Path-finding Algorithms and Solving Mazes

Oliver Temple

December 2, 2021

Contents

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, Prims 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 fist search and Kruskal's algorithms for maze generation.

1.2.2.1 Maze Generating Algorithms

Prims 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://isaaccomputerscience.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 dijkstr'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

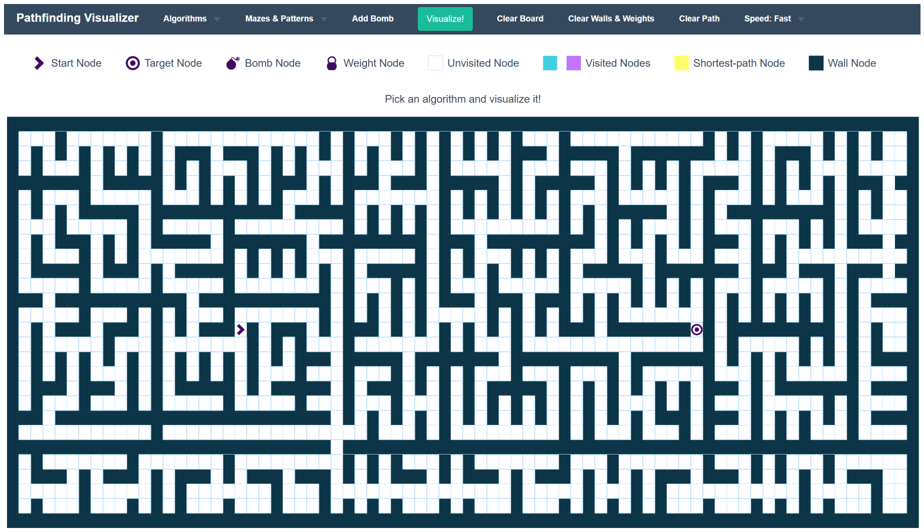
1.3.1.1 Example 1 https://clementmihailescu.github.io/Pathfinding-Visualizer

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:

- 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

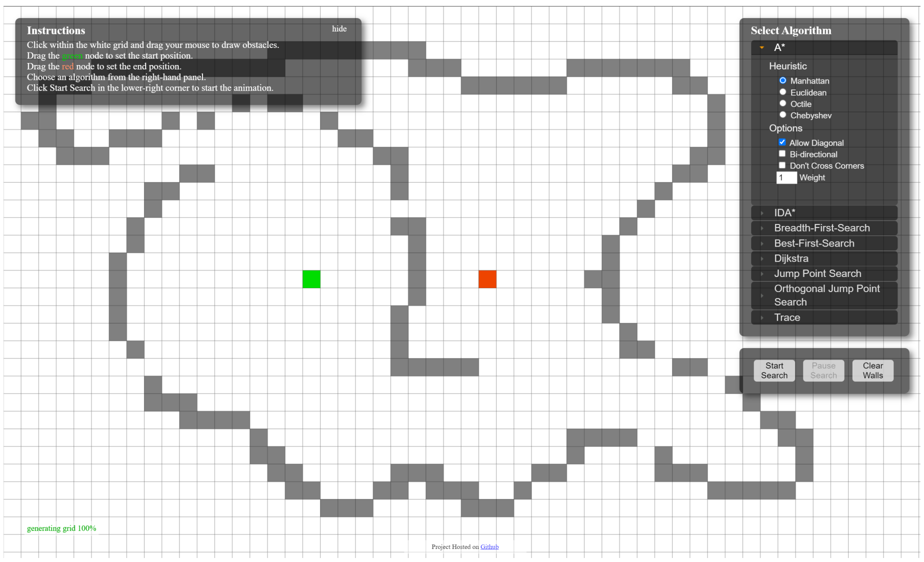
- The visualization is too slow.
- If maze is altered by user after visualization has run, then the visualization will not update.

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

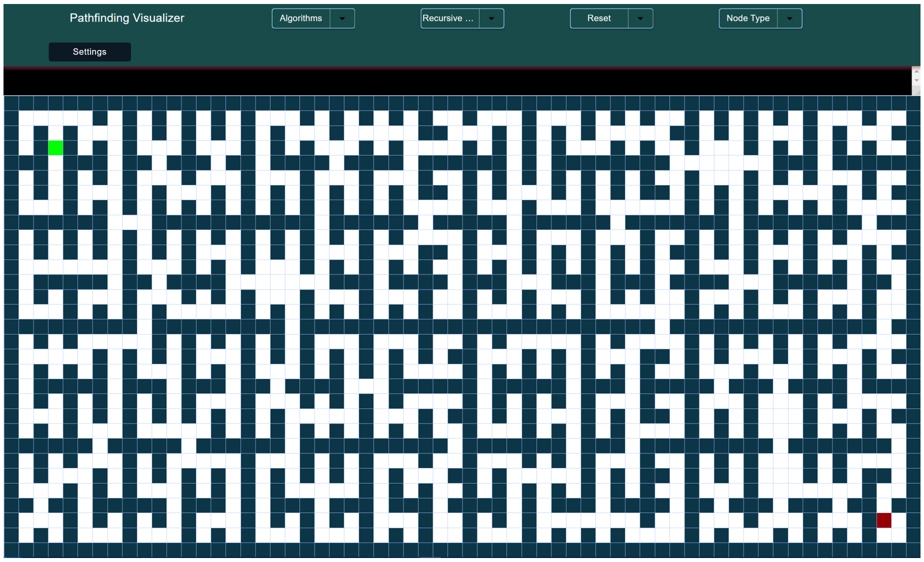
- No maze generation.
- Visualization does not update when maze or start/finnish nodes are changed.

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.
- 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 Prims 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.

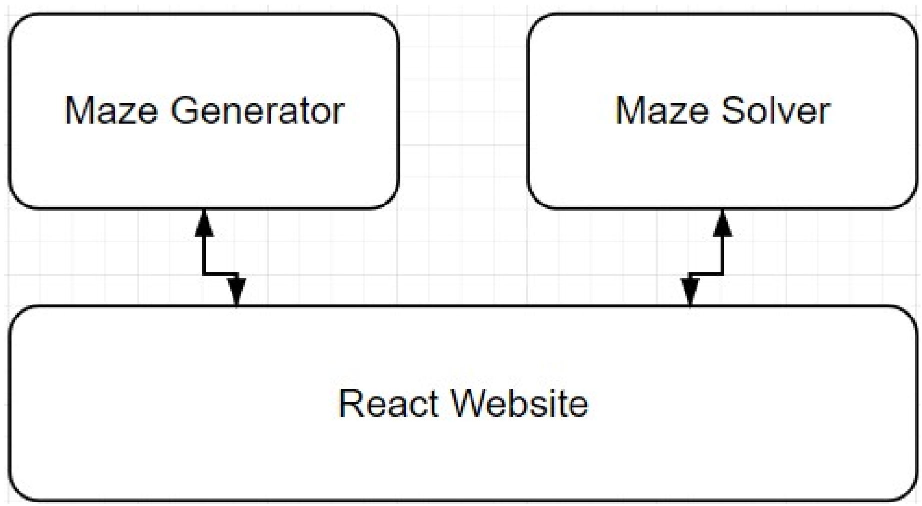
2 Documented Design

2.1 Visualization Structure

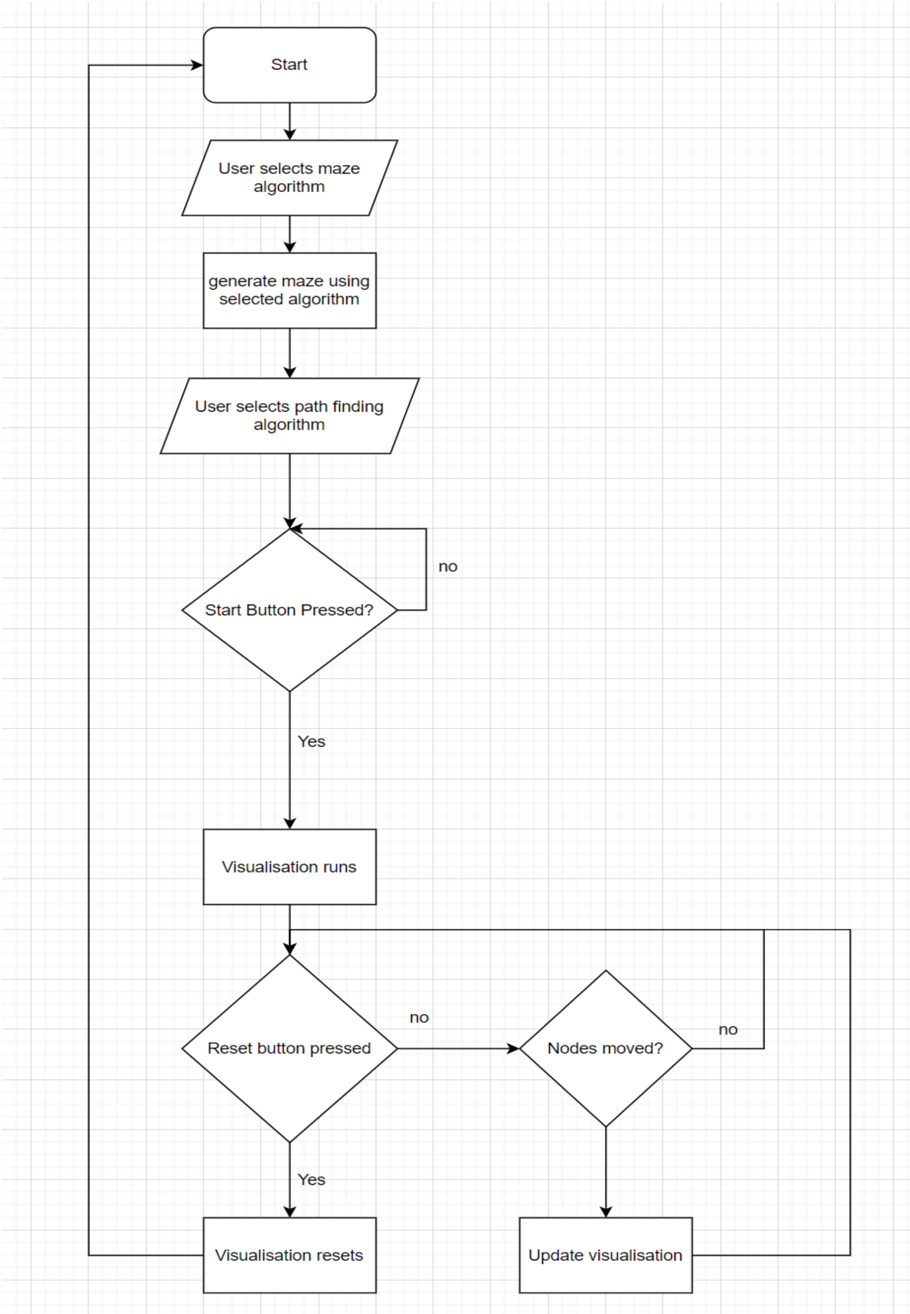
The project will be split into two main sections, the first being the visualization and the second being the python API. The python API will be for generating and solving the mazes, and the react website that will visualize the algorithms. The python API will can be subdivided into two sections, the 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.

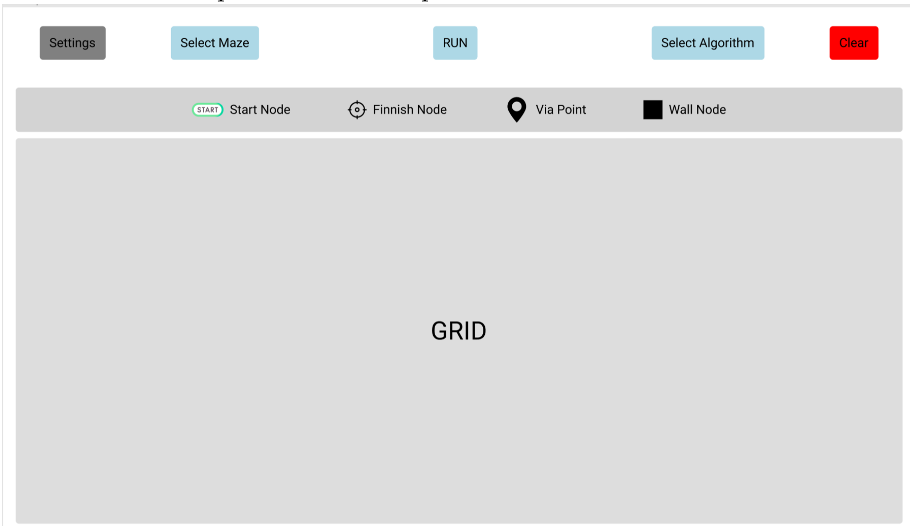

Maze Generator
Maze Solver
React Website

2.2 User Interaction Flow Chart



2.3 User Interface

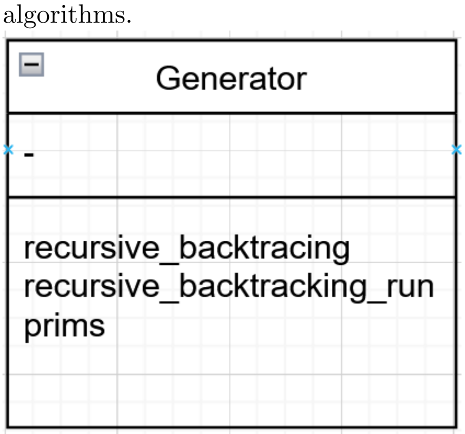
I have used Figma to create a design mockup of how the user interface will look. This allows me to plan what user inputs will be needed.



2.4 Classes Breakdown

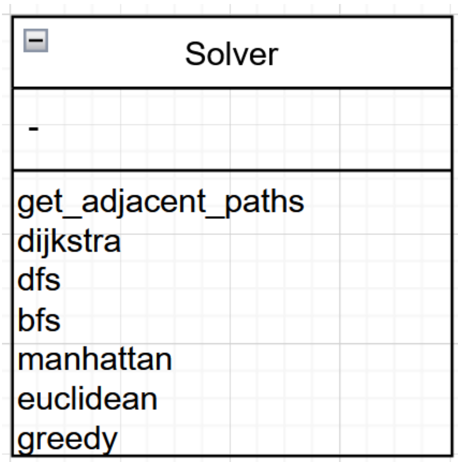
2.4.1 Genereator

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



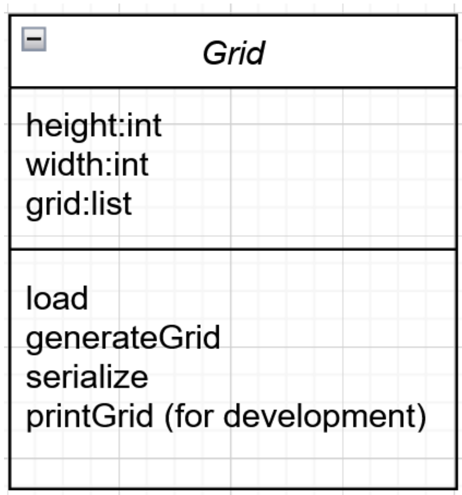
2.4.2 Solver

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



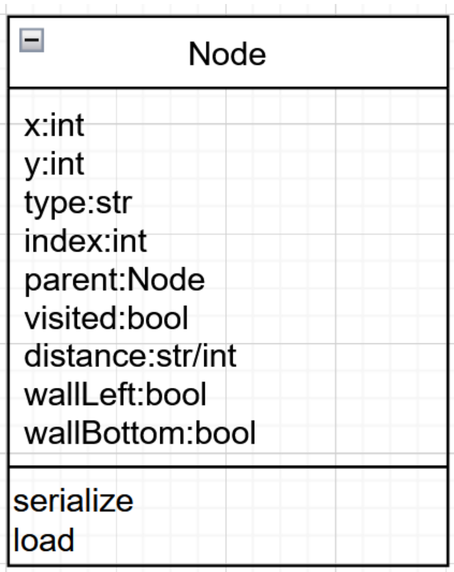
2.4.3 Maze

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.



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.



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

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

Euclidean Distance Euclidean is calculated by

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

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 ber 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

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

```python
import json
from generator import Generator
from grid import Grid
from solver import Solver
def lambda_handler(event, context):
    #get the parameters from the url query
    width = event["queryStringParameters"]["width"]
    height = event["queryStringParameters"]["height"]

    if event["queryStringParameters"]["type"] == "empty_maze":#generate an empty grid
        myGrid = Grid(int(width), int(height))

    elif event["queryStringParameters"]["type"] == "generate":#generate an empty grid, then create a maze
        #get the maze generating algorithm
        generate_algorithm = event["queryStringParameters"]["generate"]

        #create a new generator to generate the maze
        myGenerator = Generator();

        #create a new grid with the height and width algorithms from the query
        myGrid = Grid(int(width),int(height))

        #generate the maze using the algorithm from the query
```

```
        if generate_algorithm == "prims":
            myGenerator.prims(myGrid)
        elif generate_algorithm == "recursive_backtracking":
            myGenerator.recursive_backtracking(myGrid)

    elif event["queryStringParameters"]["type"] == "solve":#solve the maze using
    requested algorithm
        #get the grid from the body of the request
        grid = json.loads(event["body"])

        #generate an empty grid with the same size as the grid from the body
        myGrid = Grid(int(grid["width"]), int(grid["height"]))

        #load the grid from the body into the empty grid by iterating through the nodes
     and changing the attributes
        myGrid.load(grid["grid"])

        #get the solving algorithm from the request
        solve_algorithm = event["queryStringParameters"]["solve"]

        #get the start and node positions from the request
        start = eval(event["queryStringParameters"]["start"])
        end = eval(event["queryStringParameters"]["end"])

        #get the start and end nodes in the grid
        start_node = myGrid.grid[start[0]][start[1]]
        end_node = myGrid.grid[end[0]][end[1]]

        #create a new solver to solve the maze
        mySolver = Solver()

        #solve the maze with the requested algorithm
        if solve_algorithm == "dijkstra":
            mySolver.dijkstra(myGrid, start_node, end_node)
        elif solve_algorithm == "dfs":
            mySolver.dfs(myGrid, start_node, end_node)
        elif solve_algorithm == "bfs":
            mySolver.bfs(myGrid, start_node, end_node)
        elif solve_algorithm == "greedy":
            #get the heuristic for greedy from the request
            heuristic = event["queryStringParameters"]["heuristic"]
            mySolver.greedy(myGrid, start_node, end_node, heuristic)

    #return the json of the grid
    return {
        'statusCode': 200,
        'body': json.dumps(myGrid.serialize())
    }

if __name__ == "__main__":
    myGrid = Grid(15,15)
    myGenerator = Generator()
    myGenerator.prims(myGrid)
    mySolver = Solver()
    mySolver.greedy(myGrid, myGrid.grid[0][0], myGrid.grid[14][14], "manhattan")
    myGrid.printGrid()
```

4.2.2 Python API: grid.py

```python
#Node class for each node in the grid
class Node:
    def __init__(self, x, y, type):
        self.x = x
        self.y = y
        self.type = type

        self.index = None
        self.parent = None
        self.visited = False

        self.distance = "infinity"

        self.wallLeft = True
        self.wallBottom = True
    def __str__(self):#for development purposes
        return f"x:{self.x}, y:{self.y}, type:{self.type}, wallLeft:{self.wallLeft},
    wallBottom:{self.wallBottom}, distance:{self.distance}"
    def load(self, wallLeft, wallBottom):#when loading the grid, the walls must be set
    from the received grid
        self.wallBottom = wallBottom
        self.wallLeft = wallLeft
    def serialize(self):#for returning the grid, the node must be serialized into a
    dictionary
        return {
            "x": self.x,
            "y": self.y,
            "wallLeft": self.wallLeft,
            "wallBottom": self.wallBottom,
            "type": self.type,
            "index":self.index
        }

class Grid:#Grid class for the grid
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.grid = self.generateGrid()
    def load(self, grid):#load the grid from the grid received from the react app
        self.grid = []
        for row in grid:
            row_inner = []
            for item in row:
                node = Node(item["x"], item["y"], "space")
                node.load(item["wallLeft"], item["wallBottom"])
                row_inner.append(node)
            self.grid.append(row_inner)
    def serialize(self):#for returning the grid, the grid must be serialized into a
    dictionary
        obj = {
            "height": self.height,
            "width": self.width,
            "grid": []
        }
        for row in self.grid:
```

```
            row_inner = []
            for item in row:
                row_inner.append(item.serialize())
            obj["grid"].append(row_inner)

        return obj
    def generateGrid(self):#generate the grid
        grid = []
        for i in range(self.height):
            row = []
            for j in range(self.width):
                row.append(Node(j, i, "space"))
            row.append(Node(j, i, "space"))
            grid.append(row)

        row = []
        for j in range(self.width + 1):
            row.append(Node(j, i, "space"))
        grid.append(row)

        return grid
    def printGrid(self):#for development purposes
        width = self.width
        height = self.height

        print(" __"*width)

        for i in range(height):
            for j in range(width):
                cell = ""
                if self.grid[i][j].wallLeft:
                    cell += "|"
                else:
                    cell += " "
                if self.grid[i][j].type == "path":
                    cell += "XX"
                else:
                    cell += "  "
                print(cell, end="")
            print("|")

            for j in range(width):
                cell = ""
                if self.grid[i][j].wallLeft:
                    cell += "|"
                else:
                    cell += " "

                if self.grid[i][j].wallBottom:
                    cell += "__"
                else:
                    cell += "  "
                print(cell, end="")
            print("|")
```

4.2.3 Python API: generator.py

```python
import random
import sys

class Generator:
    def __init__(self):
        pass
    #setup for recursive backtracking
    def recursive_backtracking(self, Grid):
        sys.setrecursionlimit(2000)#set the recursion limit to 2000
        #create a list of all nodes in maze
        unvisited = []
        for row in Grid.grid:
            for item in row:
                unvisited.append(item)

        #pick a random start node
        start = random.choice(unvisited)
        unvisited.remove(start)

        #start recursive backtracking
        self.recursive_backtracking_run(Grid, unvisited, start, [])

    #recursive backtracking algorithm
    def recursive_backtracking_run(self, Grid, unvisited, current, previous):
        #work out which walls can be removed
        orientation_options = []
        if current.x > 0 and Grid.grid[current.y][current.x-1] in unvisited:
            orientation_options.append("left")
        if current.y > 0 and Grid.grid[current.y-1][current.x] in unvisited:
            orientation_options.append("top")
        if current.x < Grid.width - 1 and Grid.grid[current.y][current.x+1] in
    unvisited:
            orientation_options.append("right")
        if current.y < Grid.height - 1 and Grid.grid[current.y+1][current.x] in
    unvisited:
            orientation_options.append("bottom")

        #if there are no walls to remove, backtrack, else pick one and remove it
        if len(orientation_options) > 0:
            #pick a random wall to remove
            orientation = random.choice(orientation_options)
            #add the current node to the previous nodes list
            previous.append(current)

            #remove the wall depending on the orientation
            if orientation == "left":
                connecting_cell = Grid.grid[current.y][current.x - 1]
                current.wallLeft = False

            elif orientation == "bottom":
                connecting_cell = Grid.grid[current.y + 1][current.x]
                current.wallBottom = False

            elif orientation == "right":
                connecting_cell = Grid.grid[current.y][current.x + 1]
                connecting_cell.wallLeft = False

```

```python
            elif orientation == "top":
                connecting_cell = Grid.grid[current.y - 1][current.x]
                connecting_cell.wallBottom = False

            #remove the connecting node from the unvisited list
            unvisited.remove(connecting_cell)

            #recurse
            self.recursive_backtracking_run(Grid, unvisited, connecting_cell, previous)
        else:
            #if not back at the start, backtrack
            if len(previous) > 0:
                new = previous.pop()
                self.recursive_backtracking_run(Grid, unvisited, new, previous)

    #generate a maze using prims algorithm
    def prims(self, Grid):
        #start at the top left node
        inMaze = [[0, 0]]
        #create a list of nodes connected to the start node
        frontier = [[[0, 1],["left"]],[[1, 0],["top"]]]

        #while there are still nodes to visit
        while (len(frontier) > 0):
            #pick a random node from the frontier
            new = frontier.pop(random.randint(0, len(frontier)-1))

            toAdd = new[0]

            #pick a random wall from the available walls
            wall = new[1][random.randint(0, len(new[1])-1)]

            #add the wall to the inMaze list
            inMaze.append(toAdd)

            #remove the selected wall from the grid
            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

            #calculate the possible nodes to add to the frontier
            possible = [[toAdd[0]-1, toAdd[1]],[toAdd[0]+1,toAdd[1]],[toAdd[0],toAdd[1]-1],[toAdd[0], toAdd[1]+1]]

            #possible walls
            walls = ["bottom", "top", "right", "left"]
            #iterate through the possible nodes
            for i in range(4):
                p = possible[i]
```

```python
                wall = walls[i]
                #check that the wall can be removed
                if 0<=p[0]<Grid.height and 0<=p[1]<Grid.width:
                    #check that the node is not already in the maze
                    if p not in inMaze:
                        found = False
                        #check if the node is already in the frontier, if it is then add the wall to the possible walls for the node in the frontier
                        for v in frontier:
                            if v[0]==p:
                                v[1].append(wall)
                                found = True
                        #if the node is not in the frontier, add it to the frontier
                        if not found:
                            frontier.append([p,[wall]])
```

4.2.4 Python API: solver.py

```python
import math
class Solver:
    def __init__(self):
        pass

    def get_adjacent_paths(self, Grid, node):#Get adjacent nodes in the grid that are not blocked by a wall
        paths = []
        if node.x > 0 and not node.wallLeft: #check that the node is not on the left edge and that it doesn't have a wall on the left
            paths.append(Grid.grid[node.y][node.x - 1])
        if node.x < Grid.width - 1 and not Grid.grid[node.y][node.x + 1].wallLeft: #check that the node is not on the right edge, and that there is not a wall to the right of it
            paths.append(Grid.grid[node.y][node.x + 1])
        if node.y > 0 and not Grid.grid[node.y - 1][node.x].wallBottom: #check that the node is not at the top and that there isn't a wall above it
            paths.append(Grid.grid[node.y - 1][node.x])
        if node.y < Grid.height - 1 and not node.wallBottom:#check that the node is not at the bottom and that there is no wall below it
            paths.append(Grid.grid[node.y + 1][node.x])

        #retrun a list of the available nodes
        return paths

    def dijkstra(self, Grid, start, end):#Dijkstra's algorithm for solving the grid
        #set the distance from the start node to the start node to 0
        start.distance = 0
        #set the start index as 0
        #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.
        index = 0
        #create a queue of unvisited nodes
        unvisited = [start]
        #create a flag
        found = False
        while not found:
            #take the next node from the front of the queue
```

```
            current = unvisited.pop(0)
            #mark the node as visited
            current.visited = True
            #set the index on the node
            current.index = index
            #increment the index
            index += 1
            #iterate through the available adjacent nodes
            for node in self.get_adjacent_paths(Grid, current):
                #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
                if not node.visited:
                    node.distance = current.distance + 1
                    unvisited.append(node)

                #if the node is the end node, update the flag and exit the algorithm
                if node == end:
                    found = True
                    break

        #backtrack from the end node to the start node, picking the node with the smallest distance at every point
        current = end
        path = [end]
        while current != start:
            #get connecting cells
            connecting = self.get_adjacent_paths(Grid, current)
            #work out which node as the lowest distance
            min = None
            for node in connecting:
                if node.distance != "infinity":
                    if min == None or node.distance < min.distance:
                        min = node
            #append the node with the lowest distance to the path
            path.append(min)
            #update the current node
            current = min

        #draw the path
        for node in path:
            node.type = "path"

    def dfs(self, Grid, start, end):
        #create a stack of nodes to visit
        stack = [start]
        #set the start index as 0
        #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.
        start.index = 0
        index = 1
        #while the stack is not empty, keep searching
        while len(stack) > 0:
            #get the item at the top of the stack
            current = stack[-1]
            #mark the item as visited
```

```
            current.visited = True
            #check if the end has been found
            if current == end:
                break

            #find connecting nodes that have not been visited
            possible = []
            for node in self.get_adjacent_paths(Grid, current):
                if not node.visited:
                    possible.append(node)

            #if there are possible connections, choose the first one
            if len(possible) > 0:
                to_append = possible[0]
                #set the index
                to_append.index = index
                #set the parent node for drawing the path
                to_append.parent = current
                #increment the index
                index += 1
                #add the new node to the stack
                stack.append(to_append)
            #If there are no possible connections, remove the current node from the stack
            else:
                stack.pop()

        #backtrack from the end to the start drawing the path
        while current != start:
            current.type = "path"
            current = current.parent

        start.type = "path"

    def bfs(self, Grid, start, end):
        #create a queue of nodes that need to be visited
        queue = [start]
        #set the start index as 0
        #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.
        start.index = 0
        index = 1
        #only run the algorithm while there are nodes to visit
        while len(queue) > 0:
            #get the first item in the queue and mark as visitied
            current = queue.pop(0)
            current.visited = True

            #check to see if the end node has been found
            if current == end:
                break

            #add unvisited adjacent nodes to the queue
            for node in self.get_adjacent_paths(Grid, current):
                if not node.visited:
                    #update nodes parent for drawing path
```

```
                    node.parent = current
                    #add index
                    node.index = index
                    #increment the index
                    index += 1
                    #append node to queue
                    queue.append(node)

        #backtrack from the end to the start and draw the path
        while current != start:
            current.type = "path"
            current = current.parent

        start.type = "path"

    def manhattan(self, node1, node2):#calculate the manhattan distance between two nodes
        return abs(node1.x - node2.x) + abs(node1.y - node2.y)

    def euclidean(self, node1, node2):#calculate the euclidean distance between two nodes
        return math.sqrt((node1.x - node2.x)**2 + (node1.y - node2.y)**2)

    def greedy(self, Grid, start, end, heuristic):
        #create a priority queue for nodes that need to be visited
        queue = [start]
        #set the start index as 0
        #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.
        start.index = 0
        index = 1
        #while there are nodes to visit
        while len(queue) > 0:
            #pop the node off of the front of the queue and mark as visited
            current = queue.pop(0)
            current.visited = True

            #check if the end node has been found
            if current == end:
                break

            #iterate through unvisited adjacent nodes
            for node in self.get_adjacent_paths(Grid, current):
                if not node.visited:
                    #update the parent of the node
                    node.parent = current
                    #update the index
                    node.index = index
                    #increment the index
                    index += 1
                    #use the selected heuristic to update the distance
                    if heuristic == "manhattan":
                        node.distance = self.manhattan(node, end)
                    elif heuristic == "euclidean":
                        node.distance = self.euclidean(node, end)
                    #insert node in the priority queue
```

```
                    for item in queue:
                        if item.distance > node.distance:
                            queue.insert(queue.index(item), node)
                            break
                    queue.append(node)

        #backtrack from the start and draw the path
        while current != start:
            current.type = "path"
            current = current.parent

        start.type = "path"
```

4.2.5 React App: App.js

```
import './App.css';
import { Component } from 'react';
import DisplayGrid from './components/DisplayGrid';
import Menu from './components/Menu';
import MenuKey from './components/MenuKey';
import Footer from './components/Footer';
class App extends Component {
 constructor(props){
  super(props);
  this.state = {
   grid: null,//the grid of nodes
   algorithm: null,//algorithm for generating the maze
   solve:null,//algorithm for solving the maze
   nodes:{
    start: [0,0],//position of the start node
    end: [null, null]//position of the end node
   },
   size: {//size of the maze
    width:15,
    height:15
   },
   heuristic: "euclidean",
   speed:0.1
  }

  this.solved = false;
  this.maze = false;
  //bind the methods to the object so that the "this" keyword refers to the object no matter where the method is called from
  this.fetchGrid = this.fetchGrid.bind(this);
  this.setAlgorithm = this.setAlgorithm.bind(this);
  this.clearGrid = this.clearGrid.bind(this);
  this.setSolve = this.setSolve.bind(this);
  this.solveGrid = this.solveGrid.bind(this);
  this.setSize = this.setSize.bind(this);
  this.setStart = this.setStart.bind(this);
  this.setEnd = this.setEnd.bind(this);
  this.should_solve = this.should_solve.bind(this);
  this.setHeuristic = this.setHeuristic.bind(this);
  this.setSpeed = this.setSpeed.bind(this);
 }

```

```
componentDidMount(){
 //generate a new maze empty when the page loads
 this.clearGrid();
}

setHeuristic(heuristic){//set the heuristic for the greedy algorithm
 this.setState({heuristic: heuristic});
}
setSpeed(speed){//set the speed of the animation
 this.clearGrid();
 this.setState({speed: speed});
}
setSize(size){//set the size of the grid when changed in settings
 this.setState({
  size: size
 }, () => {//setState is asynchronous, so we need to wait for it to finish before
   running the following code
  if (size.width > 0 && size.height > 0){//if the size is valid, generate a new maze
   this.clearGrid();
  }
 })
}
setStart(node){//set the start node
 this.setState({
  nodes:{
   start: node,
   end: this.state.nodes.end
  }
 }, () => {//setState is asynchronous, so we need to wait for it to finish before
   running the following code
  this.should_solve()//solve the maze again if it is already solved
 })
}
setEnd(node){//set the end node
 this.setState({
  nodes:{
   start: this.state.nodes.start,
   end: node
  }
 }, () => {//setState is asynchronous, so we need to wait for it to finish before
   running the following code
  this.should_solve()//solve the maze again if it is already solved
 })
}

async should_solve(){//if the maze is already solved, then solve again. Only run when
   the start or end nodes are changed
 if (this.solved){
  this.clear_node_index()//clear the index of the nodes, as the maze is being solved
   again
  .then(() => {
   this.solveGrid();
  })
 }
}

async clear_node_index(){//clear the index of the nodes so that the maze can be solved
```

```
    again
  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
   let grid = this.state.grid.grid
   //iterate through the grid and clear the index of the nodes
   for (let i=0; i<this.state.size.height; i++){
    for(let j=0; j<this.state.size.width; j++){
     grid[i][j].index = null;
    }
   }
   //update the state with the grid that has been cleared of the index's
   this.setState({
    grid: {
     grid: grid,
     height: this.state.grid.height,
     width: this.state.grid.width
    }
   }, () => {
    resolve(); // resolve the promise once the state has been updated
   })
  })

 }
 async fetchGrid(){//generate a new maze from the python API using the selected algorithm
  if (this.state.algorithm){//check that there is an algorithm selected for generating the maze
   let grid = await fetch(`https://jkrlv64tsl.execute-api.eu-west-2.amazonaws.com/default/NEA?type=generate&width=${this.state.size.width}&height=${this.state.size.height}&generate=${this.state.algorithm}`)//fetch the generated grid from the python API
   grid = await grid.json();//convert the response to json
   this.setState({//update the state with the new grid
    grid:grid
   })
   this.solved = false;//the maze is no longer solved
   this.maze = true;//set the maze to true as a maze has been generated
  }else{//If there is no algorithm selected to generate the maze, alert the user
   alert("Please select a maze generating algorithm")
  }
 }
 async solveGrid(){//send the maze to the python API to be solved with the requested algorithm
  await this.clear_node_index();//clear the index of the nodes, as the maze is being solved again
  if (this.maze && this.state.solve){//check that there is a maze and that there is an algorithm selected for solving the maze
   let grid = await fetch(
    `https://jkrlv64tsl.execute-api.eu-west-2.amazonaws.com/default/NEA?type=solve&width=${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}`, {
    method: "POST",
    body: JSON.stringify(this.state.grid)//set the body of the request to the grid
   })//send the maze to the python API to be solved, with the selected algorithm as a parameter
```

```
   grid = await grid.json();//convert the response to json
   this.setState({//update the state with the new grid
    grid:grid
   })
   this.solved = true;//the maze is now solved
  }else{
   if (!this.maze){//if there is no maze, alert the user that there is no maze to solve
    alert("Please generate a maze")
   }else{
    alert("Please select a solving algorithm")//If there is no algorithm selected to solve the maze, alert the user
   }
  }

 }
 async clearGrid(){//generate an empty maze from the API
  let grid = await fetch(`https://jkrlv64tsl.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
  grid = await grid.json();//convert the response to json
  this.setState({//update the state
   grid:grid,//update the state with the new grid
   nodes:{//set the start and end nodes to default positions
    start: [0,0],
    end: [this.state.size.height - 1, this.state.size.width - 1]
   }
  })
  this.maze = false;//there is no longer a maze to solve
  this.solved = false;//the maze is no longer solved
 }
 setAlgorithm(algorithm){//set the maze generating algorithm
  this.setState({
   algorithm: algorithm
  })
 }
 setSolve(algorithm){//set the maze solving algorithm
  this.setState({
   solve:algorithm
  })
 }
 render(){
  return (
   <div className="App">
    <Menu
     setAlgorithm={this.setAlgorithm}//callback function to set the generation algorithm from the menu
     setSolve={this.setSolve}//callback function to set the solving algorithm from the menu
     generate={this.fetchGrid}//callback function to generate a new maze from the menu
     clearGrid={this.clearGrid}//callback function to clear the maze from the menu
     solve={this.solveGrid}//callback function to solve the maze from the menu
     size={this.state.size}//the size of the maze
     setSize={this.setSize}//callback function to set the size of the maze from the menu
     setHeuristic={this.setHeuristic}//callback function to set the heuristic from the menu
     setSpeed={this.setSpeed}//callback function to set the speed from the menu
```

```
     speed={this.state.speed}//the speed of the maze
    />
    <MenuKey />
    <DisplayGrid
     grid={this.state.grid}//the grid of the maze
     nodes={this.state.nodes}//the start and end nodes
     size={this.state.size}//the size of the maze
     setStart={this.setStart}//callback function to set the start node
     setEnd={this.setEnd}//callback function to set the end node
     generateAlgorithm={this.state.algorithm}//the algorithm used to generate the maze
     solveAlgorithm={this.state.solve}//the algorithm used to solve the maze
     heuristic={this.state.heuristic}//the heuristic used for the greedy algorithm
     speed={this.state.speed}//the speed of the animation
    />
    <Footer />
   </div>
  );
 }
}
export default App;
```

4.2.6 React App: Menu.jsx

```
import React from 'react';
import Settings from './Settings';
export default function Menu(props) {//Menu bar for the app
    return(
      <div className="menu">
        <Settings
          size={props.size}//size of the maze
          setSize={props.setSize}//callback to set the size of the maze from the settings
          setHeuristic={props.setHeuristic}//callback to set the heuristic from the settings
          setSpeed={props.setSpeed}//callback to set the speed from the settings
          speed={props.speed}//speed of the animation
        />
        <select className="algorithms" name="algorithms" id="algorithms" onChange={(e) => {props.setAlgorithm(e.target.value)}}>
          <option value="select">Select Generating Algorithm</option>
          <option value="prims">Prims</option>
          <option value="recursive_backtracking">recursive backtracking</option>
        </select>
        <button className="button" onClick={props.generate}>Generate</button>
        <button className="button" onClick={props.solve}>Solve</button>
        <select className="algorithms" name="algorithms" id="algorithms" onChange={(e) => {props.setSolve(e.target.value)}}>
          <option value="select">Select Solving Algorithm</option>
          <option value="dijkstra">Dijkstra</option>
          <option value="dfs">Depth First Search</option>
          <option value="bfs">Breadth First Search</option>
          <option value="greedy">Greedy</option>
        </select>
        <button className="button clear" onClick={props.clearGrid}>Reset</button>
      </div>
    )
}
```

4.2.7 React App: MenuKey.jsx

```
import React from "react";

export default function MenuKey(props){//key for showing the different types of node
    return(
      <div className="key">
        <div className="key_item">
          <div className="key_node_start"></div>
          <p>Start Node</p>
        </div>
        <div className="key_item">
          <div className="key_node_end"></div>
          <p>Finish Node</p>
        </div>
        <div className="key_item">
          <div className="key_node_path"></div>
          <p>Path Node</p>
        </div>
        <div className="key_item">
          <div className="key_node_visited_node"></div>
          <p>Visited Node</p>
        </div>
      </div>
    )
}
```

4.2.8 React App: DisplayGrid.jsx

```
import React, { Component } from "react";
import DisplayNode from "./DisplayNode";
import GeneratorInfo from "./GeneratorInfo";
import SolverInfo from "./SolverInfo";
export default class DisplayGrid extends Component{
  constructor(props){
    super(props);
    this.state = {
      dragObject: ""
    }
    //bind the methods to the object so that the "this" keyword refers to the object no
     matter where the method is called from
    this.renderTable = this.renderTable.bind(this);
    this.handelDrop = this.handelDrop.bind(this);
    this.setDragObject = this.setDragObject.bind(this);
  }
  handelDrop(pos){//move the node that was being dragged to the new position
    switch (this.state.dragObject){
      case "start":
        this.props.setStart(pos)
        break;
      case "end":
        this.props.setEnd(pos)
        break;
      default:
        break;
    }
  }
```

```
setDragObject(type){//set weather start or end node is being dragged
  this.setState({
    dragObject:type
  })
}
renderTable(){//render the grid as a table
  return(
    <table>
      <tbody className="column">
        {Array.from(Array(this.props.grid.height).keys()).map((_, i) => {//iterate through the rows of the grid
            return(
                <tr className={`row wall_right ${i === 0 ? "wall_top" : ""}`} key={i}>
                  {Array.from(Array(this.props.grid.width).keys()).map((_, j) => {//iterate through the nodes in each row
                      return(
                        <DisplayNode
                          key={j}
                          wallLeft={this.props.grid.grid[i][j].wallLeft} //bool: is there a wall to the left of this node
                          wallBottom={this.props.grid.grid[i][j].wallBottom} //bool: is there a wall below this node
                          pos={[i, j]} //position of the node
                          start={this.props.nodes.start} //position of the start node
                          end={this.props.nodes.end} //position of the end node
                          handelDrop = {this.handelDrop} //callback function to move the start or end node to a new position
                          setDragObject={this.setDragObject} // callback function to set weather the start or end node is being dragged
                          type={this.props.grid.grid[i][j].type} //type of node
                          index={this.props.grid.grid[i][j].index} //index of the node for visualization
                          speed={this.props.speed} //speed of the animation
                        />
                      )
                  })}
                </tr>
            )
        })}
      </tbody>
    </table>
  )

}
render(){
  //If there is a grid, render it, else show a message
  if (this.props.grid){
    return(
      <div className="grid" style={{padding:10}}>
        <GeneratorInfo generator={this.props.generateAlgorithm}/>
        <this.renderTable />
        <SolverInfo solver={this.props.solveAlgorithm} heuristic={this.props.heuristic}/>
      </div>
    )
  }else{
    return(
```

```
        <div className="grid message column">
          <h1>No grid to display</h1>
          <h2>Check your internet connection</h2>
        </div>
      )
    }
  }
}
```

4.2.9 React App: DisplayNode.jsx

```
import React from "react"
export default class DisplayNode extends React.Component{
  constructor(props){
    super(props)
    this.state = {
      style:{}
    }
    //bind the methods to the object so that the "this" keyword refers to the object no matter where the method is called from
    this.handelDragStart = this.handelDragStart.bind(this);
    this.handelDragLeave = this.handelDragLeave.bind(this);
    this.handelDragOver = this.handelDragOver.bind(this);
    this.handelDrop = this.handelDrop.bind(this);
  }
  handelDragStart(){//set the type of node that is being dragged
    this.props.setDragObject(this.start ? "start" : this.end ? "end" : "")
  }
  handelDrop(){//move the node that was being dragged to the new position
    this.setState({style:{}});
    this.props.handelDrop(this.props.pos)
  }
  handelDragOver(e){//when another node is dragged over this node, set the style of the node to be pink
    e.preventDefault();
    this.setState({
      style:{
        backgroundColor:"pink"
      }
    })
  }
  handelDragLeave(){//remove the pink style when the node is no longer being dragged over
    this.setState({
      style:{}
    })
  }
  render(){//render the node as a table cell
    //generate a list of css classes for this node
    this.classList = ["node"];
    //set default values for the node
    this.draggable = false;
    this.start = false;
    this.end = false;
    //add walls to the node classList
    if (this.props.wallLeft){
      this.classList.push("wall_left")
```

```
    }
    if (this.props.wallBottom){
      this.classList.push("wall_bottom")
    }
    //add path to node classList
    if (this.props.type === "path"){
      this.classList.push("node_path")
    }else{
      //Remove the "node_path" item from the classList if it isn't a path, as when maze is resolved the nodes would remain a path node if it was a path node before.
      this.classList.filter(x => {return x !== "node_path"})
    }
    //add attributes for the start node or remove them if this node is no longer the start node
    if (this.props.pos[0] === this.props.start[0] && this.props.pos[1] === this.props.start[1]){
      this.classList.push("node_start")
      this.draggable = true
      this.start = true
    }else{
      this.classList.filter(x => {return x !== "node_start"})
      this.start = false
    }
    //add attributes for the end node or remove them if this node is no longer the end node
    if (this.props.pos[0] === this.props.end[0] && this.props.pos[1] === this.props.end[1]){
      this.classList.push("node_end")
      this.draggable = true
      this.end=true
    }else{
      this.classList.filter(x => {return x !== "node_end"})
      this.end=false
    }

    if(this.props.index){//Each node is given an index when it is visited so the order of the visited nodes can be visualized
      if (this.props.type !== "path"){//Add css animations for to show the visited nodes
        this.state.style = {
          animation: `visit_node 2s linear forwards`,
          animationDelay: `${this.props.index*this.props.speed}s`
        }
      }else{//Add css animations for to show the path nodes
        this.state.style = {
          animation: "visit_node_path 2s linear forwards",
          animationDelay: `${this.props.index*this.props.speed}s`
        }
      }
    }else{
      //removes the colour if the node is no longer visited after the maze is solved again
      if (!this.state.style.backgroundColor){
        this.state.style = {}
      }
    }
    return(
```

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

5 Testing

6 Evaluation