CSCE 625: ARTIFICIAL INTELLIGENCE: PROGRAMMING ASSIGNMENT 1

- ANIKET SANJIV BONDE

Program (Python):

```
"""
CSCE 625: ARTIFICIAL INTELLIGENCE
PROGRAMMING ASSIGNMENT 1 - ANIKET SANJIV BONDE
"""

import time
import sys
from heapq import heappop, heappush, heapify
import copy
import random
import numpy as np


class Node:
    def __init__(self):
        self.parent = None
        self.state = []
        self.children = []
        self.depth = 0
        self.heuristic = None


    '''
    Heuristic number 1: Number of blocks out of place

    def set_heuristics(self):
        self.heuristic = self.depth + num_of_alphabets(self.state)
        for alphabet in self.state[0]:
            if self.state[0].index(alphabet) == MAPPING.get(alphabet):
                self.heuristic -= 1
            else:
                break

    '''
```

```
    '''
    Heuristic number 2 (Explanation in the Heuristic_Development.pdf file)

    def set_heuristics(self):
        self.heuristic = num_of_alphabets(self.state) + self.depth
        index = 0
        for alphabet in self.state[0]:
            if self.state[0].index(alphabet) == MAPPING.get(alphabet):
                self.heuristic -= 1
                index += 1
            else:
                break
        self.heuristic += len( self.state[0][index : ])

    '''

    '''
    Heuristic number 3 (The best one: Explanation in the Heuristic_Development.pdf file)
    '''
    def set_heuristics(self):
        self.heuristic = self.depth + 5*num_of_alphabets(self.state)
        len_of_sorted, len_abv_sorted, len_abv_nxt, empty_stacks_no = 0, 0, 0, 0
        for alphabet in self.state[0]:
            if self.state[0].index(alphabet) == MAPPING.get(alphabet):
                len_of_sorted += 1
            else:
                break
        len_abv_sorted = len(self.state[0]) - len_of_sorted

        for stack_number in range(1, len(self.state), 1):
            if REVERSE_MAPPING[len_of_sorted] in self.state[stack_number]:
                len_abv_nxt = len(self.state[stack_number]) -
self.state[stack_number].index(REVERSE_MAPPING[len_of_sorted])



        if ((num_of_alphabets(self.state) - len_of_sorted) > (len(self.state) - 1)):
            for stack in self.state:
                if len(stack) == 0:
                    empty_stacks_no += 1

        self.heuristic += (-5)*len_of_sorted + 2*len_abv_sorted + 2*len_abv_nxt +
empty_stacks_no
```

```
def get_heuristics(self):
    return self.heuristic


def set_depth(self, depth):
    self.depth = depth


def get_depth(self):
    return self.depth

def get_state(self):
    return self.state


def set_parent(self, parent):
    self.parent = parent


def set_state(self, state):
    self.state = state


def get_parent(self):
    return self.parent


def childstates(self):
    children = []
    # iterate over self state one stack at a time
    for i in range(len(self.state)):
        if len( self.state[i] ) == 0:
            continue
        else:
            for j in range(len(self.state)):
                if i == j:
                    continue
                curr_stack = copy.deepcopy(self.state)
                curr_stack[j].append(curr_stack[i].pop())
                children.append(curr_stack)
```

```
        return children


""" Define A* search """
def astar(root):
    frontier = []
    visited = set([])
    closed = set([])
    visited.add(tuple(tuple(ele) for ele in root.get_state()))
    heappush(frontier, (root.get_heuristics() , root))
    count = 0
    print
    while frontier :
        count += 1
        if count == 10000:
            print " Goal state not found , too many iterations "
            sys.exit()

        element = heappop(frontier)
        parent = element[1]

        print "Iter=", count, " Queue=", len(frontier), " Depth=", parent.get_depth()
        closed.add(tuple(tuple(ele) for ele in parent.get_state()))
        if goalreach(parent.get_state()):
            print "\nGOAL REACHED, SUCCESS!"
            print "Number of iterations are :" + str(count)
            print "Frontier size is :" + str(len(frontier))
            return parent
        else:
            for child_state in parent.childstates():
                if tuple(tuple(ele) for ele in child_state) in closed:
                    continue
                else:
                    # set attributes of the child
                    child = Node()
                    child.set_state(child_state)
                    child.set_parent(parent)
                    child.set_depth(parent.get_depth() + 1)

                    # set heuristic after setting depth
                    child.set_heuristics()

                    # check if child is in frontier
```

```
                    if tuple(tuple(ele) for ele in child.get_state()) in visited:
                        # iterate through heap , check if heuristic of node in heap is more
than child, if so delete the heap element and add child
                        index = 0
                        for priority, node in frontier:
                            if node.get_state() == child.get_state() :
                                if child.get_heuristics() < node.get_heuristics():
                                    frontier[index] = frontier[-1]
                                    frontier.pop()
                                    heappush(frontier, (child.get_heuristics(), child))
                                    heapify(frontier)
                                    break
                                else:
                                    # if heuristic of child is more than the state in heap
set child to None for gc
                                    child = None
                                    break
                            else:
                                index += 1
                    else:
                        heappush(frontier, (child.get_heuristics(), child))
                        visited.add(tuple(tuple(ele) for ele in child.get_state()))
```

```
def getpath(goal):
    path = []
    path.append(goal)
    while goal.get_parent() !=  goal:
        goal = goal.get_parent()
        path.insert(0, goal)
    for node in path[1:    ]:
        print "\nNext move:"
        printstate(node.get_state())
```

```
def num_of_alphabets(stacks):
    count = 0
    for stack in stacks:
        count += len(stack)
    return count
```

```
def goalreach(stacks):
    count = 0
    for alphabet in stacks[0]:
        if stacks[0].index(alphabet) == MAPPING.get(alphabet):
            count += 1
        else:
            return False
    if count == num_of_alphabets(stacks):
        return True
    else:
        return False


def printstate(state):
    for i in range(len(state)):
        print i+1,  '|' , state[i]




MAPPING = { "A" : 0 , "B" : 1 , "C" : 2 , "D" : 3 , "E" : 4, \
   "F" : 5 , "G" : 6 , "H" : 7 , "I": 8 , "J" : 9 , "K" : 10, "L" : 11, \
   "M": 12, "N" : 13, "O" : 14 , "P" : 15, "Q" : 16 , "R" : 17, "S" : 18 \
  ,"T" : 19 , "U" : 20 , "V" : 21 , "W" : 22 , "X" : 23 , "Y" : 24 , "Z" : 25 }


REVERSE_MAPPING = dict((v, k) for k, v in MAPPING.iteritems())




if (len(sys.argv) != 3):
    print("\nError, Give the input in this format: 'python blocksworld.py <number of blocks>
<number of stacks>'\n")
    Sys.exit()

numOfBlocks, numOfStacks = int(sys.argv[1]), int(sys.argv[2])
```

```
'''
Randomizing the input for multiple initializations
'''
list_of_blocks, master = [], []
for i in range(numOfBlocks):
    list_of_blocks.append(REVERSE_MAPPING[i])


random.shuffle(list_of_blocks)
master = map(list, np.array_split(list_of_blocks, numOfStacks))
print "\nInitial State: "
printstate(master)


'''
'master' is the initial state. For customized input, please set master in the following line
to the initial state
'''




#master = [['D'], ['C', 'A'], ['B', 'E']]
#master = [['D'], ['E', 'F', 'I', 'J'], ['B', 'G'], ['C', 'H'], ['A']]




root = Node()
root.set_parent(root)
root.set_state(master)
root.set_depth(0)
root.set_heuristics()
start = time.time()
goal = astar(root)
stop = time.time()
print "Run time (in seconds) :" + str(stop - start)
print "Depth of goal state is " + str(goal.get_depth())
print "\nInitial State: "
printstate(root.get_state())
print "\nSolution path is: "
getpath(goal)
```