Developing best first search and A star algorithm for real world problems

Aim:

To develop best first algorithm and a star algorithm for real world problems.

Code:

BFS:

from queue import PriorityQueue

v = 14

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

# Function For Implementing Best First Search

# Gives output path having lowest cost

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

    visited = [0] \* n

    visited = True

    pq = PriorityQueue()

    pq.put((0, source))

    while pq.empty() == False:

        u = pq.get()[1]

        # Displaying the path having 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);

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)

A\*:

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,

            'E': 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), ('E',13)],

    'B': [('D', 5)],

    'C': [('E', 12)],

    'D': [('E', 6)]

}

graph1 = Graph(adjacency\_list)

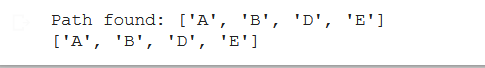
graph1.a\_star\_algorithm('A', 'E')

Output:

BFS:



A\*:



Result:

BFS and A\* algorithm for real world problems were successfully implemented.