# 一．简介/问题描述

## 1.1 待解决问题的解释

本实验用一个叫pacman的游戏让学生编写一些搜索策略控制吃豆人去吃豆子。吃豆人在行走的过程中需要遵守游戏规则，如不能翻墙、不能碰到幽灵等。该游戏以找到解决方案需要的时间、拓展的节点数等为评分指标，对于吃豆人的搜索路径给出评价。

本实验共有8个问题，其中我写的是问题三的代价一致算法和问题八的次最优搜索。由于问题八需要用到其他的一些搜索算法，所以对于其他的问题我也有所描述。

## 1.2 问题的形式化描述

状态定义：

Pos(x,y)：pacman位于点(x,y)上

destination(x,y): 点(x,y)是目标的点]

wall(x,y): 点(x,y)是墙, 无法通过

动作定义:

gWest: 向西走一步

条件: !wall(x-1,y) && pos(x,y)

结果: 移除pos(x,y), 添加pos(x-1,y)

gEast: 向东走一步

条件: !wall(x+11,y) && pos(x,y)

结果: 移除pos(x,y), 添加pos(x+1,y)

gSouth: 向南走一步

条件: !wall(x,y+1) && pos(x,y)

结果: 移除pos(x,y), 添加pos(x,y+1)

gNorth: 向北走一步

条件: !wall(x,y-1) && pos(x,y)

结果: 移除pos(x,y), 添加pos(x,y-1)

起始状态:

! Pos(x, y) && destination(x, y)

终止状态:

Pos(x, y) && destination(x, y)

## 1.3 解决方案介绍（原理）

问题一:

使用深度优先算法拓展结点, 使用栈作为Open表的数据结构。深度优先算法以搜索树为原型，每一次选择深度最大的节点进行拓展，是后生成的节点先拓展的策略。一般不能保证找到最优解。

问题二：

使用广度优先算法拓展结点，广度优先算法每一次将同样深度的节点按一定顺序逐个进行拓展，是一种先生成的节点先拓展的策略。因此使用队列作为open表的数据结构。一定能够找到最优解，但是往往搜索效率较低。

问题三：

使用代价一致算法算法进行扩展。代价一致算法是广度优先算法的推广，使用优先队列存储拓展的节点，每一次选择队列中代价最小的节点进行拓展。同样也一定能找到最优路径，并且比广度优先算法更快地找到最优路径。

问题四：

使用A\*算法进行扩展。A\*算法是对A算法的估价函数 f (n)=g(n)+h(n)加上某些限制后得到的一种启发式搜索算法，经过限制后，新的估价函数为 f \*(n)=g \*(n)+h\*(n)。其中，g \*(n)表示从开始节点到节点n的最小代价，h\*(n) 表示从节点 n 到目标节点的最小代价，通过对二者进行估计，进而 得出最优路径实际代价 f \*(n)的估计值 f (n) ，结合贪心 算法选择具有全局最小 f (n)的节点进行拓展，直到找到目标节点。

问题五~问题八：

基于以上的四种搜索算法稍作修改后进行应用。

# 二．算法介绍

## 2.1 所用方法的一般介绍及伪代码

## 问题1：应用深度优先算法找到一个特定的位置的豆

用栈作为open表存储拓展到的结点，用closed 表来对已经遍历过的位置进行标记。如果栈不为空，则取出栈顶未被遍历过的节点，将其未被遍历过的子节点（含位置、方向内容）进行压栈，一直到找到目标豆子的位置。

伪代码：

（1）把初始节点S0放入Open表, 建立一个CLOSED表，置为空；

（2）检查Open表是否为空表，若为空，则问题无解，失败退出；

（3）把Open表的第一个节点取出放入Closed表，并记该节点为n；

（4）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；

（5）若节点n不可扩展，则转第（2）步；

（6）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, 转向第（2）步。

## 问题2：广度优先算法

广度优先算法与深度优先算法唯一的不同是Open表的数据结构为队列。由于队列具有先进先出的特性，因此每一次都会逐层拓展节点，达到广度优先的目的。因此伪代码不再重复放入。

## 问题3：代价一致算法

与广度优先算法类似，只不过Open表的数据结构由队列变为了优先队列，并且是按代价为键值进行排序的。因此每次先出队列的一定是代价最小的结点，从而达到代价一致算法的目的。因此伪代码也不再重复放入。

## 问题4：A\* 算法

A\*算法如果使用的启发式函数h\*（n）为0，就退化成了代价一致算法。这里取当前点距目标点的曼哈顿距离作为启发式函数。满足A\*算法的最优性和单调性。令g\*（n）为起点到该点的距离，则f\*（n）=g\*（n）+h\*（n）作为代价,与代价一致算法类似地将节点数据放入以优先队列作为数据结构的Open表中,其余操作均与代价一致算法一样。因此伪代码也不再重复放入。

## 问题7：吃掉所有的豆子

通过A\*算法寻找路径。

## 问题8：次最优搜索

这题是需要优先吃最近的豆子，因此A\*、bfs、ucs都可以解决“寻找最近的豆子”的问题，这里我选择使用A\*，可以使搜索树更小。

（1）准备工作：得到起始位置和食物、墙的位置

（2）定义问题 AnyFoodSearchProblem(gameState)

（3）返回用 A\*算法找到的上述问题的最短路径

isGoalState(self, state):

（1） 取出坐标值(x, y)

（2） 得到食物位置 foodGrid

（3） if 该坐标处有食物或者当前地图中已经没有食物：

（4） Return True

# 三．算法实现

## 3.1 实验环境与问题规模

实验环境： ubuntu20.04

pycharm professional edition 2020.2

Python 3.8

问题规模： 本实验为基于搜索树的最优路径搜索问题。由于地图的限制，搜索树的每个节点最多只有3个子节点，代表能够行走的3个方向（已经访问过的节点不考虑）。设地图规模为m\*n，对吃豆人和 a 颗豆子， 共(m\*n)^(1+a)种状态。

## 3.2 数据结构

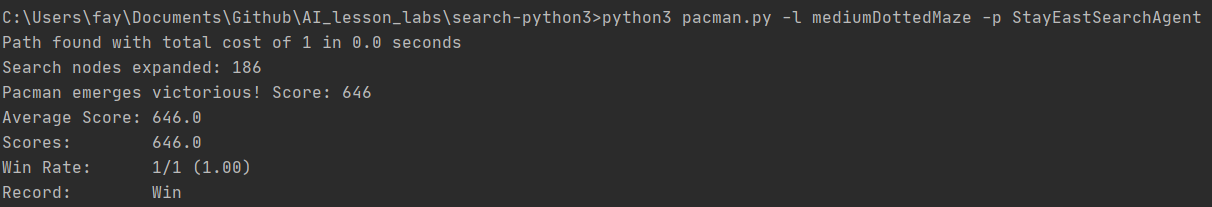
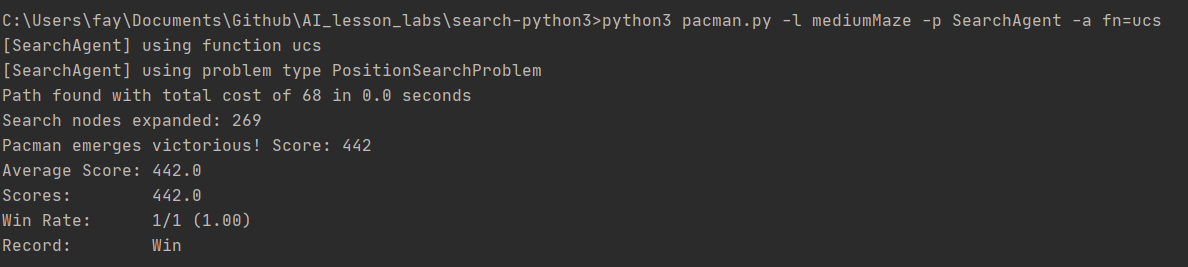
深度优先： 栈

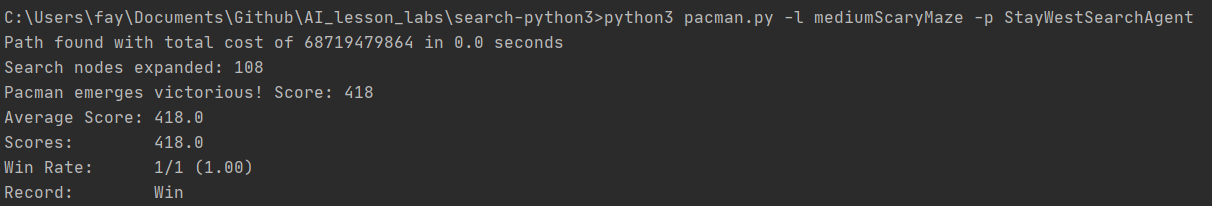
广度优先： 队列

代价一致算法和A\*： 优先队列

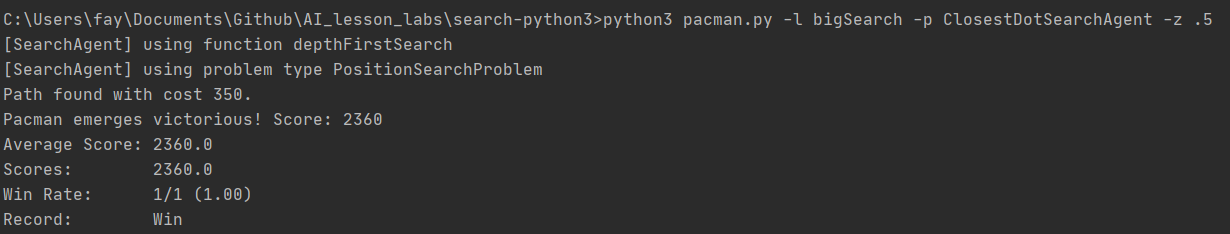
## 3.3 实验结果

代价一致算法





次最优搜索



# 四．总结及讨论（对该实验的总结以及任何该实验的启发）

该实验主要考察了学生对于搜索策略的了解程度，并且要求学生在一个实际的问题中实现他们。对于我来说，通过这次实验我对于A\*、BestFirst、ucs等搜索算法有了更加深刻的了解。这些算法的相似度较高，总体的来说，A\*是最好的，考虑到了已有的cost和对未来cost的估计，而bestfirst只考虑了对未来cost的估计，ucs只考虑了已有的cost。这也是算法不断改进的过程。

## 附录—源代码及其注释（纸质版不需要打印）

# 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: SearchProblem):

"""

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()))

"""

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

initState = (problem.getStartState(), "None")

invalidNode = None

path = util.Stack()

visited = util.Counter()

stack = util.Stack()

visited[initState[0]] = 1

stack.push(initState)

while not stack.isEmpty():

curr = stack.pop()

# 无效节点，用于标记节点层数

# 当取出一个无效节点时

# 表示result栈顶的节点没有子节点

# 或所有子节点已经访问过但无法找到可行解

if curr == invalidNode:

path.pop()

continue

path.push(curr)

if problem.isGoalState(curr[0]):

# 该节点为最终节点，结束循环

break

else:

# 当前节点不是最终节点

# 对当前节点进行扩展获得其子节点

nexts = problem.getSuccessors(curr[0])

# 节点被扩展，层数增加，推入一个invalid节点用于标记

stack.push(invalidNode)

# 选取未出现过的状态节点加入到队列中

for node in nexts:

if visited[node[0]] == 0:

visited[node[0]] = 1

stack.push((node[0], node[1]))

result = []

for i in path.list:

result.append(i[1])

if len(result) > 0:

result.pop(0)

return result

def breadthFirstSearch(problem):

"""Search the shallowest nodes in the search tree first."""

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

from game import Directions

STATE = 0

ACTION = 1

PRE\_STATE = 2

initState = problem.getStartState()

queue = util.Queue()

visited = util.Counter()

# Every node is a triple, (nodeState, action, preNode) where

# 'nodeState' is the state for current node,

# 'action is how previous node goes to current node,

# 'preNode' is the previous node

queue.push((initState, "None", None))

visited[initState] = 1

lastNode = None

while not queue.isEmpty():

curr = queue.pop()

if problem.isGoalState(curr[STATE]):

lastNode = curr

break

else:

nexts = problem.getSuccessors(curr[STATE])

for node in nexts:

if visited[node[0]] == 0:

# 未访问过

visited[node[0]] = 1

queue.push((node[0], node[1], curr))

result = []

if lastNode is not None:

while lastNode[PRE\_STATE] is not None:

result.append(lastNode[ACTION])

lastNode = lastNode[PRE\_STATE]

result.reverse()

return result

def uniformCostSearch(problem):

"""Search the node of least total cost first."""

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

from game import Directions

STATE = 0

ACTION = 1

COST = 2

PRE\_STATE = 3

initState = problem.getStartState()

queue = util.PriorityQueueWithFunction(lambda e: e[COST])

visited = util.Counter()

# Every node is a triple, (nodeState, action, cost, preNode) where

# 'nodeState' is the state for current node,

# 'action is how previous node goes to current node,

# 'preNode' is the previous node

queue.push((initState, "None", 0, None))

visited[initState] = 1

lastNode = None

while not queue.isEmpty():

curr = queue.pop()

if problem.isGoalState(curr[STATE]):

lastNode = curr

break

else:

nexts = problem.getSuccessors(curr[STATE])

for node in nexts:

if visited[node[0]] == 0:

# 未访问过

visited[node[0]] = 1

queue.push((node[0], node[1], node[2], curr))

result = []

if lastNode is not None:

while lastNode[PRE\_STATE] is not None:

result.append(lastNode[ACTION])

lastNode = lastNode[PRE\_STATE]

result.reverse()

return result

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

---------------------------------------------------------------------------------------------------------------------------------

# 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.\_\_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\_\_):

# @UndefinedVariable

if 'drawExpandedCells' in dir(\_\_main\_\_.\_display):

\_\_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

def costFn(pos): return .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

def costFn(pos): return 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

"""

Mask = [1, 2, 4, 8, 0]

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 \*\*\*"

# For display purposes

self.\_visited, self.\_visitedlist, self.\_expanded = {}, [], 0 # DO NOT CHANGE

def getStartState(self):

"""

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

space)

"""

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

return (self.startingPosition, 0)

def isGoalState(self, state):

"""

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

"""

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

return state[1] == 15

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]

cornerState = state[1]

dx, dy = Actions.directionToVector(action)

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

hitsWall = self.walls[nextx][nexty]

nextPos = (nextx, nexty)

if not hitsWall:

idx = 0

while idx < len(self.corners):

if self.corners[idx] == nextPos:

break

idx = idx + 1

successors.append((

(nextPos, cornerState | CornersProblem.Mask[idx]), 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

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

walls = problem.walls

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

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

"\*\*\* 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)

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

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

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

return self.food[x][y]

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))