Implementation of Best First Search and A\* Search

**Code for Best first Search:**

from queue import PriorityQueue

import matplotlib.pyplot as plt

import networkx as nx

*# for implementing BFS | returns path having lowest cost*

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

 visited = [0] \* n

 visited[source] = True

pq = PriorityQueue()

pq.put((0, source))

 while pq.empty() == False:

 u = pq.get()[1]

print(u, end=" ") *# the path having lowest cost*

if u == target:

 break

 for v, c in graph[u]:

 if visited[v] == False:

 visited[v] = True

pq.put((c, v))

print()

*# for adding edges to graph*

def addedge(x, y, cost):

 graph[x].append((y, cost))

 graph[y].append((x, cost))

G = nx.Graph()

v = int(input("Enter the number of nodes: "))

graph = [[] for i in range(v)] *# undirected Graph*

e = int(input("Enter the number of edges: "))

print("Enter the edges along with their weights:")

for i in range(e):

 x, y, z = list(map(int, input().split()))

addedge(x, y, z)

G.add\_edge(x, y, weight = z)

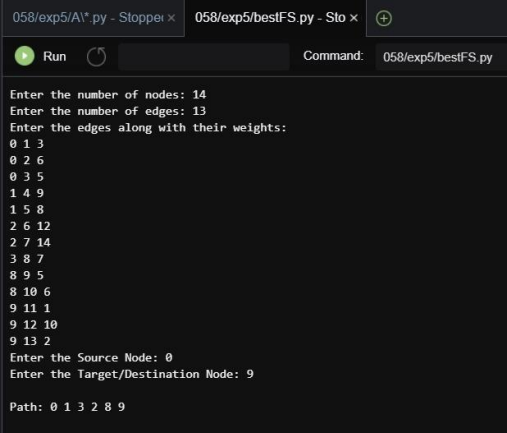
source = int(input("Enter the Source Node: "))

target = int(input("Enter the Target/Destination Node: "))

print("\nPath: ", end = "")

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

**Output:**

****

**Code for A\* search:**

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': 1,'B': 1,'C': 1,'D': 1}

 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', 1), ('C', 3), ('D', 7)],

 'B': [('D', 5)],

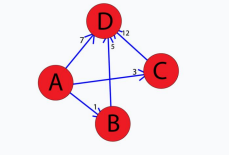
 'C': [('D', 12)]

}

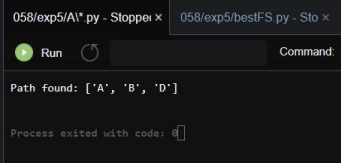
graph1 = Graph(adjacency\_list)

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

**Input Graph:**

****

**Output:**

****