCostRouteMap[nextStatus] = (currentState, action)

uniCostValue[nextStatus] = uniCostValue[currentState] + cost; #更新节点的累计代价

priQueue.update(nextStatus, uniCostValue[nextStatus]) #update果然用上了

return uniCostPath

def nullHeuristic(state, problem=None):

return 0

def aStarSearch(problem, heuristic=nullHeuristic):

#引入优先队列

from util import PriorityQueue

priQueue = PriorityQueue()

#初态入队

priQueue.push(problem.getStartState(), 0 + heuristic(problem.getStartState(), problem))

#广度算法

aStarPath = []

aStarRouteMap = {} #(5,5):((4,0),"east") #反向路由关系,键是子节点,值是父节点和动作

aStarValue = {problem.getStartState(): 0} #每个点累计代价

aStarFn = {problem.getStartState(): 0 + heuristic(problem.getStartState(), problem)} #每个点f(n)

aStarVisited = [problem.getStartState()]

while not priQueue.isEmpty():

#出队一个点

currentState = priQueue.pop()

#判断目标点

if problem.isGoalState(currentState):

pathState = currentState;

while aStarRouteMap.has\_key(pathState):

aStarPath.append(aStarRouteMap[pathState][1])

pathState = aStarRouteMap[pathState][0]

aStarPath=list(reversed(aStarPath))

break

#遍历后继，入队并填充路由信息，以及累计代价

for nextInfo in problem.getSuccessors(currentState):

nextStatus = nextInfo[0];

action = nextInfo[1];

cost = nextInfo[2];

#未放入队列的点

if not nextStatus in aStarVisited:

aStarVisited.append(nextStatus)

aStarRouteMap[nextStatus] = (currentState, action)

aStarValue[nextStatus] = aStarValue[currentState] + cost; #计算累计代价

aStarFn[nextStatus] = aStarValue[nextStatus] + heuristic(nextStatus, problem) #计算f(n)

priQueue.push(nextStatus, aStarFn[nextStatus])

#放入过队列的点

else:

#f(n)更小，换爹

if aStarFn[nextStatus]>aStarValue[currentState]+cost+heuristic(nextStatus, problem):

aStarRouteMap[nextStatus] = (currentState, action)

aStarValue[nextStatus] = aStarValue[currentState] + cost; #更新累计代价

aStarFn[nextStatus] = aStarValue[nextStatus] + heuristic(nextStatus, problem) #更新f(n)

priQueue.update(nextStatus, aStarFn[nextStatus])

return aStarPath

# Abbreviations

bfs = breadthFirstSearch

dfs = depthFirstSearch

astar = aStarSearch

ucs = uniformCostSearch

## SerchAgents.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

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

def getStartState(self):

"""

Returns the start state (in your state space, not the full Pacman state

space)

"""

#真实位置给计算次态用，后面四个顺序对应self.corners

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

def isGoalState(self, state):

"""

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

"""

#四个角落全部吃掉才是终极

if (state[1] and state[2] and state[3] and state[4]):

return True

else:

return False

def getSuccessors(self, state):

"""

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

As noted in search.py:

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

action, stepCost), where 'successor' is a successor to the current

state, 'action' is the action required to get there, and 'stepCost'

is the incremental cost of expanding to that successor

"""

#在这里返回次态，注意更新角落信息

successors = []

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

x,y = state[0]

dx, dy = Actions.directionToVector(action)

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

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

nextState = (nextx, nexty)

stateList = list(state);

for i in [1, 2, 3, 4]:

if nextState==self.corners[i-1]:

stateList[i] = True

successors.append(((nextState, stateList[1], stateList[2], stateList[3], stateList[4]), action, 1))

self.\_expanded += 1 # DO NOT CHANGE

return successors

def getCostOfActions(self, actions):

"""

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

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

"""

if actions == None: return 999999

x,y= self.startingPosition

for action in actions:

dx, dy = Actions.directionToVector(action)

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

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

return len(actions)

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)

#曼哈顿均值，满足h(n)<=h\*(n)

"""

acceptList = []

for i in [1, 2, 3, 4]:

if not state[i]:

acceptList.append(corners[i-1])

if len(acceptList)==0:

return 0

else:

sumSize = 0;

for corner in acceptList:

sumSize += (abs(corner[0]-state[0][0])+abs(corner[1]-state[0][1]))

return sumSize/len(acceptList)

"""

#曼哈顿最大值，满足h(n)<=h\*(n)

acceptList = []

for i in [1, 2, 3, 4]:

if not state[i]:

acceptList.append(corners[i-1])

if len(acceptList)==0:

return 0

else:

disList = []

for corner in acceptList:

disList.append(abs(corner[0]-state[0][0])+abs(corner[1]-state[0][1]))

return max(disList)

#先靠边(4种选择)，再转圈(2种选择)，8种取最小，满足h(n)<=h\*(n)

"""

#未实现

#self.corners = ((1,1), (1,top), (right, 1), (right, top))

all = 2\*(corners[3][1]-1)+2\*(corners[3][0]-1)

height = corners[3][1];

width = corners[3][0];

top = corners[3][1] - state[0][1]; #2

bottem = state[0][1] - 1; #1

left = state[0][0] - 1; #1

right = corners[3][0] - state[0][0]; #1

"""

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

foodList = foodGrid.asList()

#曼哈顿均值，满足h(n)<=h\*(n)

"""

if len(foodList)==0:

return 0

else:

sumSize = 0;

for food in foodList:

sumSize += (abs(food[0]-state[0][0])+abs(food[1]-state[0][1]))

return sumSize/len(foodList)

"""

#曼哈顿最大值，满足h(n)<=h\*(n)

"""

if len(foodList)==0:

return 0

else:

disList = []

for food in foodList:

disList.append(abs(food[0]-state[0][0])+abs(food[1]-state[0][1]))

return max(disList)

"""

#曼哈顿最大值+最远外部点，满足h(n)<=h\*(n)

if len(foodList)==0:

return 0

else:

disList = []

for food in foodList:

disList.append(abs(food[0]-state[0][0])+abs(food[1]-state[0][1]))

maxDis = max(disList)

farFood = foodList[disList.index(maxDis)]

upBorder = max(state[0][1], farFood[1])

downBorder = min(state[0][1], farFood[1])

leftBorder = min(state[0][0], farFood[0])

rightBorder = max(state[0][0], farFood[0])

maxExt = 0

for food in foodList:

nowExt = 0

if food[0]<leftBorder:

nowExt += (leftBorder-food[0])

if food[0]>rightBorder:

nowExt += (food[0]-rightBorder)

if food[1]<downBorder:

nowExt += (downBorder-food[1])

if food[1]>upBorder:

nowExt += (food[1]-upBorder)

if nowExt > maxExt:

maxExt = nowExt

#upBorder = max(upBorder, food[1])

#downBorder = min(downBorder, food[1])

#leftBorder = min(leftBorder, food[0])

#rightBorder = max(rightBorder, food[0])

return maxDis+ 2\*maxExt

#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)

#广度优先来做

return search.bfs(problem)

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

#有豆子的位置就是目标位置

return state in self.food.asList()

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))