Maze-Problem
LOGIC OF INFERENCE (IT485)

1.Samarth Nasit - 201501082
2.Meet Goti - 201501093
3.Jay Patel - 201501186

1.What is maze?

Maze is a puzzle with one starting-point , many dead-ends and one (more than one possible) ending point or destination. There can be more than one path from source to destination.Normally the walls are fixed of the mazes and we here only focus on fixed-wall type of mazes. There are many mazes being used for psychological experiments.

2.Maze-solving Algorithms

There are many maze-solving algorithms and all of them works fine if the maze is valid. All of them mostly run on trial-and-error manner.
We have two types of maze-problem:

1. Known-maze
2. Unknown-maze

a. Known-maze:

If we already know the maze then this problem can simply be converted into graph problem with and path-finding from one node to another node/nodes. All the shortest-path algorithms and path-finding algorithms can be applied for the maze.

B. Unknown-maze:

If we don't know maze prior then things get tricky because we can not tell if the path we are taking is not dead-ending or connecting to already visited node or tile.

We have implemented algorithm for both type of mazes.We have implemented simple DFS and A-star algo with two types of heuristic functions. DFS can be applied on both type of above mentioned algorithms while A-star is mostly focused on unknown kind of mazes.

On basis of solving the maze there can be two type:

1. There's a maze-runner that is running through the maze. This is more focused upon unknown-mazes.
2. The maze-runner first find the path and then enter the maze. More focused upon known-mazes

1.Depth first search:

This algorithm go through every possible path and tries to find the destination through every path possible from the source. It is the most path-finding algorithm and works for any type of maze possible.

2.Wall-Follower Algorithm:

Wall-follower algorithm works like dfs but at turning or cross-road it prefers one side over other sides. If i am using this algorithm with right side preference then I will always prefer right turn over straight or right.

Why we should have used this algorithm:

Better than DFS in terms of steps taken.

Why we didn't use this algorithm:

This algorithm assumes that the target is in the outer-boundary of the maze and if we arrive at the first starting point then declares "No path".So this algorithm will not work on the mazes with destination not in outer-boundary

3.Dead-End filling:

In this algorithm we mark every dead-end in the maze and remained path is the final answer-path of the algorithm.

Why we didn't use this algorithm:

This algorithm assumes that we already knows the path and can not be implemented on a maze-runner running in unknown-maze.

4.Breadth first search:

In this algorithm we track more than one path simultaneously and try to find the final destination. This algorithm gives the shortest path to the destination from source.

Why we didn't use this algorithm:

If we are developing a maze-runner robot then it is impossible to track more than one path simultaneously. Every time to run through another path it have to go back to the another path-end.

5.A* Algorithm:

A* algorithm is famous algorithm and can be seen as updated version of breadth-first-search. In this in place of queue there will be a priority queue and all the running paths will have path according to the heuristic function that is being used. A* algorithm assumes that we already know the at-least position of the destination. A* not always give faster solution because it is approximation algorithm.We can use two types of heuristic functions.

a. Manhattan-distance: $|x_1 - x_2| + |y_1 - y_2|$
b. Euclidean-distance: $\mathrm{sqrt}((x_1 - x_2)^2 + (y_1 - y_2)^2)$

3.Implementation:

We have implemented two algorithms DFS and A*. For A* we have chosen Manhattan-distance as heuristic function. We have implemented the UI in React.js and tried to keep UI as simple as possible to use and our implementation is for mazes with maximum four crossing roads and implementation is matrix based and mostly 2D based.

In DFS algorithm implementation we have not choose to take random path from every point because it is a little hard to find a random function and also “perfect randomness is myth”. We are choosing direction in Up , Left , Right , Down as sequence of preference. DFS can also be used without specifying the destination point. It will return the first visited exit-point of the maze.

A* algorithm implementation can not be used if the destination is not defined prior because then we cannot predict which running current path is better. We have used manhattan-distance from destination to current-running path end as Heuristic function.

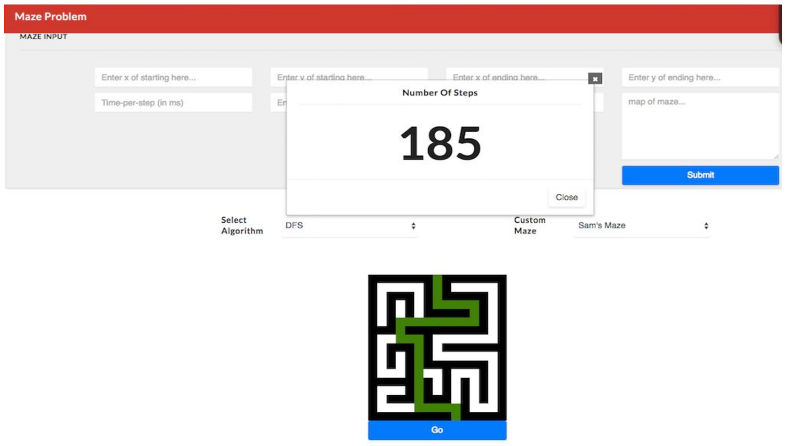
We are trying to improve the whole rendering time problem with considering cross-point to cross-point consideration rather than considering every tile.

4.Time Complexity:

We have implemented two algorithms DFS and A*. For A* we have implement two function Manhattan-distance & Euclidean-distance as heuristic function.

Using different algorithms like DFS , A*-Manhattan , A*-Euclid ,we can show the time complexity of different algorithms on different Maze in terms of steps. The time complexity of different algorithms are same and efficiency of algorithms will depends on Maze.

Below are the some Mazes and it’s time complexity in terms of steps :


Maze Problem
MAZE INPUT
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
Time-per-step (in ms)
map of maze...
Submit
Close
Go

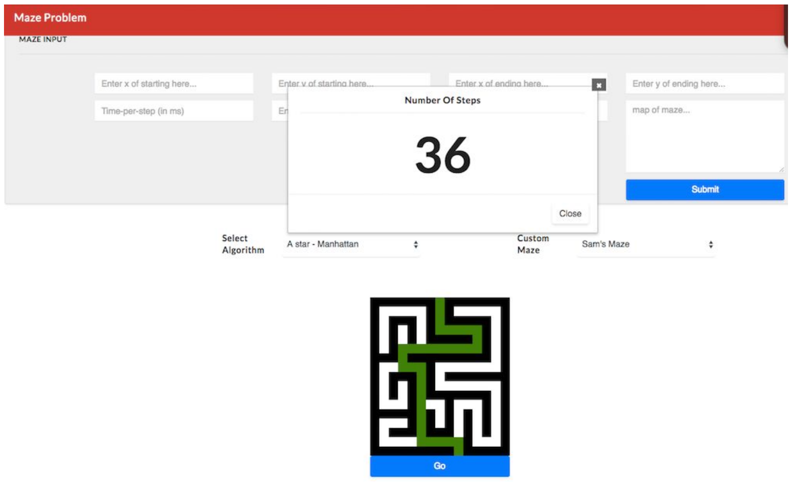
Number Of Steps

185

Select Algorithm: DFS

Custom Maze: Sam's Maze

DFS


Maze Problem
MAZE INPUT
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
Time-per-step (in ms)
map of maze...
Submit
Close
Go

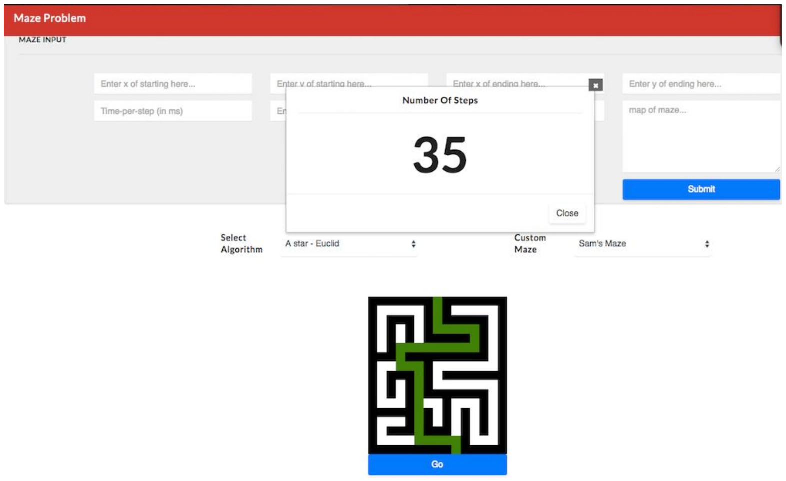
Number Of Steps

36

Select Algorithm: A star - Manhattan

Custom Maze: Sam's Maze

A*-Manhattan


Maze Problem
MAZE INPUT
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
Time-per-step (in ms)
map of maze...
Submit
Close
Go

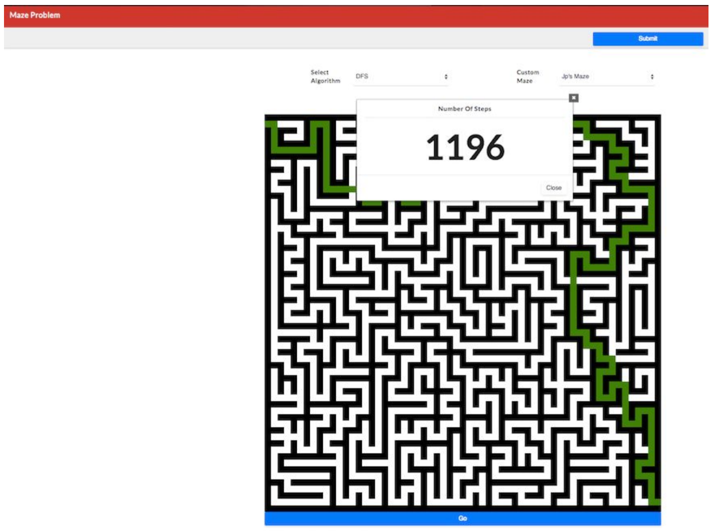
Number Of Steps

35

Select Algorithm: A star - Euclid

Custom Maze: Sam's Maze

A*-Euclid



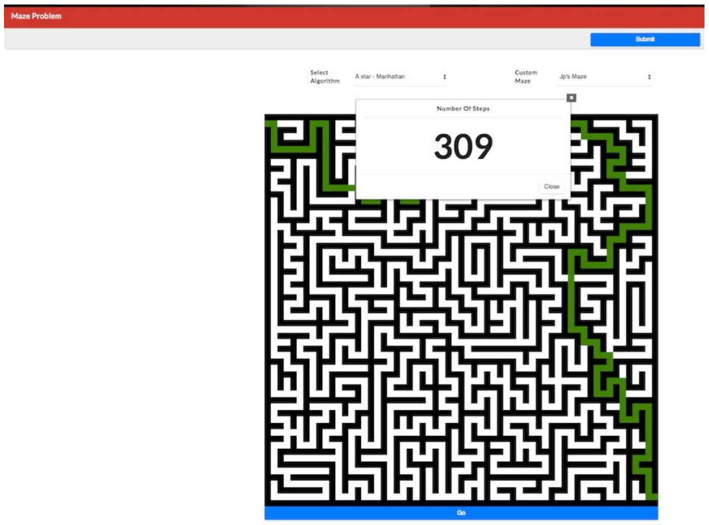
Select Algorithm: DFS

Custom Maze: Jp's Maze

Number Of Steps

1196

DFS



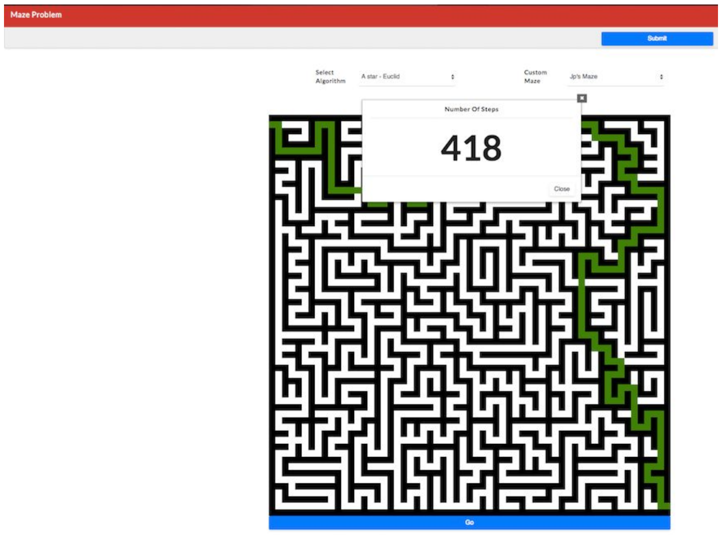
Select Algorithm: A star - Manhattan

Custom Maze: Jp's Maze

Number Of Steps

309

A*-Manhattan



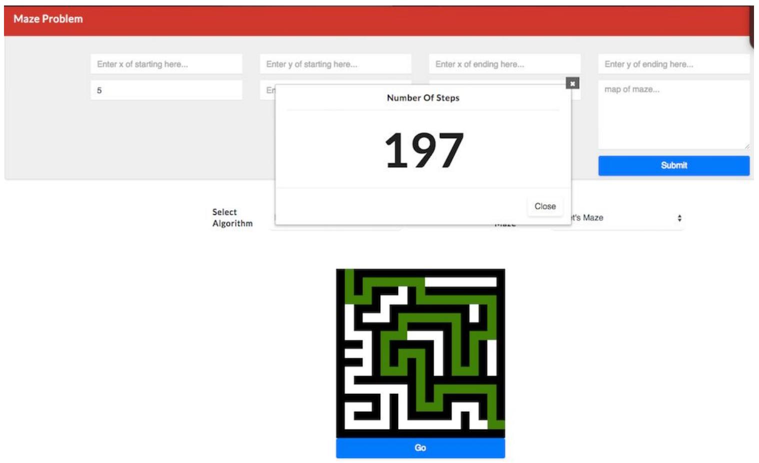
Select Algorithm: A star - Euclid

Custom Maze: Jp's Maze

Number Of Steps

418

A*-Euclid

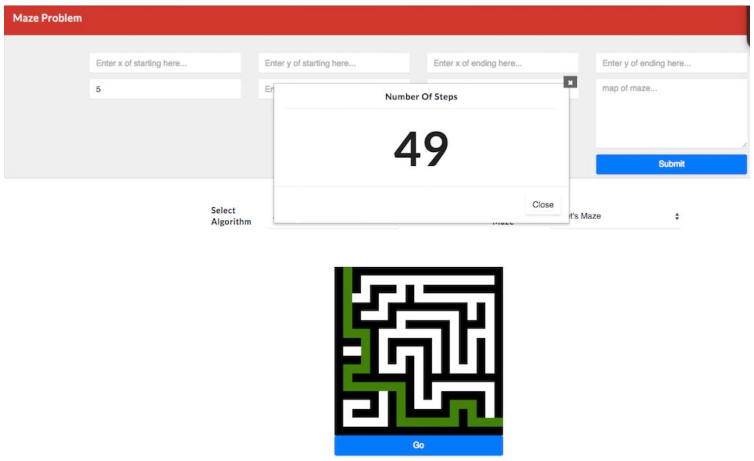

Maze Problem
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
map of maze...
Submit
Select Algorithm
t's Maze
Close
Go

Number Of Steps

197

5

DFS

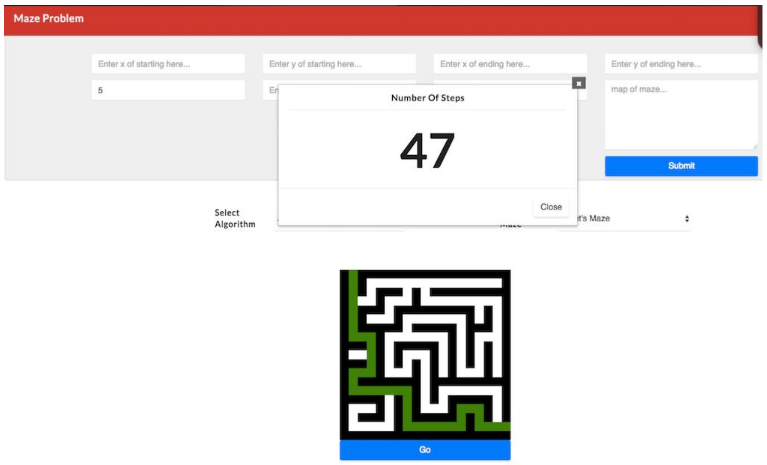

Maze Problem
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
map of maze...
Submit
Select Algorithm
t's Maze
Close
Go

Number Of Steps

49

5

A*-Manhattan


Maze Problem
Enter x of starting here...
Enter y of starting here...
Enter x of ending here...
Enter y of ending here...
map of maze...
Submit
Select Algorithm
t's Maze
Close
Go

Number Of Steps

47

5

A*-Euclid

5.Limitation:

- The implementation is only valid for matrix like mazes and will not work if the maze have more than four roads joining at a point.
- Algorithms implemented does not give surety to give the fastest route from source to destination.
- Since A* is running more than one path simultaneously it is not being suggested for actual maze-runner.
- Rendering is little slow for large size of the matrix or maze.