# Artificial Intelligence

# Anjali Agrawal

# RA1811030010096

BFS and A\*

AIM:

1. To perform Best First Search traversal on a graph.
2. To perform A\* on a graph.

PROBLEM DESCRIPTION:

BFS:

Best first search uses the concept of a priority queue and heuristic search. It is a search algorithm that works on a specific rule. The aim is to traverse the graph using best first search.

A\*:

A\* is based on using heuristic methods to achieve *optimality* and *completeness*, and is a variant of the best-first algorithm. We need to find the cost effective path.

PROBLEM FORMULATION:

BEST FIRST SEARCH

The Best first search uses the concept of a Priority queue and heuristic search. To search the graph space, the BFS method uses two lists for tracking the traversal. An ‘Open’ list which keeps track of the current ‘immediate’ nodes available for traversal and ‘CLOSED’ list that keeps track of the nodes already traversed. For Greedy Best first search the evaluation function is f(n) = h(n) while for A\* the evaluation function is f(n) = g(n) + h(n).

A\*

What A\* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

1. = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
2. = the estimated movement cost to move from that given square on the grid to the final destination.

ALGORITHM:

BFS:

* + Get number of nodes and edges o Add the edges onto the graph using a function.
  + Best first algorithm – do the following steps until priority queue is empty.
    - Print the path which has the lowest cost.
    - If the node is the target node, we have reached our solution.
    - Put all the neighbors of the node into the priority queue.

A\*:

* + Firstly, add the beginning node to the open list o Then repeat the following step
    - In the open list, find the square with the lowest F cost – and this denotes the current square.
    - Now we move to the closed square.
    - Consider 8 squares adjacent to the current square and
  + Ignore it if it is on the closed list, or if it is not workable. Do the following if it is workable
  + Check if it is on the open list; if not, add it. You need to make the current square as this square’s a parent. You will now record the different costs of the square like the F, G and H costs.
  + If it is on the open list, use G cost to measure the better path. Lower the G cost, the better the path. If this path is better, make the current square as the parent square. Now you need to recalculate the other scores – the G and F scores of this square. o If you find the path, you need to check the closed list and add the target square to it.
  + There is no path if the open list is empty and you could not find the target square.
  + Now you can save the path and work backwards starting from the target square, going to the parent square from each square you go, till it takes you to the starting square. You’ve found your path now.

Source Code:

BFS:

from queue import PriorityQueue

# best first algorithm

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

    visited = [False for \_ in range(n)]

    # keep track of visited nodes

    visited[0] = True

    pq = PriorityQueue()

    pq.put((0, source))

    while pq.empty() == False:

        u = pq.get()[1]

        # Displaying path which has lowest cost

        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()

# Function for adding edges to graph

def addedge(x, y, cost):

    graph[x].append((y, cost))

    graph[y].append((x, cost))

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

# implemented using integers addedge(x,y,cost);

N = int(input("Number of nodes: "))

E = int(input("Number of edges: "))

graph = [[] for \_ in range(N)]

for \_ in range(E):

    x,y,cost=input("Enter edge: ").split(",")

    addedge(int(x),int(y),int(cost))

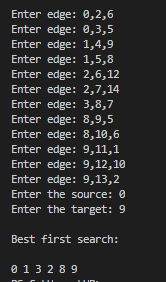
source = int(input("Enter the source: "))

target = int(input("Enter the target: "))

print("\nBest first search: \n")

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

Output:



A\*

def aStarAlgo(start\_node, stop\_node,heu):

    open\_set = set(start\_node)

    closed\_set = set()

    g = {}  # store distance from starting node

    parents = {}  # parents contains an adjacency map of all nodes

    # ditance of starting node from itself is zero

    g[start\_node] = 0

    # start\_node is root node i.e it has no parent nodes

    # so start\_node is set to its own parent node

    parents[start\_node] = start\_node

    while len(open\_set) > 0:

        n = None

        # node with lowest f() is found

        for v in open\_set:

            if n == None or g[v] + heu[v] < g[n] + heu[n]:

                n = v

        if n == stop\_node or Graph\_nodes[n] == None:

            pass

        else:

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

                # nodes 'm' not in first and last set are added to first

                # n is set its parent

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

                    open\_set.add(m)

                    parents[m] = n

                    g[m] = g[n] + weight

                # for each node m,compare its distance from start i.e g(m) to the

                # from start through n node

                else:

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

                        # update g(m)

                        g[m] = g[n] + weight

                        # change parent of m to n

                        parents[m] = n

                        # if m in closed set,remove and add to open

                        if m in closed\_set:

                            closed\_set.remove(m)

                            open\_set.add(m)

        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:

            path = []

            while parents[n] != n:

                path.append(n)

                n = parents[n]

            path.append(start\_node)

            path.reverse()

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

            return path

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

        # because all of his neighbors were inspected

        open\_set.remove(n)

        closed\_set.add(n)

    print('\nPath does not exist!')

    return None

# define fuction to return neighbor and its distance

# from the passed node

def get\_neighbors(v):

    if v in Graph\_nodes:

        return Graph\_nodes[v]

    else:

        return None

# Describe your graph here

Graph\_nodes = {}

n = int(input("Enter no. of nodes:"))

val = []

for i in range(n):

    k = input("Node: ")

    j = int(input("Enter number of neighbors: "))

    l1 = []

    for \_ in range(j):

        node, cost = input("Enter node and cost: ").split(",")

        # val=[item for item in input("Enter the neighbours: ").split()]

        l1.append(tuple((node, int(cost))))

    Graph\_nodes[k] = l1

print()

print(Graph\_nodes)

heu={}

print("\nInput for heuristics function: \n")

Node=int(input("Enter no. of total nodes:"))

val=[]

for i in range(Node):

    k=input("Node: ")

    val=int(input("Heuristic value: "))

    heu[k]=val

print()

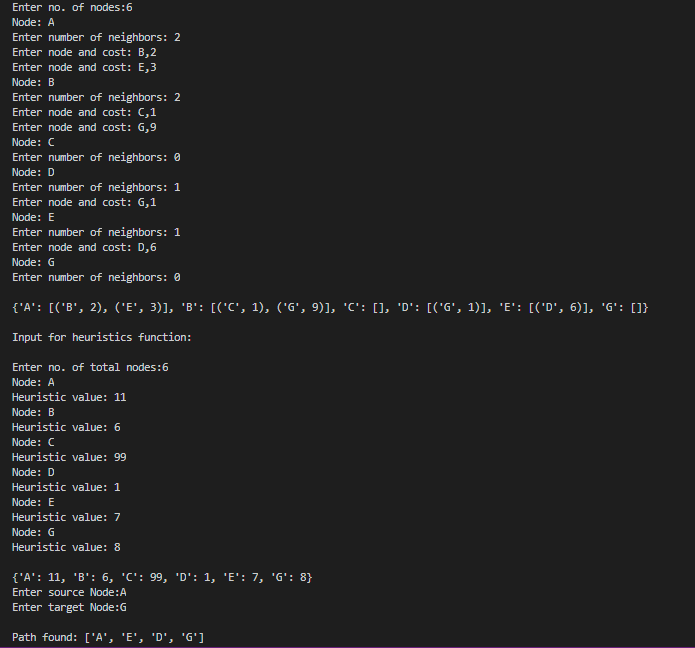
print(heu)

source=input("Enter source Node:")

target=input("Enter target Node:")

aStarAlgo(source, target,heu)

Output:



Output

RESULT: We have successfully implemented both BFS and A\* search algorithm