

# Path-finding Algorithms and Solving Mazes

Oliver Temple

December 2, 2021

## Contents

<b>1</b>	<b>analysis</b>	<b>4</b>
1.1	Project Description . . . . .	4
1.2	Research . . . . .	4
1.2.1	Overview . . . . .	4
1.2.2	Research Log . . . . .	4
1.2.2.1	Maze Generating Algorithms . . . . .	4
1.2.2.2	Path Finding Algorithms . . . . .	5
1.3	Project Background . . . . .	6
1.3.1	Current Systems . . . . .	6
1.3.1.1	Example 1 . . . . .	6
1.3.1.2	Example 2 . . . . .	7
1.3.1.3	Example 3 . . . . .	8
1.3.2	Proposed Solution . . . . .	10
1.3.3	Prospective Users . . . . .	10
1.3.3.1	Questions To Users . . . . .	10
1.4	Objectives . . . . .	11
1.4.1	Generate Mazes . . . . .	11
1.4.2	Solve Mazes . . . . .	11
1.4.3	Customisation . . . . .	11
1.4.4	User written algorithms . . . . .	12
1.4.5	Update Visualization . . . . .	12
1.4.6	Special Nodes . . . . .	12
<b>2</b>	<b>Documented Design</b>	<b>12</b>
2.1	Visualization Structure . . . . .	12
2.2	User Interaction Flow Chart . . . . .	14
2.3	User Interface . . . . .	15
2.4	Classes Breakdown . . . . .	15
2.4.1	Generator . . . . .	15
2.4.2	Solver . . . . .	15
2.4.3	Maze . . . . .	16
2.4.4	Node . . . . .	16

2.5	Algorithms	17
2.5.1	Prims	17
2.5.2	Recursive Backtracking	17
2.5.3	Dijkstra's	17
2.5.4	Depth First Search	18
2.5.5	Breadth First Search	18
2.5.6	Greedy Search	18
2.6	Subroutine Breakdown API	19
2.6.1	generator.py: recursive_backtracking	19
2.6.2	generator.py: recursive_backtracking_run	19
2.6.3	generator.py: prims	19
2.6.4	solver.py: get_adjacent_paths	19
2.6.5	solver.py: dijkstra	19
2.6.6	solver.py: dfs	19
2.6.7	solver.py: bfs	20
2.6.8	solver.py: manhattan	20
2.6.9	solver.py: euclidean	20
2.6.10	solver.py: greedy	20
2.6.11	grid.py: Node: __init__	20
2.6.12	grid.py: Node: load	20
2.6.13	grid.py: Node: serialize	21
2.6.14	grid.py: Grid: __init__	21
2.6.15	grid.py: Grid: load	21
2.6.16	grid.py: Grid: serialize	21
2.6.17	grid.py: Grid: generateGrid	21
2.6.18	grid.py: Grid: printGrid	21
2.7	Subroutine Breakdown react app	21
2.7.1	App.js: App: Constructor	21
2.7.2	App.js: App: componentDidMount	21
2.7.3	App.js: App: setHeuristic	21
2.7.4	App.js: App: setSpeed	22
2.7.5	App.js: App: setSize	22
2.7.6	App.js: App: setStart	22
2.7.7	App.js: App: setEnd	22
2.7.8	App.js: App: setAlgorithm	22
2.7.9	App.js: App: setSolve	22
2.7.10	App.js: App: async should_solve	22
2.7.11	App.js: App: async clear_node_index	23
2.7.12	App.js: App: async fetchGrid	23
2.7.13	App.js: App: async solveGrid	23
2.7.14	App.js: App: async clearGrid	23
2.7.15	App.js: App: render	23
2.7.16	DisplayGrid.jsx: DisplayGrid: handelDrop	23
2.7.17	DisplayGrid.jsx: DisplayGrid: setDragObject	23
2.7.18	DisplayGrid.jsx: DisplayGrid: renderTable	23
2.7.19	DisplayGrid.jsx: DisplayGrid: render	24

2.7.20	DisplayNode.jsx: DisplayNode: handelDragStart . . . . .	24
2.7.21	DisplayNode.jsx: DisplayNode: handelDrop . . . . .	24
2.7.22	DisplayNode.jsx: DisplayNode: handelDragOver . . . . .	24
2.7.23	DisplayNode.jsx: DisplayNode: handelDragLeave . . . . .	24
2.7.24	DisplayNode.jsx: DisplayNode: render . . . . .	24
2.7.25	Settings.jsx: Settings: componentDidMount . . . . .	24
2.7.26	Settings.jsx: Settings: componentWillUnmount . . . . .	24
2.7.27	Settings.jsx: Settings: handelClickOutsize . . . . .	24
2.7.28	Settings.jsx: Settings: handelSizeChange . . . . .	25
2.7.29	Settings.jsx: Settings: setHeuristic . . . . .	25
2.7.30	Settings.jsx: Settings: renderSettings . . . . .	25
<b>3</b>	<b>Evidence of Completeness</b>	<b>25</b>
<b>4</b>	<b>Technical Solution</b>	<b>25</b>
4.1	Summary Of Skills Used . . . . .	25
4.2	The Full Code . . . . .	26
4.2.1	Python API: lambda_function.py . . . . .	26
4.2.2	Python API: grid.py . . . . .	28
4.2.3	Python API: generator.py . . . . .	29
4.2.4	Python API: solver.py . . . . .	32
4.2.5	React App: App.js . . . . .	36
4.2.6	React App: Menu.jsx . . . . .	40
4.2.7	React App: MenuKey.jsx . . . . .	41
4.2.8	React App: DisplayGrid.jsx . . . . .	41
4.2.9	React App: DisplayNode.jsx . . . . .	43
<b>5</b>	<b>Testing</b>	<b>45</b>
5.1	Test1: Pass . . . . .	46
5.2	Test2: Pass . . . . .	46
5.3	Test3: Pass . . . . .	47
5.3.1	Prim's Algorithm . . . . .	48
5.3.2	Recursive Backtracking . . . . .	49
5.4	Test4/Test5: Pass . . . . .	49
5.4.1	Dijkstra's Algorithm . . . . .	50
5.4.2	Depth First Search . . . . .	50
5.4.3	Breadth First Search . . . . .	50
5.4.4	Greedy Search Manhattan . . . . .	50
5.4.5	Greedy Search Euclidean . . . . .	50
5.5	Test6: Pass . . . . .	50
5.6	Test7 . . . . .	50
<b>6</b>	<b>Evaluation</b>	<b>50</b>

# 1 analysis

## 1.1 Project Description

Path finding algorithms are essential in many aspects of computer science, from computer games to solving complex real world problems, however they can be complex things to visualize, especially when learning about them for the first time. This project's aim to create a path finding visualization tool that will generate and solve mazes, and to provide a general understanding of the algorithms used.

There are a number of different algorithms that can be used to solve and generate mazes, and this project will focus on the more common ones like Dijkstra's, Depth First Search, Prim's and more. I intend to include algorithms that are different from each other to show the advantages and disadvantages of each.

## 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

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 first search and Kruskal's algorithms for maze generation.

#### 1.2.2.1 Maze Generating Algorithms

**Prim's** In class, we were taught about Prim's algorithm, and how it works. I used the information that our teacher gave us to write my own implementation of Prim's algorithm.

**Recursive Backtracking** When researching recursive backtracking I can across a website that said:

*Here's the mile-high view of recursive backtracking:*

1. Choose a starting point in the field.

2. *Randomly choose a wall at that point and carve a passage through to the adjacent cell, but only if the adjacent cell has not been visited yet. This becomes the new current cell.*
3. *If all adjacent cells have been visited, back up to the last cell that has uncarved walls and repeat.*
4. *The algorithm ends when the process has backed all they way up to the starting point.*

<https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking>  
 I used this along with another visualization that I found to write my own recursive backtracking algorithm.

### 1.2.2.2 Path Finding Algorithms

**Depth First Search** For the depth first search, I looked [https://isaacomputerscience.org/concepts/dsa\\_pathfinding\\_dfs\\_bfs?examBoard=ocr&stage=a\\_level](https://isaacomputerscience.org/concepts/dsa_pathfinding_dfs_bfs?examBoard=ocr&stage=a_level), a resource I often use for studying. I used their easy to understand description to write my own implementation of depth first search. Their website said:

*A depth-first search begins at the start node and then searches as far as possible down a branch of the graph, moving forward until there are no more nodes along the current branch to be explored. If the target node is found along the way, the search can stop. Otherwise, it must backtrack and find another branch to explore.*

*This process uses a stack as a supporting data structure to keep track of the nodes that have not been fully explored. As each node is discovered, it is added to the stack.*

**Breadth First Search** Isaaccomputerscience also have a page on breadth first search, also containing a simple description that I used to implement my own breadth first search.

*The algorithm starts searching at a designated start node and searches through all the adjacent nodes (neighbours) before moving on. You can think of this process as moving out in waves from a given point. Another simple way to visualise a breadth-first search algorithm is to imagine that you are making a cake one layer at a time. You can't add the next layer unless the previous one is complete.*

*This process uses a queue as a supporting data structure to keep track of the nodes that have not been fully explored. As each node is discovered, it is added to the queue.*

**Dijkstra's** For dijkstra's, i found this website:<https://daemianmack.org/posts/2019/12/mazes-for-programmers-dijkstras-algorithm.html>, that had a very good description of Dijkstra's algorithm.

1. *Determine the starting point of the grid.*
2. *Record the cost of reaching that cell: 0.*
3. *Find that cell's navigable neighbors.*
4. *For each neighbor, record the cost of reaching that neighbor: 1.*
5. *For each neighbor, repeat steps 3-5, taking care not to revisit already-visited cells.*

**Greedy Search** Having researched the other path finding algorithms, and learnt about greedy algorithms in class, I used the knowledge I already had to write my own greedy search algorithm.

## 1.3 Project Background

### 1.3.1 Current Systems

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

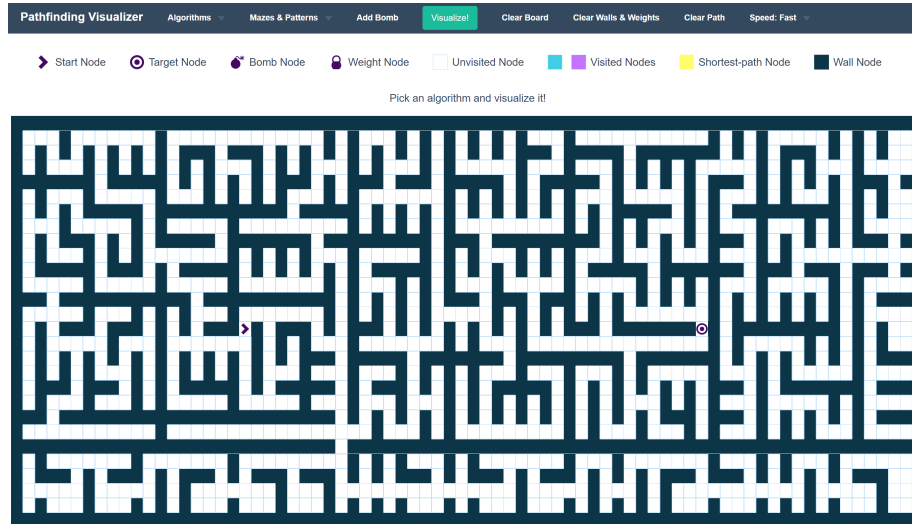
In this example, mazes can be generated with various algorithms, as well as being drawn by the user. The mazes can also be edited after they have been generated. The available maze generation algorithms are:

- 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:

- Dijkstra'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

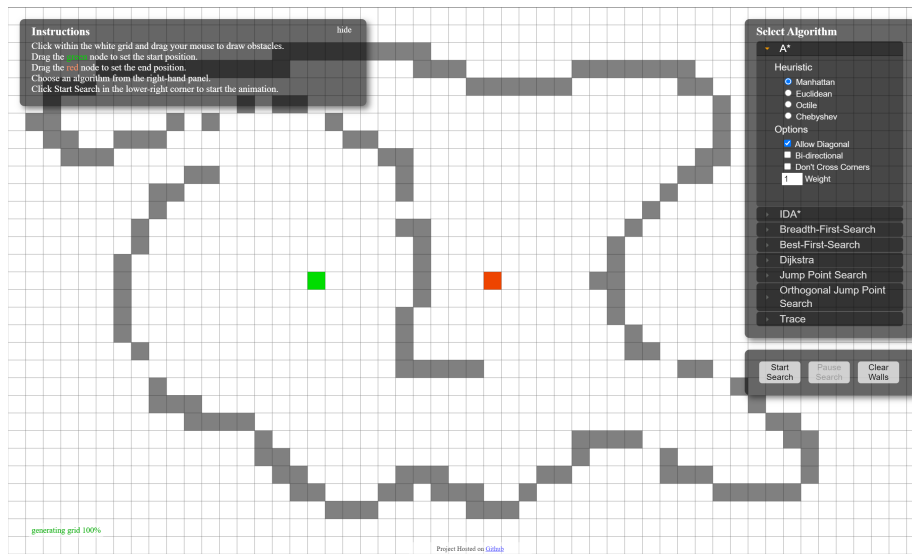
- The visualization is too slow.
- If maze is altered by user after visualization has run, then the visualization will not update.

#### 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/finish nodes are changed.

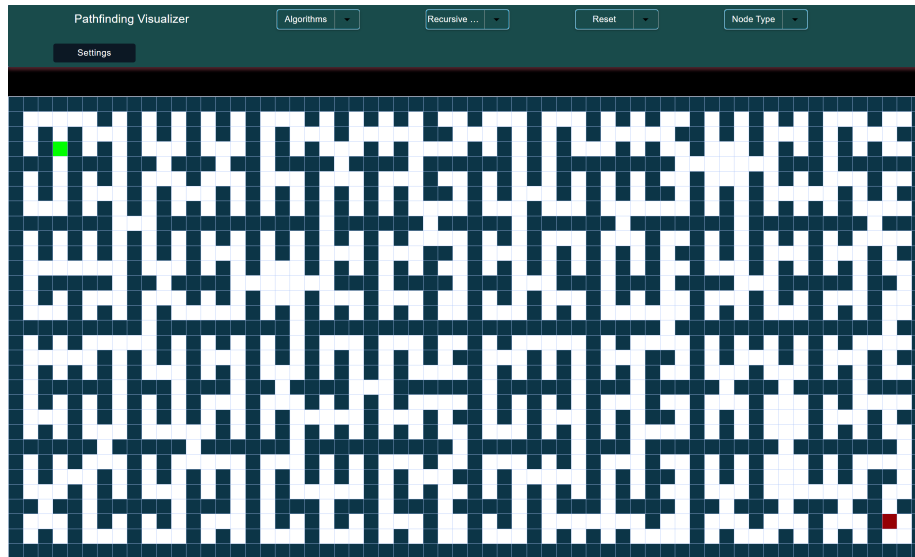
#### 1.3.1.3 Example 3 <https://pathfindout.com/>

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 pseudocode.
- Create a mockup of the user interface.
- Create the user interface in react.
- Code the algorithms in python.
- Create an AWS lambda function in python that runs the algorithms as an API.
- Create the react website to visualize the algorithms.
- 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 backtracking and Prim's algorithms used for maze generation, as well as Kruskal's (less important). This is because Kruskal's and Prim's are similar but each has their own advantages and disadvantages and recursive backtracking is different and uses recursion.
- A2. I would like to see an algorithm (such as Dijkstra's) as well as a heuristic (such as A\* or a greedy search), as this will show the difference between a heuristic and an algorithm.
- 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 want to visualize.

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 multiple algorithms, including, but not limited to: Prim's algorithm and recursive backtracking. There should also be a brief description of the algorithm that has been selected.

### 1.4.2 Solve Mazes

The website should be able to solve mazes using multiple algorithms, including, but not limited to: greedy 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.

### 1.4.3 Customisation

The user should be able to customize aspects of the visualization, including:

- Size of the maze.
- Speed of the animation.
- The heuristic used in any heuristic algorithms.

#### 1.4.4 User written algorithms

The website should be able to run algorithms written by the user for both maze generation and solving. 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.

#### 1.4.5 Update Visualization

If the start or end nodes are moved once the visualization has been run, then it should update without the user having to rerun the visualization.

#### 1.4.6 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.

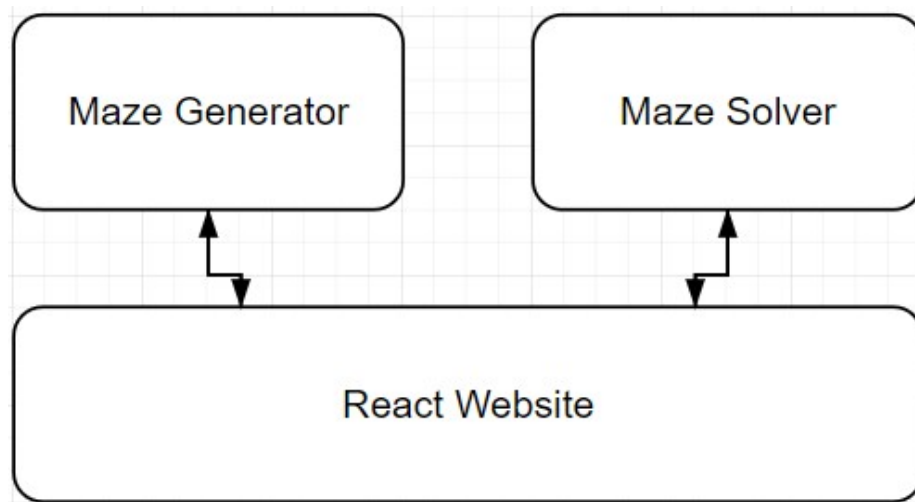
## 2 Documented Design

### 2.1 Visualization Structure

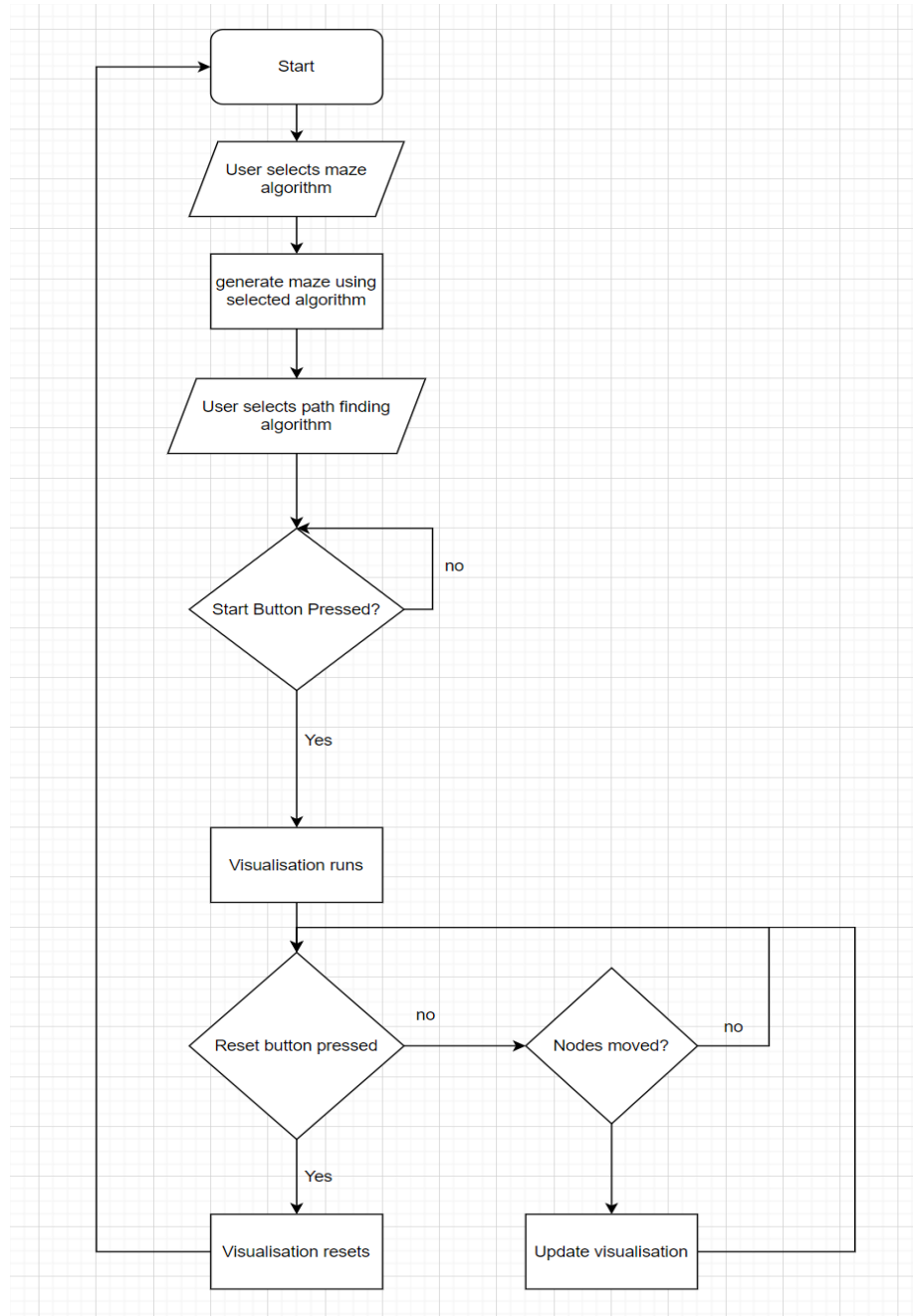
The project will be split into two main sections, the first being the visualization and the second being the python API. The python API will be for generating and solving the mazes, and the react website that will visualize the algorithms. The python API will can be subdivided into two sections, the first being the maze generation and the second being the maze solving. Different parameters will be passed to the API depending on the what the user has requested. These parameters will be:

- width
- height
- type (solve/generate/empty maze)
- generate (algorithm for generation)
- solve (algorithm for solving)
- start
- end

When solving the maze, the maze will be sent to the API as part of the body of the request.

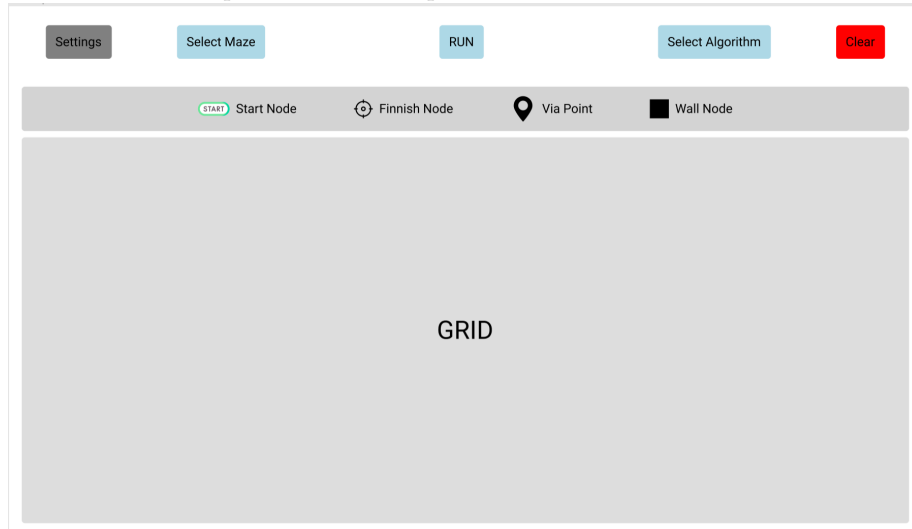


## 2.2 User Interaction Flow Chart



## 2.3 User Interface

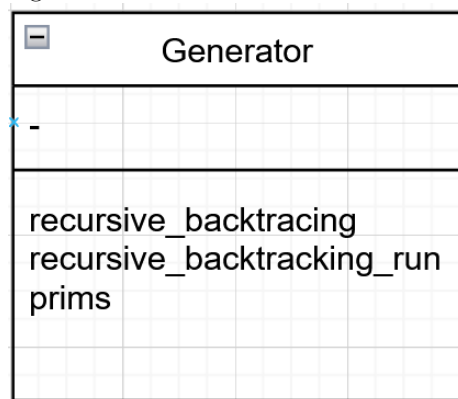
I have used Figma to create a design mockup of how the user interface will look. This allows me to plan what user inputs will be needed.



## 2.4 Classes Breakdown


### 2.4.1 Genereator

The generator class will be responsible for generating the maze with different algorithms.




### 2.4.2 Solver

The solver class will be responsible for solving the maze with different algorithms.

 Solver
-
get_adjacent_paths dijkstra dfs bfs manhattan euclidean greedy

#### 2.4.3 Maze

The maze class will be responsible for storing the maze and any other information that is needed, as well as methods for loading a generated maze to be solved and a serialize method for sending the maze to the react app.

 <i>Grid</i>
height:int width:int grid:list
load generateGrid serialize printGrid (for development)

#### 2.4.4 Node

The node class will be responsible for storing the nodes and any other information that is needed, as well as methods for loading the nodes from a generated maze and a serialize method for sending the maze to the react app.



Node
x:int y:int type:str index:int parent:Node visited:bool distance:str/int wallLeft:bool wallBottom:bool
serialize load

## 2.5 Algorithms

### 2.5.1 Prims

Prim's algorithm will have two lists, an inMaze list and a frontier, to store the nodes that are in the maze and in the frontier respectively. To start, the nodes adjacent to the start node will be added to the frontier. Then a random wall will be taken from the frontier, and the wall will be removed from the maze. The node that was connected by removing the wall will then be added to the inMaze list. Any nodes that are adjacent to the new node and not in the maze will be added to the frontier. This will continue while there are nodes in the frontier.

### 2.5.2 Recursive Backtracking

Recursive backtracking will use a stack to store the previous nodes that have been visited. The stack will be a list of Nodes that are added to the stack as they are visited. If the algorithm reaches a node that has no unvisited adjacent nodes, then the algorithm will backtrack to the last node that was visited by popping off of the stack. This will continue until the algorithm backtracks to the start node.

### 2.5.3 Dijkstra's

Dijkstra's algorithm works by keeping a priority queue of the nodes that have not been visited, however, all connections have the same weights, as it is solving a maze. The queue will be a list of Nodes, with their distance initially set to "infinity". The algorithm will then search outwards from the start node, adding

the distance from the start to each node as it is discovered, until the end node is reached. The algorithm will then backtrack from the end node to the start node, choosing the node with the lowest distance at each step.

#### 2.5.4 Depth First Search

Depth first search(dfs) works by keeping a stack of the nodes that need to be visited, and adding to the stack as new nodes are discovered. When the nodes are added to the stack, the "parent" attribute of the nodes will be updated so that a path can be drawn. If there are no unvisited nodes connected to the current node, then the algorithm will backtrack by popping a node off of the stack. This will continue until the stack is empty or the end node is reached, at which point the algorithm will return and a path will be drawn.

#### 2.5.5 Breadth First Search

Breadth first search(bfs) is similar to dfs, however a queue should be used instead of a stack. The queue will be a list of Nodes, with new Nodes being added to the end of the list as they are discovered. When the nodes are added to the queue, the "parent" attribute of the nodes will be updated so that a path can be drawn. The algorithm will keep searching until the queue is empty or the end node is reached, at which point the algorithm will return and a path will be drawn.

#### 2.5.6 Greedy Search

Greedy search works by keeping a priority queue, with the heuristic distance from the node to the end node being the priority. The algorithm will start from the start node, searching outwards, adding nodes to the priority queue as they are discovered, with the "parent" attribute being updated so that the path can be drawn. The algorithm will keep searching until the queue is empty or the end node is reached, at which point the algorithm will return and a path will be drawn.

**Heuristic** Different heuristics can be used for the greedy search. The heuristic I will include are manhattan distance and euclidean distance, as these are some of the most common heuristics.

**Manhattan Distance** Manhattan is calculated by

$$h(n) = |x_{current\ node} - x_{end\ node}| + |y_{current\ node} - y_{end\ node}| \quad (1)$$

**Euclidean Distance** Euclidean is calculated by

$$h(n) = \sqrt{(x_{current\ node} - x_{end\ node})^2 + (y_{current\ node} - y_{end\ node})^2} \quad (2)$$

## 2.6 Subroutine Breakdown API

### 2.6.1 `generator.py: recursive_backtracking`

Parameters:

Grid—2D array of Nodes

Setup the variables for recursive backtracking by creating a list in unvisited nodes.

### 2.6.2 `generator.py: recursive_backtracking_run`

Parameters:

Grid— 2D array of Nodes

unvisited— list of Nodes

current— Node

previous— list of previous Nodes

The recursive backtracking algorithm.

### 2.6.3 `generator.py: prims`

Parameters:

Grid— 2D array of Nodes

Generate the maze using Prim's algorithm.

### 2.6.4 `solver.py: get_adjacent_paths`

Parameters:

Grid— 2D array of Nodes

node— Node to find adjacent paths

Calculate and return adjacent nodes that are not blocked by walls.

### 2.6.5 `solver.py: dijkstra`

Parameters:

Grid— 2D array of Nodes

start— Node to start from

end— Node to end at

Use Dijkstra's algorithm to find the shortest path from the start node to the end node and draw a path on the Grid.

### 2.6.6 `solver.py: dfs`

Parameters:

Grid— 2D array of Nodes

start— Node to start from

end— Node to end at

Use Depth First Search to find the path from the start node to the end node and draw a path on the Grid.

**2.6.7 solver.py: bfs**

Parameters:

Grid— 2D array of Nodes  
start— Node to start from  
end— Node to end at

Use Breadth First Search to find the path from the start node to the end node and draw a path on the Grid.

**2.6.8 solver.py: manhattan**

Parameters:

node1— Node to calculate distance from  
node2— Node to calculate distance to

Calculate and return the manhattan distance between two nodes.

**2.6.9 solver.py: euclidean**

Parameters:

node1— Node to calculate distance from  
node2— Node to calculate distance to

Calculate and return the euclidean distance between two nodes.

**2.6.10 solver.py: greedy**

Parameters:

Grid— 2D array of Nodes  
start— Node to start from  
end— Node to end at  
heuristic— function to use for heuristic

Use Greedy Search to find the path from the start node to the end node and draw a path on the Grid.

**2.6.11 grid.py: Node: \_\_init\_\_**

Parameters:

x— x coordinate of node  
y— y coordinate of node  
type— type of node

Constructor for the Node class.

**2.6.12 grid.py: Node: load**

Parameters:

wallLeft— boolean if there is a wall to the left  
wallRight— boolean if there is a wall to the right

Load the walls of the node. Used when loading the maze for solving.

**2.6.13 grid.py: Node: serialize**

Returns the Node as a dictionary to be sent to the react app.

**2.6.14 grid.py: Grid: \_\_init\_\_**

Parameters:

height— height of the maze

width— width of the maze

Constructor for the Grid class.

**2.6.15 grid.py: Grid: load**

Parameters:

grid— 2D array of Nodes

Load the maze into the Grid. Used when loading the maze for solving.

**2.6.16 grid.py: Grid: serialize**

Returns the Grid as a dictionary to be sent to the react app.

**2.6.17 grid.py: Grid: generateGrid**

Generate an empty grid.

**2.6.18 grid.py: Grid: printGrid**

Print the grid, used when developing maze generation and solving algorithms.

**2.7 Subroutine Breakdown react app****2.7.1 App.js: App: Constructor**

Parameters:

props— props passed from react

Constructor for the App class.

**2.7.2 App.js: App: componentDidMount**

Method runs when the component mounts in the DOM, used to load an empty maze when the app loads.

**2.7.3 App.js: App: setHeuristic**

Parameters:

heuristic— heuristic to use

Callback function to change the heuristic used for the greedy search from the settings component.

#### **2.7.4 App.js: App: setSpeed**

Parameters:

speed— speed to use

Callback function to change the speed of the maze generation from the settings component.

#### **2.7.5 App.js: App: setSize**

Parameters:

size— size to use

Callback function to change the size of the maze from the settings component, the maze is reset when the size is changed.

#### **2.7.6 App.js: App: setStart**

Parameters:

start— start node to use

Callback function to change the start node of the maze when the start node is dragged and dropped to a new location.

#### **2.7.7 App.js: App: setEnd**

Parameters:

end— end node to use

Callback function to change the end node of the maze when the end node is dragged and dropped to a new location.

#### **2.7.8 App.js: App: setAlgorithm**

Parameters:

algorithm— algorithm to use for generating the maze

Callback function to change the algorithm used to generate the maze from the settings component.

#### **2.7.9 App.js: App: setSolve**

Parameters:

algorithm— algorithm to use for solving the maze

Callback function to change the algorithm used to solve the maze from the settings component.

#### **2.7.10 App.js: App: async should\_solve**

Method to check if the maze should be resolved after start or end nodes have moved, and if it should be resolved, then resolve the maze. This method is asynchronous, as it changes the state of the app, which is an asynchronous operation and must be awaited before continuing.

**2.7.11 App.js: App: async clear\_node\_index**

Method to clear the index of the nodes so that the maze can be solved again. This method is asynchronous, as it changes the state of the app, which is an asynchronous operation and must be awaited before continuing, and it returns a promise, which is resolved once the state has been updated.

**2.7.12 App.js: App: async fetchGrid**

Method to fetch the maze from the server. This method is asynchronous, as it changes the state of the app and is making a call to the API over the internet, both of which are asynchronous operations and must be awaited before continuing.

**2.7.13 App.js: App: async solveGrid**

Method to send the grid to the API to be solved and receive a response with a solved maze. This method is asynchronous, as it changes the state of the app and is making a call to the API over the internet, both of which are asynchronous operations and must be awaited before continuing.

**2.7.14 App.js: App: async clearGrid**

Method to get an empty grid from the API. This method is asynchronous, as it changes the state of the app and is making a call to the API over the internet, both of which are asynchronous operations and must be awaited before continuing.

**2.7.15 App.js: App: render**

Method to render the react component.

**2.7.16 DisplayGrid.jsx: DisplayGrid: handelDrop**

Parameters:

pos— position of the node that is dropped

Callback function to handle the drop of a start or end node.

**2.7.17 DisplayGrid.jsx: DisplayGrid: setDragObject**

Parameters:

type— type of node that is being dragged

Callback function to set the type of node that is being dragged.

**2.7.18 DisplayGrid.jsx: DisplayGrid: renderTable**

Method to render the table of the grid.

**2.7.19 DisplayGrid.jsx: DisplayGrid: render**

Method to render the react component. It will return a table of the grid if there is a grid, else it will show a message saying that there is no grid.

**2.7.20 DisplayNode.jsx: DisplayNode: handelDragStart**

Callback function to handle the start of the node being dragged. It will set the type of node being dragged to the type of node that is being dragged in the parent component using DisplayGrid.jsx: DisplayGrid: setDragObject.

**2.7.21 DisplayNode.jsx: DisplayNode: handelDrop**

Callback function to handle the drop of the node.

**2.7.22 DisplayNode.jsx: DisplayNode: handelDragOver**

Callback function to run when the node is dragged over this node. It will change the colour of the node to indicate that it is being dragged over.

**2.7.23 DisplayNode.jsx: DisplayNode: handelDragLeave**

Callback function to run when the node is dragged off this node. It will change the colour of the node to indicate that it is no longer being dragged over.

**2.7.24 DisplayNode.jsx: DisplayNode: render**

Method to render the react component. Selection is used to examine the props and apply the correct attributes to the node.

**2.7.25 Settings.jsx: Settings: componentDidMount**

Method to run when the component mounts in the DOM. It will add an event listener to the document to detect if the user clicks outside the settings component.

**2.7.26 Settings.jsx: Settings: componentWillUnmount**

Method to run when the component is unmounted from the DOM. It will remove the event listener from the document.

**2.7.27 Settings.jsx: Settings: handelClickOutsize**

Parameters:

event— event that is triggered when the user clicks outside the settings component

Callback function to handle the click outside the settings component.



**2.7.28 Settings.jsx: Settings: handelSizeChange**

Parameters:

event— event that is triggered when the user changes the size of the maze

Callback function to the change of the size of the maze in the parent component using App.jsx: App: setSize.

**2.7.29 Settings.jsx: Settings: setHeuristic**

Parameters:

heuristic— selected heuristic

Callback function to change the heuristic used for the greedy search in the parent component using App.jsx: App: setHeuristic.

**2.7.30 Settings.jsx: Settings: renderSettings**

Method to render the settings component if the settings button is clicked.

## 3 Evidence of Completeness

## 4 Technical Solution

A running solution of the project is available at <https://olivertemple.github.io/nea/>.

### 4.1 Summary Of Skills Used

These tables are a list of some of the skills I have used in this project, with line numbers of where they are demonstrated.

Skill	Where to find
Stacks	Solver.py: dfs
Queues	Solver.py: bfs, Solver.py: dijkstra
Priority Queue	Solver.py: greedy
Recursive Algorithms	Generator.py: recursive_backtracking_run
Complex OOP	grid.py: Grid, grid.py: Node, App.js, DisplayGrid.jsx, DisplayNode.jsx, Settings.jsx,
Dynamic generation of objects	Grid.py: Grid: generateGrid
Client-server model, parsing JSON	Called from App.js: 117, App.js: 131, App.js: 151, request handled in lambda_function.py
Dictionaries	lambda_function.py, grid.py: Grid: serialize, grid.py: Node: serialize
Multi-dimensional arrays	grid.py (used to store grid), generator.py: prims (inMaze, frontier)
Simple mathematical calculations	solver.py: manhattan, solver.py: euclidean
Complex Algorithms	solver.py, generator.py
GUI	App.js, DisplayGrid.jsx, DisplayNode.jsx, Footer.jsx, GeneratorInfo.jsx, Menu.jsx, MenuKey.jsx, Settings.jsx, SolverInfo.jsx
Graph traversals	solver.py, generator.py

## 4.2 The Full Code

### 4.2.1 Python API: lambda\_function.py

```

1 import json
2 from generator import Generator
3 from grid import Grid
4 from solver import Solver
5 def lambda_handler(event, context):
6     #get the parameters from the url query
7     width = event["queryStringParameters"]["width"]
8     height = event["queryStringParameters"]["height"]
9
10    if event["queryStringParameters"]["type"] == "empty_maze":#generate an empty grid
11        myGrid = Grid(int(width), int(height))
12
13    elif event["queryStringParameters"]["type"] == "generate":#generate an empty grid,
14        then create a maze
15        #get the maze generating algorithm
16        generate_algorithm = event["queryStringParameters"]["generate"]
17
18        #create a new generator to generate the maze
19        myGenerator = Generator();
20
21        #create a new grid with the height and width algorithms from the query
22        myGrid = Grid(int(width),int(height))
23
24        #generate the maze using the algorithm from the query

```

```

24     if generate_algorithm == "prims":
25         myGenerator.prims(myGrid)
26     elif generate_algorithm == "recursive_backtracking":
27         myGenerator.recursive_backtracking(myGrid)
28
29     elif event["queryStringParameters"]["type"] == "solve":#solve the maze using
requested algorithm
30         #get the grid from the body of the request
31         grid = json.loads(event["body"])
32
33         #generate an empty grid with the same size as the grid from the body
34         myGrid = Grid(int(grid["width"]), int(grid["height"]))
35
36         #load the grid from the body into the empty grid by iterating through the nodes
and changing the attributes
37         myGrid.load(grid["grid"])
38
39         #get the solving algorithm from the request
40         solve_algorithm = event["queryStringParameters"]["solve"]
41
42         #get the start and node positions from the request
43         start = eval(event["queryStringParameters"]["start"])
44         end = eval(event["queryStringParameters"]["end"])
45
46         #get the start and end nodes in the grid
47         start_node = myGrid.grid[start[0]][start[1]]
48         end_node = myGrid.grid[end[0]][end[1]]
49
50         #create a new solver to solve the maze
51         mySolver = Solver()
52
53         #solve the maze with the requested algorithm
54         if solve_algorithm == "dijkstra":
55             mySolver.dijkstra(myGrid, start_node, end_node)
56         elif solve_algorithm == "dfs":
57             mySolver.dfs(myGrid, start_node, end_node)
58         elif solve_algorithm == "bfs":
59             mySolver.bfs(myGrid, start_node, end_node)
60         elif solve_algorithm == "greedy":
61             #get the heuristic for greedy from the request
62             heuristic = event["queryStringParameters"]["heuristic"]
63             mySolver.greedy(myGrid, start_node, end_node, heuristic)
64
65         #return the json of the grid
66         return {
67             'statusCode': 200,
68             'body': json.dumps(myGrid.serialize())
69         }
70
71 if __name__ == "__main__":
72     myGrid = Grid(15,15)
73     myGenerator = Generator()
74     myGenerator.prims(myGrid)
75     mySolver = Solver()
76     mySolver.greedy(myGrid, myGrid.grid[0][0], myGrid.grid[14][14], "manhattan")
77     myGrid.printGrid()

```

### 4.2.2 Python API: grid.py

```

1 #Node class for each node in the grid
2 class Node:
3     def __init__(self, x, y, type):
4         self.x = x
5         self.y = y
6         self.type = type
7
8         self.index = None
9         self.parent = None
10        self.visited = False
11
12        self.distance = "infinity"
13
14        self.wallLeft = True
15        self.wallBottom = True
16    def __str__(self):#for development purposes
17        return f"x:{self.x}, y:{self.y}, type:{self.type}, wallLeft:{self.wallLeft},
18        wallBottom:{self.wallBottom}, distance:{self.distance}"
19    def load(self, wallLeft, wallBottom):#when loading the grid, the walls must be set
20        from the received grid
21        self.wallBottom = wallBottom
22        self.wallLeft = wallLeft
23    def serialize(self):#for returning the grid, the node must be serialized into a
24        dictionary
25        return {
26            "x": self.x,
27            "y": self.y,
28            "wallLeft": self.wallLeft,
29            "wallBottom": self.wallBottom,
30            "type": self.type,
31            "index":self.index
32        }
33
34class Grid:#Grid class for the grid
35    def __init__(self, height, width):
36        self.height = height
37        self.width = width
38        self.grid = self.generateGrid()
39    def load(self, grid):#load the grid from the grid received from the react app
40        self.grid = []
41        for row in grid:
42            row_inner = []
43            for item in row:
44                node = Node(item["x"], item["y"], "space")
45                node.load(item["wallLeft"], item["wallBottom"])
46                row_inner.append(node)
47            self.grid.append(row_inner)
48    def serialize(self):#for returning the grid, the grid must be serialized into a
49        dictionary
50        obj = {
51            "height": self.height,
52            "width": self.width,
53            "grid": []
54        }
55        for row in self.grid:

```

```

52         row_inner = []
53         for item in row:
54             row_inner.append(item.serialize())
55         obj["grid"].append(row_inner)
56
57     return obj
58
59 def generateGrid(self):#generate the grid
60     grid = []
61     for i in range(self.height):
62         row = []
63         for j in range(self.width):
64             row.append(Node(j, i, "space"))
65         grid.append(row)
66
67     row = []
68     for j in range(self.width + 1):
69         row.append(Node(j, i, "space"))
70     grid.append(row)
71
72     return grid
73
74 def printGrid(self):#for development purposes
75     width = self.width
76     height = self.height
77
78     print(" _"*width)
79
80     for i in range(height):
81         for j in range(width):
82             cell = ""
83             if self.grid[i][j].wallLeft:
84                 cell += "|"
85             else:
86                 cell += " "
87             if self.grid[i][j].type == "path":
88                 cell += "XX"
89             else:
90                 cell += " "
91             print(cell, end=" ")
92         print("|")
93
94     for j in range(width):
95         cell = ""
96         if self.grid[i][j].wallLeft:
97             cell += "|"
98         else:
99             cell += " "
100
101         if self.grid[i][j].wallBottom:
102             cell += "--"
103         else:
104             cell += " "
105         print(cell, end=" ")
106     print("|")

```

#### 4.2.3 Python API: generator.py

```

1 import random
2 import sys
3
4 class Generator:
5     def __init__(self):
6         pass
7     #setup for recursive backtracking
8     def recursive_backtracking(self, Grid):
9         sys.setrecursionlimit(2000) #set the recursion limit to 2000
10        #create a list of all nodes in maze
11        unvisited = []
12        for row in Grid.grid:
13            for item in row:
14                unvisited.append(item)
15
16        #pick a random start node
17        start = random.choice(unvisited)
18        unvisited.remove(start)
19
20        #start recursive backtracking
21        self.recursive_backtracking_run(Grid, unvisited, start, [])
22
23    #recursive backtracking algorithm
24    def recursive_backtracking_run(self, Grid, unvisited, current, previous):
25        #work out which walls can be removed
26        orientation_options = []
27        if current.x > 0 and Grid.grid[current.y][current.x-1] in unvisited:
28            orientation_options.append("left")
29        if current.y > 0 and Grid.grid[current.y-1][current.x] in unvisited:
30            orientation_options.append("top")
31        if current.x < Grid.width - 1 and Grid.grid[current.y][current.x+1] in
unvisited:
32            orientation_options.append("right")
33        if current.y < Grid.height - 1 and Grid.grid[current.y+1][current.x] in
unvisited:
34            orientation_options.append("bottom")
35
36        #if there are no walls to remove, backtrack, else pick one and remove it
37        if len(orientation_options) > 0:
38            #pick a random wall to remove
39            orientation = random.choice(orientation_options)
40            #add the current node to the previous nodes list
41            previous.append(current)
42
43            #remove the wall depending on the orientation
44            if orientation == "left":
45                connecting_cell = Grid.grid[current.y][current.x - 1]
46                current.wallLeft = False
47
48            elif orientation == "bottom":
49                connecting_cell = Grid.grid[current.y + 1][current.x]
50                current.wallBottom = False
51
52            elif orientation == "right":
53                connecting_cell = Grid.grid[current.y][current.x + 1]
54                connecting_cell.wallLeft = False
55

```

```

56         elif orientation == "top":
57             connecting_cell = Grid.grid[current.y - 1][current.x]
58             connecting_cell.wallBottom = False
59
60             #remove the connecting node from the unvisited list
61             unvisited.remove(connecting_cell)
62
63             #recurse
64             self.recursive_backtracking_run(Grid, unvisited, connecting_cell, previous)
65         else:
66             #if not back at the start, backtrack
67             if len(previous) > 0:
68                 new = previous.pop()
69                 self.recursive_backtracking_run(Grid, unvisited, new, previous)
70
71     #generate a maze using prims algorithm
72     def prims(self, Grid):
73         #start at the top left node
74         inMaze = [[0, 0]]
75         #create a list of nodes connected to the start node
76         frontier = [[[0, 1], ["left"]], [[1, 0], ["top"]]]
77
78         #while there are still nodes to visit
79         while (len(frontier) > 0):
80             #pick a random node from the frontier
81             new = frontier.pop(random.randint(0, len(frontier)-1))
82
83             toAdd = new[0]
84
85             #pick a random wall from the available walls
86             wall = new[1][random.randint(0, len(new[1])-1)]
87
88             #add the wall to the inMaze list
89             inMaze.append(toAdd)
90
91             #remove the selected wall from the grid
92             if wall == "bottom":
93                 Grid.grid[toAdd[0]][toAdd[1]].wallBottom = False
94
95             if wall == "left":
96                 Grid.grid[toAdd[0]][toAdd[1]].wallLeft = False
97
98             if wall == "top" and toAdd[0] > 0:
99                 Grid.grid[toAdd[0]-1][toAdd[1]].wallBottom = False
100
101             if wall == "right" and toAdd[1] < Grid.width:
102                 Grid.grid[toAdd[0]][toAdd[1]+1].wallLeft = False
103
104             #calculate the possible nodes to add to the frontier
105             possible = [[toAdd[0]-1, toAdd[1]], [toAdd[0]+1, toAdd[1]], [toAdd[0], toAdd
106             [1]-1], [toAdd[0], toAdd[1]+1]]
107
108             #possible walls
109             walls = ["bottom", "top", "right", "left"]
110             #iterate through the possible nodes
111             for i in range(4):
112                 p = possible[i]

```

```

112         wall = walls[i]
113         #check that the wall can be removed
114         if 0<=p[0]<Grid.height and 0<=p[1]<Grid.width:
115             #check that the node is not already in the maze
116             if p not in inMaze:
117                 found = False
118                 #check if the node is already in the frontier, if it is then
119                 add the wall to the possible walls for the node in the frontier
120                 for v in frontier:
121                     if v[0]==p:
122                         v[1].append(wall)
123                         found = True
124                     #if the node is not in the frontier, add it to the frontier
125                     if not found:
126                         frontier.append([p,[wall]])

```

#### 4.2.4 Python API: solver.py

```

1 import math
2 class Solver:
3     def __init__(self):
4         pass
5
6     def get_adjacent_paths(self, Grid, node):#Get adjacent nodes in the grid that are
7     not blocked by a wall
8         paths = []
9         if node.x > 0 and not node.wallLeft: #check that the node is not on the left
10         edge and that it doesn't have a wall on the left
11             paths.append(Grid.grid[node.y][node.x - 1])
12         if node.x < Grid.width - 1 and not Grid.grid[node.y][node.x + 1].wallLeft: #
13         check that the node is not on the right edge, and that there is not a wall to the
14         right of it
15             paths.append(Grid.grid[node.y][node.x + 1])
16         if node.y > 0 and not Grid.grid[node.y - 1][node.x].wallBottom: #check that the
17         node is not at the top and that there isn't a wall above it
18             paths.append(Grid.grid[node.y - 1][node.x])
19         if node.y < Grid.height - 1 and not node.wallBottom:#check that the node is not
20         at the bottom and that there is no wall below it
21             paths.append(Grid.grid[node.y + 1][node.x])
22
23         #retrun a list of the available nodes
24         return paths
25
26     def dijkstra(self, Grid, start, end):#Dijkstra's algorithm for solving the grid
27     #set the distance from the start node to the start node to 0
28     start.distance = 0
29     #set the start index as 0
30     #The index is a number that shows when that node was visited by the solving
31     algorithm, so that when the algorithm is visualized, the react app can show in what
32     order the nodes were visited.
33     index = 0
34     #create a queue of unvisited nodes
35     unvisited = [start]
36     #create a flag
37     found = False
38     while not found:
39         #take the next node from the front of the queue

```



```

32         current = unvisited.pop(0)
33         #mark the node as visited
34         current.visited = True
35         #set the index on the node
36         current.index = index
37         #increment the index
38         index += 1
39         #iterate through the available adjacent nodes
40         for node in self.get_adjacent_paths(Grid, current):
41             #if the node has not been visited already, set the distance from the
start node to be one more than the distance of the current node and add it to the
queue of unvisited nodes
42                 if not node.visited:
43                     node.distance = current.distance + 1
44                     unvisited.append(node)
45
46             #if the node is the end node, update the flag and exit the algorithm
47             if node == end:
48                 found = True
49                 break
50
51         #backtrack from the end node to the start node, picking the node with the
smallest distance at every point
52         current = end
53         path = [end]
54         while current != start:
55             #get connecting cells
56             connecting = self.get_adjacent_paths(Grid, current)
57             #work out which node as the lowest distance
58             min = None
59             for node in connecting:
60                 if node.distance != "infinity":
61                     if min == None or node.distance < min.distance:
62                         min = node
63             #append the node with the lowest distance to the path
64             path.append(min)
65             #update the current node
66             current = min
67
68         #draw the path
69         for node in path:
70             node.type = "path"
71
72     def dfs(self, Grid, start, end):
73         #create a stack of nodes to visit
74         stack = [start]
75         #set the start index as 0
76         #The index is a number that shows when that node was visited by the solving
algorithm, so that when the algorithm is visualized, the react app can show in what
order the nodes were visited.
77         start.index = 0
78         index = 1
79         #while the stack is not empty, keep searching
80         while len(stack) > 0:
81             #get the item at the top of the stack
82             current = stack[-1]
83             #mark the item as visited

```

```

84         current.visited = True
85         #check if the end has been found
86         if current == end:
87             break
88
89         #find connecting nodes that have not been visited
90         possible = []
91         for node in self.get_adjacent_paths(Grid, current):
92             if not node.visited:
93                 possible.append(node)
94
95         #if there are possible connections, choose the first one
96         if len(possible) > 0:
97             to_append = possible[0]
98             #set the index
99             to_append.index = index
100            #set the parent node for drawing the path
101            to_append.parent = current
102            #increment the index
103            index += 1
104            #add the new node to the stack
105            stack.append(to_append)
106            #If there are no possible connections, remove the current node from the
stack
107        else:
108            stack.pop()
109
110        #backtrack from the end to the start drawing the path
111        while current != start:
112            current.type = "path"
113            current = current.parent
114
115        start.type = "path"
116
117    def bfs(self, Grid, start, end):
118        #create a queue of nodes that need to be visited
119        queue = [start]
120        #set the start index as 0
121        #The index is a number that shows when that node was visited by the solving
algorithm, so that when the algorithm is visualized, the react app can show in what
order the nodes were visited.
122        start.index = 0
123        index = 1
124        #only run the algorithm while there are nodes to visit
125        while len(queue) > 0:
126            #get the first item in the queue and mark as visited
127            current = queue.pop(0)
128            current.visited = True
129
130            #check to see if the end node has been found
131            if current == end:
132                break
133
134            #add unvisited adjacent nodes to the queue
135            for node in self.get_adjacent_paths(Grid, current):
136                if not node.visited:
137                    #update nodes parent for drawing path

```

```

138         node.parent = current
139         #add index
140         node.index = index
141         #increment the index
142         index += 1
143         #append node to queue
144         queue.append(node)
145
146     #backtrack from the end to the start and draw the path
147     while current != start:
148         current.type = "path"
149         current = current.parent
150
151     start.type = "path"
152
153     def manhattan(self, node1, node2):#calculate the manhattan distance between two
nodes
154         return abs(node1.x - node2.x) + abs(node1.y - node2.y)
155
156     def euclidean(self, node1, node2):#calculate the euclidean distance between two
nodes
157         return math.sqrt((node1.x - node2.x)**2 + (node1.y - node2.y)**2)
158
159     def greedy(self, Grid, start, end, heuristic):
160         #create a priority queue for nodes that need to be visited
161         queue = [start]
162         #set the start index as 0
163         #The index is a number that shows when that node was visited by the solving
algorithm, so that when the algorithm is visualized, the react app can show in what
order the nodes were visited.
164         start.index = 0
165         index = 1
166         #while there are nodes to visit
167         while len(queue) > 0:
168             #pop the node off of the front of the queue and mark as visited
169             current = queue.pop(0)
170             current.visited = True
171
172             #check if the end node has been found
173             if current == end:
174                 break
175
176             #iterate through unvisited adjacent nodes
177             for node in self.get_adjacent_paths(Grid, current):
178                 if not node.visited:
179                     #update the parent of the node
180                     node.parent = current
181                     #update the index
182                     node.index = index
183                     #increment the index
184                     index += 1
185                     #use the selected heuristic to update the distance
186                     if heuristic == "manhattan":
187                         node.distance = self.manhattan(node, end)
188                     elif heuristic == "euclidean":
189                         node.distance = self.euclidean(node, end)
190                     #insert node in the priority queue

```

```

191         for item in queue:
192             if item.distance > node.distance:
193                 queue.insert(queue.index(item), node)
194                 break
195         queue.append(node)
196
197         #backtrack from the start and draw the path
198         while current != start:
199             current.type = "path"
200             current = current.parent
201
202         start.type = "path"

```

#### 4.2.5 React App: App.js

```

1 import './App.css';
2 import { Component } from 'react';
3 import DisplayGrid from './components/DisplayGrid';
4 import Menu from './components/Menu';
5 import MenuKey from './components/MenuKey';
6 import Footer from './components/Footer';
7 class App extends Component {
8     constructor(props){
9         super(props);
10        this.state = {
11            grid: null, //the grid of nodes
12            algorithm: null, //algorithm for generating the maze
13            solve: null, //algorithm for solving the maze
14            nodes: {
15                start: [0,0], //position of the start node
16                end: [null, null] //position of the end node
17            },
18            size: { //size of the maze
19                width: 15,
20                height: 15
21            },
22            heuristic: "euclidean",
23            speed: 0.1
24        }
25
26        this.solved = false;
27        this.maze = false;
28        //bind the methods to the object so that the "this" keyword refers to the object no
29        //matter where the method is called from
30        this.fetchGrid = this.fetchGrid.bind(this);
31        this.setAlgorithm = this.setAlgorithm.bind(this);
32        this.clearGrid = this.clearGrid.bind(this);
33        this.setSolve = this.setSolve.bind(this);
34        this.solveGrid = this.solveGrid.bind(this);
35        this.setSize = this.setSize.bind(this);
36        this.setStart = this.setStart.bind(this);
37        this.setEnd = this.setEnd.bind(this);
38        this.should_solve = this.should_solve.bind(this);
39        this.setHeuristic = this.setHeuristic.bind(this);
40        this.setSpeed = this.setSpeed.bind(this);
41    }

```

```

42 componentDidMount(){
43   //generate a new maze empty when the page loads
44   this.clearGrid();
45 }
46
47 setHeuristic(heuristic){//set the heuristic for the greedy algorithm
48   this.setState({heuristic: heuristic});
49 }
50 setSpeed(speed){//set the speed of the animation
51   this.clearGrid();
52   this.setState({speed: speed});
53 }
54 setSize(size){//set the size of the grid when changed in settings
55   this.setState({
56     size: size
57   }, () => {//setState is asynchronous, so we need to wait for it to finish before
58     //running the following code
59     if (size.width > 0 && size.height > 0){//if the size is valid, generate a new maze
60       this.clearGrid();
61     }
62   })
63 }
64 setStart(node){//set the start node
65   this.setState({
66     nodes:{
67       start: node,
68       end: this.state.nodes.end
69     }, () => {//setState is asynchronous, so we need to wait for it to finish before
70       //running the following code
71       this.should_solve()//solve the maze again if it is already solved
72     })
73 }
74 setEnd(node){//set the end node
75   this.setState({
76     nodes:{
77       start: this.state.nodes.start,
78       end: node
79     }, () => {//setState is asynchronous, so we need to wait for it to finish before
80       //running the following code
81       this.should_solve()//solve the maze again if it is already solved
82     })
83 }
84 async should_solve(){//if the maze is already solved, then solve again. Only run when
85   //the start or end nodes are changed
86   if (this.solved){
87     this.clear_node_index()//clear the index of the nodes, as the maze is being solved
88     //again
89     .then(() => {
90       this.solveGrid();
91     })
92   }
93 }
94
95 async clear_node_index(){//clear the index of the nodes so that the maze can be solved

```

```

    again
94   return new Promise(resolve => { //since there are asynchronous calls, we need to wait
        for them to finish before running the code after the clear_node_index function,
        hence we use a promise that is resolved once this code is finished
95   let grid = this.state.grid.grid
96   //iterate through the grid and clear the index of the nodes
97   for (let i=0; i<this.state.size.height; i++){
98       for(let j=0; j<this.state.size.width; j++){
99           grid[i][j].index = null;
100       }
101   }
102   //update the state with the grid that has been cleared of the index's
103   this.setState({
104       grid: {
105           grid: grid,
106           height: this.state.grid.height,
107           width: this.state.grid.width
108       }
109   }, () => {
110       resolve(); // resolve the promise once the state has been updated
111   })
112 })
113
114 }
115 async fetchGrid(){//generate a new maze from the python API using the selected
    algorithm
116 if (this.state.algorithm){//check that there is an algorithm selected for generating
    the maze
117 let grid = await fetch('https://jkr1v64tsl.execute-api.eu-west-2.amazonaws.com/
    default/NEA?type=generate&width=${this.state.size.width}&height=${this.state.size.
    height}&generate=${this.state.algorithm}')//fetch the generated grid from the python
    API
118 grid = await grid.json();//convert the response to json
119 this.setState({//update the state with the new grid
120     grid:grid
121 })
122 this.solved = false;//the maze is no longer solved
123 this.maze = true;//set the maze to true as a maze has been generated
124 }else{//If there is no algorithm selected to generate the maze, alert the user
125     alert("Please select a maze generating algorithm")
126 }
127 }
128 async solveGrid(){//send the maze to the python API to be solved with the requested
    algorithm
129 await this.clear_node_index();//clear the index of the nodes, as the maze is being
    solved again
130 if (this.maze && this.state.solve){//check that there is a maze and that there is an
    algorithm selected for solving the maze
131 let grid = await fetch(
132     'https://jkr1v64tsl.execute-api.eu-west-2.amazonaws.com/default/NEA?type=solve&
    width=${this.state.size.width}&height=${this.state.size.height}&solve=${this.state.
    solve}&start=${this.state.nodes.start}&end=${this.state.nodes.end}&heuristic=${this.
    state.heuristic}', {
133     method: "POST",
134     body: JSON.stringify(this.state.grid)//set the body of the request to the grid
135 })//send the maze to the python API to be solved, with the selected algorithm as a
    parameter

```

```

136   grid = await grid.json();//convert the response to json
137   this.setState({//update the state with the new grid
138     grid:grid
139   })
140   this.solved = true;//the maze is now solved
141 }else{
142   if (!this.maze){//if there is no maze, alert the user that there is no maze to solve
143     alert("Please generate a maze")
144   }else{
145     alert("Please select a solving algorithm");//If there is no algorithm selected to
146     solve the maze, alert the user
147   }
148 }
149 }
150 async clearGrid(){//generate an empty maze from the API
151   let grid = await fetch('https://jkr1v64tsl.execute-api.eu-west-2.amazonaws.com/
    default/NEA?type=empty_maze&width=${this.state.size.width}&height=${this.state.size.
    height}')//fetch the empty grid from the python API
152   grid = await grid.json();//convert the response to json
153   this.setState({//update the state
154     grid:grid,//update the state with the new grid
155     nodes:{//set the start and end nodes to default positions
156       start: [0,0],
157       end: [this.state.size.height - 1, this.state.size.width - 1]
158     }
159   })
160   this.maze = false;//there is no longer a maze to solve
161   this.solved = false;//the maze is no longer solved
162 }
163 setAlgorithm(algorithm){//set the maze generating algorithm
164   this.setState({
165     algorithm: algorithm
166   })
167 }
168 setSolve(algorithm){//set the maze solving algorithm
169   this.setState({
170     solve:algorithm
171   })
172 }
173 render(){
174   return (
175     <div className="App">
176       <Menu
177         setAlgorithm={this.setAlgorithm}//callback function to set the generation
178         algorithm from the menu
179         setSolve={this.setSolve}//callback function to set the solving algorithm from the
180         menu
181         generate={this.fetchGrid}//callback function to generate a new maze from the menu
182         clearGrid={this.clearGrid}//callback function to clear the maze from the menu
183         solve={this.solveGrid}//callback function to solve the maze from the menu
184         size={this.state.size}//the size of the maze
185         setSize={this.setSize}//callback function to set the size of the maze from the
186         menu
187         setHeuristic={this.setHeuristic}//callback function to set the heuristic from the
188         menu
189         setSpeed={this.setSpeed}//callback function to set the speed from the menu

```

```

186   speed={this.state.speed} //the speed of the maze
187   />
188   <MenuKey />
189   <DisplayGrid
190     grid={this.state.grid} //the grid of the maze
191     nodes={this.state.nodes} //the start and end nodes
192     size={this.state.size} //the size of the maze
193     setStart={this.setStart} //callback function to set the start node
194     setEnd={this.setEnd} //callback function to set the end node
195     generateAlgorithm={this.state.algorithm} //the algorithm used to generate the maze
196     solveAlgorithm={this.state.solve} //the algorithm used to solve the maze
197     heuristic={this.state.heuristic} //the heuristic used for the greedy algorithm
198     speed={this.state.speed} //the speed of the animation
199   />
200   <Footer />
201 </div>
202 );
203 }
204 }
205 export default App;

```

#### 4.2.6 React App: Menu.jsx

```

1 import React from 'react';
2 import Settings from './Settings';
3 export default function Menu(props) { //Menu bar for the app
4   return(
5     <div className="menu">
6       <Settings
7         size={props.size} //size of the maze
8         setSize={props.setSize} //callback to set the size of the maze from the
9         settings
10        setHeuristic={props.setHeuristic} //callback to set the heuristic from the
11        settings
12        setSpeed={props.setSpeed} //callback to set the speed from the settings
13        speed={props.speed} //speed of the animation
14      />
15      <select className="algorithms" name="algorithms" id="algorithms" onChange={(e)
16      => {props.setAlgorithm(e.target.value)}}>
17        <option value="select">Select Generating Algorithm</option>
18        <option value="prims">Prims</option>
19        <option value="recursive_backtracking">recursive backtracking</option>
20      </select>
21      <button className="button" onClick={props.generate}>Generate</button>
22      <button className="button" onClick={props.solve}>Solve</button>
23      <select className="algorithms" name="algorithms" id="algorithms" onChange={(e)
24      => {props.setSolve(e.target.value)}}>
25        <option value="select">Select Solving Algorithm</option>
26        <option value="dijkstra">Dijkstra</option>
27        <option value="dfs">Depth First Search</option>
28        <option value="bfs">Breadth First Search</option>
29        <option value="greedy">Greedy</option>
30      </select>
31      <button className="button clear" onClick={props.clearGrid}>Reset</button>
32    </div>
33  )
34 }

```



#### 4.2.7 React App: MenuKey.jsx

```

1 import React from "react";
2
3 export default function MenuKey(props){//key for showing the different types of node
4   return(
5     <div className="key">
6       <div className="key_item">
7         <div className="key_node_start"></div>
8         <p>Start Node</p>
9       </div>
10      <div className="key_item">
11        <div className="key_node_end"></div>
12        <p>Finish Node</p>
13      </div>
14      <div className="key_item">
15        <div className="key_node_path"></div>
16        <p>Path Node</p>
17      </div>
18      <div className="key_item">
19        <div className="key_node_visited_node"></div>
20        <p>Visited Node</p>
21      </div>
22    </div>
23  )
24 }

```

#### 4.2.8 React App: DisplayGrid.jsx

```

1 import React, { Component } from "react";
2 import DisplayNode from "../DisplayNode";
3 import GeneratorInfo from "../GeneratorInfo";
4 import SolverInfo from "../SolverInfo";
5 export default class DisplayGrid extends Component{
6   constructor(props){
7     super(props);
8     this.state = {
9       dragObject: ""
10    }
11    //bind the methods to the object so that the "this" keyword refers to the object no
12    //matter where the method is called from
13    this.renderTable = this.renderTable.bind(this);
14    this.handelDrop = this.handelDrop.bind(this);
15    this.setDragObject = this.setDragObject.bind(this);
16  }
17  handelDrop(pos){//move the node that was being dragged to the new position
18    switch (this.state.dragObject){
19      case "start":
20        this.props.setStart(pos)
21        break;
22      case "end":
23        this.props.setEnd(pos)
24        break;
25      default:
26        break;
27    }
28  }
29 }

```

```

28 setDragObject(type){//set weather start or end node is being dragged
29   this.setState({
30     dragObject:type
31   })
32 }
33 renderTable(){//render the grid as a table
34   return(
35     <table>
36       <tbody className="column">
37         {Array.from(Array(this.props.grid.height).keys()).map((_, i) => {//iterate
38           through the rows of the grid
39             return(
40               <tr className={`row wall_right ${i === 0 ? "wall_top" : ""}` key={i}>
41                 {Array.from(Array(this.props.grid.width).keys()).map((_, j) => {//
42                   iterate through the nodes in each row
43                     return(
44                       <DisplayNode
45                         key={j}
46                         wallLeft={this.props.grid.grid[i][j].wallLeft} //bool: is there
47                         a wall to the left of this node
48                         wallBottom={this.props.grid.grid[i][j].wallBottom} //bool: is
49                         there a wall below this node
50                         pos={[i, j]} //position of the node
51                         start={this.props.nodes.start} //position of the start node
52                         end={this.props.nodes.end} //position of the end node
53                         handelDrop = {this.handelDrop} //callback function to move the
54                         start or end node to a new position
55                         setDragObject={this.setDragObject} // callback function to set
56                         weather the start or end node is being dragged
57                         type={this.props.grid.grid[i][j].type} //type of node
58                         index={this.props.grid.grid[i][j].index} //index of the node
59                         for visualization
60                         speed={this.props.speed} //speed of the animation
61                       />
62                     )
63                   )}}
64                 </tr>
65               )
66             )}}
67         </tbody>
68       </table>
69     )
70 }
71 render(){
72   //If there is a grid, render it, else show a message
73   if (this.props.grid){
74     return(
75       <div className="grid" style={{padding:10}}>
76         <GeneratorInfo generator={this.props.generateAlgorithm}/>
77         <this.renderTable />
78         <SolverInfo solver={this.props.solveAlgorithm} heuristic={this.props.
79           heuristic}/>
80       </div>
81     )
82   }else{
83     return(

```

```

77     <div className="grid message column">
78       <h1>No grid to display</h1>
79       <h2>Check your internet connection</h2>
80     </div>
81   )
82 }
83 }
84 }

```

#### 4.2.9 React App: DisplayNode.jsx

```

1  import React from "react"
2  export default class DisplayNode extends React.Component{
3    constructor(props){
4      super(props)
5      this.state = {
6        style:{}
7      }
8      //bind the methods to the object so that the "this" keyword refers to the object no
      //matter where the method is called from
9      this.handelDragStart = this.handelDragStart.bind(this);
10     this.handelDragLeave = this.handelDragLeave.bind(this);
11     this.handelDragOver = this.handelDragOver.bind(this);
12     this.handelDrop = this.handelDrop.bind(this);
13   }
14   handelDragStart(){//set the type of node that is being dragged
15     this.props.setDragObject(this.start ? "start" : this.end ? "end" : "")
16   }
17   handelDrop(){//move the node that was being dragged to the new position
18     this.setState({style:{}});
19     this.props.handelDrop(this.props.pos)
20   }
21   handelDragOver(e){//when another node is dragged over this node, set the style of the
    //node to be pink
22     e.preventDefault();
23     this.setState({
24       style:{
25         backgroundColor:"pink"
26       }
27     })
28   }
29   handelDragLeave(){//remove the pink style when the node is no longer being dragged
    //over
30     this.setState({
31       style:{}
32     })
33   }
34   render(){//render the node as a table cell
    //generate a list of css classes for this node
35     this.classList = ["node"];
36     //set default values for the node
37     this.draggable = false;
38     this.start = false;
39     this.end = false;
40     //add walls to the node classList
41     if (this.props.wallLeft){
42       this.classList.push("wall_left")
43     }

```

```

44     }
45     if (this.props.wallBottom){
46         this.classList.push("wall_bottom")
47     }
48     //add path to node classList
49     if (this.props.type === "path"){
50         this.classList.push("node_path")
51     }else{
52         //Remove the "node_path" item from the classList if it isn't a path, as when maze
53         //is resolved the nodes would remain a path node if it was a path node before.
54         this.classList.filter(x => {return x !== "node_path"})
55     }
56     //add attributes for the start node or remove them if this node is no longer the
57     //start node
58     if (this.props.pos[0] === this.props.start[0] && this.props.pos[1] === this.props.start[1]){
59         this.classList.push("node_start")
60         this.draggable = true
61         this.start = true
62     }else{
63         this.classList.filter(x => {return x !== "node_start"})
64         this.start = false
65     }
66     //add attributes for the end node or remove them if this node is no longer the end
67     //node
68     if (this.props.pos[0] === this.props.end[0] && this.props.pos[1] === this.props.end[1]){
69         this.classList.push("node_end")
70         this.draggable = true
71         this.end=true
72     }else{
73         this.classList.filter(x => {return x !== "node_end"})
74         this.end=false
75     }
76     if(this.props.index){//Each node is given an index when it is visited so the order
77     //of the visited nodes can be visualized
78     if (this.props.type !== "path"){//Add css animations for to show the visited
79     //nodes
80         this.state.style = {
81             animation: 'visit_node 2s linear forwards',
82             animationDelay: `${this.props.index*this.props.speed}s`
83         }
84     }else{//Add css animations for to show the path nodes
85         this.state.style = {
86             animation: "visit_node_path 2s linear forwards",
87             animationDelay: `${this.props.index*this.props.speed}s`
88         }
89     }
90     }else{
91         //removes the colour if the node is no longer visited after the maze is solved
92         //again
93         if (!this.state.style.backgroundColor){
94             this.state.style = {}
95         }
96     }
97     }
98     return(

```

```
93   <td style={this.state.style} className={this.classList.join(" ")} draggable={this
    .draggable} onDragStart={this.handelDragStart} onDrop={this.handelDrop} onDragOver={
    this.handelDragOver} onDragLeave={this.handelDragLeave}>
94   </td>
95   )
96   }
97 }
```

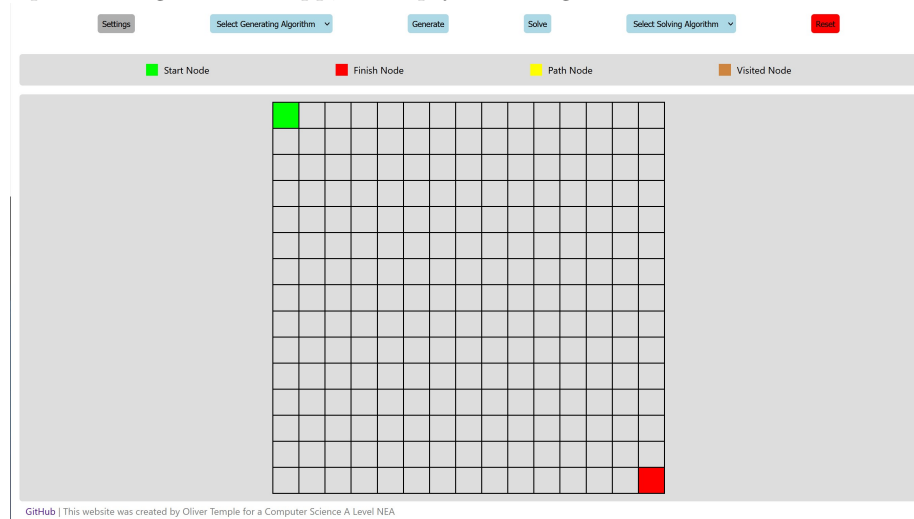
## 5 Testing

To test the project as a whole, I will check the following:

- Test1. Load the react app and check that an empty maze is generated correctly.
- Test2. Change the size of the maze and check that the size changes and doesn't go below 1.
- Test3. Run each of the maze generation algorithms and check that the maze is generated correctly and is solvable.
- Test4. Solve an assortment of mazes with assorted sizes using each of the algorithms (including heuristics for the greedy search) and check that they run correctly.
- Test5. Use different speeds when solving the mazes and check that the speed changes correctly.
- Test6. Reset the maze and check that the maze is reset correctly.
- Test7. Drag and drop the start and finish nodes. Check that the maze resolves correctly.

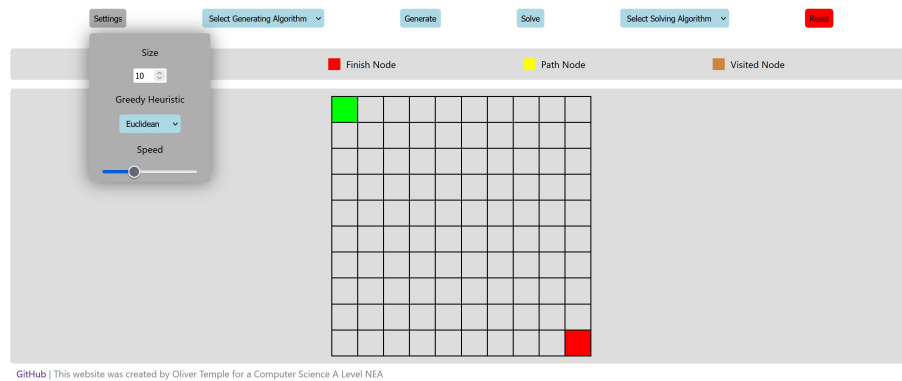
## 5.1 Test1: Pass

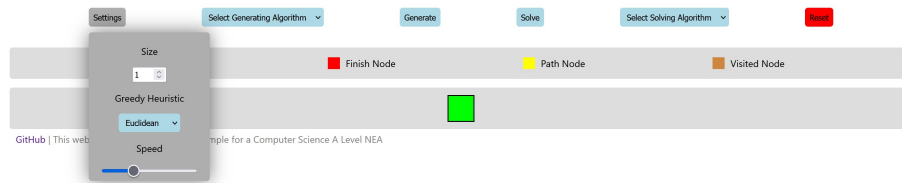
Upon loading the react app, an empty maze is generated.



## 5.2 Test2: Pass

The size can be changed down to 1 and the maze is still generated correctly, you cannot go blow 1.

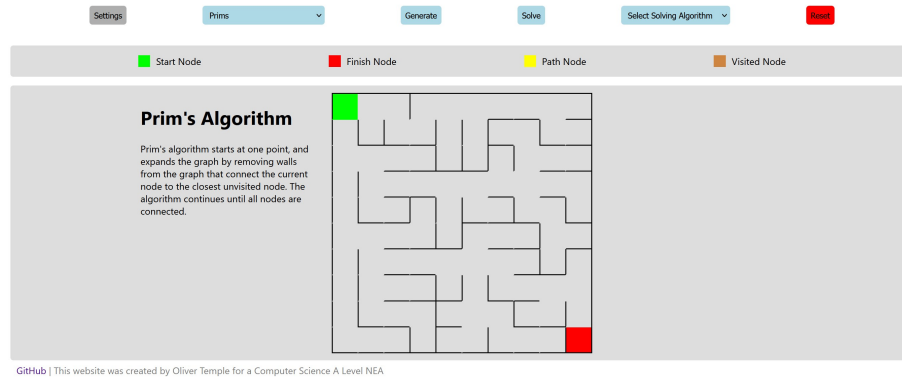
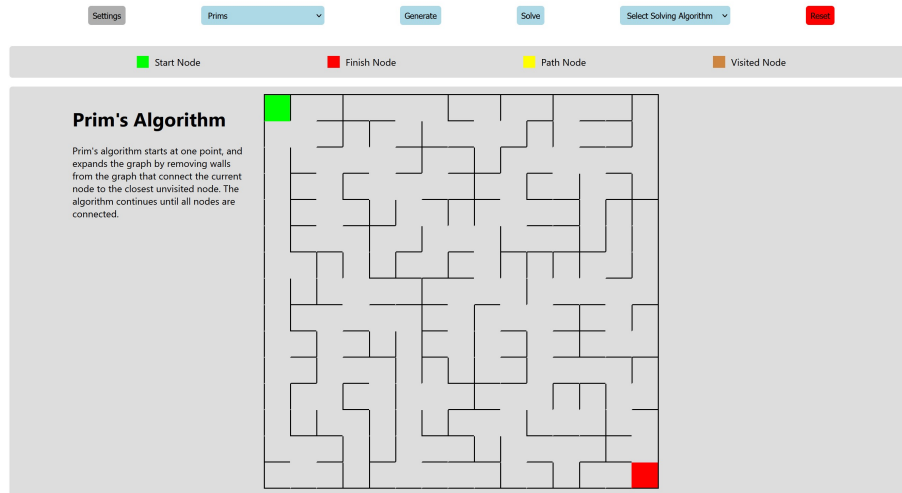




### 5.3 Test3: Pass

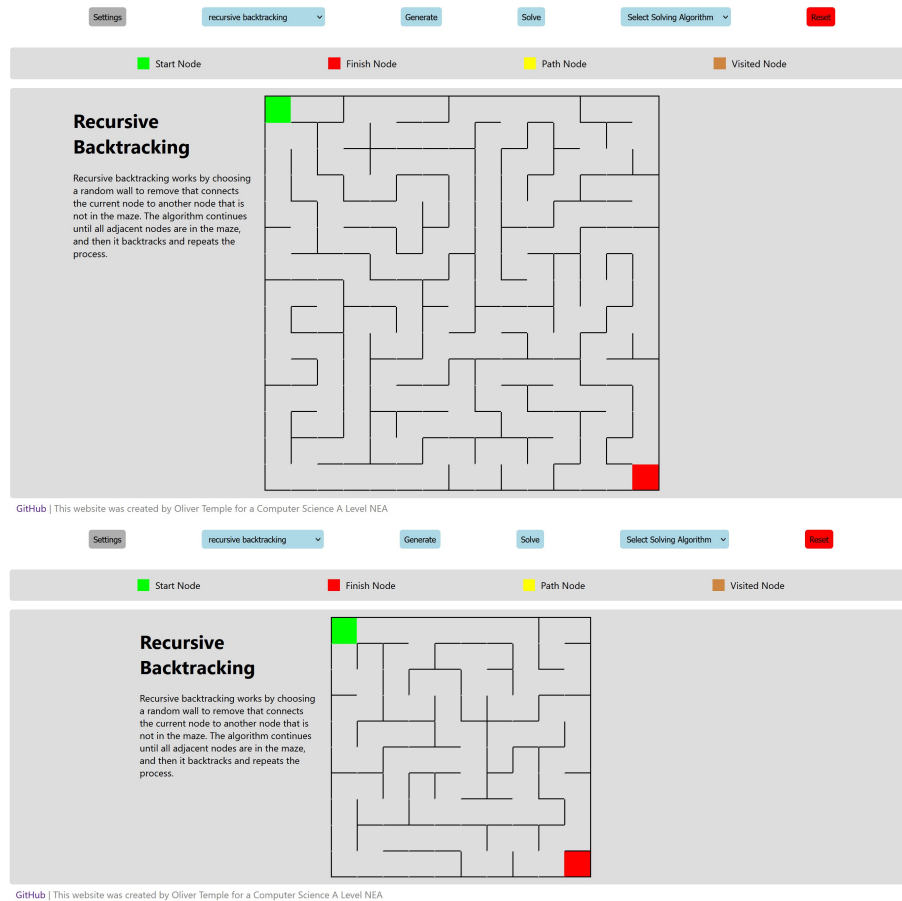
Below are screenshots of a maze being generated using each of the different algorithms, with different size mazes.

### 5.3.1 Prim's Algorithm





### 5.3.2 Recursive Backtracking



### 5.4 Test4/Test5: Pass

Below are links to videos assorted mazes being solved using each of the different algorithms, with different speeds and sizes.

**5.4.1 Dijkstra's Algorithm****5.4.2 Depth First Search****5.4.3 Breadth First Search****5.4.4 Greedy Search Manhattan****5.4.5 Greedy Search Euclidean****5.5 Test6: Pass**

When the reset button is pressed, an empty grid of the current size is generated.

**5.6 Test7**

When moving the start and finish nodes, the maze is resolved, however, sometimes the colours of the start or end nodes are not update properly when they are moved.

**6 Evaluation**