BST

from queue import PriorityQueue

# Filling adjacency matrix with empty arrays

vertices = 14

graph = [[] for i in range(vertices)]

# Function for adding edges to graph

def add\_edge(x, y, cost):

graph[x].append((y, cost))

graph[y].append((x, cost))

# Function For Implementing Best First Search

# Gives output path having the lowest cost

def best\_first\_search(source, target, vertices):

visited = [0] \* vertices

pq = PriorityQueue()

pq.put((0, source))

print("Path: ")

while not pq.empty():

u = pq.get()[1]

# Displaying the path having the lowest cost

print(u, end=" ")

if u == target:

break

for v, c in graph[u]:

if not visited[v]:

visited[v] = True

pq.put((c, v))

print()

if \_\_name\_\_ == '\_\_main\_\_':

# The nodes shown in above example(by alphabets) are

# implemented using integers add\_edge(x,y,cost);

add\_edge(0, 1, 1)

add\_edge(0, 2, 8)

add\_edge(1, 2, 12)

add\_edge(1, 4, 13)

add\_edge(2, 3, 6)

add\_edge(4, 3, 3)

source = 0

target = 2

best\_first\_search(source, target, vertices)

A\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

from collections import deque

class Graph:

# example of adjacency list (or rather map)

# adjacency\_list = {

# 'A': [('B', 1), ('C', 3), ('D', 7)],

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

# 'C': [('D', 12)]

# }

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

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

# heuristic function with equal values for all nodes

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

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

# open\_list is a list of nodes which have been visited, but who's neighbors

# haven't all been inspected, starts off with the start node

# closed\_list is a list of nodes which have been visited

# and who's neighbors have been inspected

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

closed\_list = set([])

# g contains current distances from start\_node to all other nodes

# the default value (if it's not found in the map) is +infinity

g = {}

g[start\_node] = 0

# parents contains an adjacency map of all nodes

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

# find a node with the lowest value of f() - evaluation function

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 the current node is the stop\_node

# then we begin reconstructin the path from it to the start\_node

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 all neighbors of the current node do

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

# if the current node isn't in both open\_list and closed\_list

# add it to open\_list and note n as it's parent

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

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

# otherwise, check if it's quicker to first visit n, then m

# and if it is, update parent data and g data

# and if the node was in the closed\_list, move it to open\_list

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)

# remove n from the open\_list, and add it to closed\_list

# because all of his neighbors were inspected

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

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

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')