**Artificial Intelligence Lab**

**Name** - N Aditya Sai

**Register Number**- RA1911030010075

**Topic**: Constraint Satisfaction Problem

**Experiment** – 5B

**Developing Best First Search And A\* Algorithm for Real World Problems**

**Aim:** To implement Best First Algorithm and A\* Algorithm using python.

**Best First Search:**

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

**Algorithm:**

* Define a list, OPEN, consisting solely of a single node, the start node, s.
* IF the list is empty, return failure.
* Remove from the list the node n with the best score (the node where f is the minimum), and move it to a list, CLOSED.
* Expand node n.
* IF any successor to n is the goal node, return success and the solution (by tracing the path from the goal node to s).
* FOR each successor node:
  1. Apply the evaluation function, f, to the node.
  2. IF the node has not been in either list, add it to OPEN.
* looping structure by sending the algorithm back to the second step.

**Code:**

from queue import PriorityQueue

v = 14

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

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

    visited = [0] \* n

    visited[0] = True

    pq = PriorityQueue()

    pq.put((0, source))

    while pq.empty() == False:

        u = pq.get()[1]

        print(u, end=" ")

        if u == target:

            break

        for v, c in graph[u]:

            if visited[v] == False:

                visited[v] = True

                pq.put((c, v))

    print()

def addedge(x, y, cost):

    graph[x].append((y, cost))

    graph[y].append((x, cost))

addedge(0, 1, 3)

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

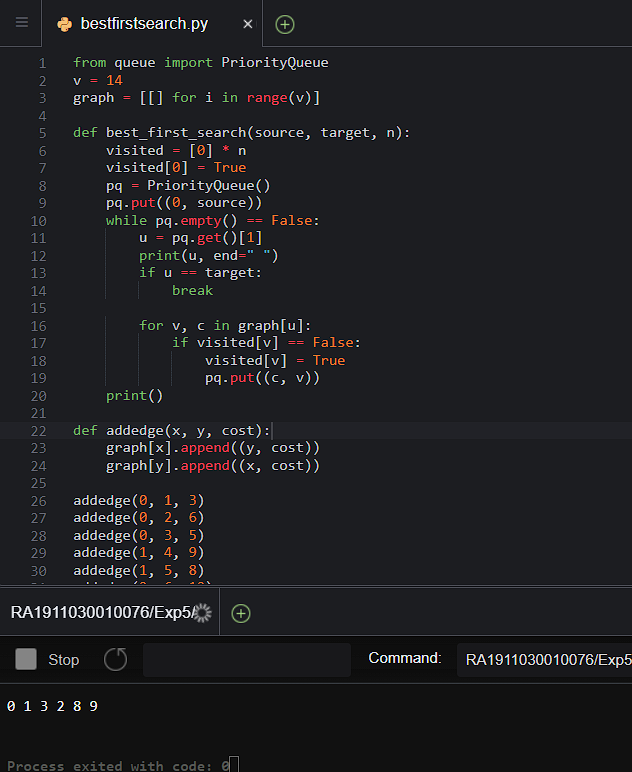
addedge(9, 13, 2)

source = 0

target = 9

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

**Output:**

****

**A\* Search Algorithm:**

A\* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

**Algorithm:**

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

* Initialize the open list
* Initialize the closed list put the starting node on the open list (you can leave its f at zero)
* while the open list is not empty

1. find the node with the least f on the open list, call it "q"
2. pop q off the open list
3. generate q's 8 successors and set their parents to q
4. for each successor
   1. if successor is the goal, stop search
   2. else, compute both g and h for successor

successor.g = q.g + distance between successor and q successor.h = distance from goal to successor(This can be done using many ways, we will discuss three heuristics- Manhattan, Diagonal and Euclidean Heuristics)

successor.f = successor.g + successor.h

1. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
2. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list

end (for loop)

1. push q on the closed list end (while loop)

**Code:**

from collections import deque

class Graph:

    def \_init\_(self, adjac\_lis):

        self.adjac\_lis = adjac\_lis

    def get\_neighbors(self, v):

        return self.adjac\_lis[v]

    # This is heuristic function which is having 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, stop):

        # In this open\_lst is a lisy of nodes which have been visited, but who's

        # neighbours haven't all been always inspected, It starts off with the start

  #node

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

        # and who's neighbors have been always inspected

        open\_lst = set([start])

        closed\_lst = set([])

        # poo has present distances from start to all other nodes

        # the default value is +infinity

        poo = {}

        poo[start] = 0

        # par contains an adjac mapping of all nodes

        par = {}

        par[start] = start

        while len(open\_lst) > 0:

            n = None

            # it will find a node with the lowest value of f() -

            for v in open\_lst:

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

                    n = v;

            if n == None:

                print('Path does not exist!')

                return None

            # if the current node is the stop

            # then we start again from start

            if n == stop:

                reconst\_path = []

                while par[n] != n:

                    reconst\_path.append(n)

                    n = par[n]

                reconst\_path.append(start)

                reconst\_path.reverse()

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

                return reconst\_path

            # for all the neighbors of the current node do

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

              # if the current node is not presentin both open\_lst and closed\_lst

                # add it to open\_lst and note n as it's par

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

                    open\_lst.add(m)

                    par[m] = n

                    poo[m] = poo[n] + weight

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

                # and if it is, update par data and poo data

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

                else:

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

                        poo[m] = poo[n] + weight

                        par[m] = n

                        if m in closed\_lst:

                            closed\_lst.remove(m)

                            open\_lst.add(m)

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

            # because all of his neighbors were inspected

            open\_lst.remove(n)

            closed\_lst.add(n)

        print('Path does not exist!')

        return None

adjac\_lis = {

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

    'B': [('D', 5)],

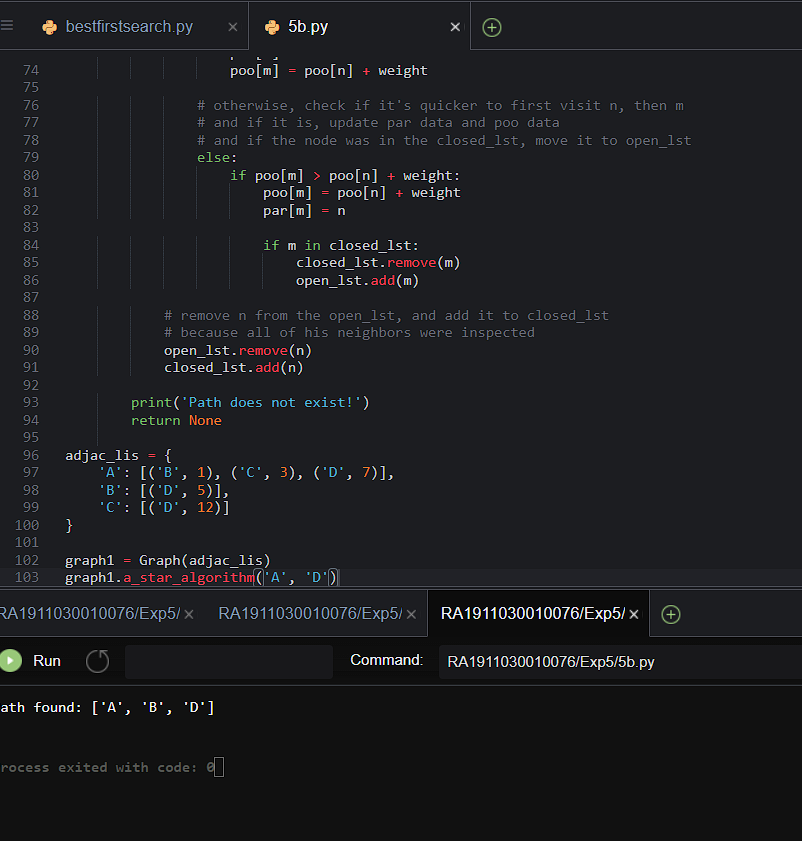
    'C': [('D', 12)]

}

graph1 = Graph(adjac\_lis)

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

**Output:**

****

**Result:**

Best first search and A\* search algorithm were successfully executed in python.