THE MARS COLONIZATION MISSION

NAVIGATE THE MARS ROVER

-BY TEAM MANGAL- YATRI

OUR TEAM

NIKHIL NISCHAL 2018EEB1169 IIT ROPAR	PREETESH VERMA 2018EEB1171 IIT ROPAR	GARIMA SONI 2018CSB1089 IIT ROPAR

INTRODUCTION

As a part of the Microsoft Engage 2020 program we were asked to help the Mars Curiosity Rover and find the shortest path between two points while avoiding obstacles on the way and to present it with a web interface.The Algorithm has been written in JavaScript and the web interface has been developed using HTML5,CSS, Node JS and Javascript.It runs on Node.js and is capable of supporting all the browsers.

It aims to provide a path-finding library that can find the path between two points(Source and destination) while avoiding the obstacles.

PATH FINDING ALGORITHM

At its core, a pathfinding algorithm seeks to find the shortest path between two points.This application visualises various pathfinding algorithms in action.

Here, We are using all the algorithms for a 2D grid.

Currently, we are using 5 path finding Algorithms -

- A* - AStar
- Dijkstra
- Greedy Best-First Search
- Breadth-First Search (BFS)
- Depth-First Search(DFS)

Note : Some algorithms are weighted(Dijkstra, Bellman Ford,etc) and some are unweighted(BFS,etc). Unweighted algorithms don't take turns or weight nodes into account, whereas weighted ones do. Also, not all algorithms guarantee the shortest path.

ABOUT THE ALGORITHMS

- **A* Search (Weighted)** - The best path finding Algorithm. It uses heuristics to guarantee the shortest path much faster than Dijkstra's Algorithm.
- **Dijkstra's Algorithm (Weighted)** - The father of path finding algorithms - guarantees the shortest path.
- **Greedy Best-first Search Algorithm(Weighted)** - a faster, more heuristic heavy version of A*does not guarantee the shortest path.
- **Breadth-First Search (Unweighted)** - A great algorithm, guarantees the shortest path.
- **Depth-First Search(Unweighted)** - A very bad algorithm for path finding, does not guarantee the shortest path.

ADDING WALLS

We can add walls by clicking on the grid .Here, Walls are impenetrable, meaning that a path cannot cross through them.

DETAILED ANALYSIS OF ALGORITHMS

1. **A* Search Algorithm -** It is an informed search or best-first search meaning it is

   formulated in terms of weighted graphs from a specific starting node of a graph and aims to find the smallest cost(or least distance travelled , shortest time). It does this by maintaining a tree of paths originating at the start node and extending those edges at a time until termination condition is satisfied.It is one of the best and popular techniques used in path-finding and graph traversals.It is really a smart algorithm which separates it from other conventional algorithms.

   How?

At each iteration of its main loop, A* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes

f(n) = g(n) + h(n)

Where **n** is the next node on the path, ***g*(*n*)** is the cost of the path from start node to *n*, and ***h*(*n*)** is a heuristic function that estimates cost of the cheapest path from *n* to the goal.

Algorithm -

1. Initialize the open list.

2. Initialize the closed list and 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 successor.**g** = q.**g** + distance between

         successor and q successor.**h** = distance from goal to successor

      ii) if a node with the same position as successor is in the OPEN list which has a

         lower **f** than successor, skip this successor.

      iii) if a node with the same position as successor is in the CLOSED list which
      . has 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)

2. **Dijkstra's Algorithm -** It is a greedy Algorithm.

At every step of the Algorithm, we find a vertex which is in the other set (not yet included) and has minimum distance from the source.It is a very famous algorithm for finding the shortest path(positive weight edges only).

Algorithm -

1)Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While *sptSet* doesn't include all vertices

a) Pick a vertex u which is not there in *sptSet* and has minimum distance value.

b) Include u to *sptSet*.

c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

3. **Greedy Best-first Search Algorithm**

It always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the

best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Algorithm -

1. Place the starting node into the OPEN list.
2. If the OPEN list is empty, Stop and return failure.
3. Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
4. Expand the node n, and generate the successors of node n.
5. Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
6. For each successor node, the algorithm checks for the evaluation function f(n), and then checks if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
7. Return to Step 2.

4. Breadth-First Search -

BFS is a traversing algorithm where we start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node).

Algorithm -

1. Create an empty queue and create two sets for checking visited and unvisited elements.
2. Push the source node into the queue.
3. Mark the source as visited.
4. While q is not empty :
    - Initialize the top of the queue as tp and pop it.
    - Traversing all neighbours of tp
    - If they are not visited - Push them into the queue and mark them as visited.

5. Depth-first Search -

DFS is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.

Algorithm:

a. Create a recursive function that takes the index of node and a visited array.

b. Mark the current node as visited and print the node.

c. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent nodes.

A FLOW CHART DESCRIBING THE WALK THROUGH OF THE APP.