PCB Auto-routing

Computer Science Coursework

Edwin Shepherd

# Contents

[Contents 1](#_Toc3112653)

[Analysis 3](#_Toc3112654)

[The Problem 3](#_Toc3112655)

[What is routing 3](#_Toc3112656)

[Different types of routing 4](#_Toc3112657)

[Problems with routing 5](#_Toc3112658)

[Definition of distances 5](#_Toc3112659)

[Users 6](#_Toc3112660)

[Limitations 7](#_Toc3112661)

[Prerequisites and missing requirements 8](#_Toc3112662)

[Requirements 9](#_Toc3112663)

[Core Requirements 9](#_Toc3112664)

[Extensions 10](#_Toc3112665)

[Prototyping 11](#_Toc3112666)

[Objectives 11](#_Toc3112667)

[What I found out 11](#_Toc3112668)

[Data structure 12](#_Toc3112669)

[Documented Design 12](#_Toc3112670)

[Overview 12](#_Toc3112671)

[Server Side 14](#_Toc3112672)

[Client Side 15](#_Toc3112673)

[Pesudo Code snippets 16](#_Toc3112674)

[A\* 16](#_Toc3112675)

[Highlevel Routing 18](#_Toc3112676)

[User Interface 22](#_Toc3112677)

[UML Digram 26](#_Toc3112678)

[Data structures 27](#_Toc3112679)

[Screen Shots 28](#_Toc3112680)

[Testing 30](#_Toc3112681)

[Tests 30](#_Toc3112682)

[Tests 30](#_Toc3112683)

[ScreenShots 31](#_Toc3112684)

[Demonstration of overall effectiveness 34](#_Toc3112685)

[Evaluation 35](#_Toc3112686)

[Evaluation with End User 35](#_Toc3112687)

[Comment from end user 35](#_Toc3112688)

[Evaluation of comment from end user 35](#_Toc3112689)

[Completeness 36](#_Toc3112690)

[Objective review 36](#_Toc3112691)

[Extensions 39](#_Toc3112692)

[Effectiveness 42](#_Toc3112693)

[Strengths 42](#_Toc3112694)

[Improvements 42](#_Toc3112695)

[Bibliography 43](#_Toc3112696)

[Technical solution 44](#_Toc3112697)

[Contents 44](#_Toc3112698)

[Server side 46](#_Toc3112699)

[Index.js 46](#_Toc3112700)

[BoardRouter.js 51](#_Toc3112701)

[NetRouter.js 56](#_Toc3112702)

[Board.js 59](#_Toc3112703)

[Colour.js 65](#_Toc3112704)

[SVG.js 66](#_Toc3112705)

[Client Side 69](#_Toc3112706)

[Index.html 69](#_Toc3112707)

[Index.css 70](#_Toc3112708)

[Indexscript.js 72](#_Toc3112709)

# Analysis

## The Problem

### What is routing

Electronics engineers develop circuits which are often printed on to a circuit board (PCB), keeping wiring tidy. However routing the wires is a very tedious task. Additionally the wiring must conform to rules or else malfunctions in the circuit may occur. For example, high current tracks must be of a set width, certain components cannot have tracks under them. This is takes more time and is difficult to ensure 100% accuracy.

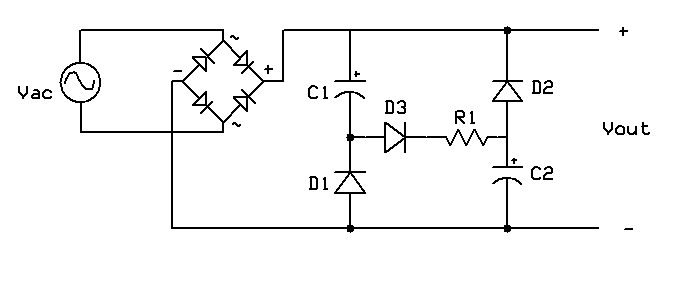
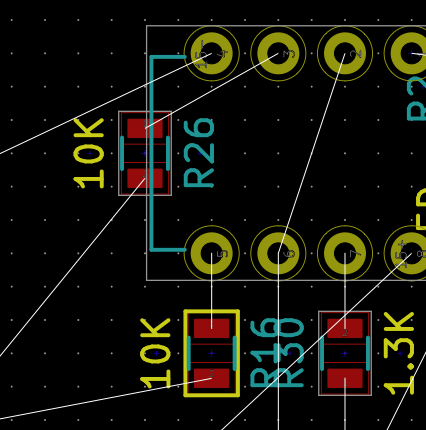
[](https://www.google.co.uk/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjSqtfQ-NPdAhVIWBoKHcJ_DAsQjRx6BAgBEAU&url=https://en.wikipedia.org/wiki/File:Valley-fill_circuit_schematic_1.png&psig=AOvVaw1u6-1MKZyxtGEkY9-Jxwfn&ust=1537888868345021)To solve this problem PCB Routing Software is common place in PCB design tools. Its purpose is to automatically route all of the components and to perform “Design Rule Checking” DRC. PCB software usually has two modes one where the circuit is designed:

Figure 1 a schematic for a basic diode rectifier coupled with a valley-fill circuit - Image credit: WikiMedia available: <https://goo.gl/NJdaqG> , accessed 16:28 24/09/2018

[](https://www.google.co.uk/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwj-0Oz0rqveAhUOThoKHfDXAwgQjRx6BAgBEAU&url=https://electronics.stackexchange.com/questions/200093/what-does-yellow-border-around-component-in-kicad-pcbnew-mean&psig=AOvVaw12652bQlXOE6Df1Zygvx7C&ust=1540892724447123)

There is then another mode where the connections between all of the contact points (pads) for the components are connected. EDA (Electronic Design Application) tools usually help with this by showing which nets need to be routed with straight grey lines. The connections then need to be made between all of the points. Making these connections is the role of what my program is meant to do. Finally the connections must be converted into a format in which they can be printed.

In this project I will focus on the challenges that are presented by the routing.

Figure 2 - Nets are shown in white straight lines. This screen shot is taken from KiCAD's PCBnew software. Image credit; Scanny available: <https://goo.gl/muyghy> date accessed 29/10/2018

### Different types of routing

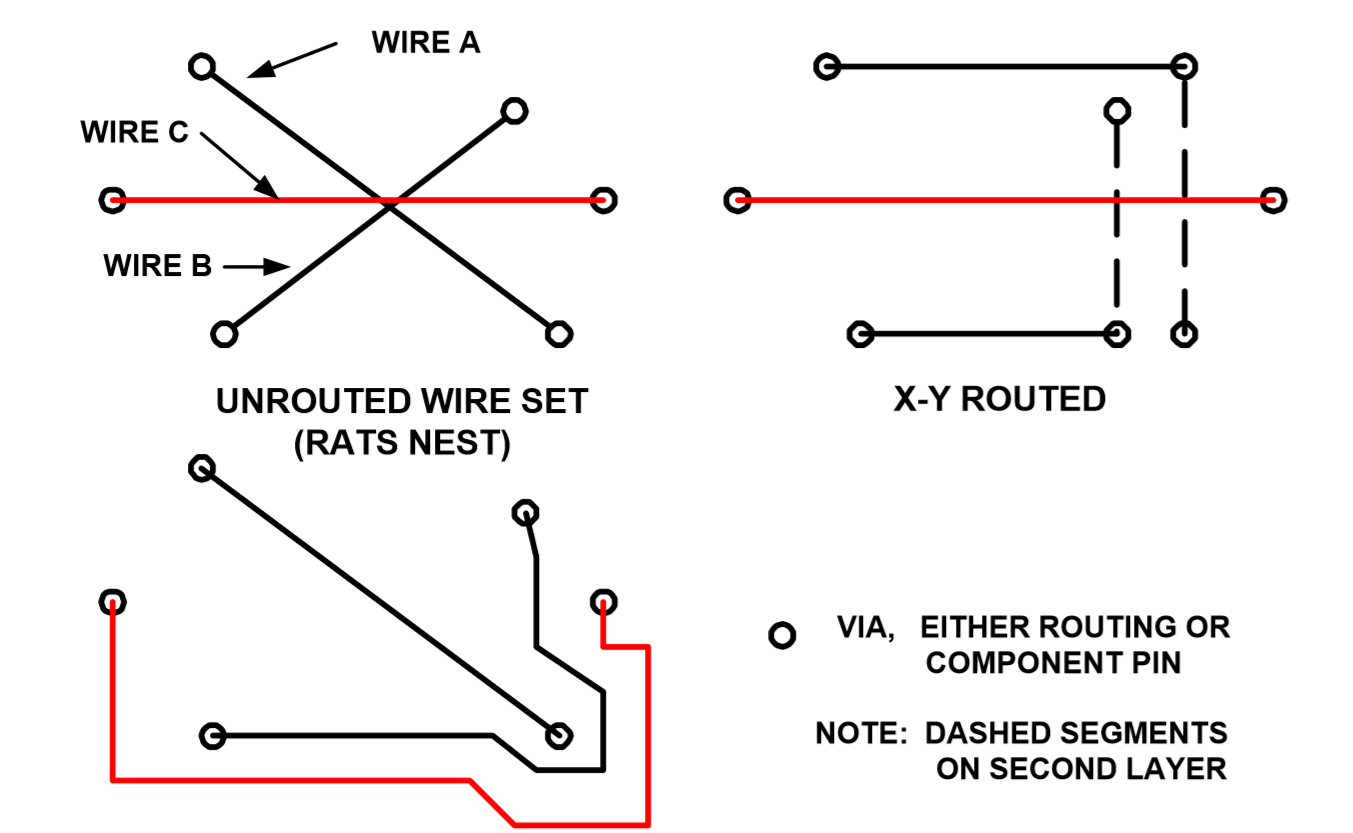
There are two main types of routing X-Y routing and Maze routing (LEE W. RITCHEY, 1999).

Figure 3 - A diagram to show the different methods of routing. Image Credit LEE W. RITCHEY, 1999 Accessed: 21/11/2018 Available <http://www.speedingedge.com/PDF-Files/pcbrouters.pdf>

Initially, I will focus on maze routing as it the preferred method of routing by industry, due to the fact that X-Y routing is almost always impossible with less than 4 layers due to the component footprints[[1]](#footnote-2) disallowing straight lines to be drawn. However as the project progresses adding support for X-Y routing or even creating a hybrid router would have its uses. The routes can move between the layers of the board using what are called “via’s”, which are essentially just holes in the board allowing a current to pass through.

### Problems with routing

#### Boxing in

Consider the bellow board, when routing the 2 pairs of green points were successfully connected however the red pair was not as it was “boxed in” by the other two preventing it from being routed there are three ways that this can be resolved:

* A board with multiple layers can be used
* A “jump”[[2]](#footnote-3) could be inserted
* The routed tracks could be ripped up and routing could be reattempted.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

### Definition of distances

There are a couple distances that will be referred to throughout this document.

**Euclidean Distance –** The shortest distance between the two points found using Pythagoras, e.g. to find the distance between the points and:

**Manhattan Distance -** Sum of the absolute values of the differences, e.g. for the previous points.

## Users

My user Stuart Jessop has experience with electrical engineering and I interviewed him to ask about what requirements my system should have. It should be noted that the bellow is written up from my notes on the interview and show the way the conversation went rather than being direct quotes.

**Me: I need to support being able to find a route between two points otherwise I just have a sheet of copper. I am thinking about supporting thermal sinks and keep out areas, thoughts?**

Jessop: A sheet of copper would not be very useful. It would be useful to support keep out areas, and thermal sinks are essentially a keep in area. The most important thing is to be able to find a route between two points without shorts.

**Me: I am also considering integration with EDA software what are your thoughts on this?**

Jessop:Integration with EDA software is going to be a lot of hard and monotonous work.

**Me: I guess that it is also at the end of the design pipeline as after routing all that needs to be done is exporting. Speaking of exporting which would be better, to export in SVG to Gerber?**

Jessop: Again you probably want to avoid just complying with a specification, I do not know how much work each of those is but you probably want to try and make the router as good as you can and only worry about exporting if you have time.

**Me: Ok, another thing that I am thinking of doing is isometric routing[[3]](#footnote-4) how useful would this be?**

Jessop: Isometric routing would be very interesting, I however do not know if it is reasonable to expect you to be able to implement it.

**Me: Many EDA tools support multiple layers, should I also try to make my router support many layers?**

Jessop: Single layer boards are fine and you can use jumps to effectively have two layers. This also means that it is fine if your router does not complete the routing and leaves the rest of the job up to the user. It is a tool for helping the user find routes.

**Me: Okay, thank you for helping me**

## Limitations

Limitations

* The software must run on a school computer. I have to demonstrate this to my user at school and I don’t have my own laptop so I either need to compile to an “.exe” or use a language with a run-time environment on the schools machines.
* My own coding abilities, I should do this project in a language that I am very familiar with such as node.js or python
  + Need to perform I/O in those languages which could difficult
* Time – This project needs to be done in the time allocated in the time allocated for the project in my computer science course, some objectives whilst not significantly harder may have to be class as extension objectives as there is a limit to what software I can develop in the time.

### Prerequisites and missing requirements

After my interview with my user I investigated other useful features which auto-routers have which are perquisites to being able to implement the features requested. The definition for what auto-routing is on Wikipedia is “*the routing step adds wires needed to properly connect the placed components while obeying all*[*design rules*](https://en.wikipedia.org/wiki/Design_rules)” (Wikimedia Foundation, 2018). This definition clearly points out a few areas which my project would need to support.

1. Being able to input data
2. Being able to find a route between two points
3. The route such does not interfere with other routes
4. Ability to export files

Other features which auto routers sell themselves on is the ability to route across multiple layers (Altium LLC, 2018), support for X-Y routing, for nets with multiple contacts as well as their speed.

The requirements listed above are quite vague and need to be made more precise.

Being able to find a route between two points– The route should not waste board space, this means that it is likely to be the shortest possible route and so is “efficient”.

The route such does not interfere with other routes– Interfere here means cause a short or a break in another track.

## Requirements

### Core Requirements

1. The program needs to run on a school computer.
2. User can input the positions of the nets into the system. This involves:
   1. Being able to create new nets
   2. Being able to change the start and end points of a nets
   3. Being able to remove nets
3. Finding an efficient routes between two points where efficient is defined as:
   1. Taking reasonable time to find the route, no more than a 5 seconds.
   2. Taking reasonable routes to the point and does not waste board space.
4. The user can create keep outs which are areas defined by the user that a net from outside may not enter.
5. The user can create thermal sinks where:
   1. The sink is of a minimum size specified by the user.
   2. The sink is entirely flooded
6. The user can create floods where:
   1. The flood has a “seed point” from which is spreads out marking those areas of the board as covered with copper
   2. The flood stops at non routable boundaries.
7. Program can export result in SVG format, this should allow the user to:
   1. Interact with the program to allow redesigning of the board
   2. Print the SVG to allow for etching of the board.
8. Being able to find the efficient route between multiple points:
   1. Being able to run the router multiple times on the same board
   2. Each run of the router routes a track which does not interfere with any other tracks

### Extensions

1. Being able to raise error messages to the user for when routing has not been completed and they will need to use jumps.
2. If the router boxes itself in its self in it is able to:
   1. Detect this
   2. Attempt Rip up tracks that it routed and boxed its self in with
   3. Attempt to route the track that failed to route
   4. Attempt to reroute any tracks that were ripped up
3. Support for push and shove routing.[[4]](#footnote-5)
   1. The user moving an endpoint triggers rerouting
   2. Selectively reroute routes only in the area of the board that the user has altered, to improve latency.
4. Multi-threading:
   1. Support flooding on multiple threads
   2. Attempt to route multiple nets at once.
5. Support for nets with multiple contacts
6. Support for isometric routing
7. Integration with EDA[[5]](#footnote-6) software, KiCAD[[6]](#footnote-7)
   1. Parse files generated to load the initial board conditions into the program
   2. Route and then output the routed board in a file format recognized by the program.

## Prototyping

### Objectives for the Prototyping Period

From the prototyping period I needed answers to the following questions:

* What language should I use?
  + If it’s not Delphi how do I get it to work on the school PC’s
  + How do I compile/run code
* How hard is this problem:
  + What solution and algorithms can I implement?
  + Is it of the right difficultly level for my coursework?
* What should a basic outline of the classes for my project look like?

### What I found out

#### Language

I do not want to learn a new language for my project and of the languages which I know I feel most confident in python or JS, graphical output is hard in python (as pygame and other popular graphics libraries are not installed on the schools computers) and so I chose to use JS. The schools computers have node.js installed which means that I can write my code in a text editor and execute it using PowerShell.

#### Algorithms

In-order to perform my prototyping I needed a basic awareness of the available algorithms. My problem is effectively how to solve a maze as if I can find a route between two points then that route just becomes another “wall” of the maze for the remaining routes. This means that I can use any maze solving algorithm. There are two main types of routing algorithms which apply “breadth-first” and “depth-first” (Zhou, 2000). Breadth-first algorithms are slower, require more resources but often find better routes as they will explore all possible routes over just prioritising one route. Depth-first algorithms attempt a single and likely path which is much faster (particularly for long tracks). For prototyping I wanted to show that it would be possible and so I chose a breadth first approach, however for my final project I may want to use a hybrid algorithm.

For my prototyping I chose to implement the “Lee-algorithm” (Techie Delight, 2018). Which works by propagating a wave out from one of the nodes leaving behind it a record of how long it to the wave to reach each point. When the wave hits the other contact it is then able to route to the point by tracing back through the record of distances selecting for the smallest distance to the start on the way back.

### Data structure

There are three main entities that my program needs to be able to handle as shown in figure 4. So during the prototyping period I created a class diagram as shown in figure 5

Figure 4 - An entity relationship diagram showing how the data about the board is related.

Point

Board

Net

Route

No route area

This class structure divides the problem into multiple clear sections. The Nets are represented by an object in which there are four integer properties which store the start and end position of the points. This object is contained within a board object which is then manipulated by a Board Router Object. The Net router object never gets instantiated but instead contains all of the functions associated with routing one net; the board router inherits all of these functions and then adds functions to prioritize which nets to route first and which nets to rip up. The route is represented as an array in the BoardRouter Class, which also contains the no-route areas.

Net

NetRouter

BoardRouter

Board

Figure 5 - A class diagram showing my OOP model at the end of the prototyping period

# Documented Design

## Overview

A requirement of my program is that it has to run on the school’s computers. As I have chosen to write my program in JavaScript, I have the choice of using the node V8 runtime environment or the Edge Charkra JS engine which does not fully support the ES6[[7]](#footnote-8) JS language standard. Given that node also has the added ability to be able to use the vs-code debug tools, thus I have decided to use Node and move as much of my code server side to support this.

The client server interaction can be sumarised as followsD:\Documents\My Documents (Edwin)\Downloads\TO-nJWCn44Jx-ugfg8Ze8uGYWQ12I4JGE-S9jkIpm-wQaB_7XaDS4SgHvis-tQWviJgbuJP5PiFg1blA1mLh2DyhrH1iIfTGEoKgNFRGKUB1z9UvNYqPFPHMEKOBFPGg8tK8kHXHUcU6rdZHbc8sYXjDVuXTSUBFkUnfOy2Lb6fSLA7cZ6sbHNzQdFaZxsFJwTm.emf

### Server Side

The board is represented as a 2D array of cell objects. The rest of the program is based around altering the properties of these objects. A netrouter class is used to find the shortest path between two points of the grid, a boardRouter class then controls the order in which the nets are routed in. The board is converted into and SVG and sent client side. Here is a high-level flow diagram of my server side code.

D:\Documents\My Documents (Edwin)\Downloads\TLF1QXin4BtlLmov50g17dBQGy4c9JKaRN1fBsqFYZLsHDQIAesSBY7_Nkbfbwn8S_BiqvltdcRaA4gO8eSJ1thgY8WgTFPPMJqm6RjpHsCF5FXFuYXdf6AfEt1_llzDv-SVNu1Sz7IfHEL6-hwzfgn2tiLexeQnO6kdnvc9rgrBmckdWkvfMpr8N8LfTRlfbg6.emf

### Client Side

The user loads and places components on the board this calls the board update method which sends the current board state to the server through a **RESTful web API** and gets a response with the routed SVG and displays this behind intractable parts of the UI.

The user is able to change the state of the board by adding, removing and dragging nodes around the board. These nodes represent the start and end positions for nodes on the board. The file can then be downloaded into the users file system.

D:\Documents\My Documents (Edwin)\Downloads\RL0nJiGm4EpzYb5Af8W-1HBLYGOGwAv152PP8ak77NRNbmBnTnJ70P-vAhCxdfsPY5eb2F69s9Xrwe-ElEBB0EkYsP5YfhV9ZdJD925oJu9t3XzPsi0AHjAdZJJ0ozKhslQcNfTZuhCvClYdWVaByk0HXHYsLnTyShMpTReNuaktI_8NdpCTxeFYO8zkE2_C8If.emf

## Pesudo Code snippets

### A\*

To route individual nets I model the board as a grid which can then be modelled as a graph. From this point all common graph traversal algorithms can be applied.

I use my own implementation of the A\* path finding algorithm. Each node in the grid has a node.g property which represents the length of the shortest known path to that cell. Each cell also has a node.h value which is a heuristic value. In this implementation I am using the Euclidian distance. These two values are summed to produce the node.f value ().

As the algorithm explores the graph it adds the neighbours of the node that it is currently exploring to a heap[[8]](#footnote-9). The heap is a data structure like a binary tree that satisfies the Heap property where the parent node is always greater than or equal to a child node. (Wikimedia, 2019) In my implementation of A\* the algorithm nodes are sorted in the heap by their node.f value with the lowest value first. To determine which node to investigate next the algorithm pops the values from the top of the heap.

A common method of building A\* is to use recursion however, JS has limited stack space and does not optimize well for tail recursion. I therefore use a while loop to check if the heap is empty. If the heap is empty then all possible nodes have been explored and an error is raised. If we find the end node then we trace the steps back, by selecting the cell in the neighbours which has the lowest node.g value each of the nodes are pushed to an array until the start node is reached and the array is returned as the result of the function call.

Pseudo code is based upon my netrouter.route function.

1. toCheck = **new** Heap(sortedBy item.f)
3. startCell.g = 0
4. startCell.f = 0
5. startCell.checked = True
7. toCheck.push(startCell)
9. **while** toCheck not empty {
10. cell = toCheck.pop()
12. **if** (cell = end) {
13. **while** (cell != start) {
14. trace = current.push
15. current = current.**super**
16. }
17. **return** trace
18. }
20. neighbours = cell.getneighbours()
22. **for** n **in** neighbour {
23. ng = cell.g + 1;
25. **if** (ng < n.g or neighbour not checked) {
26. n.**super** = cell
27. n.checked = **true**
29. n.g = ng;
30. n.h = EuclideanDist(neighbour, end)
31. n.f = n.g + n.h
33. toCheck.push(neighbour)
34. }
35. }
36. }

### Highlevel Routing

When deciding on the order in which to route the nets I want to try and minimize interference. Therefore the nets are placed into a queue in which they are initially sorted by Manhattan length. Sorting them by Manhattan length means that the smaller routes do not unnecessarily take longer paths and so reduces the amount of weird routes and keeps the board as clear as possible. The pseudo code for my board routing looks like this.

1. toRoute.sortByManhattan()
3. **while** (toRoute.length > 0) {
4. **try** {
5. currentNet = toRoute.shift()
6. tracks.push(route(currentNet))
7. } **catch** (err) {
8. currentNet.errorCount++
9. **if** (currentNet.errorCount < threshold) {
10. netsToClear = getNetsToClear(currentNet)
11. ripup(netsToClear)
13. toRoute.unshift(netsToClear)
14. toRoute.unshift(currentNet)
15. }
16. }
17. }

The algorithm will try to route the nets in order of shortest first. If however it finds that a net is impossible to route then it checks to make sure that this net hasn’t failed many times already (maybe it is just impossible) and then removes the tracks that might be causing the problem and tries again.

#### Get nets To Clear

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

It is possible however for the router to box itself in and consequently fail to route more nets when this happens I need to be able to decide which nets to rip up and reroute. In the example above the two nets starting from each of the green nodes were routed and consequently blocked the red net from being routing.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

The nets to be removed are calculated by drawing an L between the two end points and ripping up and nets that cross this L-shape. In the bellow diagram the orange cells are the ones that are checked and as a result the right hand blue net is removed, and added to the front of the queue of nets to be routed. The reason why the nets are prepended to this queue is that the reason why they have already been routed is that they are shorter than the net that is to be routed and so to reduce long routes I push them to the front of the queue.

1. **function** getNetIDsToClear(net) {
2. netIDsToRipup = **new** Set
4. start = net.startCell
5. end = net.endCell
7. bigX = Bigger(start.x, end.x)
8. smallX = Smaller(start.x, end.x)
10. bigY = Bigger(start.y, end.y)
11. smallY = Smaller(start.y, end.y)
13. **for** (let x = smallX; x < bigX; x++) {
14. netIDs =  getcontrollingNetOfCell(x,start.y)
15. netIDs.forEach(controllingNet =>
16. netIDsToRipup.add(controllingNet.controllingNetID)
17. )
18. }
20. **for** (let y = smallY; y < bigY; y++) {
21. netIDs = getcontrollingNetOfCell(y,end.x)
22. netIDs.forEach(controllingNet =>
23. netIDsToRipup.add(controllingNet.controllingNetID)
24. )
25. }
27. //convert the set to an array
28. Array = []
29. netIDsToRipup.reduce(item => Array.append(item))
30. netIDsToRipup = Array
31. //remove the current net from the array
32. index = netIDsToRipup.indexOf(net)
33. **if** (index > -1) {
34. netIDsToRipup.remove(index)
35. }
37. **return** netIDsToRipup
39. }

In order to ripup the nets I need to get their identifying property an unique identifier that is assigned to them at the start of the routing process. I use a set to ensure that I do not add duplicate entries to nodes to rip up. I then find the smaller starting coordinates and iterate along an axis from there adding nets to the list of nets to ripup.

Sets in JS have few methods so I convert the output to an array for ease of use and also removal of the starting net.

However it’s not just the net that needs to be ripped up, the area around the net also needs to be made routable again but only if that area is not marked as unrouteable by any other net which is not being ripped up.

A set of objects is assigned to each cell object representing the nets that control that cell and the priority that is required to make that cell routable. This allows prevention of ripping up the end cells and their neighbours. To ensure that we don’t get duplicates in the set I first use the array.find method before adding anything to the set.

#### Ripping up the nets

To decide whether we should allow a cell to marked as routable or unrouteable I have implemented a overridelevel cells can only be marked as routable if they were marked as unrouteable with an equal to or lower than overridelevel, here is a summary of some of the override levels that I have used:

|  |  |
| --- | --- |
| **Board Object** | **Overidelevel** |
| End of a net | 1 |
| Neighbour of end of a net | 1 |
| Cell in the middle of a net | 0 |
| Neighbour of middle of net cell | 0 |
| Flood Seed | 1 |
| Keepout controlling point | 1 |
| Sides of keepout | 1 |

These mean that when tracks are being ripped up (an operation performed with a routing level of 0) I only remove the nets and nothing that the user has placed down.

1. **function** markCellAsUnrouteable (cell, ID, overide) {
3. **for** (let i = 0; i < cell.controllingNetID.length; i++) {
5. **if** (cell.controllingNetID[i]== ID)   {
7. **if** (cell.controllingNetID[i] < overide) {
8. cell.controllingNetID[i] = overide
9. cell.routeable = **false**
10. }
12. **return**
14. }
16. }
18. //Not in list
20. cell.controllingNetID.push({
21. controllingNetID: ID,
22. overide: overide
23. })
25. cell.routeable = **false**
27. }

### User Interface

#### The Board

One of my design constraints was that I had to be able to be able to run this on a school computer, which vastly limits my JS runtime environments to either the edge 15 browser or the node.js V6. For development purposes I wanted to be able to use vs-code’s inbuilt debugging tools and so for this reason I decided to use node with as much code as I could possibly move server side.

All of the UI is displayed in the browser and the user interacts with the board using AJAX to interface with the RESTful server where the routing and construction of the DOM takes place. Here is the basic outline of how this works:

D:\Documents\My Documents (Edwin)\Downloads\TO-nJWCn44Jx-ugfg8Ze8uGYWQ12I4JGE-S9jkIpm-wQaB_7XaDS4SgHvis-tQWviJgbuJP5PiFg1blA1mLh2DyhrH1iIfTGEoKgNFRGKUB1z9UvNYqPFPHMEKOBFPGg8tK8kHXHUcU6rdZHbc8sYXjDVuXTSUBFkUnfOy2Lb6fSLA7cZ6sbHNzQdFaZxsFJwTm.emf

To update the DOM without making everything unresponsive for the user an asynchronous request is performed to the server in the following format.

1. async update() {
2. //Fetch the SVG DOM from the server but store the value as a promise
3. let resGrid = fetch(‘/route?’ + params, body: JSON.stringify(requestContent))
5. //Await for the promise to be fullied then assign the DOM
6. let response = await resGrid
8. //Convert the response into objects that we can manipulate
9. response = JSON.parse(response)
10. **this**.grid.innerHTML = response.board
11. **if** (response.errors.length > 0) {
12. show response.errors
13. }
14. }

To trigger the routing the client side makes a HTTP Request using the JS fetch API (Mozilla, 2018). This is made to /route with the routing parameters passed in the URL like so: /route?foo =”bar”.I then pass an encoded JSON string in the body of the request which represents the state of the board. This is then received by the server which parses the JSON, performs the routing, builds an SVG representation of the board and returns that in the body of its response along with any problems that happened whilst routing, encoded using JSON. The client then parses this response and updates the DOM accordingly.

#### Movement of end nodes

The only processing that really goes on client side is the handling of the end nodes which are used to control the positioning of nets, floods and keep outs. To do this I override the onMouseDown Method of each of the squares with my own dragMouseDown.

In order to keep with my grid design for the simplification of routing I rounded the mouse’s value to the nearest grid cell size. This forced everything to work in the grid.

To allow to the user to delete items I check if the control key is pressed when the dragMouseDown event is triggered, if so it executes the deleteCallBack function which was passed to the function when it was initialized, which destroys the object that the dragMouseDown function is assigned to.

So as not to overload the server (routing is an extremely CPU intensive task) the update function is only called when the user stops dragging, however if the routing algorithm was better optimised this could be moved into the elementDrag function to provide real time routing support.

1. **function** dragMouseDown(e) {
2. e.preventDefault()   //Prevents the default method from running
4. // get the mouse cursor position at startup:
5. oldXPos = getRoundedMouseX(e);
6. oldYPos = getRoundedMouseY(e);
8. /\*\*
9. \* If the control key is down then the user wants to delete this item
10. \*/
11. **if** (e.ctrlKey) {
12. deleteCallBack()
13. board.update();
14. } **else** {
15. document.onmouseup = stopDragging;  //Override the onmouseupmethod
16. document.onmousemove = elementDrag;     //Assign the on mousemove elementDrag method to elementDrag
17. }
18. }
20. **function** elementDrag(e) {
21. e.preventDefault();
23. // calculate the new cursor position:
24. newXPos = oldXPos - getRoundedMouseX(e);
25. newYPos = oldYPos - getRoundedMouseY(e);
27. // set the element's new position:
28. elmnt.style.top = newYPos
29. elmnt.style.left = newXPos
31. //Update the old positions
32. oldXPos = getRoundedMouseX(e);
33. oldYPos = getRoundedMouseY(e);
34. }

37. **function** stopDragging() {
38. /\* stop moving when mouse button is released:\*/
39. document.onmouseup = **null**;
40. document.onmousemove = **null**;
42. //find new the cells position in grid terms.
43. let newCellX = Math.ceil(elmnt.offsetLeft/board.cellSize);
44. let newCellY = Math.ceil(elmnt.offsetTop/board.cellSize);
46. cell.x = newCellX;
47. cell.y = newCellY;
49. board.update();
50. }

### SVG Files

My program generates SVG files. SVG files that are generated comprise of the following hierarchy of information stored in XML.

* SVG FILE
  + Header
    - SVG Version
    - Width & Height
    - Base Profile
  + /Header
  + Body
    - Rectangle
      * Styling
      * Position
    - /Rectangle
  + /Body
* /SVG

Here is an example SVG file generated by my program:

1. <svg version="1.1" baseProfile="full" width="1400" height="800" xmlns="http://www.w3.org/2000/svg">
2. <rect id="#undefined" **class**="Rect" x="0" y="0" width="10" height="10" style="fill:#007cae;, stroke:#ffffff;, stroke-width: 5">
3. </rect>
4. <rect id="#undefined" **class**="Rect" x="0" y="10" width="10" height="10" style="fill:#007cae;, stroke:#ffffff;, stroke-width: 5">
5. </rect>
6. <rect id="#undefined" **class**="Rect" x="0" y="20" width="10" height="10" style="fill:#007cae;, stroke:#ffffff;, stroke-width: 5">
7. </rect>
8. </svg>

These are moved into the users file system by creating a “phantom” anchor object which the program then clicks for the user to trigger the download as JS can’t write directly into the users file system and so I need to create the whole file first before I can ask the browser to write it into the users file system.

#### Object orientation

D:\Documents\My Documents (Edwin)\Downloads\SohDp4bnyVTLoDVLLIWkI_TDp4ujuk82ie8p4WjI5Mo0ma6foIM9UUavgLmm1GfQAC9m5jWKv_oS_D8Y0000.emfThere are many types of SVG object and so I created an **abstract class** (svgShape)to deal with them, this allows me to safely write code in svgMaker for a generic SVG object as I know that all of the svgShape objects have to implement certain methods and properties. This is a real strength of using polymorphic objects. On the right there is a class diagram showing the relations.

## UML Digram

D:\Documents\My Documents (Edwin)\Downloads\TP7DReCm383lVWgVAz47c7QewAJzRBHIsGj4O57KD471wGNrtOSJIiR4JjXVlZYssVMi7FkhXavL5zKGbkHkv7200p7QBCCzQGsvLQx-csx6TBJ4F5dFvFuXmOvt4XrmbjBHuWBqjwOyguxmUMGdgbYPHfFWGbqcDU4vYcrQRRq3E3DtSQIeooCVvV40SnJqtBU.emf

## Class lookup table

For the readers convenience some of the smaller classes have been omitted.

|  |  |
| --- | --- |
| Name | Description |
| Cell | Contains information about each cell, e.g. is it routable which nets control it etc. |
| Net | Contains the start/end points, and the tracks |
| Board | Contains the cells and all of the methods that are used for handling them |
| Net Router | Manipulates the board object to find a route for its contained net object. |
| Board Router | Dynamically creates Net Routers to try and reduce to the toRoute queue to zero |
| RouteJSON | Controls the servers components:   * Decodes JSON received from the client into useable objects * Initiates routing on the board * Requests the conversion of the track list into DOM * Passes result back to the server |
| BuildDOM | Generates valid DOM from the current state of the board. Utilizes SVG making classes. |
| httpServer | Handles client connections |
| httpClient | Requests resources and routing jobs from the server. Displays the response to the user |
| Grid | The client side object that the user interacts with |

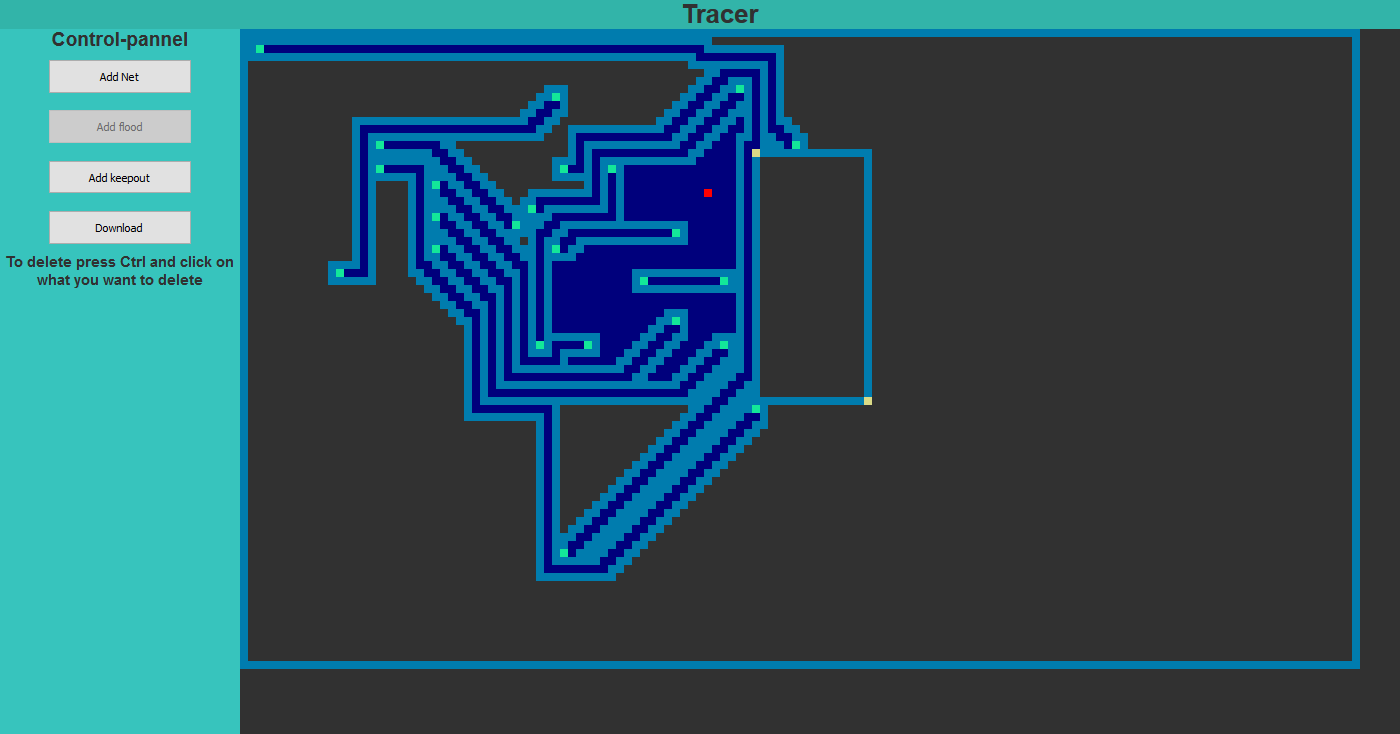
## Data structures

Here is a sample of some of the data structures that I have used.

|  |  |  |
| --- | --- | --- |
| FileName/Name | **Type** | **Purpose** |
| index.js/fileTypes | Dictionary | To allow for the mimetype of a file to be looked up from its file extension |
| main/board.js/Grid | 2D Array of Cell objects representing a graph | A 2D array of cell objects contained within the Board object that represents everything that is happening on the board.  The router traverses this array as if it were a graph by interpreting the neighbours of each of the squares adjacent nodes. |
| index.js/responseContent | JSON String | A JSON string which is transferred between client and server. It contains a complete representation of all of the client side objects. |
| main/netRouter.js/toCheck | Heap | Hold the list of items which have are on the edge of where has been checked and poping them gives the next one to check |
| main/boardRouter.js/errors | Stack | All the non-crashing error messages that need to be displayed to the user. |
| main/svg.js/DOM | DOM | Holds the Document Object Model to be displayed to the user that has been generated using the SVG class. |
| main/boardRouter.js/netsToRipup | Set | Used to store each of the nets that need to be ripped up, due to the way that these are selected the algorithm may detect them on multiple instances should only store them once as they can only be ripped up once. |
| main/boardRouter.js/netlist | List | Stores the nets and the order in which to route them. |
| indexScript.js/deleteCallBack | Function | Passed to the makeInteractable function, used to allow the polymorphic like behaviour of the makeInteractable function. |
| main/boardRouter.js/toRoute | Queue | A queue of items that are to be routed allows for controlling the order in which the items are to be routed. |
| SVG OUTPUT | File | To allow the user to be able to make PCB’s from this. I have created my own SVG class to handle this which is compliant with the SVG specification. |

## Screen Shots

Here is an overview of what my program looks like with a set of routes.



|  |  |
| --- | --- |
| Screen shot | Explanation |
|  | On the left hand side there is a control panel with buttons, which when clicked add items to the board. There is a message to the user to remind them how to delete nets (by control clicking) and if a net failed to route or any other error occurred then it raises it bellow. |
|  | **Nets** are displayed as a series of green dots that the user can move to where ever they want. The picture in the red square shows these dots by themselves. Then the update() method is called the server generate an SVG which is placed behind all of the dots creating the illusion that they are being connected. |
|  | Floods are generated by spreading out from a point, represented in the router by a red dot, which |
|  | Areas that nets are not able to cross (called keepout areas) are defined by two points (represented here in yellow, which define two corners for the keep out area to make unrouteable. |
| SVG DOWNLOAD | The download complies with the SVG specification for displaying data. |

This program is really shown off better in video format to see some videos please see the videos in *Testing.Demonstraion of overall effectiveness*.

# Testing

## Tests

### Tests

The tests that failed are displayed in this shade of red.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No | Objective being tested | Test Data/Action | Reason/Explanation | Expected Result | Actual Result |
| 0 | 1 | Run the program on a schools computer | Even though some parts of the project were developed at home it still runs on a school machine. | A webserver is created on port 1337. | Going to [http://localhost:](http://localhost:5555)1337 shows a webpage being served by the user. |
| 1 | 2a, b & c | A new net can be created and its positions set to a desired location and then deleted | Checking that the user interaction code is working. | The net appears is rerouted between those points and removed. | As expected |
| 2 | 4 | Creating a keep out and placing the end nodes around the area. | This is to test that keep out areas cannot be routed through. | A Route is found around the keep out | See test screen shot 1 |
| 3 | 4 | Placing one end of a net on the boarder of the keepout and the other outside of the keepout | Tests that even if the user asks something impossible of the router it throws and error | The routing to fail | FAILED: The net is routed through the keep out see screen shot 2 |
| 4 | 3a | Create a route with 5 nets that deliberately causes non-straight forward routing to occur, and time how long it takes for the screen to get updated. See Screenshot 3 for the routing that I did. | This tests that even with non-trivial routing that the router performs fast enough to keep up with the person. | The routing and screen update is faster than a human can discern | The routing averaged at 45ms which is about 22fps. Easily bellow my 5 second goal. |
| 5 | 3b | Attempt a couple of routes and try to come up with a board with less wasted board space. | Objective 3b is quite hard to assess as it is asking how well did I do on an np-complete problem using heuristics. The only reasonable test is to ask if it is better than a human | Each route will be the shortest given the state of the board which is probably going to be better than anything I can do. | I was unable to beat the auto router. |
| 6 | 5a & b | Place a flood seed inside a keep out area | This is how I have chosen to fulfil objective 5, about creating thermal sinks. | The keep out area to be completely filled by the flood | As expected see screen shot 4. |
| 7 | 6a & b | Place some a few nets such that they create a box, and then place a seed point for a flood in the middle of the box. | This is testing that nets are able to properly contain floods. | The boxed in area will be flooded. | As expected see screen shot 5 |
| 8 | 7b | Clicking the download board button and seeing if the downloaded file is a valid SVG by opening it in another program | This tests that program is able to export valid SVG’s | The program to download a file and it can be opened by other software capable of opening SVG’s | As expected see screen shot 6 |
| 9 | 8a | Creating multiple routes. | Checking that the router is able to route more than one route | Routes to be found between the points and displayed as lines on screen. | As expected. |
| 10 | 9 | Place one end of a route in a keepout and the other outside | Test if the program detects that the route is impossible and raises and error for the user | An error message to be shown identifying which net couldn’t be routed. | As expected see screen shot 7 |
| 11 | General | Placing a node outside the board | Test for the user doing something unexpected | The program to catch the error | The program identified that it wasn’t able to perform the route because of the perimeter of the board. |
| 12 | General | Placing the end of a net on another net | Test if the program can reject impossible routes | The program to return the net to its original position. | The program worked as expected and gave a custom warning message. |
| 13 | General | Placing a net end on the edge controlling point of a keep out | Test if the program can reject impossible configurations | The program to return the net to its original position. | Keep outs can’t be dragged onto nets but nets can be dragged onto keep outs. |
| 14 | General | Placing a node in the path of another track | Tests the avoidance of the routes. | The program to route around this. | In most cases this works however it is possible to end up with screen shot 8 when the longer one is placed in the route of the shorter one at the right angle at close proximity to the end node. |
| 15 | General | Resizing the screen after the program has initialized | Tests that the UI is robust to changes that the user might make to it. | The UI to scale with the screen | The end points moved until a reroute was retrigged and then the end points ended up mostly where they were meant to be. See screen shot 9. |
| 16 | General | Deleting all the nets on the board | The program can cope with nulls and nothing | The program to do nothing | As expected |
| 17 | General | Go to a sub-url and then go back to the original page | Test the usage robustness of the program. | The program to pass all the above tests as before when the user returns | The browser does not trigger a fetch to the routing API. See screen shot 10. |
| 18 | General | Run the router and kill the server | Tests robustness against the server crashing. | The router to allow movement of end nodes and do nothing else | As expected |
| 19 | General | Attempt to add two floods | The idea of flooding with an autorouter in hindsight is a questionable feature and I have chosen to not allow adding two at any time. | The Add flood button to be greyed out | As expected |
| 20 | General | Pressing the add net button many times without moving any of the nets. | Checks the program is able to cope with a large number of errors | The program to display all the error messages. | As expected. |

### ScreenShots

|  |  |
| --- | --- |
| **Screenshot Number** | **Screen Shot** |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 |  |
| 8 |  |
| 9 |  |
| 10 |  |

## Demonstration of overall effectiveness

Demonstration 1, general demonstration of the router showing that I have achieved all of my objectives: <https://youtu.be/Ti5vIHvVJkc>

Demonstration 2, demonstrating the very specific edge cases that break the auto router as revealed in testing, in normal use it is quite hard to set up the above conditions: <https://youtu.be/TjGX_9jJu44>

# Evaluation

## Evaluation with End User

### Comment from end user

Dear Edwin

Thanks for demonstrating your PCB routing application.  As an electronic designer in my past life, it is clear that your application can be used to design single-sided printed circuit boards, particularly for prototyping purposes.  It meets all the basic requirements of such a tool and the auto-routing and flooding work well and give big advantages over designing PCBs using a general purpose drawing tool, or by hand.  The ability to output an SVG file means that the application could be used in practice to create a PCB.  It is only the fact that this is a software project that has stopped us getting the ferric chloride out and etching PCBS!

The application would have been useable without the extensions you implemented, but I especially like the ‘jumper’ feature that we discussed, and which you added, to solve the problem of trapped routes.  The push and shove auto-reroute feature you implemented also makes the application so much easier to use.  Should you wish to continue development you have a sound foundation here to build on.

In summary, this is a practical, usable application – well done.

Mr Jessup

### Evaluation of comment from end user

The comment from my end user shows that I have been able to fulfil all of my core objectives to a standard that is useful to the end user. It also shows that I have managed to usefully fulfil many of my extension objectives to add functionality to my program. I have also selected useful extension objectives to achieve such as the “jumper feature” mentioned. I therefore think that my project has been a success.

The expression of the desirability of the push and shove feature that was implemented shows how this is an important area which, if I had more time, should work to make even more usable. My main goal with this would be to get it to update the routes with every mouse movement rather than just when the user releases a point.

## Completeness

### Objective review

1. **The program needs to run on a school computer.**

This objective was met by using JavaScript a very portable scripting language with wide support. The client server nature of my solution would have allowed me to potentially host the majority of my program on a server and load it as a webpage at school. The one major incompatibility with the schools browsers Edge and IE. Chrome and Firefox do however run on a school machine and so this is acceptable.

1. **User can input the positions of the nets into the system. This involves:**
2. **Being able to create new nets**
3. **Being able to change the start and end points of a nets**
4. **Being able to remove nets**

Objective 2.1 was met by use of a button to add new nets to the same area of the board. This was functional and worked. However it did have the shortcoming that new nets were added to the exact same position on the board this is possibility a bit confusing for the user.

This is a consequence of my program not having good integration with existing tools and so it was considering points opposed to components with footprints. If the user was first instructed to place all of the footprints that they wanted onto the board then the nets would be added implicitly by adding footprints. If I was to support EDA tools rather than build my own basic one as was outlined in extension objective 15 then this issue would be resolved.

Objective 2.2 was met in an intuitive way by dragging and dropping the end points to change their coordinates. This was done by creating an event Listener for a mouse down event over an end point and then binding its position to a rounded value of the mousse until a mouse up was triggered.

Objective 2.3 was met by detecting if the CTRL key was pressed when the nets controlling points was clicked on then deleting that net. This did allow for very fast use of the program if you were familiar with it however made the program less easy to navigate for first time users. I attempted to solve this by displaying a message to the user specifying how to delete a net, however it may have been better to have a button to toggle a “delete-mode”.

**3) Finding an efficient routes between two points where efficient is defined as:**

1. **Taking reasonable time to find the route, no more than a 5 seconds.**
2. **Taking reasonable routes to the point and does not waste board space.**

As test 4 showed the time that it takes to perform routing is almost imperceptible, hence I have easily achieved 3.1. I did this by making heavy use of heuristics be it in A\* where I was using the distance between nodes to try and reduce time or in the board router class where I sorted all of the nets by length.

3.2 is an objective which is very hard to quantify as it is asking how close to the perfect solution can I get of an NP-complete problem. Due to the nature of NP-complete problem’s I do not know the perfect solution. The implementation of A\* that I am using should in theory find the shortest route most of the time as providing that there isn’t too much back tracking then the Euclidian distance should be considerably greater than the shortest route. When the heuristic is greater than the shortest distance then A\* will behave optimally.

I as a person with good special awareness I did try to beat the router and found that I was unable to. Therefore to the best of my quantitative ability I would say that this objective has been achieved.

1. **The user can create keep outs which are areas defined by the user that a net from outside may not enter.**

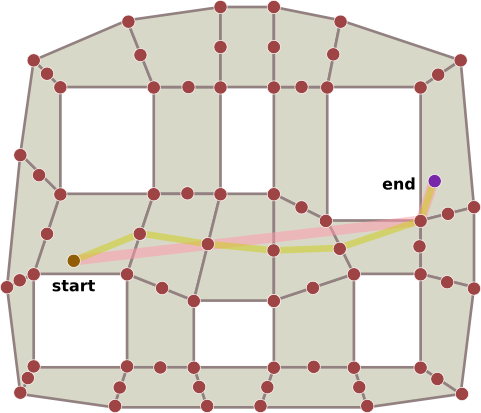
The way that I solved this is by having a square of nodes that are marked as unrouteable. This does fulfil the objective. Possibly a more useful way for the user would be to define straight lines between which there is an unrouteable line, which would mean the user would be able to create many different shapes of non-routable area. Rather than just the current rectangles.

1. **The user can create thermal sinks where:**
   1. **The sink is of a minimum size specified by the user.**
   2. **The sink is entirely flooded**

The way that a user can do this with my program is to create a keep out area and then flood that. This means that the user is able to vary the size by varying the size of the keep out and it is entirely flooded because no nets can enter the area. Although the original intent of this algorithm was to create something that would spread out to cover a minimum area regardless of shape this approach works and fulfils the objective.

1. **The user can create floods where:**
   1. **The flood has a “seed point” from which is spreads out marking those areas of the board as covered with copper**
   2. **The flood stops at non routable boundaries.**

6.a was achieved by pushing the valid neighbours of a cell to a stack and running until the stack was empty, whilst this worked it turned out to be quite slow (noticeable to the user but less than 5s) when flooding large areas of the board. The ways in which it could be speed up is to do pre-processing on the board. This would involve blocking out areas of the board before flooding so that they could be treated as one giant cell.



Alternative representations of maps such as the one above (image credit: (Amit, 2019)). In the representation above the by performing this pre-processing I would greatly be able to speed up the time that it took to flood large areas.

The flood does stop at the unrouteable areas which means that objective 6.b was entirely satisfied.

1. **Program can export result in SVG format, this should allow the user to:**
   1. **Interact with the program to allow redesigning of the board**
   2. **Print the SVG to allow for etching of the board.**

The server creates a valid SVG which is transmitted across the internet which is then displayed to the user completing 7.a. This is then downloaded into the user file system. This was non-trial to do as JS does not have write permissions into the users file system so to move the contents of memory into the users file system by creating an anchor link that is “clicked” on by the program and then removed. This triggers a download into the users file system. Once the file is in the users file system they can print it at whatever scaling they desire.

1. **Being able to find the efficient route between multiple points:**
   1. **Being able to run the router multiple times on the same board**
   2. **Each run of the router routes a track which does not interfere with any other tracks**

Objective 8.a is completed by use of a queue for sorting which nets should be routed next and as each route was routed it added a property to the cells which were associated with the track as “unrouteable” this means that subsequent passes of the router would not be allowed to move through those areas of the board. This completes objective 8.b

### Extensions

1. **Being able to raise error messages to the user for when routing has not been completed and they will need to use jumps.**

A\* fails after it has found no routes from an exhaustive search, when this occurred I then threw an error. It would the attempt to perform a reroute of the board however if a specific net kept throwing a non-routable error then it would give up and show a message to the user. This is a relatively good approach however it was not able to distinguish between the user creating impossible routes and the router boxing its self in.

1. **If the router boxes itself in its self in it is able to:**
   1. **Detect this**
   2. **Attempt Rip up that it routed and boxed its self in with**
   3. **Attempt to route the track that failed to route**
   4. **Attempt to reroute any tracks that were ripped up**

I managed to complete this objective with partial success. 10.a is easy as that is just error catching when A\* fails however then working out which nets need to be ripped up and rerouted is hard. This is hard because sometimes the router would rip up nets that could only be routed through the area of the board that it had routed. This meant that it was possible for the user to create setups where the router would trap its self in a loop of trying to route two nets as each would require the other to be ripped up (according to its heuristic).

The way that I solved this was to add a maximum error count to each net after which was reached it would just give up on the net. This did solve the infinite loop problems however meant that it would not always be able to attempt to route the track that had failed to route as such objectives 10.c and 10.d could not always be completed as the way in which 10.b was solved prevented this.

A better router may spawn a number of threads and consider what would happen if more nets in different directions were routed. If a suitable heuristic (maybe the maximum number of nets that could be routed) was used then a minimax algorithm could be used here, to decide how to traverse a decision tree.

1. **Support for push and shove routing.[[9]](#footnote-10)**
   1. **The user moving an endpoint triggers rerouting**
   2. **Selectively reroute routes only in the area of the board that the user has altered, to improve latency.**

I managed to achieve objective 11.a when the user drops the end point of a net, this allowed for a primitive implementation of “pushing and shove” routing however a major improvement would be to reroute the board as the user drags around end points. To build such a real time system I probably would have to move to either a socket based communication system (currently I have a REST API) or to move all of the routing client side.

I did not start on objective 11.b however to build such a system I would probably have to rip up nets in a local area and then reroute them.

1. **Multi-threading:**
   1. **Support flooding on multiple threads**
   2. **Attempt to route multiple nets at once.**

Node.js 11 does have experimental support for multithreading (Node.js, 2019), so it is theoretically possible to implement objective 12 however I did not have time to do this.

1. **Support for nets with multiple contacts**

This should be possible by calling the net router multiple times where one of the ends are each of the ends. However to do this I would need to perform a complete overhaul with how my user interacts with my program and hence this objective was not achieved.

1. **Support for isometric routing**

Isometric routing is where two tracks are made to be of the same length (to within a tolerance) this means that signals sent down both wires in parallel will arrive at the same time. To perform routing like this I would first route both nets and tune the shorter routes length up until both of the routes were even in length. I did not manage to achieve this objective.

1. **Integration with EDA**[[10]](#footnote-11) **software, KiCAD**[[11]](#footnote-12)
   1. **Parse files generated to load the initial board conditions into the program**
   2. **Route and then output the routed board in a file format recognized by the program.**

This would have allowed me to lavage the power and existing development of current EDA. To parse KiCAD files, would have involved a lot of regular expressions. The type of file that I would be concerned with .sch files are simply text files. Here is an extract of one:

1. Wire Wire Line
2. 4800 14500 4800 15050
3. Wire Wire Line
4. 5150 14900 5150 15050
5. Wire Wire Line
6. 4450 15000 5450 15000
7. $Comp
8. L power:GND #PWR0106
9. U 1 1 5C5F551F
10. P 5350 14700
11. F 0 "#PWR0106" H 5350 14450 50  0001 C CNN
12. F 1 "GND" H 5355 14527 50  0000 C CNN
13. F 2 "" H 5350 14700 50  0001 C CNN
14. F 3 "" H 5350 14700 50  0001 C CNN
15. 1    5350 14700
16. 1    0    0    -1
17. $EndComp

As you can see the file structure is mostly concerned with specifying coordinates for components on the board. This means that it could be possible to load these coordinates into my system and then run the router.

## Effectiveness

As I have completed all my core objective my system is capable of performing the intended task, to use the words of Stuart Jessop[[12]](#footnote-13): *“It is only the fact that this is a software project that has stopped us getting the ferric chloride out and etching PCBS!”*

### Strengths

The main simplification that I made to my project’s modelling was to model the board as a grid and then a graph. Whilst there are routers which rely solely on graph theory (such as the spectra router (Wikimedia)) these can often produce messy results and this method is much cleaner. This also allowed me access to all of the brilliant ideas that come from graph theory empowering me to make a better router.

One of the main strengths of my project are that is able to work in real time. This allows the user to quickly drag nets around and decide on positioning of items and other related board features. This allows for the user to quickly prototype their design as the system gives feedback in real time, which as my user pointed out was a real strength of this project.

A strength of my project is that it works on any machine with a modern browser. This makes my project highly accessible to many users.

### Improvements

#### Failed tests

The tests that my program failed were acceptable as they were edge cases. The cause of both of these bugs is likely something due to the rip up code accidentally removing areas that should not be ripped up. This is due to the fact that when the route is routed the areas around it are marked as routable when they shouldn’t.

#### General

There are however many ways in which my program could be improved. The main ones of these outlined in my extension objectives. I think that the improvements that would be best to now make is integration with EDA software. This would best lavage on current software as auto-routers are not a replacement for EDA software but rather an enhancement to a stage in the design phase.

Another feature that my autorouter lacks that much support is support for multi-layer boards. This extension should be relatively easy as A\* works with grids of any dimensions. There may need to be some heuristic added that punishes use of via’s as they consume more board space however this addition would bring the router up towards the requirements of industry.

With giving the user greater power to input more and increasingly complex nets into the system the routing algorithms whilst efficient at the numbers of nets that I have been dealing with I doubt they will scale well to 100s of nets and so greater efficiency gains will be needed. To achieve this I am likely to want to develop pre-processing algorithms (Microsoft Research, 2004). These can include blocking as was outlined earlier as well as more advanced heuristics that are derived from deep learning (Aleksandr, Konstantin, & Roman, 2018).

I could also achieve another efficiency gain of 100 times through the use of web assembly. I would write the really CPU intensive components of my program (the board router and all of its dependencies) in a highly performant language like C or Rust, then set my compilation target to be WebAssembly.

# Bibliography

Aleksandr, I., Konstantin, S., & Roman, S. (2018, January 5). *Grid Path Planning with Deep Reinforcement Learning: Preliminary Result.* Retrieved from Science Direct: https://www.sciencedirect.com/science/article/pii/S1877050918300553

Altium LLC. (2018). *BEST AUTOROUTER*. Retrieved November 18, 2018, from Altium: https://www.altium.com/solution/best-autorouter

Amit. (2019, 02 24). *Thoughts on pathfinding*. Retrieved from Stanford: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html

LEE W. RITCHEY, S. E. (1999). *PCB ROUTERS AND ROUTING METHODS.* Retrieved from Speeding Edge: http://www.speedingedge.com/PDF-Files/pcbrouters.pdf

Microsoft Research. (2004, March). *Computing the shortest path: A\* Search Meets Graph Theory.* Retrieved from Microsoft reaserach: https://www.microsoft.com/en-us/research/wp-content/uploads/2004/07/tr-2004-24.pdf

Mozilla. (2018, October 23). *Fetch API*. Retrieved from MDN: https://developer.mozilla.org/en-US/docs/Web/API/Fetch\_API

Node.js. (2019, 3 05). *Worker Threads*. Retrieved from nodejs.org: https://nodejs.org/dist/latest-v11.x/docs/api/worker\_threads.html

Techie Delight. (2018). *Shortest path in a maze | Lee algorithm*. Retrieved from Techie Delight: https://www.techiedelight.com/lee-algorithm-shortest-path-in-a-maze/

Wikimedia. (2019). *Heap (data structure)*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Heap\_%28data\_structure%29

Wikimedia Foundation. (2018, October 22). *Routing (electronic design automation)*. Retrieved November 18, 2018, from Wikipedia: https://en.wikipedia.org/wiki/Routing\_(electronic\_design\_automation)

Wikimedia. (n.d.). *Spectra*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Specctra

Zhou, P. H. (2000). *EECS 357 Introduction to VLSI CAD.* Retrieved from eecs.northwestern.edu: http://users.eecs.northwestern.edu/~haizhou/357/lec6.pdf

# Technical solution

## Contents

|  |  |  |  |
| --- | --- | --- | --- |
| **Implementation** | **File** | **Line** | **Page Number** |
| Importing defined and system modules | Index.js | 1 | 46 |
| Webserver | Index.js | 200 | 49 |
| Parameterized file handling of multiple file types. | Index.js | 247, 252 | 49 |
| File system error handling | Index.js | 248 | 50 |
| Use of dictionary | Index.js | 254 | 50 |
| Accepting data from client | Index.js | 222 | 49 |
| Parsing JSON from client | Index.js | 231 | 49 |
| Map Reduce | Index.js | 143 | 48 |
| Optimisation that adapts to user use in dynamic generation of objects in complex OOP model. | Index.js | 158 | 48 |
| Web API interface | Index.js | 231 | 49 |
| Web API parameter passing | Index.js | 200 | 49 |
| Creating JSON strings | Index.js | 235 | 49 |
| Use of constants | Index.js | 1 | 51 |
| Use of a stack | BoardRouter.js | 85,95,104 | 52 |
| Use of a queue | BoardRouter.js | 159, 161, 163 | 53 |
| Cumulative error handling | BoardRouter.js | 177 | 53 |
| Use of a set | BoardRouter.js | 182 | 53 |
| Nested higher order functions | BoardRouter.js | 224-248 | 54 |
| Dynamic object creation dependant on errors and other parameters. | BoardRouter.js | 164 | 53 |
| List operations | BoardRouter.js | 226 | 54 |
| Use of Heap | netRouter.js | 23 | 56 |
| Implementation of A\* | netRouter.js | 35 | 56 |
| Throwing error | netRouter.js | 114 | 57 |
| Graph traversal | netRouter.js | 49 | 56 |
| Definition of objects | Board.js | 1 | 59 |
| Extension of object prototype | Board.js | 139 | 61 |
| Calculation of Manhattan and Euclidean distances | Board.js | 255 | 62 |
| Containment of objects | Board.js | 110 | 60 |
| Abstract class:  NB JS does not by default have support for abstract classes and so this had to be added | svg.js | 8 | 66 |
| Inheritance from abstract class | svg.js | 37 | 66 |
| Selective module exports | svg.js | 157 | 68 |
| Generation of SVG file | svg.js | 127 | 68 |
| EDP[[13]](#footnote-14) | indexScript.js | 74 | 73 |
| Conditional EDP | IndexScript.js | 119 | 73 |
| Call-back function | IndexScript.js | 74, 120 | 73 |
| Passing parametrized data to web API and fetching result | IndexScript.js | 314 | 76 |
| Asynchronous function | IndexScript.js | 306 | 76 |
| Saving file into users file system | IndexScript.js | 356 | 77 |
| Creation of objects when user specifies | IndexScript.js | 340 | 77 |
| Dynamic creation of DOM | Indexscript.js | 215 | 75 |

## Server side

### Index.js

1. **const** http = require('http');
2. **const** fs = require('fs');
3. **const** url = require('url');
5. **const** br = require('./scripts/boardRouter')
6. **const** b = require('./scripts/board');
7. **const** svg = require('./scripts/svg');
8. **const** colour = require('./scripts/colour');
10. **const** port = 1337;
11. **const** debug = **false**;
13. **const** path = "."
14. **const** indexLocation = "/index.html"
16. //F
17. **const** boardWidth = 140;
18. **const** boardHeight = 80;
20. //!Use of a dictionary
21. **const** fileTypes = {
22. '.html' : 'text/html',
23. '.css' : 'text/css',
24. '.js' : 'text/javascript'
25. };
27. /\*\*
28. \* Converts a JSON netlist into an SVG represenation of the routed board.
29. \* @param {JSON} JSONData
30. \* @param {Number} cellSize
31. \*/
32. **var** routeJSON = **function**(JSONData, cellSize) {
34. let errors = []
36. JSONnetList = JSONData.netList;
37. JSONFloodList = JSONData.floodList;
38. JSONKeepoutList = JSONData.keepoutList;
40. **const** trackWidth = cellSize;
42. /\*\*This is where we take the JSON inputed into the system and turn it into objects
43. \* that we can manipulate
44. \*/
45. **try** {
47. **var** netList = []
48. let length = JSONnetList.length;
50. /\*\*Creates each of the net objects from the specification in the
51. \* recived JSON
52. \*/
53. **for** (i = 0; i < length; i++) {
54. let start = **new** b.Cell( JSONnetList[i].start.x,
55. JSONnetList[i].start.y);
57. let end = **new** b.Cell(   JSONnetList[i].end.x,
58. JSONnetList[i].end.y);
60. netList.push(**new** b.Net(start, end));
61. }
63. } **catch** (err) {
64. **throw** err
65. }
67. board = **new** b.Board(boardWidth,boardHeight);
69. BR = **new** br.BoardRouter(board, netList);
71. //Create board features
73. //Create the boarder of the board
74. let topLeft = **new** b.Cell(0,0);
75. let bottomRight = **new** b.Cell(boardWidth-1,boardHeight-1);
76. BR.createKeepOut(topLeft,bottomRight);
78. //Create any user specified keepouts
79. JSONKeepoutList.forEach( keepout => {
80. let topLeft = **new** b.Cell(keepout.start.x,keepout.start.y);
81. let bottomRight = **new** b.Cell(keepout.end.x,keepout.end.y);
83. BR.createKeepOut(topLeft,bottomRight,"keepout",1);
84. });
86. **try** {
87. let floodCellJSON = JSONFloodList[0];
88. **var** floodCell = board.getCell(floodCellJSON.x,floodCellJSON.y)
89. board.markNeighboursAsUnrouteable(floodCell, "Flood", 1);
90. } **catch**(err) {
91. **if** (err.name == "TypeError") {
92. //This is to be expected if we have nothing to route, nothing needs to be done here
93. } **else** {
94. //This is unexpected
95. **throw** err
96. }
97. }
99. //Find a route for all of the specified objects
100. let route = BR.route();
102. errors = route.errors;
103. let tracks = route.tracks;
105. //Apply the flood now we know where the nets are going.
106. **try** {
107. BR.flood(floodCell);
108. } **catch**(err) {
109. **if** (err.name == "TypeError") {
110. //This is to be expected if we have nothing to route
111. } **else** {
112. **throw** err
113. }
114. }
116. **return** {
117. DOM: BuildDOM(tracks, trackWidth),
118. errors: errors
119. };
121. }

124. **var** BuildDOM = **function**(tracks, trackWidth) {
125. //We now need to actually build the output, using the SVGmaker class
126. **var** SvgMaker = **new** svg.Maker;
128. //Shows the area on the board which have been marked as not routeable
129. **for** (**var** x = 0; x < board.width; x++) {
130. **for** (**var** y = 0; y < board.height; y++) {
131. **if** (!board.getCell(x,y).routeable) {
132. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
134. /\*\*
135. \* Example of Map Reduce used for debuging,
136. \* Shows which of the cells have at least one cell with a routing overide level of
137. \* at least 1.
138. \* Map - Go through all of the controlling cells and get the overide level
139. \* Reduce - Sum these overide levels if any of them are 1 then it will total to a non-zero
140. \* number and the statement will evaluate to true.
141. \*/
142. **if** (debug &&
143. board.getCell(x,y).controllingNet.map((i) => {
144. **return** i.overide;
145. }).reduce((a,c) => {
146. **return** a + c;
147. },0))
148. {
149. Rect.fillColour = **new** colour.Colour(124,0,0);
150. } **else** {
151. Rect.fillColour = **new** colour.Colour(0,124,174);
152. }
153. SvgMaker.addElement(Rect);
154. }
155. }
156. }
158. /\*\*If the flood list is empty then all of the tracked cells will be along the routes and so
159. \* we can speed up our generation times by only going along the nets.
160. \*
161. \* Otherwise we need to scan the whole board
162. \*/
163. **if** (JSONFloodList.length > 0) {
164. **for** (let x = 0; x < board.width; x++) {
165. **for** (let y = 0; y < board.height; y++) {
166. **if** (board.getCell(x,y).tracked) {
167. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
168. Rect.fillColour = **new** colour.Colour(0,0,124);
169. SvgMaker.addElement(Rect);
170. }
171. }
172. }
173. } **else** {
174. //The default where we can effiecently draw along the tracks rather than scan the whole
175. //Board
176. **for** (**var** track = 0; track < tracks.length; track++) {
177. **for** (**var** cell = 0; cell < tracks[track].length; cell++) {
178. let x = tracks[track][cell].x;
179. let y = tracks[track][cell].y;
181. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
182. Rect.fillColour = **new** colour.Colour(0,0,124)
183. SvgMaker.addElement(Rect);
184. }
185. }
186. }
188. **return** SvgMaker.getImage();
189. }
191. /\*\*
192. \* ===========SERVER===========
193. \*/

196. /\*\*
197. \* This serves all the files to the client
198. \* @type {module:http.Server}
199. \*/
200. let server = http.createServer(**function** (req, res) {
202. **var** parsedURL = url.parse(req.url, **true**);
204. //This switch allows assignment specific urls to files.
205. **switch** (parsedURL.pathname) {
206. //The entry point
207. **case** ('/'):
209. fs.readFile(path + indexLocation, **function** read(err, indexFile) {
210. **if** (err) **throw** err;
211. res.writeHead(200, {"Content-Type": "text/html"});
212. res.write(indexFile);
213. res.end();
214. });
216. **break**;
218. //This handels requests for a routed version of the board
219. **case** ('/route'):
221. //NetLists etc passed to the server inside the body of the request
222. let requestBody = **new** String;
223. req.on('data', chunk => {
224. requestBody += chunk.toString();
225. });
227. req.on('end', () => {
228. //The request has ended and so we have all of the data
229. // lets give them their new route
231. let route = routeJSON(JSON.parse(requestBody), parseInt(parsedURL.query.cellSize));
232. let svg = route.DOM;
233. let errors = route.errors;
235. responseContent = JSON.stringify({
236. board: svg,
237. errors: errors
238. })
240. res.end(responseContent);
241. });
243. **break**;
245. //If they are requesting styles or other scripts
246. **default**:
247. fs.readFile('./' + path + req.url, **function**(err, data) {
248. **if** (err) {
249. res.writeHead(404, "Not Found");
250. res.write('THERE WAS A 404')
251. res.end();
252. } **else** { //There was no error
254. /\*\*
255. \* We now need to automatically serve the file however
256. \* we don't know what mimetype (e.g. is it a text/html)
257. \* this gets the file extentsion and assumes that is the
258. \* value of the sign we want. We can do this by finding the
259. \* last dot in the file name, then looking that extention
260. \* up in a dictionary.
261. \*/
262. let dotPosFromEnd = req.url.lastIndexOf('.');
263. let mimetype = 'text/plain';
265. **if** (!(dotPosFromEnd == -1)) {
266. //Perform a look up for the text using the suffix in a dictionary
267. mimetype = fileTypes[req.url.substr(dotPosFromEnd)];
268. }
270. res.setHeader('Content-type' , mimetype);
271. res.end(data);
272. }
273. });
274. }
275. }).listen(port);
277. console.log('Sever listening on port ', port);

### BoardRouter.js

1. **const** nr = require('./netRouter');
2. **const** b = require('./board');
4. **const** hurestristicWeight = 1.5;
5. **const** errorThreshold = 2;
7. /\*\*
8. \* The class for routing a whole board
9. \* @param {Board} board
10. \* @param {Array<Board.Net>} netList
11. \*/
12. BoardRouter = **function** (board, netList) {
13. **this**.board = board;
14. **this**.netList = netList;
15. }
17. /\*\*
18. \* Creates a non routeable box
19. \* @param {Cell} cell1 The cell which defines one coner
20. \* @param {Cell} cell2 The cell which defines one coner
21. \* @param {Boolean} borderOnly Do we allow routing with in the area
22. \*/
23. BoardRouter.prototype.createKeepOut = **function**(cell1, cell2, borderOnly = **true**) {
25. **if** (**this**.board.CordsOnBoard(cell1.x, cell1.y)
26. && **this**.board.CordsOnBoard(cell2.x, cell2.y)) {
27. //find the smaller of the cords so that we can itterate using a for loop
28. **if** (cell1.x > cell2.x) {
29. **var** bigX = cell1.x;
30. **var** smallX = cell2.x;
31. } **else** {
32. **var** bigX = cell2.x;
33. **var** smallX = cell1.x;
34. }
36. **if** (cell1.y > cell2.y) {
37. **var** bigY = cell1.y;
38. **var** smallY = cell2.y;
39. } **else** {
40. **var** bigY = cell2.y;
41. **var** smallY = cell1.y;
42. }
44. **if** (borderOnly) {
45. //Mark the horizontal walls as unrouteable
46. **for** (let x = smallX + 1; x < bigX; x++) {
47. **this**.board.markCordsAsUnrouteable(x,smallY)
48. **this**.board.markCordsAsUnrouteable(x,bigY)
49. }
50. //Mark the verticle walls as unrouteable
51. **for** (let y = smallY; y < bigY + 1; y++) {
52. **this**.board.markCordsAsUnrouteable(smallX,y)
53. **this**.board.markCordsAsUnrouteable(bigX,y)
54. }
55. } **else** {
56. //Mark the whole areas as unrouteable
57. **for** (let x = smallX; x < bigX; x++) {
58. **for** (let y = smallY; y < bigY; y++) {
59. **this**.board.markCordsAsUnrouteable(x,y);
60. }
61. }
62. }
63. }
64. }
66. /\*\*
67. \* Floods all of the cells possible from a cell
68. \* This could have been made recursively in theory however
69. \* JS doesn't support tail optimaisation and so it would have caused
70. \* StackOver Flows left right and center.
71. \* @param {Cell} Cell Where we start the flood from
72. \* @returns {<Cell>} Returns list of cells in the flood
73. \*/
74. BoardRouter.prototype.flood = **function**(Cell) {

77. //Must check this as we use weather cell is defined in the while loop later
78. //!EXAMPLE OF ERROR HANDELING
79. **if** (**typeof**(Cell) == undefined) {
80. **throw** Error ("Flood method passed undefined cell");
81. }
83. **this**.board.markNeighboursAsRouteable(Cell, "Flood", 1)
85. unchecked = [];
86. flood = [];
87. current = Cell;
89. //Check if we can still pop items from the stack
90. **while** (current) {
91. **if** (current.routeable) {
92. //Gets the routeable neighbours and pushes them to the unchecked list
94. **this**.board.getValidNeighbours(current).forEach(cell => {
95. unchecked.push(cell)
96. });
98. current.routeable = **false**;
99. current.tracked = **true**;
101. flood.push(current);
102. }
104. current = unchecked.pop();
105. }
107. **return** flood;
108. }
110. /\*\*
111. \* A function decide which order the nets are to be routed in
112. \*/
113. BoardRouter.prototype.route = **function**() {
114. let errors = [];
116. let tracks = **new** Array;
118. //If i have to go through the list any ways then this is linner time and I could
119. //Implement my own sorting algorithium
120. **this**.netList.sort((cellA, cellB) =>
121. cellA.manhattanLength() - cellB.manhattanLength()
122. );
124. /\*\*
125. \* PRE PROCESSING
126. \*/
128. **try** {
130. **this**.netList = **this**.netList.filter((net) => {
131. **if** (!**this**.board.CordsOnBoard(net.start.x, net.start.y)
132. || !**this**.board.CordsOnBoard(net.end.x, net.end.y)) {
133. errors.push('Not all nets on the board');
134. **return** **false**
135. } **else** {
136. **return** **true**
137. }
138. })
140. **this**.netList.forEach((net, i) => {
141. net.routingErrors = 0;
142. net.id = i;
143. **this**.board.markNeighboursAsUnrouteable(net.startCell,**true**,net.id,1,)
144. **this**.board.markNeighboursAsUnrouteable(net.endCell,**true**,net.id,1)
145. });

148. } **catch** (err) {
149. **if** (err.name == "TypeError") {
150. //This is to be expected if there are no items on the board and we don't need to tell the user that
151. } **else** {
152. //This \*should\* never happen but if it does we want to know what happened
153. **throw** err;
154. }
155. }
157. //Clone the array
158. //use of a queue
159. let toRoute = **this**.netList.slice();
161. **while** (toRoute.length > 0) {
162. **try** {
163. currentNet = toRoute.shift();
164. let myNetRouter = **new** NetRouter(**this**.board,
165. currentNet,
166. hurestristicWeight,
167. currentNet.id);
168. let trace = myNetRouter.route();
169. tracks.push(trace);
171. } **catch** (err) {
173. currentNet.routingErrors++;
175. //It is possible to build a set of nets that will cause tracks to be ripped up in a loop
176. //This says just give up on a net after it has caused so many problems so we don't run forever
177. **if** (currentNet.routingErrors < errorThreshold) {
179. //Draws an L shape to find which nets that it should rip up
180. let getNetIDsToClear = **function**(net) {
181. //Use of set
182. let netIDsToRipup = **new** Set;
184. start = net.startCell;
185. end = net.endCell;
187. bigX = Math.max(start.x, end.x);
188. smallX = Math.min(start.x, end.x);
190. bigY = Math.max(start.y, end.y);
191. smallY = Math.min(start.y, end.y);
193. **for** (let x = smallX; x < bigX; x++) {
194. let netIDs = **this**.board.getCell(x,start.y).controllingNet;
195. netIDs.forEach(controllingNet => netIDsToRipup.add(controllingNet.controllingNetID));
196. }
198. **for** (let y = smallY; y < bigY; y++) {
199. let netIDs = **this**.board.getCell(y,end.x).controllingNet;
200. netIDs.forEach(controllingNet => netIDsToRipup.add(controllingNet.controllingNetID));
201. }
203. //Convert the set to an array
204. netIDsToRipup = [...netIDsToRipup];
206. //remove the current net from the array
207. **var** index = netIDsToRipup.indexOf(net);
208. **if** (index > -1) {
209. netIDsToRipup.splice(index, 1);
210. }
212. **return** netIDsToRipup
214. }

217. //Make the net that just failed the first net to try again
218. let toReRoute = [currentNet];
220. //Use of alot of functional methods
222. //Converts the net ID's to actual net objects and filter out the undefineds
223. //caused when netList is empty
224. let netIDsToClear = getNetIDsToClear(currentNet);
226. let netsToClear = netIDsToClear.map((ID) => {
227. let net = **this**.netList.find((net) => {
228. let **boolean** = (net.id == ID)
229. **return** **boolean**
230. });
231. **return** net
232. }, **this**).filter((element) => { element != undefined })
234. netIDsToClear.forEach((ID) => {
235. **this**.netList.splice(
236. **this**.netList.findIndex((net) => { **return** net.id == ID })
237. ,1);
238. })
240. //RipUp the nets
241. netsToClear.forEach(net => {
242. toReRoute.push(net)
244. **var** currentTrace = net.trace;
246. //This could get slow with alot of nets
247. //Delete the trace from the trace array
248. tracks.splice(tracks.findIndex((value) => {**return** (currentTrace == value)}),1);
250. currentTraceLength = currentTrace.length;
251. **for**(traceCellIndex = 0; traceCellIndex < currentTraceLength; traceCellIndex++) {
253. **this**.board.markNeighboursAsRouteable(currentTrace[traceCellIndex],
254. **true**,
255. net.id,
256. 0);
257. //Remove the tracks from the board
258. **this**.board.markCellAsUntracked(currentTrace[traceCellIndex]);
259. }
261. net.trace = [];
262. })
264. //Prepend the nets that need to be ReRouted to the front of the array
265. toRoute.unshift(...toReRoute);
266. }  **else** {
267. //Compile the error message for the user
268. errors.push("Routing failed on: {start: (" + currentNet.startCell.x + ',' + currentNet.startCell.y +
269. '), end: ('+ currentNet.endCell.x + ',' + currentNet.endCell.y + ')}')
270. }
271. }
272. }
274. **return** {
275. tracks: tracks,
276. errors: errors
277. };
278. }

281. module.exports = {BoardRouter};

### NetRouter.js

1. **var** BoardObject = require('./board');
2. **var** Heap = require('../Modules/Heap');
4. NetRouter = **class** {
5. /\*\*
6. \* A class for performing the routing of a net. A board of where the net can go is passed to it
7. \* and it will return the path the net will be routed along.
8. \* @param {Array<Array<Cell>>} board A board conataining all the tracks that aren't routed.
9. \* @param {Net} net The net to be routed
10. \* @param {Number} heuristicWeight the weight of the heuristic
11. \*/
12. constructor(board, net, heuristicWeight, ID) {
13. **this**.board = board;
14. **this**.net = net;
15. **this**.heuristicWeight = heuristicWeight;
16. **this**.ID = ID;
18. /\*\* Create a new heap which sorts the entities by the difference in their f values,
19. \*  where f = g + h
20. \*  g - Shortest known path from start
21. \*  h - Manhattan distance
22. \*/
23. **this**.toCheck = **new** Heap(**function**(cellA, cellB) {
24. **return** cellA.f - cellB.f;
25. })
27. //Add the starting point of the route to our list toCheck nodes.
28. **this**.startCell = **this**.net.startCell;
29. **this**.endCell = **this**.net.endCell;
30. }
32. /\*\*
33. \* The function that performs the path finding for a single net
34. \*/
35. route() {
36. let B = **this**.board;
38. **this**.startCell.g = 0;
39. **this**.startCell.f = 0;
41. **this**.startCell.checked = **true**;
43. B.markNeighboursAsRouteable(**this**.startCell,**true**,**this**.ID,1);
44. B.markNeighboursAsRouteable(**this**.endCell,**true**,**this**.ID,1);
46. **this**.toCheck.push(**this**.net.startCell);
48. //While there are still possible cells that there could be a route for.
49. **while** (!**this**.toCheck.empty()) {
51. let cell = **this**.toCheck.pop();
53. cell.checked = **true**;
55. //Have we found the endCell yet?
56. **if** (cell.x == **this**.endCell.x && **this**.endCell.y == cell.y) {
58. let current = cell;
60. B.markNeighboursAsUnrouteable(current, **true**, **this**.ID, 1);
62. **this**.net.trace.push(current);
64. /\*\* Starting from the endCell perform a trace back to
65. \*  the start, adding the cells that are on the trace to a list and marking
66. \*  all of the neighbours as unrouteable
67. \*/
68. **do** {
69. B.markNeighboursAsUnrouteable(current, **true**, **this**.ID, 0);
71. current = current.**super**;
73. current.tracked = **true**;
74. **this**.net.trace.push(current);
76. } **while** (current != **this**.startCell)
78. B.markNeighboursAsUnrouteable(**this**.startCell, **true**, **this**.ID, 1);
80. **this**.cleanUp();
82. **return** **this**.net.trace
84. }
86. /\*\*We've not found the end yet.
87. \* Lets work out work out which cell to expand into next
88. \*/
89. let neighbours = B.getValidNeighbours(cell);
91. let neightbourLength = neighbours.length;
92. **for** (let i = 0; i < neightbourLength; i++) {
94. let neighbour = neighbours[i];
95. let ng = cell.g + 1; //neighbough g -> ng
97. //Is there a route to that cell that we don't know abouts
98. **if** (ng < neighbour.g || !neighbour.checked) {
99. neighbour.**super** = cell;
101. neighbour.g = ng;
102. neighbour.h = **this**.heuristicWeight \* B.getEuclidean(neighbour, **this**.endCell);
103. neighbour.f = neighbour.g + neighbour.h;
105. neighbour.checked = **true**;
107. **this**.toCheck.push(neighbour);
108. }
109. }
110. }
112. **this**.cleanUp()
114. **throw** Error("No route found");
116. }
118. cleanUp() {
119. **for** (let x = 0; x < **this**.board.width; x++) {
120. **for** (let y = 0; y < **this**.board.height; y++) {
121. let cell = **this**.board.getCell(x,y);
122. cell.checked = **false**;
123. cell.**super** = **null**;
124. }
125. }
126. }
128. reset() {
129. **this**.net.trace = **new** Array;
130. }

133. }
135. module.exports = {NetRouter};

### Board.js

1. **class** Cell {
2. /\*\*
3. \*
4. \* @param {Number} x
5. \* @param {Number} y
6. \* @param {Number} layer
7. \* @param {Boolean} routeable
8. \*/
9. constructor (x,y,layer=1,routeable=**true**) {
11. /\*\*
12. \* @type {number}
13. \*/
14. **this**.x = x;
16. /\*\*
17. \* @type {number}
18. \*/
19. **this**.y = y;
21. /\*\*
22. \* This can be deleted now
23. \* @type {number}
24. \*/
25. **this**.layer = layer;
27. /\*\*
28. \* @type {boolean}
29. \*/
30. **this**.routeable = routeable;
32. **this**.hardRouteable = **true**;
33. /\*\*
34. \* If the Cell has a track going through it.
35. \* Defaults to false because at start of day we have no tracks
36. \* @type {boolean}
37. \*/
38. **this**.tracked = **false**;
40. **this**.controllingNet = **new** Array;
42. **this**.hardControllingNetID = **new** Array;
43. }
44. }
46. //Alow all of the Net class to be acessed from other files
47. /\*\*
48. \* A net for a single track.
49. \* This class holds the information about a single net
50. \* @param {Cell} start The start of the net
51. \* @param {Cell} end   The end of the net
52. \*/
53. **function** Net (start, end) {
55. /\*\*
56. \* @type {Cell}
57. \*/
58. **this**.startCell = start;
60. /\*\*
61. \* @type {Cell}
62. \*/
63. **this**.endCell = end;
65. **this**.trace = **new** Array;
66. }
68. /\*\*
69. \* This maybe too similar to the get manhattan distance board method
70. \* and theere may be a way of subcalssing these to make more sense
71. \* !This is the Euclidian length need to refactor
72. \*/
73. Net.prototype.manhattanLength = **function**() {
74. **return** Math.sqrt(Math.pow(**this**.startCell.x-**this**.endCell.x, 2) + Math.pow(**this**.startCell.y-**this**.endCell.y, 2));
75. }


79. //Allow all of the board class to be accessed from other files
81. /\*\*
82. \* The Class that holds all of the infomation about the board
83. \* @constructor
84. \* @param {number} boardWidth The width of the board
85. \* @param {number} boardHeight The height of the board
86. \* @param {Array<Array<number|boolean>>} routeMask An boolean grid showing all the places that the route can not go
87. \*/
88. **var** Board = **function**(boardWidth, boardHeight, routeMask=[[]]) {
90. /\*\*
91. \* @type {number}
92. \*/
93. **this**.width = boardWidth;
95. /\*\*
96. \* @type {number}
97. \*/
98. **this**.height = boardHeight;
100. /\*\*Create a matrix for the grid to be stored in then go through and
101. \* populate it with cells
102. \*/
104. **this**.grid = **new** Array(**this**.height);
106. **for** (let y = 0; y < **this**.height; y++) {
107. **this**.grid[y] = **new** Array(**this**.width);
109. **for** (let x = 0; x < **this**.width; x++) {
110. **this**.grid[y][x] = **new** Cell(x, y);
111. }
113. }
115. /\*\*
116. \* Check if each of the cells are routeable according to the inputted routing guide.
117. \*/
119. **if** ((routeMask != undefined)
120. && (routeMask.length > **this**.height)
121. && (routeMask[0].length > **this**.width)) {
123. //NOTE TO SELF - Possibly look at merging this and the for loop above together
125. **for** (let y = 0; y < routeMask.length; y++) {
126. **for** (let x = 0; x < routeMask[y].length; x++) {
127. **if** (routeMask[x][y] = 1) grid[y][x].routeable = **false**;
128. }
129. }
130. }
132. };

135. /\*\*
136. \* Checks if a cell is on the board
137. \* @param {Cell} Cell
138. \*/
139. Board.prototype.CordsOnBoard = **function**(x ,y) {
140. **return** (x >= 0 && x < **this**.width)
141. && (y >= 0 && y < **this**.height);
142. }
144. /\*\*
145. \* Checks weather we can route a track through a given Cell
146. \* @param {Cell} Cell
147. \*/
149. Board.prototype.validCord = **function** (x,y) {
150. **return** **this**.grid[y][x].routeable && !**this**.grid[y][x].checked
151. }

154. Board.prototype.getNeighbours = **function**(Cell, diagonals) {
155. let neighbours = **new** Array;
157. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y)) {
158. neighbours.push(**this**.grid[Cell.y + 1][Cell.x]);
159. };
161. **if** (**this**.CordsOnBoard(Cell.x, Cell.y + 1)) {
162. neighbours.push(**this**.grid[Cell.y][Cell.x + 1]);
163. };
165. **if** (**this**.CordsOnBoard(Cell.x, Cell.y -1)) {
166. neighbours.push(**this**.grid[Cell.y - 1][Cell.x]);
167. };
169. **if** (**this**.CordsOnBoard(Cell.x - 1, Cell.y)) {
170. neighbours.push(**this**.grid[Cell.y][Cell.x - 1]);
171. };
173. **if** (diagonals) {
174. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y + 1)) {
175. neighbours.push(**this**.grid[Cell.y + 1][Cell.x + 1]);
176. };
178. **if** (**this**.CordsOnBoard(Cell.x - 1, Cell.y + 1)) {
179. neighbours.push(**this**.grid[Cell.y + 1][Cell.x - 1]);
180. };
182. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y -1)) {
183. neighbours.push(**this**.grid[Cell.y - 1][Cell.x + 1]);
184. };
186. **if** (**this**.CordsOnBoard(Cell.x - 1, Cell.y - 1)) {
187. neighbours.push(**this**.grid[Cell.y - 1][Cell.x - 1]);
188. };
189. }
191. **return** neighbours
192. }
194. /\*\*
195. \* Returns a list of the Neighbours of a Cell, if the
196. \* cell is valid it will return the cell's neighbours object
197. \* @param {Cell} Cell
198. \*/
199. Board.prototype.getValidNeighbours = **function** (Cell){
201. let neighbours = **new** Array;
203. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y)
204. && **this**.grid[Cell.y + 1][Cell.x].routeable) {
206. neighbours.push(**this**.grid[Cell.y + 1][Cell.x]);
207. };
209. **if** (**this**.CordsOnBoard(Cell.x, Cell.y + 1)
210. && **this**.grid[Cell.y][Cell.x + 1].routeable) {
212. neighbours.push(**this**.grid[Cell.y][Cell.x + 1]);
213. };
215. **if** (**this**.CordsOnBoard(Cell.x, Cell.y -1)
216. && **this**.grid[Cell.y - 1][Cell.x].routeable) {
218. neighbours.push(**this**.grid[Cell.y - 1][Cell.x]);
219. };
221. **if** (**this**.CordsOnBoard(Cell.x, Cell.y - 1)
222. && **this**.grid[Cell.y][Cell.x - 1].routeable) {
224. neighbours.push(**this**.grid[Cell.y][Cell.x - 1]);
225. };
227. **return** neighbours;
228. }
230. /\*\*
231. \* Gets the neighbours of a cell including diagonals
232. \* @param {Cell} Cell
233. \*/
234. Board.prototype.getCellAndAllNeighbours = **function**(Cell) {
235. let cells = **new** Array
237. **for** (let x = -1; x <= 1; x++) {
238. **for** (let y = -1; y <= 1; y++) {
240. **if** (**this**.CordsOnBoard(Cell.x + x,Cell.y + y)
241. && !(x == 0 && y == 0)) {
242. cells.push(**this**.grid[Cell.y + y][Cell.x + x]);
243. }
244. }
245. }
247. **return** cells;
248. }
250. /\*\*
251. \* Gets the manhattan distance between two cells
252. \* @param {BoardObject.Cell} cell1 Cells for distance to be found between
253. \* @param {BoardObject.Cell} cell2 Cells for distance to be found between
254. \*/
255. Board.prototype.getManhattan = **function**(cell1, cell2) {
256. //Pythagouses theorem to get the Manhattan distance
257. **return** (Math.abs(cell1.x-cell2.x) + Math.abs(cell1.y-cell2.y));
258. }
260. /\*\*
261. \*
262. \* @param {BoardObject.Cell} cell1
263. \* @param {BoardObject.Cell} cell2
264. \*/
265. Board.prototype.getEuclidean = **function**(cell1, cell2) {
266. **return** Math.sqrt(Math.pow(cell1.x-cell2.x, 2) + Math.pow(cell1.y-cell2.y, 2));
267. }
269. /\*\*
270. \* Finds all of the cells which are neighbours (that are also on the)
271. \* board and marks them and all of their neighours as not routeable.
272. \* It then specifies the reason why they are not routeable and the level
273. \* of overide required to make it routebale
274. \*/
275. Board.prototype.markNeighboursAsUnrouteable = **function**(Cell, diagonals=**false**, NetID=**null**, overide=0) {
276. **this**.getNeighbours(Cell, diagonals).forEach((neighbour) => {
277. **this**.markCellAsUnrouteable(neighbour,NetID,overide);
278. });
279. }
281. /\*\*
282. \* @param {\*} Cell
283. \* @param {\*} diagonals
284. \* @param {\*} ID
285. \*/
286. Board.prototype.markNeighboursAsRouteable = **function**(Cell, diagonals=**false**, ID, overide=0) {
287. **this**.getNeighbours(Cell, diagonals).forEach(neighbour => {
288. **this**.markCellAsRouteable(neighbour,ID,overide)
289. },**this**);
290. };
292. Board.prototype.markCellAsRouteable = **function**(cell, ID=**null**, overide) {
293. /\*\*Need to check weather this is the only net that
294. \* Controlls this cell, if not then we don't want to
295. \* remove it from this nets trace
296. \*
297. \* If it is the only one we remove it from the controlling array
298. \*
299. \* If the controlling array is then empty then we the demark it
300. \*/
301. **for** (let i = 0; i < cell.controllingNet.length; i++) {
302. **if** (cell.controllingNet[i].controllingNetID == ID) {
303. **if** (overide >= cell.controllingNet[i].overide) {
304. cell.controllingNet.splice(i, 1);
306. **if** (cell.controllingNet.length == 0) {
307. cell.routeable = **true**;
308. };
309. };
311. **break**; //No point to continue itteration
312. }
313. }
314. }
316. Board.prototype.markCellAsUnrouteable = **function**(cell, ID=**null**, overide=0) {
317. //Convert to set
318. **for** (let i = 0; i < cell.controllingNet.length; i++) {
319. **if** (cell.controllingNet[i].controllingNetID == ID)   {
320. **if** (cell.controllingNet[i].controllingNetID < overide) {
321. cell.controllingNet[i].controllingNetID = overide;
322. cell.routeable = **false**;
323. }
324. **return**;
325. }
326. }
328. //Not in list
329. cell.controllingNet.push({
330. controllingNetID: ID,
331. overide: overide
332. })
333. cell.routeable = **false**;
334. }
336. //!This should be removed but needs to have uses removed first
337. Board.prototype.markCordsAsUnrouteable = **function**(x,y, ID=**null**, overide=0) {
338. **this**.markCellAsUnrouteable(**this**.getCell(x,y),ID, overide);
339. }

342. Board.prototype.markCordsAsTracked = **function**(x,y) {
343. **this**.grid[y][x].tracked = **true**;
344. }
346. Board.prototype.markCellAsUntracked = **function**(cell) {
347. cell.tracked = **false**;
348. }
350. Board.prototype.getCell = **function**(x,y) {
351. **try** {
352. **return** **this**.grid[y][x];
353. } **catch** (err) {
354. **if** (err.message == "TypeError") {
355. **throw** **new** err ("Cords not on Board")
356. } **else** {
357. **throw** err
358. }
359. }
360. }

363. module.exports = {Cell, Board, Net};

### Colour.js

1. **class** Colour {
2. /\*\*
3. \* A prototype that allows storage and manipulation of colour objects
4. \* @param {Number} red
5. \* @param {Number} green
6. \* @param {Number} blue
7. \*/
8. constructor(red, green, blue) {
9. **this**.red = red;
10. **this**.green = green;
11. **this**.blue = blue;
12. };
14. /\*\* Add the colour values up in decimal, shifting the bits for each colour value to the left then adding them together
15. \*  The hex value is then converted to a string and the leading value is lost as it was just put there to absorb overflows
16. \*  @returns {string} returns 7 Char string with the first char being a #
17. \*/
18. toHexString() {
19. **return** '#' + (0x10000 \* **this**.red + 0x100 \* **this**.green + **this**.blue + 0x1000000).toString(16).substr(1);
20. };
21. }
22. module.exports = {Colour}

### SVG.js

1. **const** colour = require('./colour');
3. /\*\*
4. \* An Abstract prototype for SVG shapes
5. \* @param {Number} x
6. \* @param {Number} y
7. \*/
8. **var** svgShape = **function**(x, y, id, DOMClass) {
9. //Make it so that the svgShape class is abstract
10. **if** (**new**.target === svgShape) {
11. **throw** **new** TypeError("Cannot construct Abstract instance of svgShape directly");
12. }
14. //Ensure that all subclasses implement the generateSVGString method
15. **if** (**this**.generateSVGString === undefined) {
16. **throw** **new** TypeError("Must override generateSVGString method of the abstract class svgShape");
17. }
19. /\*\*Ensure that all subclasses implement the getCordinatesOfBottomRight method
20. \* Used in working out weather to exand the size of the svgShape.
21. \*/
22. **if** (**this**.getCordinatesOfBottomRight === undefined) {
23. **throw** **new** TypeError("Must override getCordinatesOfBottomRight method of the abstract class svgShape");
24. }
26. **this**.xPos = x;
27. **this**.yPos = y;
29. **this**.id = id;
30. **this**.DOMClass = DOMClass
32. **this**.nameOf = "svgShape";
34. }
36. //Sub-class svgShape
37. **class** Rectangle **extends** svgShape {
39. /\*\*
40. \* A class for storing svg rectangles
41. \* @param {Number} xPos The x position of the top left coner
42. \* @param {Number} yPos The y position of the top left coner
43. \* @param {Number} xLen The length of the shape in the x direction
44. \* @param {Number} yLen The length of the shape in the y direction
45. \* @param {String} id   The id of the rectangle.
46. \*/
47. constructor (xPos = 0, yPos = 0, xLen = 10, yLen = 10, id, DOMClass="Rect") {
48. **super**(xPos, yPos, id, DOMClass);
49. **this**.xLen = xLen;
50. **this**.yLen = yLen;
51. **this**.fillColour = **new** colour.Colour(255,0,0);
52. **this**.borderColour = **new** colour.Colour(255,255,255);
53. **this**.borderWidth = 5;
54. }
55. }
57. /\*\*
58. \* Takes all of the parameters of this SVG shape and returns a valid DOM
59. \* @returns {String} returns valid svg DOM
60. \*/
61. Rectangle.prototype.generateSVGString = **function**() {
62. **return**  '<rect id="#' + **this**.id +
63. '" class="' + **this**.DOMClass +
64. '" x=' + **this**.xPos +
65. ' y=' + **this**.yPos +
66. ' width=' + **this**.xLen +
67. ' height=' + **this**.yLen +
68. ' style="fill:' + **this**.fillColour.toHexString() +
69. ';, stroke:' + **this**.borderColour.toHexString() +
70. ';, stroke-width: ' + **this**.borderWidth +
71. '"></rect>';
72. }
74. /\*\*
75. \* Used for board resizing
76. \* @returns {object}
77. \*/
78. Rectangle.prototype.getCordinatesOfBottomRight = **function**() {
79. **return** {
80. xCord: **this**.xPos + **this**.xLen,
81. yCord: **this**.yPos + **this**.yLen
82. }
83. }
85. /\*\*
86. \* A Class used to store an array of SVG entities and generate them into valid DOM
87. \* @param {Number} \_width
88. \* @param {Number} \_height
89. \* @param {Boolean} overflows Controlls weather the SVG resizes to make sure that none of its elements overflow.
90. \*/
91. **var** Maker = **function**(\_width=500, \_height=500, overflows=**true**) {
92. **this**.footer = "</svg>"
94. /\*\*
95. \* @type {<svgShape>}
96. \*/
97. **this**.entities = [];
99. **this**.width = \_width;
100. **this**.height = \_height;
102. **this**.overflows = overflows;
104. }
106. /\*\*
107. \* Generates the correct SVG header
108. \*/
109. Maker.prototype.generateHeader = **function** () {
110. **return** "<svg version='1.1'"
111. + "baseProfile='full'"
112. + "width='" + **this**.width.toString() + "' height='" + **this**.height.toString() + "'"
113. + "xmlns=http://www.w3.org/2000/svg>";
114. }
116. /\*\*
117. \* Stores the element in the svgShape class ready for the next render
118. \* @param {svgShape} element
119. \*/
120. Maker.prototype.addElement = **function** (element) {
121. **this**.entities.push(element)
122. }
124. /\*\*
125. \* @returns {String} Returns valid SVG dom of all of the shapes that it conatains
126. \*/
127. Maker.prototype.getImage = **function**() {
128. let DOM = ''
129. let MaxX = **this**.width
130. let MaxY = **this**.height
132. **for** (let i = 0; i < **this**.entities.length; i++) {
133. **if** (**typeof** **this**.entities[i] == "string") {
134. DOM += **this**.entities[i]
135. } **else** **if** (**this**.entities[i].nameOf == "svgShape") {
136. DOM += **this**.entities[i].generateSVGString();
137. } **else** {
138. **throw** **new** TypeError("Unknown object passed to Maker");
139. }
141. **if** (**this**.overflows) {
142. **if** (MaxX < **this**.entities[i].getCordinatesOfBottomRight().xCord) {
143. MaxX = **this**.entities[i].getCordinatesOfBottomRight().xCord;
144. }
146. **if** (MaxY < **this**.entities[i].getCordinatesOfBottomRight().yCord) {
147. MaxY = **this**.entities[i].getCordinatesOfBottomRight().yCord;
148. }
149. }
150. **this**.width = MaxX;
151. **this**.height = MaxY;
152. };
153. **return** (**this**.generateHeader() + DOM + **this**.footer);
154. }

157. module.exports = {Maker, Rectangle}

## Client Side

### Index.html

1. <!DOCTYPE html**>**
2. **<html>**
3. **<head>**
4. **<meta** charset="utf-8"**/>**
5. **<link** type="text/css" rel="stylesheet" href="styles/index.css"**/>**
6. **</head>**
7. **<body>**

10. **<div** class="grid-container"**>**
12. **<div** class="title"**>**
13. **<h1>**Tracer**</h1>**
14. **</div>**
16. **<div** class="control-pannel"**>**
17. **<h2>**Control-pannel**</h2>**
18. **<button** id="addNet" onclick="addNetButtonListener()"**>**Add Net**</button><br/>**
19. **<button** id="addFlood" onclick="addFloodButtonListener()"**>**Add flood**</button><br/>**
20. **<button** id="addKeepout" onclick="addKeepoutButtonListener()"**>**Add keepout**</button><br/>**
21. **<button** id="DownloadBoard" onclick="downloadButtonListener()"**>**Download**</button><br/>**
22. **<h3>**To delete press Ctrl and click on what you want to delete**</h3><br/>**
23. **<h3** id="warning"**></h3>**
24. **<h3** id="errors"**></h3>**
25. **</div>**
27. **<div** id="board"**>**
29. **</div>**
31. **<div** id="node-container"**></div>**
33. **<div** id="flood-container"**></div>**
35. **<div** id="keepout-container"**></div>**
37. **</div>**
39. **<script** type="text/javascript" src="indexScript.js"**></script>**
41. **</body>**
42. **</html>**

### Index.css

1. \* { **margin**:0; **padding**:0; } /\* to remove the top and left whitespace \*/
3. html, body { **width**:100%; **height**:100%; } /\* just to be sure these are full screen\*/
5. body {
6. **background**: rgb(49, 49, 49);
7. **color**: rgb(49, 49, 49);
8. **font-family**: 'Ubuntu', sans-serif;
9. **position**: relative;
10. **text-align**: center;
11. }
13. button {
14. **margin**: 1vh;
15. **padding**: 1vh;
16. **width**: 10vw;
17. }
19. .Point {
20. **position**: absolute;
21. **z-index**: 9;
22. **background-color**: #13e799;
23. **text-align**: center;
24. }
26. .flood {
27. **background-color**: #ff0000;
28. }
30. .keepout {
31. **background-color**: #dada86;
32. }
34. .h3 {
35. **margin**: 20px;
36. }
38. #warning #errors {
39. **color**: #790202;
40. }
42. .PointPadder {
43. **padding**: 5px;
44. **cursor**: move;
45. **z-index**: 10;
46. }
48. svg {
49. **position**: relative;
50. **float**: left;
51. }
53. @supports (**display**: grid) {
54. .grid-container {
55. **display**: grid;
56. grid-template-columns: 1fr 5fr;
57. grid-template-rows: 1fr auto;
58. grid-template-areas:
59. "titleHeader titleHeader"
60. "controlPannel boardGrid";
61. }
63. .control-pannel {
64. grid-area: controlPannel;
65. **width**: 100%;
66. **height**: 90vh;
67. **margin**: auto;
68. **background**: #37c4bd;
69. }
71. .board {
72. grid-area: boardGrid;
73. **background**: #dada86;
74. **z-index**: -9;
75. **max-width**: 50%;
76. **max-height**: 50%;
77. }
79. .title {
80. grid-area: titleHeader;
81. **background**: #32b4a9
82. }
83. }

### Indexscript.js

1. //Define some functions for use just here
2. **function** getRoundedMouseX (event, roundToNearest = cellSize) {
3. **return** Math.ceil(event.clientX / roundToNearest) \* roundToNearest;
4. }
6. **function** getRoundedMouseY (event, roundToNearest = cellSize) {
7. **return** Math.ceil(event.clientY / roundToNearest) \* roundToNearest;
8. }
10. **class** clientSideCell {
11. constructor(x,y){
12. **this**.x = x;
13. **this**.y = y;
14. }
16. buildDOM (elementID, ...classes) {
17. **var** cellPadder = document.createElement("div");
18. cellPadder.className = "PointPadder";
19. cellPadder.id = elementID + "Padding";
21. classes.forEach(DOMclass => {
22. cellPadder.classList.add(DOMclass);
23. });
25. **var** cell = document.createElement("div");
26. cell.className = "Point";
27. cell.id = elementID;
29. cell.appendChild(cellPadder);
31. **this**.el = cell;
32. **this**.elementID = elementID;
34. **return** cell;
35. }
36. }
38. **class** clientSideNet {
39. constructor(start, end, id) {
40. **this**.start = start;
41. **this**.end = end;
42. **this**.id = id;
43. }
44. }
46. **class** clientsideKeepout {
47. constructor(start, end, id) {
48. **this**.start = start;
49. **this**.end = end;
50. }
51. }
53. **class** Grid {
54. /\*\*
55. \* @pram {Number} width the width cell units
56. \* @pram {Number} width the height in cell units
57. \* @pram {Number} cellSize the size in cell units
58. \*/
59. constructor (gridID, endPointContainerID, floodContainerID ,keepoutContainerID, width, height, cellSize) {
60. **this**.grid = document.getElementById(gridID);
61. **this**.endPointContainer = document.getElementById(endPointContainerID);
62. **this**.floodContainer = document.getElementById(floodContainerID);
63. **this**.keepoutContainer = document.getElementById(keepoutContainerID);
64. **this**.width = width;
65. **this**.height = height;
66. **this**.cellSize = cellSize;
67. **this**.netList = [];
68. **this**.floodList = [];
69. **this**.keepoutList = [];
70. **this**.cellCounter = 0;
71. **this**.svg = **null**;
72. }
74. makeInteractable(cell, deleteCallBack) {
75. **var** board = **this**;
76. **var** startx = cell.x \* **this**.cellSize;
77. **var** starty = cell.y \* **this**.cellSize;
78. **var** newXPos = 0;
79. **var** newYPos = 0;
80. **var** oldXPos = 0;
81. **var** oldYPos = 0;
82. **var** elmnt = cell.el; //el is standard shorthand for element in web frameoworks
83. **var** mouseDownPos = {
84. cellx: cell.x,
85. celly: cell.y,
86. mousex: startx,
87. mousey: starty
88. }

91. //Overide default method
92. document.getElementById(cell.elementID + "Padding").onmousedown = dragMouseDown ;
94. //Offset by the correct amount given css grid layout
95. elmnt.style.top = (starty)
96. + board.grid.getBoundingClientRect().top
97. + "px";
99. elmnt.style.left = (startx)
100. + board.grid.getBoundingClientRect().left
101. + "px";
103. **function** dragMouseDown(e) {
104. e.preventDefault();   //Prevents the default method from running

107. // get the mouse cursor position at startup:
108. oldXPos = getRoundedMouseX(e);
109. oldYPos = getRoundedMouseY(e);
111. mouseDownPos.mousex = elmnt.offsetLeft;
112. mouseDownPos.mousey = elmnt.offsetTop;
113. mouseDownPos.cellx = cell.x;
114. mouseDownPos.celly = cell.y;
116. /\*\*
117. \* If the control key is down then the user wants to delete this item
118. \*/
119. **if** (e.ctrlKey) {
120. deleteCallBack()
121. board.update();
122. } **else** {
123. document.onmouseup = stopDragging;  //Override the onmouseupmethod
124. document.onmousemove = elementDrag;     //Assign the on mousemove elementDrag method to elementDrag
125. }
126. }
128. **function** elementDrag(e) {
129. e.preventDefault();
131. // calculate the new cursor position:
132. newXPos = oldXPos - getRoundedMouseX(e);
133. newYPos = oldYPos - getRoundedMouseY(e);
135. // set the element's new position:
136. elmnt.style.top = (elmnt.offsetTop - newYPos) + "px";
137. elmnt.style.left = (elmnt.offsetLeft - newXPos) + "px";
139. //Update the old positions
140. oldXPos = getRoundedMouseX(e);
141. oldYPos = getRoundedMouseY(e);
142. }

145. **function** stopDragging() {
146. /\* stop moving when mouse button is released:\*/
147. document.onmouseup = **null**;
148. document.onmousemove = **null**;
150. //find new the cells position in grid terms.
151. let newCellX = Math.ceil((elmnt.offsetLeft - board.grid.getBoundingClientRect().left)/board.cellSize);
152. let newCellY = Math.ceil((elmnt.offsetTop - board.grid.getBoundingClientRect().top)/board.cellSize);
154. /\*\*Test if the new position for the cell is where any of the current ends of nets
155. \* keepouts and floods are, or if it is adjacent to any other cells
156. \*/
157. let isCellInvalid = (cell) => {
158. //Check weather it was the orginal cell
159. **if** ((mouseDownPos.celly) != cell.y || (mouseDownPos.cellx) != cell.x) {
160. **for** (let x = - 1; x <= 1; x++) {
161. **for** (let y = - 1; y <= 1; y++) {
162. **if** ((cell.x - x == newCellX
163. && cell.y - y == newCellY)
164. || (cell.x - x == newCellX
165. && cell.y - y == newCellY))
166. {
167. **return** **true**
168. }
170. }
171. }
172. }
173. **return** **false**
174. };
176. **if** (
177. (board.netList.some(net => {
178. **return** isCellInvalid(net.start)
179. || isCellInvalid(net.end)
180. } ))
182. || (board.floodList.some(isCellInvalid))
184. || (board.keepoutList.forEach(keepout => {
185. **return** isCellInvalid(keepout.start)
186. || isCellInvalid(keepout.end)
187. } ))
189. ){
190. //Raise the error
191. document.getElementById('warning').innerHTML = "Nodes too close!";
192. setTimeout(() => document.getElementById('warning').innerHTML = "", 2500);
194. elmnt.style.top = (mouseDownPos.mousey) + "px";
195. elmnt.style.left = (mouseDownPos.mousex) + "px";
196. } **else** {
197. cell.x = newCellX;
198. cell.y = newCellY;
199. }
201. board.update();
202. }
203. }
205. /\*\*
206. \* Inalizes a net
207. \*/
208. createNet(startx,starty,endx,endy) {
209. let start = **new** clientSideCell(startx, starty);
210. let end = **new** clientSideCell(endx, endy);
211. let startID = "start" + **this**.cellCounter;
212. let endID = "end" + **this**.cellCounter;
213. **this**.cellCounter++;
215. **this**.endPointContainer.appendChild(
216. start.buildDOM(startID)
217. );
219. **this**.endPointContainer.appendChild(
220. end.buildDOM(endID)
221. );
223. let net = **new** clientSideNet(start,end);
225. **this**.makeInteractable(start,deleteCallBack);
226. **this**.makeInteractable(end,deleteCallBack);
228. let netList = **this**.netList
229. netList.push(net);
231. **function** deleteCallBack() {
232. start.el.remove(); //Remove the Dom
233. end.el.remove();
234. **var** index = netList.indexOf(net);
235. **if** (index > -1) {
236. netList.splice(index, 1);
237. }
238. }

241. }
243. createFlood(x,y) {
244. let cell = **new** clientSideCell(x,y);
246. **this**.floodContainer.appendChild(
247. cell.buildDOM("flood","flood")
248. )
250. **this**.makeInteractable(cell, deleteCallBack);
252. **this**.floodList.push(cell);
254. let floodList = **this**.floodList;
256. **function** deleteCallBack() {
257. cell.el.remove(); //Remove the Dom
258. document.getElementById("addFlood").disabled = **false**;
259. floodList.pop();
260. }
261. }
263. createKeepout(startx, starty, endx, endy) {
264. let start = **new** clientSideCell(startx, starty);
265. let end = **new** clientSideCell(endx, endy);
267. **this**.keepoutContainer.appendChild(
268. start.buildDOM("keepoutStart" + **this**.cellCounter,"keepout")
269. );
271. **this**.keepoutContainer.appendChild(
272. end.buildDOM("keepoutEnd" + **this**.cellCounter,"keepout")
273. );
275. let keepout = **new** clientsideKeepout(start,end,**this**.cellCounter);
277. **this**.cellCounter++;
279. **this**.makeInteractable(start,deleteCallBack);
280. **this**.makeInteractable(end,deleteCallBack);
282. **this**.keepoutList.push(keepout);
284. let keepoutList = **this**.keepoutList;
286. **function** deleteCallBack() {
287. start.el.remove(); //Remove the Dom
288. end.el.remove();
289. **var** index = keepoutList.indexOf(keepout);
290. **if** (index > -1) {
291. keepoutList.splice(index, 1);
292. }
293. }
294. }
296. addNet(net) {
297. **this**.netList.push(net)
298. **return** **this**.netList.length - 1;
299. }
301. //This is quite hacky and not very OOP should fix at somepoint
302. net(id, contents) {
303. **this**.netList[id] = contents;
304. }
306. async update() {
307. //Fetch the SVG DOM from the server but store the value as a promise
308. let requestContent = {
309. netList: **this**.netList,
310. floodList: **this**.floodList,
311. keepoutList: **this**.keepoutList
312. }
314. let resGrid = fetch('/route?cellSize=' + **this**.cellSize, {
316. method: 'POST',
317. headers: {
318. 'Content-Type': 'application/json'
319. },
320. body: JSON.stringify(requestContent)
322. })
324. //Await for the promise to be fullied when it is set the value of the dom to be that
325. let response = await resGrid.then(response => response.text());
326. response = JSON.parse(response)
327. **this**.grid.innerHTML = response.board;
328. **this**.svg = response.board;
329. //!Example of reduce
330. **if** (response.errors.length > 0) {
331. let errorHeader = "Warning: <br/>"
332. let groupedErrors = errorHeader + response.errors.reduce((accumulator, current) => accumulator + "<br/>" + current);
333. document.getElementById('errors').innerHTML = groupedErrors;
334. } **else** {
335. document.getElementById('errors').innerHTML = "";
336. }
337. }
338. }
340. **function** addNetButtonListener() {
341. grid.createNet(2,2,4,4)
342. grid.update();
343. }
345. **function** addFloodButtonListener() {
346. document.getElementById("addFlood").disabled = **true**;
347. grid.createFlood(5,5);
348. grid.update();
349. }
351. **function** addKeepoutButtonListener() {
352. grid.createKeepout(15,15,20,20);
353. grid.update();
354. }
356. **fu****nction** downloadButtonListener() {
357. **var** anchor = window.document.createElement('a');
358. anchor.download = 'board.svg';
359. anchor.href = window.URL.createObjectURL(**new** Blob([grid.grid.innerHTML], {type: 'text/svg'}));
361. document.body.appendChild(anchor);
362. anchor.click();
364. document.body.removeChild(anchor);
365. }
366. //------------MAIN------------
368. **const** cellSize = 10;
369. **const** gridWidth = 10;
370. **const** gridHeight = 10;
371. **const** nodeContainerID = "node-container";
372. **const** floodContainerID = "flood-container";
373. **const** keepoutContainerID = "keepout-container"
375. **const** gridID = "board";
377. **var** grid = **new** Grid(gridID,
378. nodeContainerID,
379. floodContainerID,
380. keepoutContainerID,
381. gridWidth,
382. gridHeight,
383. cellSize);
385. grid.createNet(2,2,3,3)
387. grid.update();

1. Footprints are the shape of the area on the board that a component will occupy as well as the location of its contact points. Most EDA software treats footprints as entities. [↑](#footnote-ref-2)
2. Jumps are pieces of wire which effectively act as another layer as they travel outside the board. [↑](#footnote-ref-3)
3. Isometric routing – is where both of the routes are the same length, this is useful for wires that carry signals so that if the signals are traveling at the same speed over the same distance then they will both reach the end at the same time. [↑](#footnote-ref-4)
4. Push and shove routing is a method of routing where the user alters the end points that the router is routing to and from on the fly giving the user real time feedback for their placement of the end points. [↑](#footnote-ref-5)
5. EDA - Electronic Design Application, CAD software for modelling PCB’s [↑](#footnote-ref-6)
6. KiCAD was chosen as the target platform as it is open source and so there is better documentation on the file structure. [↑](#footnote-ref-7)
7. ES6 short for ECMA Script is a JS standard for the web that was released in 2016 and adds a lot of nice to have features to JS. [↑](#footnote-ref-8)
8. The heap module that I’ve been using is made by Qiao <https://raw.githubusercontent.com/qiao/heap.js/master/lib/heap.js> [↑](#footnote-ref-9)
9. Push and shove routing is a method of routing where the user alters the end points that the router is routing to and from on the fly giving the user real time feedback for their placement of the end points. [↑](#footnote-ref-10)
10. EDA - Electronic Design Application, CAD software for modelling PCB’s [↑](#footnote-ref-11)
11. KiCAD was chosen as the target platform as it is open source and so there is better documentation on the file structure. [↑](#footnote-ref-12)
12. See the end user comment [↑](#footnote-ref-13)
13. EDP – Event Driven Programing [↑](#footnote-ref-14)