***Muhammad Kaleem***

***56614***

***Lab Task no 8***

***Q1***

***Code:***

from copy import deepcopy

import numpy as np

import time

def bestsolution(state):

bestsol = np.array([], int).reshape(-1, 9)

count = len(state) - 1

while count != -1:

bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

count = (state[count]['parent'])

return bestsol.reshape(-1, 3, 3)

# Checks for the uniqueness of the iteration(it).

def all(checkarray):

set=[]

for it in set:

for checkarray in it:

return 1

else:

return 0

# Number of misplaced tiles

def misplaced\_tiles(puzzle, goal):

mscost = np.sum(puzzle != goal) - 1

return mscost if mscost > 0 else 0

def coordinates(puzzle):

pos = np.array(range(9))

for p, q in enumerate(puzzle):

pos[q] = p

return pos

# Start of puzzle evaluation, using Misplaced tiles heuristics

def evaluate\_misplaced(puzzle, goal):

steps = np.array([

('up', [0, 1, 2], -3),

('down', [6, 7, 8], 3),

('left', [0, 3, 6], -1),

('right', [2, 5, 8], 1)

], dtype=[('move', str, 1), ('position', list), ('head', int)])

dtstate = [('puzzle', list), ('parent', int), ('gn', int), ('hn', int)]

costg = coordinates(goal)

# Initializing the parent, gn and hn, where hn is misplaced\_tiles function call

parent = -1

gn = 0

hn = misplaced\_tiles(coordinates(puzzle), costg)

state = np.array([(puzzle, parent, gn, hn)], dtstate)

# Priority queues with position as keys and fn as value.

dtpriority = [('position', int), ('fn', int)]

priority = np.array([(0, hn)], dtpriority)

while True:

priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

position, fn = priority[0]

# Sort priority queue using merge sort, the first element is picked for exploring.

priority = np.delete(priority, 0, 0)

puzzle, parent, gn, hn = state[position]

puzzle = np.array(puzzle)

blank = int(np.where(puzzle == 0)[0])

gn += 1

c = 1

start\_time = time.time()

for s in steps:

c += 1

if blank not in s['position']:

openstates = deepcopy(puzzle)

openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

end\_time = time.time()

if (end\_time - start\_time) > 2:

print(" The puzzle is unsolvable \n")

break

hn = misplaced\_tiles(coordinates(openstates), costg)

# Generate and add new state in the list

q = np.array([(openstates, position, gn, hn)], dtstate)

state = np.append(state, q, 0)

# f(n) is the sum of cost to reach node

fn = gn + hn

q = np.array([(len(state) - 1, fn)], dtpriority)

priority = np.append(priority, q, 0)

if np.array\_equal(openstates, goal):

print(' The puzzle is solvable \n')

return state, len(priority)

return state, len(priority)

# Updated initial state using append method

puzzle = []

puzzle.append(1)

puzzle.append(2)

puzzle.append(3)

puzzle.append(4)

puzzle.append(0)

puzzle.append(5)

puzzle.append(7)

puzzle.append(8)

puzzle.append(6)

# Updated goal state using append method

goal = []

goal.append(1)

goal.append(2)

goal.append(3)

goal.append(4)

goal.append(5)

goal.append(0)

goal.append(7)

goal.append(8)

goal.append(6)

state, visited = evaluate\_misplaced(puzzle, goal)

bestpath = bestsolution(state)

print(str(bestpath).replace('[', ' ').replace(']', ''))

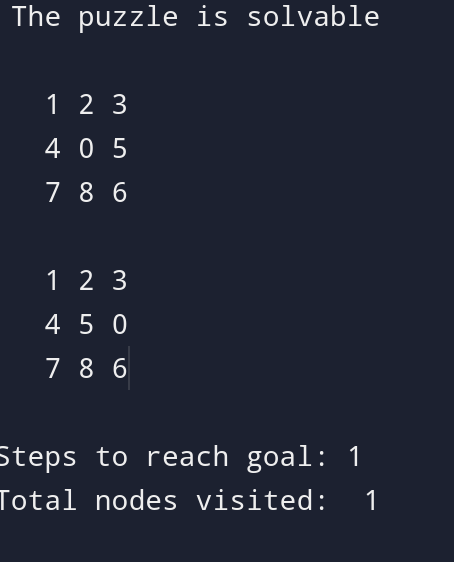
totalmoves = len(bestpath) - 1

print('\nSteps to reach goal:', totalmoves)

visit = len(state) - visited

print('Total nodes visited: ', visit, "\n")

***Output:***

******

***Q2***

***Code:***

import heapq

class Node:

def \_\_init\_\_(self, name, g=0, h=0):

self.name = name # Name of the node

self.g = g # Cost from start node

self.h = h # Heuristic cost to goal

self.f = g + h # Total cost

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

return self.f < other.f # For priority queue comparison

def heuristic(node, goal):

# For simplicity, the heuristic is defined as the absolute difference

# between the node name (assumed to be an integer) and goal.

return abs(goal - node)

def a\_star(graph, start, goal):

open\_list = []

closed\_set = set()

start\_node = Node(start, 0, heuristic(start, goal))

heapq.heappush(open\_list, start\_node)

came\_from = {}

g\_costs = {start: 0}

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.name == goal:

# Reconstruct path

path = []

while current\_node.name in came\_from:

path.append(current\_node.name)

current\_node = came\_from[current\_node.name]

path.append(start)

return path[::-1] # Return reversed path

closed\_set.add(current\_node.name)

for neighbor, cost in graph[current\_node.name].items():

if neighbor in closed\_set:

continue

tentative\_g\_cost = g\_costs[current\_node.name] + cost

if neighbor not in g\_costs or tentative\_g\_cost < g\_costs[neighbor]:

came\_from[neighbor] = current\_node

g\_costs[neighbor] = tentative\_g\_cost

h\_cost = heuristic(neighbor, goal)

neighbor\_node = Node(neighbor, tentative\_g\_cost, h\_cost)

if neighbor\_node not in open\_list:

heapq.heappush(open\_list, neighbor\_node)

return None # No path found

# Define the graph as a dictionary

graph = {

0: {1: 1, 2: 4},

1: {0: 1, 3: 2, 4: 5},

2: {0: 4, 4: 1},

3: {1: 2, 5: 3},

4: {1: 5, 2: 1, 5: 1},

5: {3: 3, 4: 1}

}

start\_node = 0

goal\_node = 5

# Find the shortest path using A\*

path = a\_star(graph, start\_node, goal\_node)

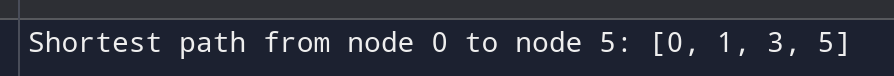
if path is not None:

print("Shortest path from node {} to node {}: {}".format(start\_node, goal\_node, path))

else:

print("No path found from node {} to node {}.".format(start\_node, goal\_node))

***Output:***

******