Lab 3

# **Name: Hassan Shahzad**

# **Class: BSCS 7C**

# **CMS ID: 211798**

## **Code Search.py:**

# 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 Node:

"""A node in a search tree. Contains a pointer to the parent (the node

that this is a successor of) and to the actual state for this node. Note

that if a state is arrived at by two paths, then there are two nodes with

the same state. Also includes the action that got us to this state, and

the total path\_cost (also known as g) to reach the node. Other functions

may add an f and h value; see best\_first\_graph\_search and astar\_search for

an explanation of how the f and h values are handled. You will not need to

subclass this class."""

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=0):

"Create a search tree Node, derived from a parent by an action."

self.state = state

self.parent = parent

self.action = action

self.path\_cost = path\_cost

self.depth = 0

if parent:

self.depth = parent.depth + 1

def \_\_repr\_\_(self):

return "<Node %s>" % (self.state,)

def \_\_lt\_\_(self, node):

return self.state < node.state

def expand(self, problem):

"List the nodes reachable in one step from this node."

return [self.child\_node(problem, action)

for action in problem.getSuccessors(self.state)]

def child\_node(self, problem, action):

"[Figure 3.10]"

next = action[0]

return Node(next, self, action[1], self.path\_cost+action[2])

def solution(self):

"Return the sequence of actions to go from the root to this node."

return [node.action for node in self.path()[1:]]

def path(self):

"Return a list of nodes forming the path from the root to this node."

node, path\_back = self, []

while node:

path\_back.append(node)

node = node.parent

return list(reversed(path\_back))

# We want for a queue of nodes in breadth\_first\_search or

# astar\_search to have no duplicated states, so we treat nodes

# with the same state as equal. [Problem: this may not be what you

# want in other contexts.]

def \_\_eq\_\_(self, other):

return isinstance(other, Node) and self.state == other.state

def \_\_hash\_\_(self):

return hash(self.state)

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):

"""

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 \*\*\*"

node = Node(problem.getStartState())

if problem.isGoalState(problem.getStartState()): return node.solution()

frontier = util.Stack()

frontier.push(node)

explored = set()

while not frontier.isEmpty():

node = frontier.pop()

if problem.isGoalState(node.state): return node.solution()

explored.add(node.state)

for child in node.expand(problem):

if child.state not in explored:

frontier.push(child)

return []

def breadthFirstSearch(problem):

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

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

node = Node(problem.getStartState())

if problem.isGoalState(problem.getStartState()): return node.solution()

frontier = util.Queue()

frontier.push(node)

explored = set()

while not frontier.isEmpty():

node = frontier.pop()

if problem.isGoalState(node.state): return node.solution()

explored.add(node.state)

for child in node.expand(problem):

if (child.state not in explored) and (child not in frontier.list):

frontier.push(child)

return []

def uniformCostSearch(problem):

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

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

node = Node(problem.getStartState())

if problem.isGoalState(problem.getStartState()): return node.solution()

frontier = util.PriorityQueue()

frontier.update(node, node.path\_cost)

explored = set()

while not frontier.isEmpty():

node = frontier.pop()

if problem.isGoalState(node.state): return node.solution()

explored.add(node.state)

for child in node.expand(problem):

if (child.state not in explored) and (child not in frontier.heap):

frontier.update(child, child.path\_cost)

def nullHeuristic(state, problem=None):

"""

A heuristic function estimates the cost from the current state to the nearest

goal in the provided SearchProblem. This heuristic is trivial.

"""

return 0

def aStarSearch(problem, heuristic=nullHeuristic):

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

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

node = Node(problem.getStartState())

if problem.isGoalState(problem.getStartState()): return node.solution()

frontier = util.PriorityQueue()

frontier.update(node, node.path\_cost+heuristic(node.state, problem))

explored = set()

while not frontier.isEmpty():

node = frontier.pop()

if problem.isGoalState(node.state): return node.solution()

explored.add(node.state)

for child in node.expand(problem):

if (child.state not in explored) and (child not in frontier.heap):

frontier.update(child, child.path\_cost+heuristic(child.state, problem))

# Abbreviations

bfs = breadthFirstSearch

dfs = depthFirstSearch

astar = aStarSearch

ucs = uniformCostSearch

**Output**:





