Bahria University, Islamabad

Department of Software Engineering

Artificial Intelligence Lab

(Fall-2021)

Teacher: Engr. M Waleed Khan

Student : M Iqrar Ijaz Malik

Enrollment : 01-131182-021

Lab Journal:

Date: 13-12-202۱

Lab 8

Task No:	Task Wise Marks		Documentation Marks		Total Marks (20)
	Assigned	Obtained	Assigned	Obtained	
1					

Comments:

Signature

Lab 8: Implementing AI Agents

Introduction

In artificial intelligence, an intelligent agent (IA) refers to an autonomous entity which acts, directing its activity towards achieving goals (i.e. it is an agent), upon an environment using observation through sensors and consequent actuators (i.e. it is intelligent). Intelligent agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex. A reflex machine, such as a thermostat, is considered an example of an intelligent agent. Tools Used

- Google Colab
- MS Word

Task 1:

Implement and AI agent to solve 8 Puzzle Problem using A* Algorithm Code:

```
import numpy as np import
copy
initial_state=np.array([
[1,2,3],
                    [0,4,6],
                    [7,5,8]
]) goal_state=np.array([
                    [1,2,3],
                    [4,5,6],

[7,8,0]
]) class GameNode():
  def __init__(self,state,parent):
    self.state=state
self.parent=parent
self.f=0     self.g=0
   def find(self,value):
if value<0 or value>8:
      raise Exception("Value Out of Range!!")
for row in range(len(self.state)):       for
col in range(len(self.state)):         if
self.state[row][col]==value:
          return (row,col)
   def
avaliable_moves(self):
```

```
    row,col=self.find(0)
free_moves=[]     if
row>0:
      free_moves.append((row-1,col)) #up
if row<2:
      free_moves.append((row+1,col)) #down
if col>0:
      free_moves.append((row,col-1)) #left
if col<2:
      free_moves.append((row,col+1)) #right
return free_moves
     def
clone(self):
    state=copy.deepcopy(self.state) #make a deepcopy of state and then
return as new GameNode     node=GameNode(state,self)     return node
     def
generate_child_nodes(self):
    empty_tile=self.find(0)
free_moves=self.avaliable_moves()
game_nodes=[]     for move in
free_moves:
      node=self.clone()
node.swap(empty_tile,move)
node.g=self.g+1
game_nodes.append(node)     return
game_nodes
     def
swap(self,empty_tile,free_move):
    self.state[empty_tile[0]][empty_tile[1]],self.state[free_move[0]][free_mov
e[1]]=self.state[free_move[0]][free_move[1]],self.state[empty_tile[0]][empty_t
ile[1]]  class Game():   def __init__(self,initial_state,goal_state):
    self.goal_state=goal_state
    self.node=GameNode(state=initial_state,parent=None)
  # heuristic function   def
misplaced_tiles(self,state):
    misplaced=0     for row in
range(len(state)):       for col in
range(len(state)):
        if state[row][col]!=self.goal_state[row][col] and state[row][col]!=0:
          misplaced+=1
return misplaced
  # a_star   def
solve(self):
```

```
    self.node.f=self.misplaced_tiles(self.node.state)
frontier=[self.node]     explored=[]
solution_path=[]     move_count=0     while frontier:
      current_node=frontier[0]
explored.append(current_node)       move_count+=1
if (current_node.state==self.goal_state).all():
while current_node.parent is not None:
          solution_path.append(current_node.state)
current_node=current_node.parent
solution_path.reverse()         print("Goal State
Achieved!")         return solution_path,move_count
for node in current_node.generate_child_nodes():
if node not in frontier and node not in explored:
          node.f=self.misplaced_tiles(node.state)
frontier.append(node)
        #calculate the cost to validate if is lower on the one that is in the
frontier         elif node in open_set:
          #get the value that is currently on the frontier
current_f=filter(lambda elem:elem.state==node.state,self.frontier)
#if the cost of the state in the frontier is higher than the
neighbor,replace it with the lower value           if current_f.f>node.f:
frontier.remove(current_f)             frontier.append(node)
frontier.sort(key=lambda node:node.f,reverse=False) # sort based on f_score
    return solution_path,0 if
__name__=='__main__':
  game=Game(initial_state,goal_state)
path,move_count=game.solve()
  print("Initial State\n",game.node.state,"\n")
for p in path:
    print("Current State after making an optimal move")
print(p)     print()
  print("\nSolved With Misplaced Tiles Heuristics by exploring ",move_count,"
States")
```

Screenshot

```
Goal State Achieved!
Initial State
 [[1 2 3]
 [0 4 6]
 [7 5 8]]

Current State after making an optimal move
[[1 2 3]
 [4 0 6]
 [7 5 8]]

Current State after making an optimal move
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Current State after making an optimal move
[[1 2 3]
 [4 5 6]
 [7 8 0]]


Solved With Misplaced Tiles Heuristics by exploring  4  States
```

Conclusion

The given tasks were completed successfully.