# 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)

"""

return self.startingPosition, self.corners

def isGoalState(self, state):

"""

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

"""

return len(state[1]) == 0 #state[1] contains all unvisited corners, len == 0 means all corners visited

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]

# Add a successor state to the successor list if the action is legal

x, y = state[0] #Starting position of pacman

dx, dy = Actions.directionToVector(action)

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

unfoundCorners = state[1] #Contains the unvisited corners

cornerTup = []

for corner in unfoundCorners:

if corner != (nextx, nexty):

cornerTup.append(corner)

nextState = (nextx, nexty), tuple(cornerTup) #nextState holds the new position of Pacman, along with a tuple containing corners not at next position

if not self.walls[nextx][nexty]: #check if hitsWall

successors.append((nextState, action, 1)) #Add tuple(newState, action, cost) to successors if new position does not hit wall

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)

currentPosition = state[0]

unvisitedCorners = list(state[1])

heuristic = 0

while unvisitedCorners:

cornerDists = []

for corner in unvisitedCorners: #set up heuristics

cornerDist = abs(currentPosition[0] - corner[0]) + abs(currentPosition[1] - corner[1])

cornerDists.append((cornerDist, corner))

closeCorner = min(cornerDists)

unvisitedCorners.remove(closeCorner[1]) #remove the found closest corner from unvisitedCorners list

heuristic += closeCorner[0] #append the closest corner distance to total distance

currentPosition = closeCorner[1] #set position to be the closest corner

return heuristic

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 \*\*\*"

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))