***Practical 3***

***Aim***: *Implement A\* search algorithm for Romanian map problem.*

***Theory:***

What is A\* Search Algorithm?

A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

***Why A\* Search Algorithm?***

Informally speaking, A\* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

***Explanation***

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.

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.

We define ‘g’ and ‘h’ as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.

**A\* Search Algorithm**

1. Initialize the open list

2. Initialize the closed list put the starting node on the open list (you can leave its f at zero)

3. while the open list is not empty

a) find the node with the least f on the open list, call it "q"

b) pop q off the open list

c) generate q 8 successors and set their parents to q

d) for each successor

i) if successor is the goal, stop search

ii) 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

iii) if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor

iV) 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)

e) push q on the closed list

end (while loop)

So, suppose as in the below figure if we want to reach the target cell from the source cell, then the A\* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristic.

***Code:***

import heapq

class priorityQueue:

    def \_\_init\_\_(self):

        self.cities = []

    def push(self, city, cost):

        heapq.heappush(self.cities, (cost, city))

    def pop(self):

        return heapq.heappop(self.cities)[1]

    def isEmpty(self):

        if (self.cities == []):

            return True

        else:

            return False

    def check(self):

        print(self.cities)

class ctNode:

    def \_\_init\_\_(self, city, distance):

        self.city = str(city)

        self.distance = str(distance)

romania = {}

def makedict():

    file = open("romania.txt", 'r')

    for string in file:

        line = string.split(',')

        ct1 = line[0]

        ct2 = line[1]

        dist = int(line[2])

        romania.setdefault(ct1, []).append(ctNode(ct2, dist))

        romania.setdefault(ct2, []).append(ctNode(ct1, dist))

def makehuristikdict():

    h = {}

    with open("romania\_sld.txt", 'r') as file:

        for line in file:

            line = line.strip().split(",")

            node = line[0].strip()

            sld = int(line[1].strip())

            h[node] = sld

    return h

def heuristic(node, values):

    return values[node]

def astar(start, end):

    path = {}

    distance = {}

    q = priorityQueue()

    h = makehuristikdict()

    q.push(start, 0)

    distance[start] = 0

    path[start] = None

    expandedList = []

    while (q.isEmpty() == False):

        current = q.pop()

        expandedList.append(current)

        if (current == end):

            break

        for new in romania[current]:

            g\_cost = distance[current] + int(new.distance)

            # print(new.city, new.distance, "now : " + str(distance[current]), g\_cost)

            if (new.city not in distance or g\_cost < distance[new.city]):

                distance[new.city] = g\_cost

                f\_cost = g\_cost + heuristic(new.city, h)

                q.push(new.city, f\_cost)

                path[new.city] = current

    printoutput(start, end, path, distance, expandedList)

def printoutput(start, end, path, distance, expandedlist):

    finalpath = []

    i = end

    while (path.get(i) != None):

        finalpath.append(i)

        i = path[i]

    finalpath.append(start)

    finalpath.reverse()

    print("Path From")

    print("\tArad => Bucharest")

    print("=======================================================")

    print("Exploreable cities : " + str(expandedlist))

    print("The number of possible cities: " + str(len(expandedlist)))

    print("=======================================================")

    print("The city that is passed the shortest distance: " + str(finalpath))

    print("Number of cities passed: " + str(len(finalpath)))

    print("Total Distance : " + str(distance[end]))

def main():

    src = "Arad"

    dst = "Bucharest"

    makedict()

    astar(src, dst)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

***romania\_sld.txt***

Arad, 366

Bucharest, 0

Craiova, 160

Dobreta, 242

Eforie, 161

Fagaras, 176

Giurgiu, 77

Hirsowa, 151

Lasi, 226

Lugoj, 244

Mehadia, 241

Neamt, 234

Oradea, 380

Pitesti, 100

Rimnicu Vilcea, 193

Sibiu, 253

Timisoara, 329

Urziceni, 80

Vaslui, 199

Zerind, 374

***romania.txt***

Arad,Zerind, 75

Arad,Sibiu, 140

Arad,Timisoara, 118

Zerind,Oradea, 71

Oradea,Sibiu, 151

Timisoara,Lugoj, 111

Sibiu,Fagaras, 99

Sibiu,Rimnicu Vilcea, 80

Lugoj,Mehadia, 70

Fagaras,Bucharest, 211

Rimnicu Vilcea,Pitesti, 97

Rimnicu Vilcea,Craiova, 146

Mehadia,Dobreta, 75

Bucharest,Pitesti, 101

Bucharest,Urziceni, 85

Bucharest,Giurglu, 90

Pitesti,Craiova, 138

Craiova,Dobreta, 120

Urziceni,Hirsova, 98

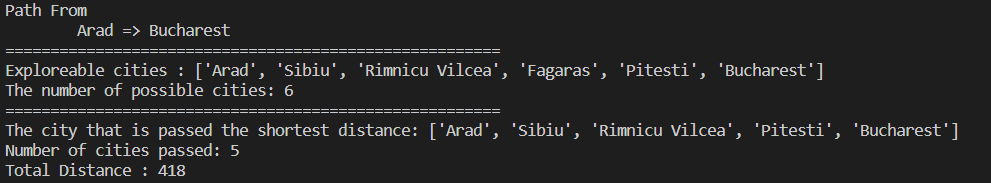
Urziceni,Vaslui, 142

Hirsova,Eforie, 86

Vaslui,Lasi, 92

Lasi,Neamt, 87

***Output:***

******

**Conclusion:**

*Implemented A\* search algorithm for Romanian map problem.*