AI(2180703)

Tutorial-5

Name : Yogesh Bavishi
Enrollment No.: 170200107003
Division/Batch: E/E1

Q: Write a program to implement A* Algorithm.

Code(pract5.py):

```
import heapq

class Node(object):
    n = 0
    def __init__(self, board,prev_state = None):
        assert len(board) == 9
        self.board = board[:];
        self.prev = prev_state
        self.step = 0
        Node.n += 1
        if self.prev:
            self.step = self.prev.step + 1


    def __eq__(self,other):
        return self.board == other.board

    def __hash__(self):
        h = [0,0,0]
        h[0] = self.board[0] << 6 | self.board[1] << 3 | self.board[2]
        h[1] = self.board[3] << 6 | self.board[4] << 3 | self.board[5]
        h[2] = self.board[6] << 6 | self.board[7] << 3 | self.board[8]
        h_val = 0
        for h_i in h:
            h_val = h_val *31 + h_i
        return h_val

    def __str__(self):
        string_list = [str(i)+' ' for i in self.board]
        sub_list = (string_list[:3],string_list[3:6],string_list[6:])
        return "\n".join(["".join(l)for l in sub_list ])

    def manhattan_distance(self):
        distance = 0
        goal = [1,2,3,4,5,6,7,8,0]
```

```
        for i in range(1,9):
            xs,ys = self.pos(self.board.index(i))
            xg,yg = self.pos(goal.index(i))
            distance += abs(xs-xg) + abs(ys-yg)
        return distance

    def hamming_distance(self):
        distance = 0
        goal = [1,2,3,4,5,6,7,8,0]
        for i in range(9):
            if goal[i] != self.board[i]: distance += 1
        return distance

    def next(self):
        next_moves = []
        i = self.board.index(0)
        next_moves = (self.moveUp(i),self.moveDown(i),self.moveRight(i),
self.moveLeft(i))
        return [s for s in next_moves if s]


    def moveLeft(self,i):
        x,y = self.pos(i)
        if y > 0:
            left_state = Node(self.board,self)
            left = self.sop(x,y-1)
            left_state.swap(i,left)
            return left_state

    def moveRight(self,i):
        x,y = self.pos(i)
        if y < 2 :
            right_state = Node(self.board,self)
            right = self.sop(x,y+1)
            right_state.swap(i,right)
            return right_state

    def moveUp(self,i):
        x,y = self.pos(i)
        if x > 0:
            up_state = Node(self.board,self)
            up = self.sop(x-1,y)
            up_state.swap(i,up)
            return up_state

    def moveDown(self , i):
        x,y = self.pos(i)
        if x < 2 :
            down_state = Node(self.board,self)
```

```
            down = self.sop(x+1,y)
            down_state.swap(i,down)
            return down_state

    def swap(self,i,j):
        self.board[j],self.board[i] = self.board[i],self.board[j]

    def pos(self,index):
        return (int(index/3),index%3)


    def sop(self,x,y):
        return x * 3 + y

class PriorityQueue:
    def  __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

def printPath(state):
    path = []
    while state:
        path.append(state)
        state = state.prev
    path.reverse()
    print("\n    \n".join([str(state) for state in path]))

def astar(start,goal):
    depth = 75
    priotity_queue = PriorityQueue()
    h_val = start.manhattan_distance() + start.hamming_distance()

    f_val = h_val + start.step
    priotity_queue.push(start, f_val)
    visited = set()
    found = False
    while not priotity_queue.isEmpty():
        state = priotity_queue.pop()
```

```
        if state == goal:
            found = state
            break

        if state in visited or state.step > depth:
            continue

        visited.add(state)

        for s in state.next():
            h_val_s = s.manhattan_distance() + s.hamming_distance()
            f_val_s = h_val_s + s.step
            priotity_queue.push(s, f_val_s)

    if found:
        print('\nFollow Below Steps To Solve Puzzle\n')
        printPath(found)
        print("\nSolution Founded Successfully")
    else:
        print("No solution found")

print("\n8-Puzzle Problem is Solved  using A* Algorithm")

print("\nGoal State is : 1 2 3\n\t\t4 5 6\n\t\t7 8 0")
print("\n0 represents the empty tile.")

puzzle = list(map(int,input("\nEnter Current Puzzle State : ").strip().
split()))[:9]

start = Node(puzzle)
goal = Node([1,2,3,4,5,6,7,8,0])
astar(start,goal)
```

Output:

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
D:\PROJECTS\AI>python pract5.py

8-Puzzle Problem is Solved  using A* Algorithm

Goal State is : 1 2 3
                4 5 6
                7 8 0

0 represents the empty tile.

Enter Current Puzzle State : 1 2 3 4 6 0 7 5 8

Follow Below Steps To Solve Puzzle

1 2 3
4 6 0
7 5 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Solution Founded Successfully

D:\PROJECTS\AI>
```