**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.")

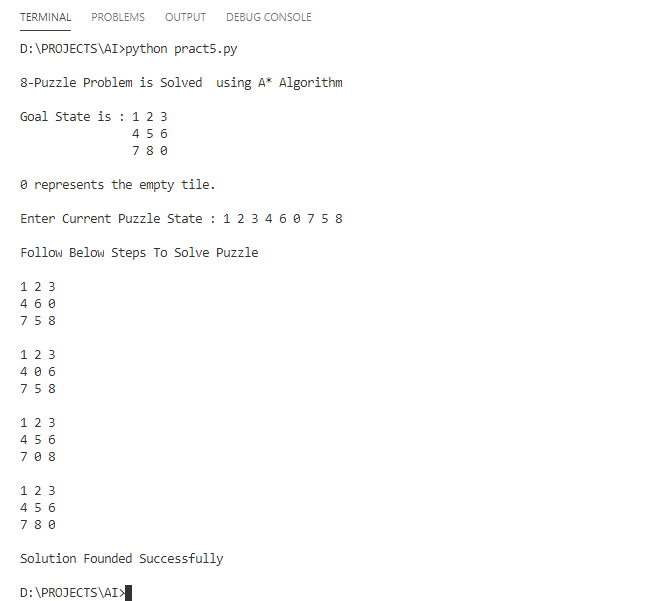
puzzle = list(map(int,input("\nEnter Current Puzzle State : ").strip().split()))[:9]

start = Node(puzzle)

goal = Node([1,2,3,4,5,6,7,8,0])

astar(start,goal)

**Output:**

****