CMPSC442 AI Project 1

Below is my latest Code:

Modified code, i wrote it myself while looking at previous solution

# 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).

# does this work

"""

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]

## Project1 Question 1

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()))

## Utilize Stack as defined in util.py """

start = problem.getStartState()

visited = []

fringe = util.Stack()

fringe.push((start, ()))

while not fringe.isEmpty():

state, direction = fringe.pop()

if problem.isGoalState(state):

return list(direction)

if state not in visited:

visited.append(state)

succ = problem.getSuccessors(state)

for i in succ:

if i[0] not in visited:

dirlist = list(direction)

dirlist.append(i[1])

fringe.push((i[0], tuple(dirlist)))

## Project1 Question 2

## Similar to DFS, but BFS needs Queue

def breadthFirstSearch(problem):

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

start = problem.getStartState()

visited = []

fringe = util.Queue()

fringe.push((start, ()))

while not fringe.isEmpty():

state, direction = fringe.pop()

if problem.isGoalState(state):

return list(direction)

if state not in visited:

visited.append(state)

succ = problem.getSuccessors(state)

for i in succ:

if i[0] not in visited:

dirlist = list(direction)

dirlist.append(i[1])

fringe.push((i[0], tuple(dirlist)))

## Project 1 Problem 3

def uniformCostSearch(problem):

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

## Utilize PriorityQueue as defined in util.py

start = problem.getStartState()

visited = []

fringe = util.PriorityQueue()

fringe.push((start, ()), 0)

while not fringe.isEmpty():

state, direction = fringe.pop()

if problem.isGoalState(state):

return list(direction)

if state not in visited:

visited.append(state)

succ = problem.getSuccessors(state)

for i in succ:

if i[0] not in visited:

dirlist = list(direction)

dirlist.append(i[1])

cost = problem.getCostOfActions(dirlist)

fringe.push((i[0], tuple(dirlist)), 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

## Project 1 Problem 4

def aStarSearch(problem, heuristic=nullHeuristic):

"""Search the node that has the lowest combined cost and heuristic first."""

## Utilize PriorityQueue as defined in util.py

start = problem.getStartState()

visited = []

fringe = util.PriorityQueue()

fringe.push((start, ()), heuristic(start, problem))

while not fringe.isEmpty():

state, direction = fringe.pop()

if problem.isGoalState(state):

return list(direction)

if state not in visited:

visited.append(state)

succ = problem.getSuccessors(state)

for i in succ:

if i[0] not in visited:

dirlist = list(direction)

dirlist.append(i[1])

cost = problem.getCostOfActions(dirlist) + heuristic(i[0], problem)

fringe.push((i[0], tuple(dirlist)), cost)

# Abbreviations

bfs = breadthFirstSearch

dfs = depthFirstSearch

astar = aStarSearch

ucs = uniformCostSearch