Path-finding Algorithms and Solving Mazes

Oliver Temple

October 12, 2021

Contents

1 analysis

1.1 Project Description

Path finding algorithms are essential in many aspects computer science, especially in games and simulations. However, they can be complex things that are hard to visualize, especially when being taught about them. Path finding algorithm visualizers do exist, however, I feel that none of them are perfect, and that they lack features.

Because of this, I am going to make a path finding visualizer that ticks all of the boxes. It will be simple and easy to use, have advanced customization and have the ability to run a path finding algorithms given by the user.

1.2 Research

1.2.1 Overview

To complete this project, I will need a strong understanding of maze generation and path finding algorithms, how they work and how to model them, knowledge of react and javascript to create a website for the visualization as well as user opinions on what features are needed.

1.2.2 Research Log

Class I was introduced to path finding algorithms in one of my lessons, where we learned what they were used for and some examples, as well as how they can be modeled. For example, modelling the maze as a graph with weighted nodes of 1 and 0, for the walls and space respectively. The algorithms we looked at were Dijkstra's and A*. We also looked at Prim's algorithm, recursive backtracking, depth fist search and Kruskal's algorithms for maze generation.

1.3 Project Background

1.3.1 Current Systems

1.3.1.1 Example 1 https://clementmihailescu.github.io/Pathfinding-Visualizer

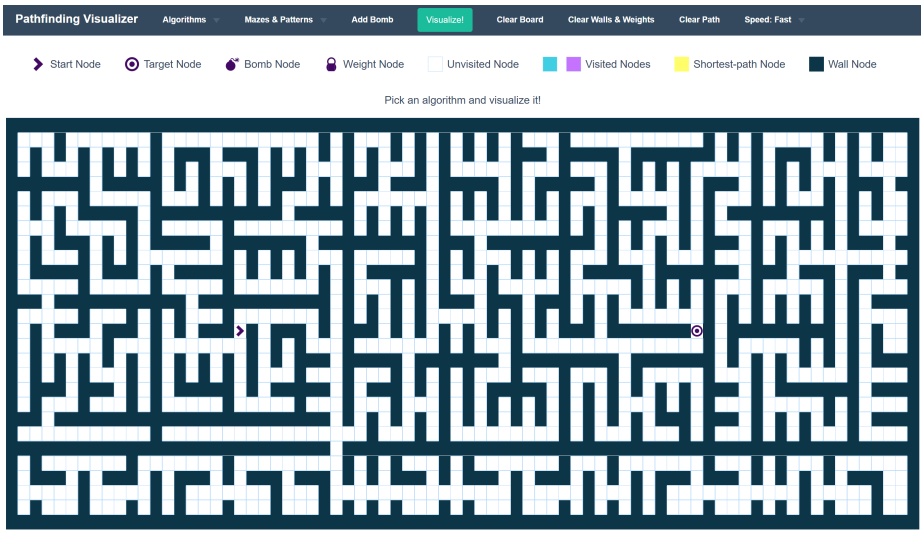
In this example, mazes can be generated with various algorithms, as well as being drawn by the user. The mazes can also be edited after they have been generated. The available maze generation algorithms are:

- Recursive Division

- Recursive Division (vertical skew)
- Recursive Division (horizontal skew)
- Basic Random Maze
- Basic Weight Maze
- Simple Stair Pattern

These mazes can then be solved with a number of different path finding algorithms. The available path finding algorithms are:

- Djikstra's Algorithm
- A* Search
- Greedy Best-first Search
- Swarm Algorithm
- Convergent Swarm Algorithm
- Bidirectional Swarm Algorithm
- Breadth-first Search
- Depth-first Search



pros

- Many different algorithms to choose from.
- Start and end nodes can be moved.
- Maze can be altered.
- If nodes are moved after visualization has run, then the visualization will update.
- "Bomb" node, adds a via point that the path must go through.

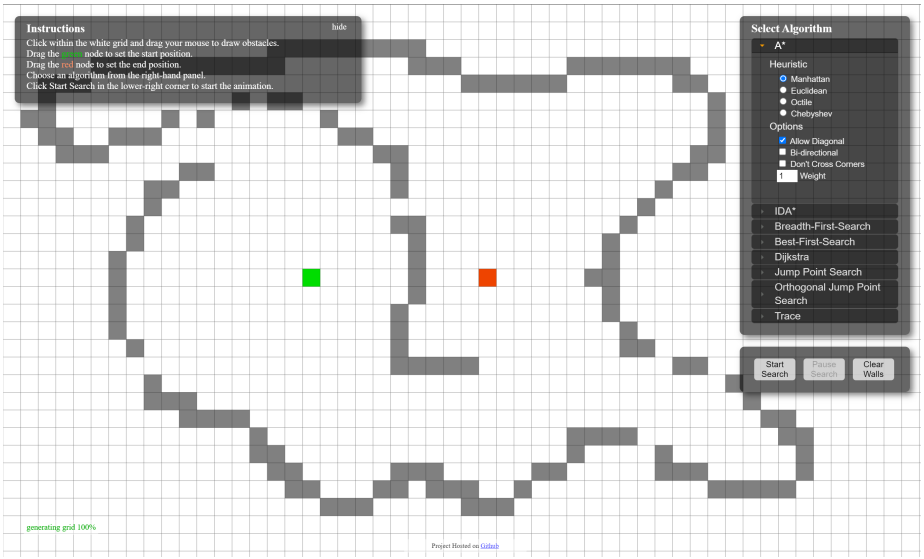
cons

- The visualization is too slow.
- If maze is altered by user after visualization has run, then the visualization will not update.

1.3.1.2 Example 2 https://qiao.github.io/PathFinding.js/visual/

In this example, mazes have to be drawn by the user. The maze can then be solved with a number of different algorithms, however, these algorithms have more choice. For example, in the A* option, you can change the heuristic that is used. The available algorithms are:

- A*
- IDA*
- Breadth-First-Search
- Best-First-Search
- Dijkstra
- Jump Point Search
- Orthogonal Jump Point Search
- Trace



pros

- More options to choose from within each algorithm.

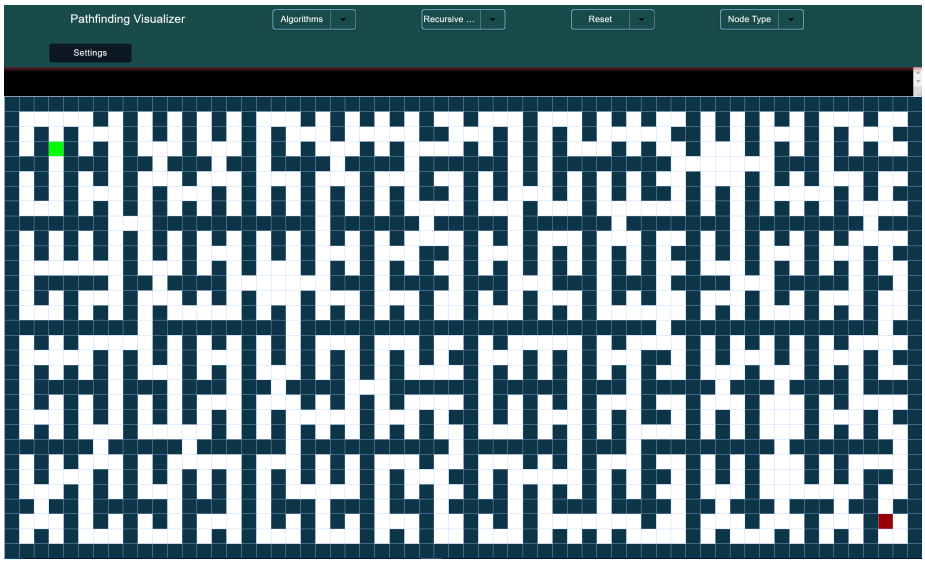
cons

- No maze generation.
- Visualization does not update when maze or start/finnish nodes are changed.

1.3.1.3 Example 3 https://pathfindout.com/

In this example, mazes can be generated or drawn, however, mazes can only be generated with the recursive division algorithm. There are fewer path finding algorithms to solve the mazes than the others. The available algorithms are:

- Dijkstra's Algorithm
- A* Search
- Breadth First Search
- Depth First Search



pros

- Different weighted nodes available.
- Shows how many nodes visited.
- Shows final path length.
- Data structure for some algorithms can be changed.
- Weights of specific node types can be changed.
- Node size can be changed.

cons

- Sometimes generates mazes that cannot be solved.
- Cannot edit maze after visualization has run.
- Only one maze generation algorithm.
- Fewer path finding algorithms to solve the maze.

1.3.2 Proposed Solution

I will make a path-finding visualization that encompasses as many of the merits of the existing solutions as possible, while also tackling as many of the drawbacks.

- Research the different algorithms needed and write the corresponding pseudoscode.

- Create a mockup of the user interface.
- Create the user interface in react native.
- Code the algorithms in javascript.
- Add additional features suggested by end users.
- Test visualization and check that it meets all of the objectives.

1.3.3 Prospective Users

The users of this system will most likely consist of teachers and students who are learning about path finding algorithms as it will clearly show how the algorithms function.

1.3.3.1 Questions To Users I asked some prospective users some questions about what they would like to see in the visualizer.

Questions

Q1. What algorithms would you like to see for maze generation?

Q2. What algorithms would you like to see for solving the maze?

Q3. What additional features would you like to see in the path finding visualization?

Q4. Would it be useful to be able to write your own path finding algorithms that can be run in the visualization?

Answers

A1. I would like to see recursive division and Prim's algorithms used for maze generation, as well as Kruskal's (less important). This is because Kruskal's and Prims are similar but each has their own advantages/disadvantages and recursive division is completely different.

A2. I would like to see Dijkstra's algorithm and A* search, as these are very popular algorithms and one is a heuristic.

A3. Option to add your own png image for the drawing of final path - "a sussy imposter running around the maze".

A4. I would find it useful to be able to code my own algorithms. This would allow me to use the visualization with other, lesser known algorithms that I may need.

The answers to these questions confirms what is required from the visualization, which will be reflected in the objectives. The need for contrasting algorithms is something that I will consider when deciding what algorithms to use.

1.4 Objectives

1.4.1 Generate Mazes

The website should be able to generate mazes using a number of different algorithms, including, but not limited to: Prim's algorithm, recursive division and random generation. There should also be a brief description of the algorithm that has been selected.

1.4.2 User Drawn Mazes

The user should be able to draw mazes and obstacles easily on the grid using the mouse.

1.4.3 Solve Mazes

The website should be able to solve mazes using a number of different algorithms, including, but not limited to: A* search, Dijkstra's algorithms, depth-first search and breadth-first search. There should also be a brief description of the algorithm that has been selected.

The website should also be able to take in an algorithm written by the user, and use it to solve the maze. This will be done by supplying documentation on what parameters need to be taken in and what will need to be returned from the function for the visualizer to work.

1.4.4 Update Visualization

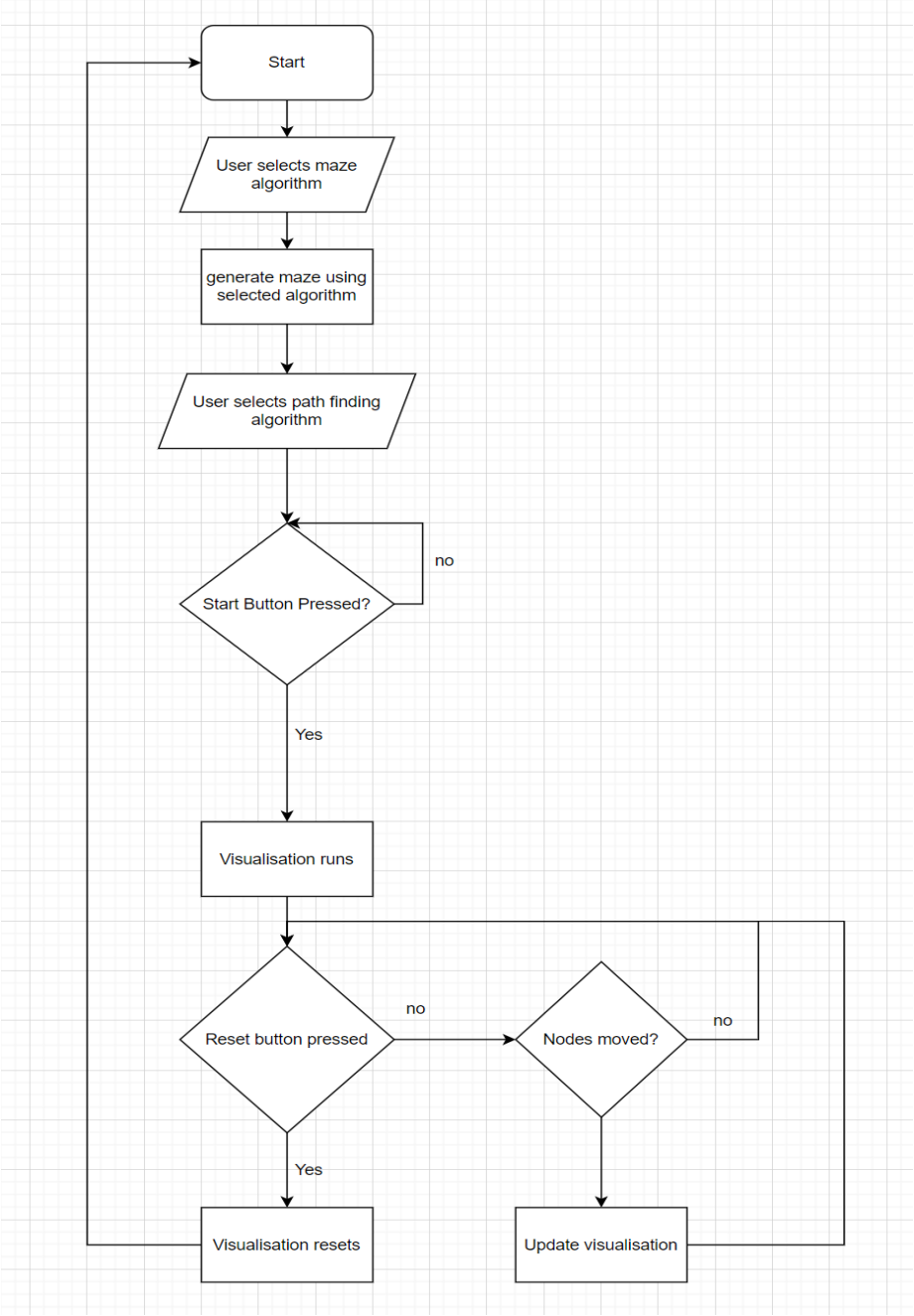
If the maze is altered or and nodes moved once the visualization has been run, then it should update without rerunning the visualization

1.4.5 Special Nodes

There should be "special" nodes that are different from walls or space. For example, nodes with different weights or a "via point" node that the path must go through.

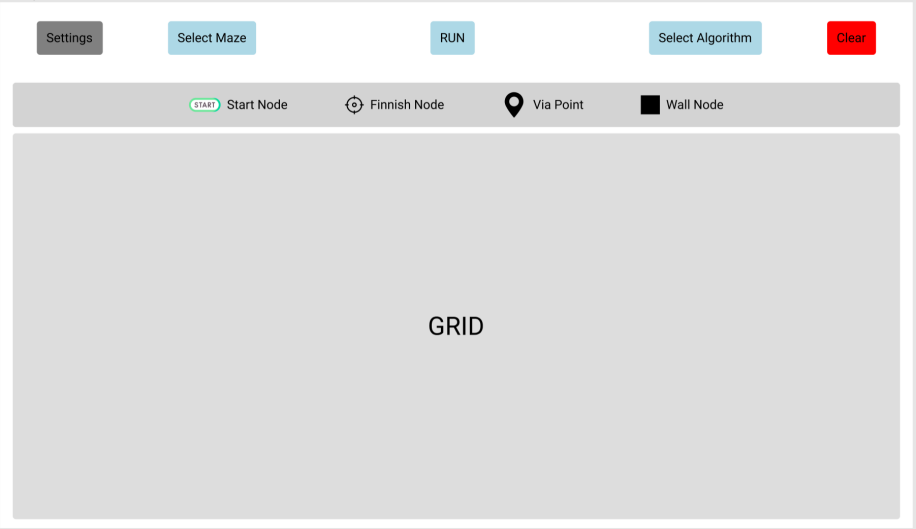
2 Documented Design

2.1 Flow Chart

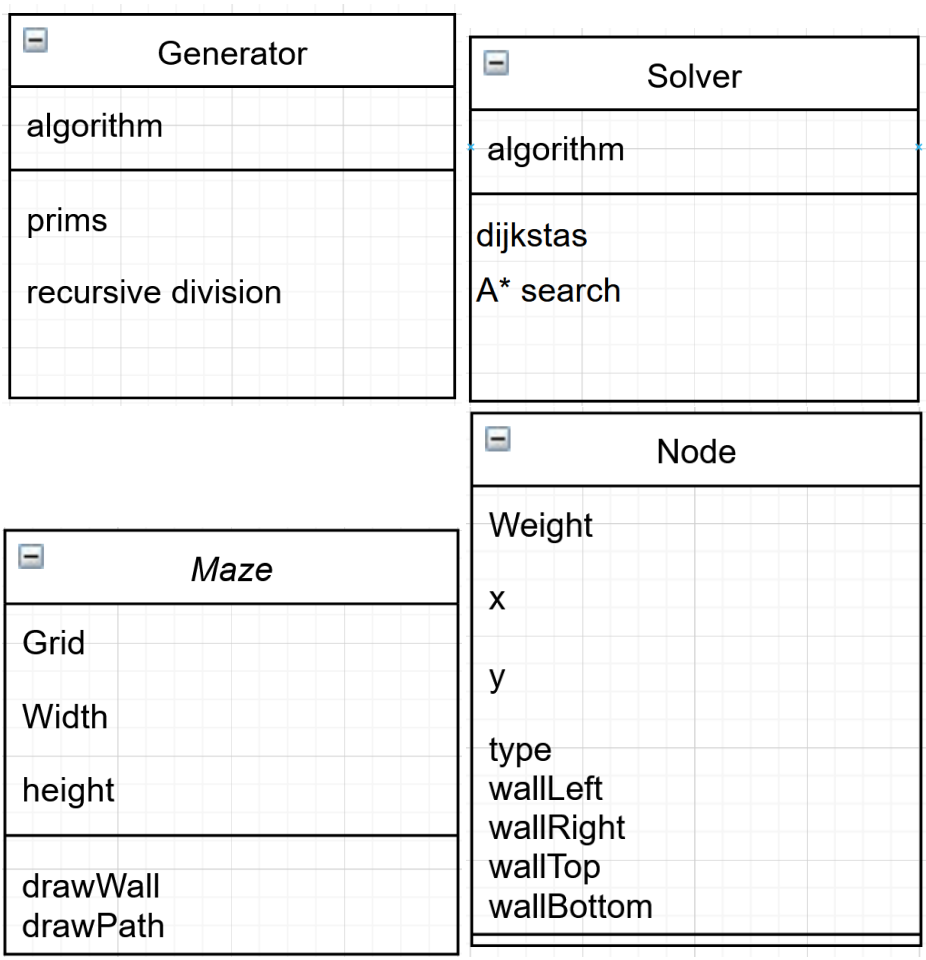


2.2 User Interface

I have used Figma to create a design mockup of how the user interface will look. This allows me to plan what user inputs will be needed.



2.3 Class Diagrams



2.4 Data Strucure

The Grid for the maze will be stored in a 2D array with each item being a Node with the information about it. For example, a 2x2 grid would look like: [[Node, Node], [Node, Node]]

2.5 Algorithms

2.5.1 Maze generating algorithms

2.5.1.1 prims

```
function prims(Grid){
```

```
let start = [0,0]
let inMaze = [start]
let frontier = [
    [[start[0],start[1]+1],["left"]],
    [[start[0]+1, start[1]],["top"]]
]

while (frontier.length > 0){
    let new = frontier.pop(randint(0, frontier.length - 1))
    let toAdd = new[0]
    let wall = new[1][randint(0, new[1].length - 1)]
    inMaze.push(toAdd)

    if (wall == "bottom"){
        Grid.grid[toAdd[0]][toAdd[1]].wallBottom = False
    }

    if (wall == "left"){
        Grid.grid[toAdd[0]][toAdd[1]].wallLeft = False
    }

    if (wall == "top" and toAdd[0] > 0){
        Grid.grid[toAdd[0]-1][toAdd[1]].wallBottom = False
    }

    if (wall == "right" and toAdd[1] < Grid.width){
        Grid.grid[toAdd[0]][toAdd[1]+1].wallLeft = False
    }

    let possible = [
        [toAdd[0]-1, toAdd[1]],
        [toAdd[0]+1,toAdd[1]],
        [toAdd[0],toAdd[1]-1],
        [toAdd[0], toAdd[1]+1]
    ]

    for i in range 4{
        p = possible[i]
        walls = ["bottom", "top", "right", "left"]
        wall = walls[i]

        if (0<=p[0]<Grid.height and 0<=p[1]<Grid.width){
            if p not in inMaze{
                found = False
                for v in frontier{
                    if (v[0] == p){
```

```
                            v[1].push(wall)
                            found = True
                        }
                    }
                    if not found{
                        frontier.push([p, [wall]])
                    }
                }
            }
        }
    }
}
```

3 Evidence of Completeness

4 Technical Solution

5 Testing

6 Evaluation