**Name:** Moin Tamboli

**Roll no.:** COTC64

**Practical No. 2**

**Source Code:**

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list=adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

def h(self, n):

H={

'A':3,

'B':4,

'C':2,

'D':6,

'G':0,

'S':5

}

return H[n]

def a\_star\_algorithm(self,start\_node,stop\_node):

open\_list=set([start\_node])

closed\_list=set([])

g={}

g[start\_node]=0

parents={}

parents[start\_node]=start\_node

while len(open\_list)>0:

n=None

for v in open\_list:

if n==None or g[v]+self.h(v)<g[n]+self.h(n):

n=v;

if n==None:

print('path does not exist!')

return None

if n==stop\_node:

reconst\_path=[]

while parents[n]!=n:

reconst\_path.append(n)

n=parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('path found:{}'.format(reconst\_path))

return reconst\_path

for(m,weight) in self.get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m]=n

g[m]=g[n]+weight

else:

if g[m]>g[n]+weight:

g[m]=g[n]+weight

parents[m]=n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('path does not exist!')

return None

adjacency\_list={

'A':[('B',2),('C',1)],

'B':[('D',5)],

'C':[('D',3),('G',4)],

'D':[('G',2)],

'S':[('A',1),('G',10)]

}

graph1=Graph(adjacency\_list)

graph1.a\_star\_algorithm('S','G')

# --------------------------------------------------------------------8 puzzle ------------------------------------------------------------------

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 8 puzzle evaluvation, using Misplaced tiles heuristics

def evaluvate\_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 1:

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 = gn + 1

c = 1

start\_time = time.time()

for s in steps:

c = 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 8 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 8 puzzle is solvable \n')

return state, len(priority)

return state, len(priority)

# initial state

puzzle = []

puzzle.append(2)

puzzle.append(8)

puzzle.append(3)

puzzle.append(1)

puzzle.append(6)

puzzle.append(4)

puzzle.append(7)

puzzle.append(0)

puzzle.append(5)

#goal state

goal = []

goal.append(1)

goal.append(2)

goal.append(3)

goal.append(8)

goal.append(0)

goal.append(4)

goal.append(7)

goal.append(6)

goal.append(5)

state, visited = evaluvate\_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:**

path found:['S', 'A', 'C', 'G']

The 8 puzzle is solvable

2 8 3

1 6 4

7 0 5

2 8 3

1 0 4

7 6 5

2 0 3

1 8 4

7 6 5

0 2 3

1 8 4

7 6 5

1 2 3

0 8 4

7 6 5

1 2 3

8 0 4

...

Steps to reach goal: 5

Total nodes visited: 6