PATH FINDING

A MINOR PROJECT REPORT

Submitted by

GUDARI SAI PRASAD – RA2011003011112
BHOGI SARANYA – RA2011003011100
KRISHNA PRAKASH – RA2011003011092

Faculty Name: S INIYAN

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

of

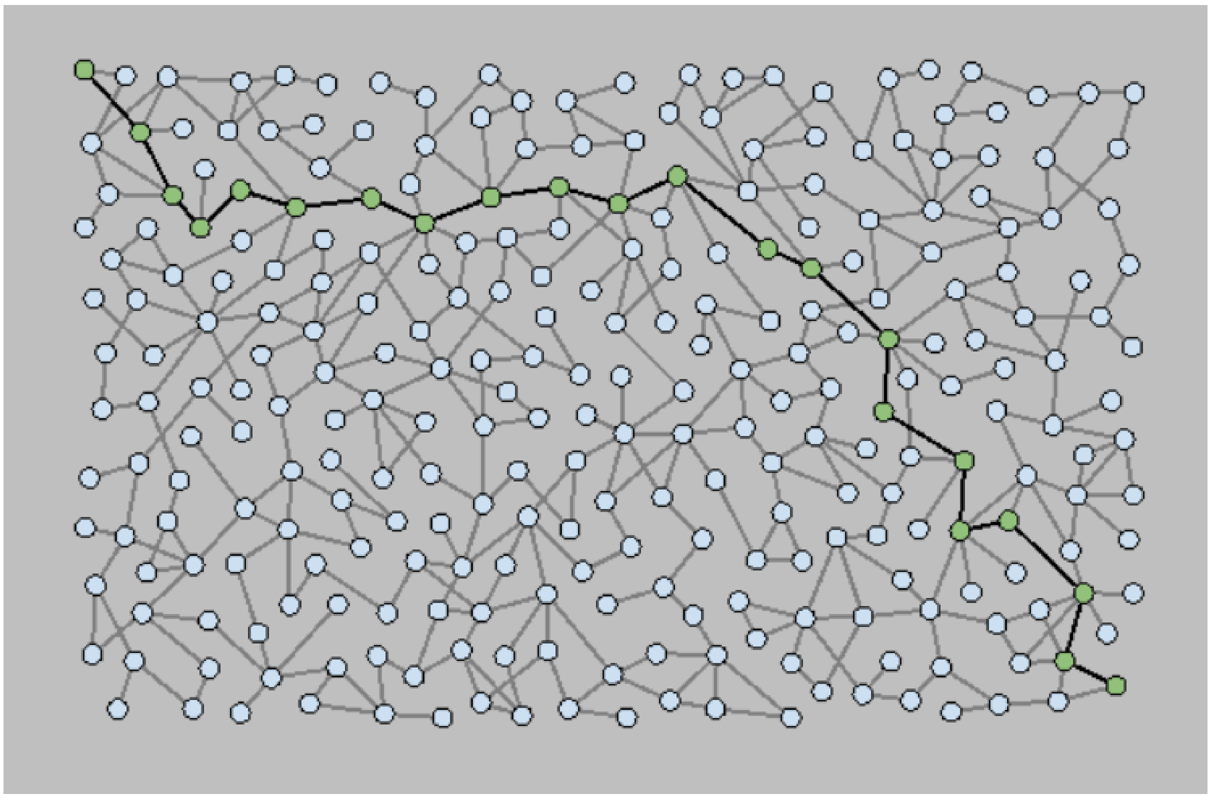
FACULTY OF ENGINEERING AND TECHNOLOGY

S.R.M. Nagar, Kattankulathur, Kancheepuram District

June,2022

DAA PROJECT

PATH FINDING



Contents

ROLE TABLE

Name	Reg. No.	Role
SAI PRASAD GUDARI	RA2011003011112	**CODER**
KRISHNA PRAKASH	RA2011003011092	**TESTER**
BHOGI SARANYA	RA2011003011100	**DOCUMENTATION**

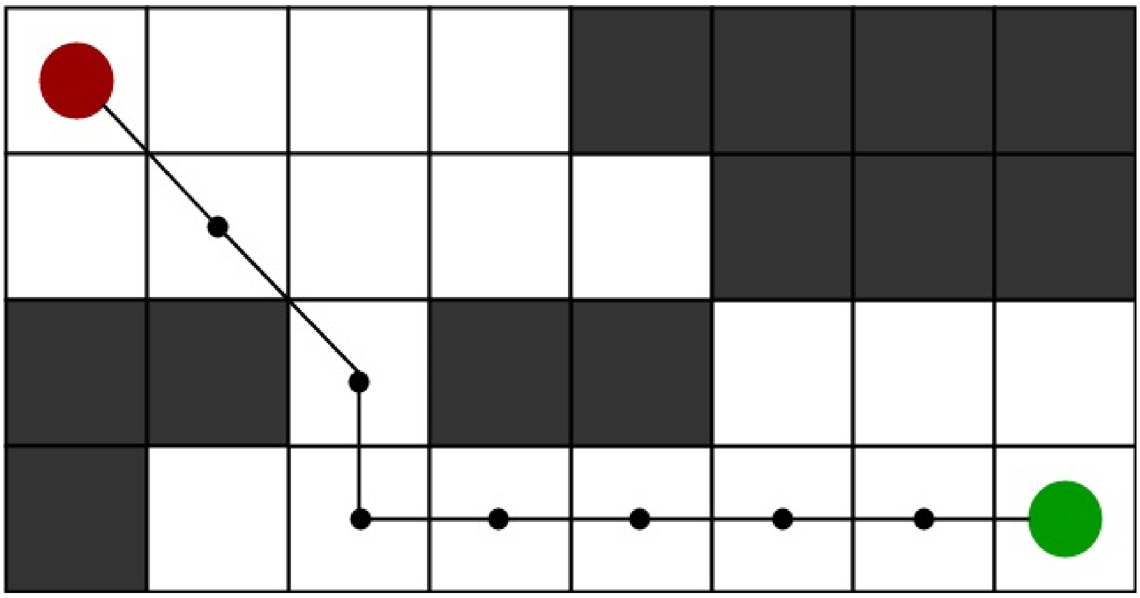
PROBLEM DEFINITION:

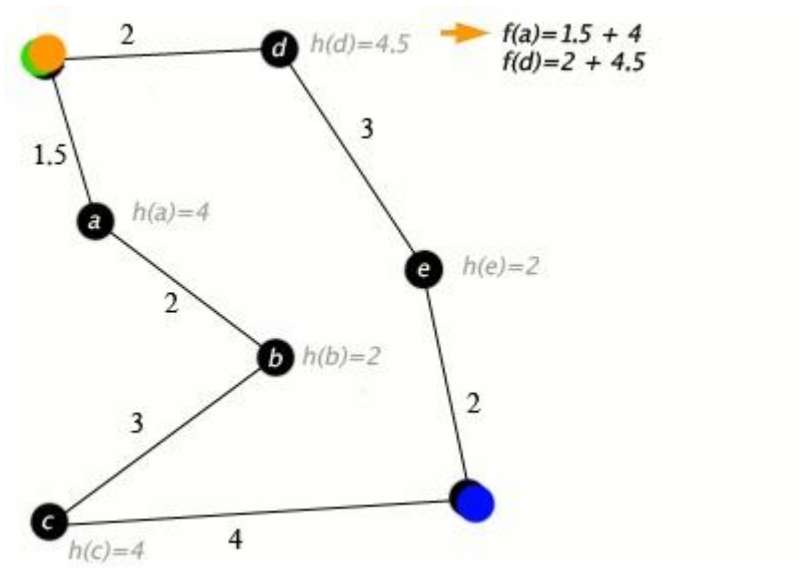
We will be executing PATH FINDING by coding methods- ASTAR

PROBLEM STATEMENT:

We have given two points in a graphs with multiple paths between them we need to find the shortest one.

Pathfinding or pathing is the plotting, by a computer application, of the shortest route between two points. It is a more practical variant on solving mazes.





PROBLEM EXPLANATION AND ITS APPROCH:

The Approch forn this problem is GRAPH TRAVERSAL

The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E)

We need to traverse nodes in the graph and we have to find the shortest path.

A-star (also referred to as A*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.

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's 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)
```

CODE:

[5:54 pm, 16/06/2022] Sai Prasad Srm: from collections import deque

```
class Graph:

    # example of adjacency list (or rather map)

    # adjacency_list = {

    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],

    # 'C': [('D', 12)]

    # }


    def _init_(self, adjacency_list):

        self.adjacency_list = adjacency_list


    def get_neighbors(self, v):

        return self.adjacency_list[v]


    # heuristic function with 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_node, stop_node):

        # open_list is a list of nodes which have been visited, but who's
neighbors
```

```
# haven't all been inspected, starts off with the start node
# closed_list is a list of nodes which have been visited
# and who's neighbors have been inspected
open_list = set([start_node])
closed_list = set([])

# g contains current distances from start_node to all other nodes
# the default value (if it's not found in the map) is +infinity
g = {}

g[start_node] = 0

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

while len(open_list) > 0:
    n = None

    # find a node with the lowest value of f() - evaluation function
    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 the current node is the stop_node
# then we begin reconstructin the path from it to the start_node
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 all neighbors of the current node do
for (m, weight) in self.get_neighbors(n):
        # if the current node isn't in both open_list and closed_list
        # add it to open_list and note n as it's parent
        if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight

        # otherwise, check if it's quicker to first visit n, then m
        # and if it is, update parent data and g data
        # and if the node was in the closed_list, move it to open_list
        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)

    # remove n from the open_list, and add it to closed_list
    # because all of his neighbors were inspected
    open_list.remove(n)
    closed_list.add(n)
```

```
        print('Path does not exist!')

        return None

[5:54 pm, 16/06/2022] Sai Prasad Srm: 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')
```

Design Techniques:

In computer science, GRAPH TRAVERSAL (also known as graph search) refers to **the process of visiting (checking and/or updating) each vertex in a graph**. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

COMPLEXITY ANALYSIS:

- Time-Complexity: O(E). Where E= Edges
- Auxiliary Space: O(V). Where V=Vertices

CONCLUSION:

We have created a solution for path finding. It helps us to write logic and maintain clean structure in code.