PCB Auto-routing

Computer Science Coursework

Edwin Shepherd

# Contents

[Contents 1](#_Toc2019334)

[Analysis 3](#_Toc2019335)

[The Problem 3](#_Toc2019336)

[What is routing 3](#_Toc2019337)

[Different types of routing 4](#_Toc2019338)

[Users, Limitations and Requirements 5](#_Toc2019339)

[Research 6](#_Toc2019340)

[Interview 6](#_Toc2019341)

[Perquisites and missing requirements 6](#_Toc2019342)

[Objectives 6](#_Toc2019343)

[Prototyping 8](#_Toc2019344)

[Objectives 8](#_Toc2019345)

[What I found out 8](#_Toc2019346)

[Data structure 9](#_Toc2019347)

[Documented Design 10](#_Toc2019348)

[Program overview 10](#_Toc2019349)

[A Star 10](#_Toc2019350)

[Highlevel Routing 11](#_Toc2019351)

[User Interface 12](#_Toc2019352)

[UML Digramoverview 16](#_Toc2019353)

[Data structures 17](#_Toc2019354)

[Screen Shots 18](#_Toc2019355)

[Technical solution 19](#_Toc2019356)

[Server side 19](#_Toc2019357)

[Index.js 19](#_Toc2019358)

[BoardRouter.js 24](#_Toc2019359)

[NetRouter.js 29](#_Toc2019360)

[Board.js 32](#_Toc2019361)

[Colour.js 38](#_Toc2019362)

[SVG.js 39](#_Toc2019363)

[Client Side 42](#_Toc2019364)

[Index.html 42](#_Toc2019365)

[INDEX.CSS 43](#_Toc2019366)

[Indexscript.js 45](#_Toc2019367)

[Bibliography 52](#_Toc2019368)

# 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.

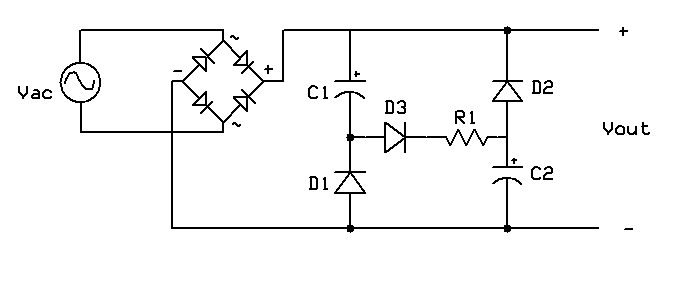
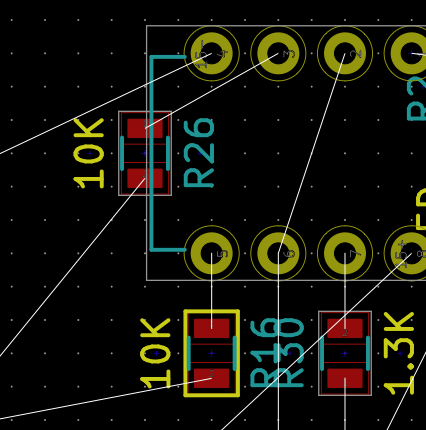
[](https://www.google.co.uk/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&uact=8&ved=2ahUKEwjSqtfQ-NPdAhVIWBoKHcJ_DAsQjRx6BAgBEAU&url=https://en.wikipedia.org/wiki/File:Valley-fill_circuit_schematic_1.png&psig=AOvVaw1u6-1MKZyxtGEkY9-Jxwfn&ust=1537888868345021)To solve this problem PCB Routing Software 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)

Then another mode where the “nets”[[1]](#footnote-2) highlighted, these are the connections between components (in figure 2 they are blue) and then can be routed by the user (or by software). Finally it’s converted into a file that can be printed.

In this project I will solely 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

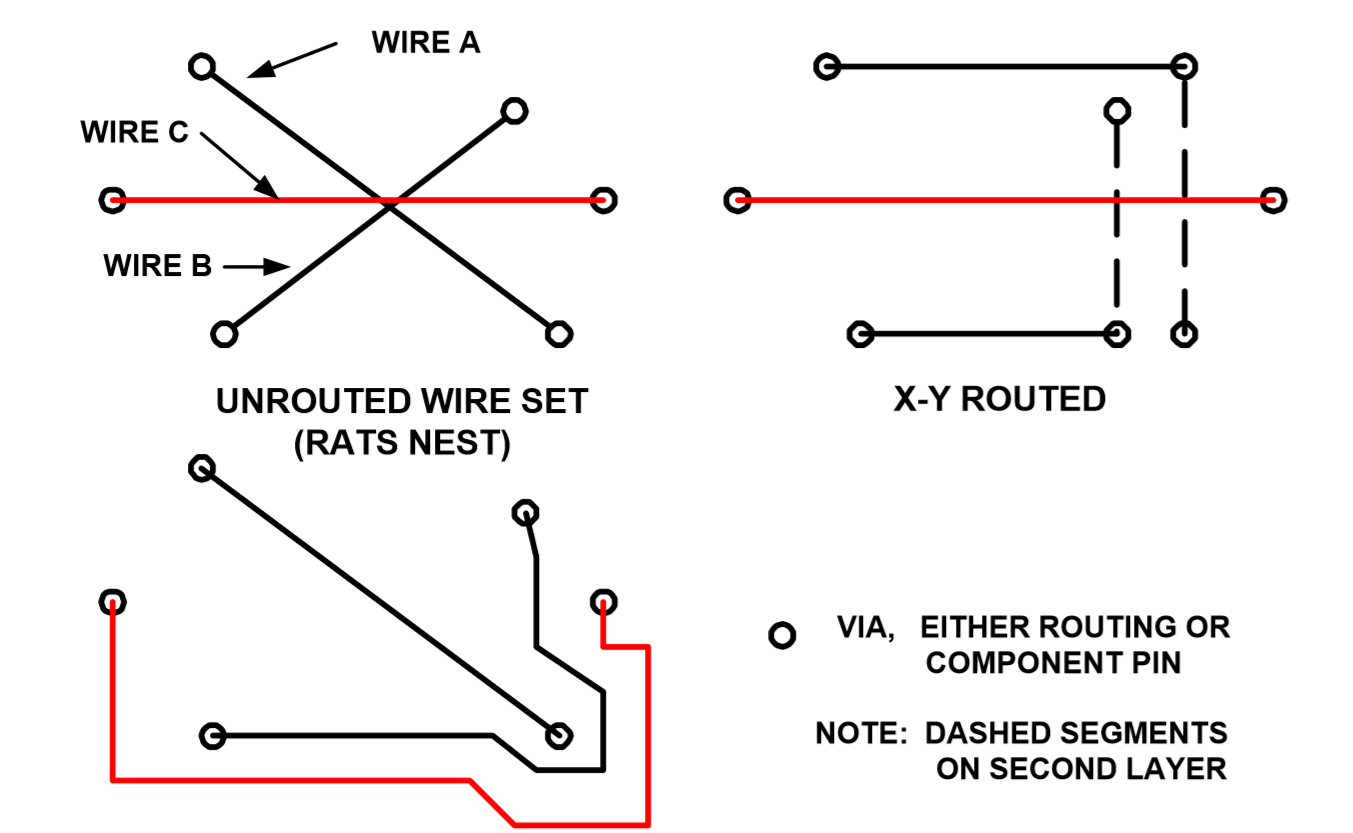
There are two main types of routing X-Y routing and Maze routing (LEE W. RITCHEY, 1999).

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 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.

### Users, Limitations and Requirements

Auto-routing is a tool to make electronic design easier for engineers to design a board. The user for the system, who I have reached out to, Stuart Jessop, should be able to use the final version to carry out the following tasks:

* Import a Net either by manually entering the coordinates of the pads or by parsing a file
* Set design constraints as detailed in the “interview”
* Route the board:
  + This will involve finding a route from two points
  + Moving on to the next two points
  + If routing impossible ripping up and retrying
* Export the routed file in a format that can be printed (SVG[[2]](#footnote-3))

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 a “.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.

## Research

### Interview

After speaking with my user there are a range of areas that my project could support and they were ranked according to value. The list below is ranked.

1. Different track widths
2. Keep out areas
3. Thermal sinks[[3]](#footnote-4)
4. Isometric routing[[4]](#footnote-5)
5. Integration with EDA (Electronic Design Application) software.

### Perquisites 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 before the requirements suggested by my user:

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 and for nets with multiple contacts. Auto-routers also prize themselves on their speed.

### Objectives

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.

The other requirements have been reworded and ordered as bellow:

1. User can input the positions of the points into the system and what routes they belong to.
2. Finding an efficient routes between two points
3. The user can create keep outs and floods
4. Program can export result in SVG format
5. Being able to find the efficient route between multiple points:
   1. Being able to run the router multiple times on the same board
6. Ensuring that the routes do not violate any the DRC requirements listed in order bellow:
   1. Shorts do not occur
   2. No tracks cross “keep out area’s”
   3. Areas of board which are specified as thermal sinks meet the minimum size requirements
7. A general requirement for all of the above steps is that the auto-router should be as fast as I can make it.

Extensions

1. These extend objective 5
   1. Being able to detect when the router has boxed it’s self in
   2. Being able to rip-up poorly routed tracks and reroute
2. Multi-threading for better performance
3. Support for nets with multiple contacts
4. Isometric Routing
5. Integration with EDA[[5]](#footnote-6) software, KiCAD[[6]](#footnote-7)

## Prototyping

### Objectives

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 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[[7]](#footnote-8)). 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

## Program overview

### A Star

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. At the core of this implementation there is a Heap, which stores all of the nodes by their value where and node.h is a heuristic value in this case determined by the Euclidian distance between the start and end node and node.g is the shortest known distance to that node.

The algorithm pops values from the heap and adds their neighbours to the heap until it runs out of nodes or reaches the end point.

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. }

The above function does not use recursion like many implementations of A\* would as tail recursion is not supported in JS, this meant that I had to design the algorithm differently to the “textbook” approach.

### 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. 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

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 blue net would be removed by not the green net.

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

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. Array = []
28. netIDsToRipup.reduce(item => Array.append(item))
29. netIDsToRipup = Array
30. //remove the current net from the array
31. index = netIDsToRipup.indexOf(net)
32. **if** (index > -1) {
33. netIDsToRipup.remove(index)
34. }
36. **return** netIDsToRipup
38. }

In the above function I use a set to ensure that I do not add duplicate entries to the array. I then iterate through from the smaller of the two numbers to the larger adding anything to the list of things to be removed that cause a problem.

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

It is not just the net that needs to be ripped up, the area around the net also needs to be made routable but only if that area is not marked as unrouteable by any other net.

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.

### 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 via and AJAX connection to the server where the routing and construction of the DOM takes place. Here is the basic outline of how this works:

<http://www.plantuml.com/plantuml/png/TO-nJWCn44Jx-ugfg8Ze8uGYWQ12I4JGE-S9jkIpm-wQaB_7XaDS4SgHvis-tQWviJgbuJP5PiFg1blA1mLh2DyhrH1iIfTGEoKgNFRGKUB1z9UvNYqPFPHMEKOBFPGg8tK8kHXHUcU6rdZHbc8sYXjDVuXTSUBFkUnfOy2Lb6fSLA7cZ6sbHNzQdFaZxsFJwTmilwinUMnV7yxfFo_rCu-zXtsH-Te5ADBIH5NtzjzV1z-TB11tJuzmVSv5-vwfIb5aVXhQqx7aOSFiwvI-0000> 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( 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. }

The state of the client side board is represented using a series of objects which are then converted into a JSON string for transmission in the body of the request. 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. }

## D:\Documents\My Documents (Edwin)\Downloads\TP1FQkim48VtSufP5v83X493KcZpIco8Um5X_R155Dd8esnCxbxBIYnQydOpdpxDdzaEeho4YwLUrMVLW2luApoFL42uMwru3sjfrocl_wOlKpfPqJnrGU3_GsOx_Sbqv4NAJxy-v2AnxoTKpvk3gstG-6xDK3gy4GtNfcfL3twTd1DgKQwnYFYeplTo6IzHMgM.emfUML Digramoverview

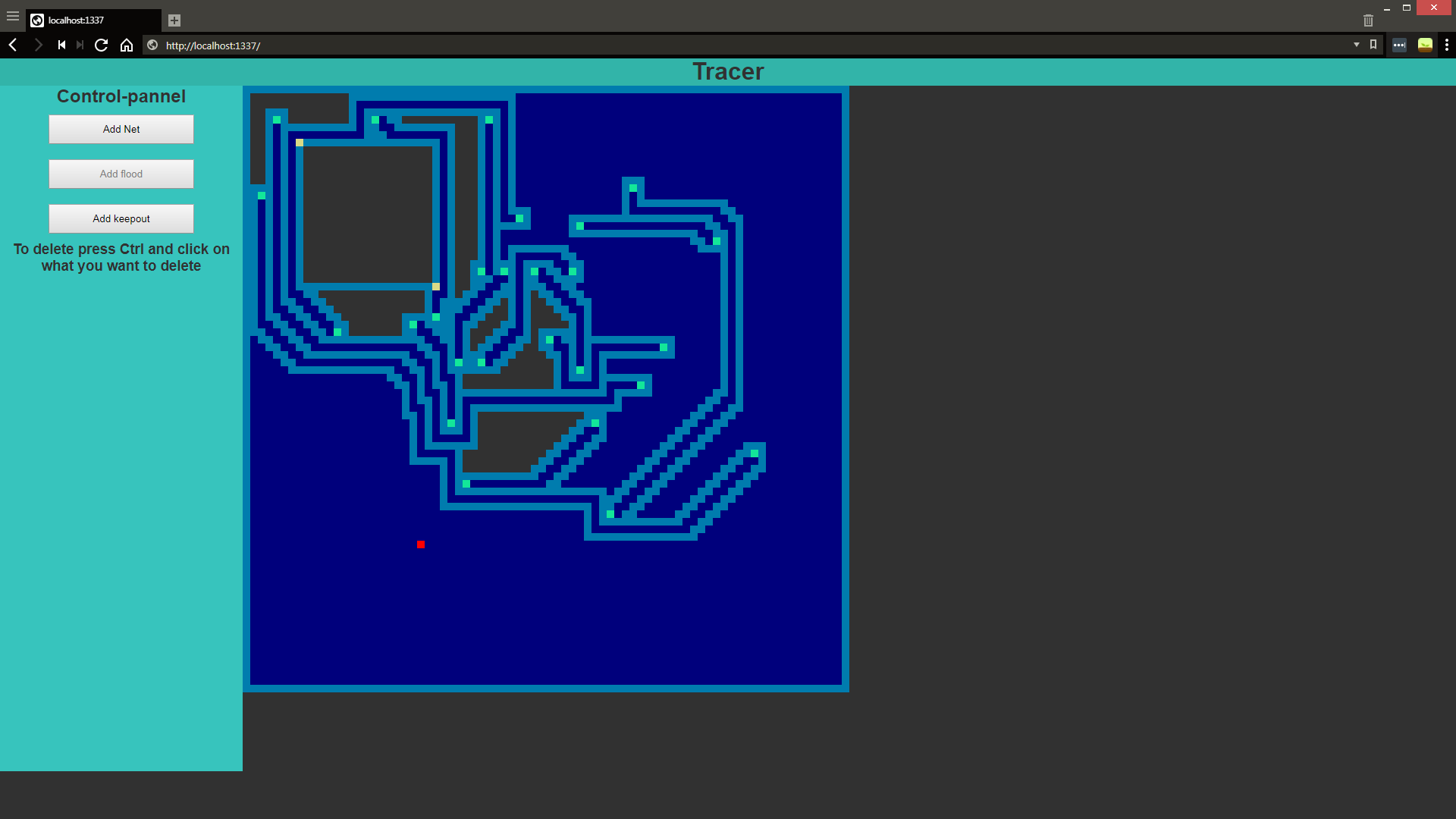
#### Board

The board is represented by

## Data structures

|  |  |  |
| --- | --- | --- |
| 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 and |
|  |  |  |

## Screen Shots



Sources for route finding:

Need to look at heuristics and preprocessing alogrithims

<http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/#1>"

<https://qiao.github.io/PathFinding.js/visual/>

<http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>

<https://www.microsoft.com/en-us/research/wp-content/uploads/2004/07/tr-2004-24.pdf>

<http://aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/>

# Technical solution

## 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. //When running in debug the path is different
14. **const** path = "."
15. **const** indexLocation = "/index.html"
17. //Make this come from client side
18. **const** boardWidth = 140;
19. **const** boardHeight = 80;
21. //!Use of a dictionary
22. **const** fileTypes = {
23. '.html' : 'text/html',
24. '.css' : 'text/css',
25. '.js' : 'text/javascript'
26. };
28. /\*\*
29. \* Converts a JSON netlist into an SVG represenation of the routed board.
30. \* @param {JSON} JSONData
31. \* @param {Number} cellSize
32. \*/
33. **var** routeJSON = **function**(JSONData, cellSize) {
35. let errors = []
37. JSONnetList = JSONData.netList;
38. JSONFloodList = JSONData.floodList;
39. JSONKeepoutList = JSONData.keepoutList;
41. **const** trackWidth = cellSize;
43. /\*\*This is where we take the JSON inputed into the system and turn it into objects
44. \* that we can manipulate
45. \*
46. \* If all goes to plan then this should be an array but we don't know
47. \* Maybe someone evil is using the API
48. \*/
49. **try** {
51. **var** netList = []
52. let length = JSONnetList.length;
54. /\*\*Creates each of the net objects from the specification in the
55. \* recived JSON
56. \*/
57. **for** (i = 0; i < length; i++) {
58. let start = **new** b.Cell( JSONnetList[i].start.x,
59. JSONnetList[i].start.y);
61. let end = **new** b.Cell(   JSONnetList[i].end.x,
62. JSONnetList[i].end.y);
64. netList.push(**new** b.Net(start, end));
65. }
67. } **catch** (err) {
68. **throw** err
69. }
71. board = **new** b.Board(boardWidth,boardHeight);
73. BR = **new** br.BoardRouter(board, netList);
75. //? Move this to clientside and to depend of the screen res&size
77. //Create the boarder of the board
78. let topLeft = **new** b.Cell(0,0);
79. let bottomRight = **new** b.Cell(boardWidth-1,boardHeight-1);
80. BR.createKeepOut(topLeft,bottomRight);
82. //Create any user specified keepouts
83. JSONKeepoutList.forEach( keepout => {
84. let topLeft = **new** b.Cell(keepout.start.x,keepout.start.y);
85. let bottomRight = **new** b.Cell(keepout.end.x,keepout.end.y);
87. BR.createKeepOut(topLeft,bottomRight,"keepout",1);
88. });
90. **try** {
91. let floodCellJSON = JSONFloodList[0];
92. **var** floodCell = board.getCell(floodCellJSON.x,floodCellJSON.y)
93. board.markNeighboursAsUnrouteable(floodCell, "Flood", 1);
94. } **catch**(err) {
95. **if** (err.name == "TypeError") {
96. //This is to be expected if we have nothing to route
97. } **else** {
98. **throw** err
99. }
100. }
102. //Find a route for all of the specified objects
103. let route = BR.route();
105. errors = route.errors;
106. let tracks = route.tracks;
108. //Apply the flood now we know where the nets are going.
109. **try** {
110. BR.flood(floodCell);
111. } **catch**(err) {
112. **if** (err.name == "TypeError") {
113. //This is to be expected if we have nothing to route
114. } **else** {
115. **throw** err
116. }
117. }
119. **return** {
120. DOM: BuildDOM(tracks, trackWidth),
121. errors: errors
122. };
124. }

127. **var** BuildDOM = **function**(tracks, trackWidth) {
128. //We now need to actually build the output, using the SVGmaker class
129. **var** SvgMaker = **new** svg.Maker;
131. //Shows the area on the board which have been marked as not routeable
132. **for** (**var** x = 0; x < board.width; x++) {
133. **for** (**var** y = 0; y < board.height; y++) {
134. **if** (!board.getCell(x,y).routeable) {
135. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
137. **if** (board.getCell(x,y).controllingNet.map((i) => {
138. **return** i.routingOverrideLevel;
139. }).reduce((a,c) => {
140. **return** a + c;
141. },0)) {
142. Rect.fillColour = **new** colour.Colour(124,0,0);
143. } **else** {
144. Rect.fillColour = **new** colour.Colour(0,124,174);
145. }
146. SvgMaker.addElement(Rect);
147. }
148. }
149. }
151. /\*\*If the flood list is empty then all of the tracked cells will be along the routes and so
152. \* we can speed up our generation times by only going along the nets.
153. \*
154. \* Otherwise we need to scan the whole board
155. \*/
156. **if** (JSONFloodList.length > 0) {
157. **for** (let x = 0; x < board.width; x++) {
158. **for** (let y = 0; y < board.height; y++) {
159. **if** (board.getCell(x,y).tracked) {
160. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
161. Rect.fillColour = **new** colour.Colour(0,0,124);
162. SvgMaker.addElement(Rect);
163. }
164. }
165. }
166. } **else** {
167. //The default where we can effiecently draw along the tracks rather than scan the whole
168. //Board
169. **for** (**var** track = 0; track < tracks.length; track++) {
170. **for** (**var** cell = 0; cell < tracks[track].length; cell++) {
171. let x = tracks[track][cell].x;
172. let y = tracks[track][cell].y;
174. let Rect = **new** svg.Rectangle(x\*trackWidth,y\*trackWidth,trackWidth,trackWidth);
175. Rect.fillColour = **new** colour.Colour(0,0,124)
176. SvgMaker.addElement(Rect);
177. }
178. }
179. }
181. **return** SvgMaker.getImage();
182. }
184. /\*\*
185. \* ===========SERVER===========
186. \*/

189. /\*\*
190. \* This serves all the files to the client
191. \* @type {module:http.Server}
192. \*/
193. let server = http.createServer(**function** (req, res) {
195. **var** parsedURL = url.parse(req.url, **true**);
197. //This switch allows assignment specific urls to files.
198. **switch** (parsedURL.pathname) {
199. //The entry point
200. **case** ('/'):
202. fs.readFile(path + indexLocation, **function** read(err, indexFile) {
203. **if** (err) **throw** err;
204. res.writeHead(200, {"Content-Type": "text/html"});
205. res.write(indexFile);
206. res.end();
207. });
209. **break**;
211. //This handels requests for a routed version of the board
212. **case** ('/route'):
214. //NetLists etc passed to the server inside the body of the request
215. let requestBody = **new** String;
216. req.on('data', chunk => {
217. requestBody += chunk.toString();
218. });
220. req.on('end', () => {
221. //The request has ended and so we have all of the data
222. // lets give them their new route
224. let route = routeJSON(JSON.parse(requestBody), parseInt(parsedURL.query.cellSize));
225. let svg = route.DOM;
226. let errors = route.errors;
228. responseContent = JSON.stringify({
229. board: svg,
230. errors: errors
231. })
233. res.end(responseContent);
234. });
236. **break**;
238. //If they are requesting styles or other scripts
239. **default**:
240. fs.readFile('./' + path + req.url, **function**(err, data) {
241. **if** (err) {
242. res.writeHead(404, "Not Found");
243. res.write('THERE WAS A 404')
244. res.end();
245. } **else** { //There was no error
247. /\*\*
248. \* We now need to automatically serve the file however
249. \* we don't know what mimetype (e.g. is it a text/html)
250. \* this gets the file extentsion and assumes that is the
251. \* value of the sign we want. We can do this by finding the
252. \* last dot in the file name, then looking that extention
253. \* up in a dictionary.
254. \*/
255. let dotPosFromEnd = req.url.lastIndexOf('.');
256. let mimetype = 'text/plain';
258. **if** (!(dotPosFromEnd == -1)) {
259. //Perform a look up for the text using the suffix in a dictionary
260. mimetype = fileTypes[req.url.substr(dotPosFromEnd)];
261. }
263. res.setHeader('Content-type' , mimetype);
264. res.end(data);
265. }
266. });
267. }
268. }).listen(port);
270. console.log('Sever listening on port ', port);

### BoardRouter.js

1. **const** nr = require('./netRouter');
2. **const** b = require('./board');
4. **const** errorThreshold = 2;
6. /\*\*
7. \* The class for routing a whole board
8. \* @param {Board} board
9. \* @param {Array<Board.Net>} netList
10. \*/
11. BoardRouter = **function** (board, netList) {
12. **this**.board = board;
13. **this**.netList = netList;
14. }
16. /\*\*
17. \* Creates a non routeable box
18. \* @param {Cell} cell1 The cell which defines one coner
19. \* @param {Cell} cell2 The cell which defines one coner
20. \* @param {Boolean} borderOnly Do we allow routing with in the area
21. \*/
22. BoardRouter.prototype.createKeepOut = **function**(cell1, cell2, borderOnly = **true**) {
24. **if** (**this**.board.CordsOnBoard(cell1.x, cell1.y)
25. && **this**.board.CordsOnBoard(cell2.x, cell2.y)) {
26. //find the smaller of the cords so that we can itterate using a for loop
27. **if** (cell1.x > cell2.x) {
28. **var** bigX = cell1.x;
29. **var** smallX = cell2.x;
30. } **else** {
31. **var** bigX = cell2.x;
32. **var** smallX = cell1.x;
33. }
35. **if** (cell1.y > cell2.y) {
36. **var** bigY = cell1.y;
37. **var** smallY = cell2.y;
38. } **else** {
39. **var** bigY = cell2.y;
40. **var** smallY = cell1.y;
41. }
43. **if** (borderOnly) {
44. //Mark the horizontal walls as unrouteable
45. **for** (let x = smallX + 1; x < bigX; x++) {
46. **this**.board.markCordsAsUnrouteable(x,smallY)
47. **this**.board.markCordsAsUnrouteable(x,bigY)
48. }
49. //Mark the verticle walls as unrouteable
50. **for** (let y = smallY; y < bigY + 1; y++) {
51. **this**.board.markCordsAsUnrouteable(smallX,y)
52. **this**.board.markCordsAsUnrouteable(bigX,y)
53. }
54. } **else** {
55. //Mark the whole areas as unrouteable
56. **for** (let x = smallX; x < bigX; x++) {
57. **for** (let y = smallY; y < bigY; y++) {
58. **this**.board.markCordsAsUnrouteable(x,y);
59. }
60. }
61. }
62. }
63. }
65. /\*\*
66. \* Floods all of the cells possible from a cell
67. \* This could have been made recursively in theory however
68. \* JS doesn't support tail optimaisation and so it would have caused
69. \* StackOver Flows left right and center
70. \* @param {Cell} Cell Where we start the flood from
71. \* @returns {<Cell>} Returns list of cells in the flood
72. \*/
73. BoardRouter.prototype.flood = **function**(Cell) {

76. //Must check this as we use weather cell is defined in the while loop later
77. //!EXAMPLE OF ERROR HANDELING
78. **if** (**typeof**(Cell) == undefined) {
79. **throw** Error ("Flood method passed undefined cell");
80. }
82. **this**.board.markNeighboursAsRouteable(Cell, "Flood", 1)
84. unchecked = [];
85. flood = [];
86. current = Cell;
88. //Check if we can still pop items from the stack
89. **while** (current) {
90. **if** (current.routeable) {
91. //Gets the routeable neighbours and pushes them to the unchecked list
93. **this**.board.getValidNeighbours(current).forEach(cell => {
94. unchecked.push(cell)
95. });
97. current.routeable = **false**;
98. current.tracked = **true**;
100. flood.push(current);
101. }
103. current = unchecked.pop();
104. }
106. **return** flood;
107. }
109. /\*\*
110. \* A function decide which order the nets are to be routed in
111. \*/
112. BoardRouter.prototype.route = **function**() {
113. let errors = [];
115. **const** hurestristicWeight = 1.5; //DO NOT TOUCH
117. let tracks = **new** Array;
119. //If i have to go through the list any ways then this is linner time and I could
120. //Implement my own sorting algorithium
121. **this**.netList.sort((cellA, cellB) =>
122. cellA.manhattanLength() - cellB.manhattanLength()
123. );
125. /\*\*
126. \* PRE PROCESSING
127. \*/
129. **try** {
131. **this**.netList = **this**.netList.filter((net) => {
132. **if** (!**this**.board.CordsOnBoard(net.start.x, net.start.y)
133. || !**this**.board.CordsOnBoard(net.end.x, net.end.y)) {
134. errors.push('Not all nets on the board');
135. **return** **false**
136. } **else** {
137. **return** **true**
138. }
139. })
141. **this**.netList.forEach((net, i) => {
142. net.routingErrors = 0;
143. net.id = i;
144. **this**.board.markNeighboursAsUnrouteable(net.startCell,**true**,net.id,1,)
145. **this**.board.markNeighboursAsUnrouteable(net.endCell,**true**,net.id,1)
146. });

149. } **catch** (err) {
150. **if** (err.name == "TypeError") {
151. //\*I am sliencing this error, maybe raise it to the user later
152. } **else** {
153. //This \*should\* never happen but if it does we want to know what happened
154. **throw** err;
155. }
156. }
158. let toRoute = **this**.netList.slice();
160. //for (let c = 0; c < 3; c++) {
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. **if** (currentNet.routingErrors < errorThreshold) {
177. let getNetIDsToClear = **function**(net) {
178. let netIDsToRipup = **new** Set;
180. start = net.startCell;
181. end = net.endCell;
183. bigX = Math.max(start.x, end.x);
184. smallX = Math.min(start.x, end.x);
186. bigY = Math.max(start.y, end.y);
187. smallY = Math.min(start.y, end.y);
189. **for** (let x = smallX; x < bigX; x++) {
190. let netIDs = **this**.board.getCell(x,start.y).controllingNet;
191. netIDs.forEach(controllingNet => netIDsToRipup.add(controllingNet.controllingNetID));
192. }
194. **for** (let y = smallY; y < bigY; y++) {
195. let netIDs = **this**.board.getCell(y,end.x).controllingNet;
196. netIDs.forEach(controllingNet => netIDsToRipup.add(controllingNet.controllingNetID));
197. }
199. //Convert the set to an array
200. netIDsToRipup = [...netIDsToRipup];
202. //remove the current net from the array
203. **var** index = netIDsToRipup.indexOf(net);
204. **if** (index > -1) {
205. netIDsToRipup.splice(index, 1);
206. }
208. **return** netIDsToRipup
210. }

213. //Make the net that just failed the first net to try again
214. let toReRoute = [currentNet];
216. //Converts the net ID's to actual net objects and filter out the undefineds
217. //caused when netList is empty
218. //!Example of Functional programing
219. let netIDsToClear = getNetIDsToClear(currentNet);
221. let netsToClear = netIDsToClear.map((ID) => {
222. let net = **this**.netList.find((net) => {
223. let **boolean** = (net.id == ID)
224. **return** **boolean**
225. });
226. **return** net
227. }, **this**).filter((element) => { element != undefined })
229. netIDsToClear.forEach((ID) => {
230. **this**.netList.splice(
231. **this**.netList.findIndex((net) => { **return** net.id == ID })
232. ,1);
233. })
235. //RipUp the nets
236. netsToClear.forEach(net => {
237. toReRoute.push(net)
239. **var** currentTrace = net.trace;
241. //This could get slow with alot of nets
242. //Delete the trace from the trace array
243. tracks.splice(tracks.findIndex((value) => {**return** (currentTrace == value)}),1);
245. currentTraceLength = currentTrace.length;
246. **for**(traceCellIndex = 0; traceCellIndex < currentTraceLength; traceCellIndex++) {
248. **this**.board.markNeighboursAsRouteable(currentTrace[traceCellIndex],
249. **true**,
250. net.id,
251. 0);
252. //Remove the tracks from the board
253. **this**.board.markCellAsUntracked(currentTrace[traceCellIndex]);
254. }
256. net.trace = [];
257. })
259. //Prepend the nets that need to be ReRouted to the front of the array
260. toRoute.unshift(...toReRoute);
261. }  **else** {
262. //Compile the error message for the user
263. errors.push("Routing failed on: {start: (" + currentNet.startCell.x + ',' + currentNet.startCell.y +
264. '), end: ('+ currentNet.endCell.x + ',' + currentNet.endCell.y + ')}')
265. }
266. }
267. }
269. **return** {
270. tracks: tracks,
271. errors: errors
272. };
273. }

276. module.exports = {BoardRouter};

### NetRouter.js

1. **var** BoardObject = require('./board');
2. //var ADT = require('./test');
3. **var** Heap = require('../Modules/Heap');
5. NetRouter = **class** {
6. /\*\*
7. \* A class for performing the routing of a net. A board of where the net can go is passed to it
8. \* and it will return the path the net will be routed along.
9. \* @param {Array<Array<Cell>>} board A board conataining all the tracks that aren't routed.
10. \* @param {Net} net The net to be routed
11. \* @param {Number} heuristicWeight the weight of the heuristic
12. \*/
13. constructor(board, net, heuristicWeight, ID) {
14. **this**.board = board;
15. **this**.net = net;
16. **this**.heuristicWeight = heuristicWeight;
17. **this**.ID = ID;
19. /\*\* Create a new heap which sorts the entities by the difference in their f values,
20. \*  where f = g + h
21. \*  g - Shortest known path from start
22. \*  h - Manhattan distance
23. \*/
24. **this**.toCheck = **new** Heap(**function**(cellA, cellB) {
25. **return** cellA.f - cellB.f;
26. })
28. //Add the starting point of the route to our list toCheck nodes.
29. **this**.startCell = **this**.net.startCell;
30. **this**.endCell = **this**.net.endCell;
31. }
33. /\*\*
34. \* The function that performs the path finding for a single net
35. \*/
36. route() {
37. let B = **this**.board;
39. **this**.startCell.g = 0;
40. **this**.startCell.f = 0;
42. **this**.startCell.checked = **true**;
44. B.markNeighboursAsRouteable(**this**.startCell,**true**,**this**.ID,1);
45. B.markNeighboursAsRouteable(**this**.endCell,**true**,**this**.ID,1);
47. **this**.toCheck.push(**this**.net.startCell);
49. //While there are still possible cells that there could be a route for.
50. **while** (!**this**.toCheck.empty()) {
52. let cell = **this**.toCheck.pop();
54. cell.checked = **true**;
56. //Have we found the endCell yet?
57. **if** (cell.x == **this**.endCell.x && **this**.endCell.y == cell.y) {
59. let current = cell;
61. B.markNeighboursAsUnrouteable(current, **true**, **this**.ID, 1);
63. **this**.net.trace.push(current);
65. /\*\* Starting from the endCell perform a trace back to
66. \*  the start, adding the cells that are on the trace to a list and marking
67. \*  all of the neighbours as unrouteable
68. \*/
69. **do** {
70. B.markNeighboursAsUnrouteable(current, **true**, **this**.ID, 0);
72. current = current.**super**;
74. current.tracked = **true**;
75. **this**.net.trace.push(current);
77. } **while** (current != **this**.startCell)
79. B.markNeighboursAsUnrouteable(**this**.startCell, **true**, **this**.ID, 1);
81. let n = B.getNeighbours(**this**.startCell);
83. **this**.cleanUp();
85. **return** **this**.net.trace
87. }
89. /\*\*We've not found the end yet.
90. \* Lets work out work out which cell to expand into next
91. \*/
92. let neighbours = B.getValidNeighbours(cell);
94. let neightbourLength = neighbours.length;
95. **for** (let i = 0; i < neightbourLength; i++) {
97. let neighbour = neighbours[i];
98. let ng = cell.g + 1; //neighbough g -> ng
100. //Is there a route to that cell that we don't know abouts
101. **if** (ng < neighbour.g || !neighbour.checked) {
102. neighbour.**super** = cell;
104. neighbour.g = ng;
105. neighbour.h = **this**.heuristicWeight \* B.getEuclidean(neighbour, **this**.endCell);
106. neighbour.f = neighbour.g + neighbour.h;
108. neighbour.checked = **true**;
110. **this**.toCheck.push(neighbour);
111. }
112. }
113. }
115. **this**.cleanUp()
117. **throw** Error("No route found");
119. }
121. cleanUp() {
122. **for** (let x = 0; x < **this**.board.width; x++) {
123. **for** (let y = 0; y < **this**.board.height; y++) {
124. let cell = **this**.board.getCell(x,y);
125. cell.checked = **false**;
126. cell.**super** = **null**;
127. }
128. }
129. }
131. reset() {
132. **this**.net.trace = **new** Array;
133. }

136. }
138. module.exports = {NetRouter};

### Board.js

1. //var ADT = require('./test')
3. //All all of the Cell class to be acessed from other files
5. **class** Cell {
6. /\*\*
7. \*
8. \* @param {Number} x
9. \* @param {Number} y
10. \* @param {Number} layer
11. \* @param {Boolean} routeable
12. \*/
13. constructor (x,y,layer=1,routeable=**true**) {
15. /\*\*
16. \* @type {number}
17. \*/
18. **this**.x = x;
20. /\*\*
21. \* @type {number}
22. \*/
23. **this**.y = y;
25. /\*\*
26. \* This can be deleted now
27. \* @type {number}
28. \*/
29. **this**.layer = layer;
31. /\*\*
32. \* @type {boolean}
33. \*/
34. **this**.routeable = routeable;
36. **this**.hardRouteable = **true**;
37. /\*\*
38. \* If the Cell has a track going through it.
39. \* Defaults to false because at start of day we have no tracks
40. \* @type {boolean}
41. \*/
42. **this**.tracked = **false**;
44. **this**.controllingNet = **new** Array;
46. **this**.hardControllingNetID = **new** Array;
47. }
48. }
50. //Alow all of the Net class to be acessed from other files
51. /\*\*
52. \* A net for a single track.
53. \* This class holds the information about a single net
54. \* @param {Cell} start The start of the net
55. \* @param {Cell} end   The end of the net
56. \*/
57. **function** Net (start, end) {
59. /\*\*
60. \* @type {Cell}
61. \*/
62. **this**.startCell = start;
64. /\*\*
65. \* @type {Cell}
66. \*/
67. **this**.endCell = end;
69. **this**.trace = **new** Array;
70. }
72. /\*\*
73. \* This maybe too similar to the get manhattan distance board method
74. \* and theere may be a way of subcalssing these to make more sense
75. \* !This is the Euclidian length need to refactor
76. \*/
77. Net.prototype.manhattanLength = **function**() {
78. **return** Math.sqrt(Math.pow(**this**.startCell.x-**this**.endCell.x, 2) + Math.pow(**this**.startCell.y-**this**.endCell.y, 2));
79. }


83. //Allow all of the board class to be accessed from other files
85. /\*\*
86. \* The Class that holds all of the infomation about the board
87. \* @constructor
88. \* @param {number} boardWidth The width of the board
89. \* @param {number} boardHeight The height of the board
90. \* @param {Array<Array<number|boolean>>} routeMask An boolean grid showing all the places that the route can not go
91. \*/
92. **var** Board = **function**(boardWidth, boardHeight, routeMask=[[]]) {
94. /\*\*
95. \* @type {number}
96. \*/
97. **this**.width = boardWidth;
99. /\*\*
100. \* @type {number}
101. \*/
102. **this**.height = boardHeight;
104. /\*\*Create a matrix for the grid to be stored in then go through and
105. \* populate it with cells
106. \*/
108. **this**.grid = **new** Array(**this**.height);
110. **for** (let y = 0; y < **this**.height; y++) {
111. **this**.grid[y] = **new** Array(**this**.width);
113. **for** (let x = 0; x < **this**.width; x++) {
114. **this**.grid[y][x] = **new** Cell(x, y);
115. }
117. }
119. /\*\*
120. \* Check if each of the cells are routeable according to the inputted routing guide.
121. \*/
123. **if** ((routeMask != undefined)
124. && (routeMask.length > **this**.height)
125. && (routeMask[0].length > **this**.width)) {
127. //NOTE TO SELF - Possibly look at merging this and the for loop above together
129. **for** (let y = 0; y < routeMask.length; y++) {
130. **for** (let x = 0; x < routeMask[y].length; x++) {
131. **if** (routeMask[x][y] = 1) grid[y][x].routeable = **false**;
132. }
133. }
134. }
136. };

139. /\*\*
140. \* Checks if a cell is on the board
141. \* @param {Cell} Cell
142. \*/
143. Board.prototype.CordsOnBoard = **function**(x ,y) {
144. **return** (x >= 0 && x < **this**.width)
145. && (y >= 0 && y < **this**.height);
146. }
148. /\*\*
149. \* Checks weather we can route a track through a given Cell
150. \* @param {Cell} Cell
151. \*/
153. Board.prototype.validCord = **function** (x,y) {
154. **return** **this**.grid[y][x].routeable && !**this**.grid[y][x].checked
155. }

158. Board.prototype.getNeighbours = **function**(Cell, diagonals) {
159. let neighbours = **new** Array;
161. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y)) {
162. neighbours.push(**this**.grid[Cell.y + 1][Cell.x]);
163. };
165. **if** (**this**.CordsOnBoard(Cell.x, Cell.y + 1)) {
166. neighbours.push(**this**.grid[Cell.y][Cell.x + 1]);
167. };
169. **if** (**this**.CordsOnBoard(Cell.x, Cell.y -1)) {
170. neighbours.push(**this**.grid[Cell.y - 1][Cell.x]);
171. };
173. **if** (**this**.CordsOnBoard(Cell.x - 1, Cell.y)) {
174. neighbours.push(**this**.grid[Cell.y][Cell.x - 1]);
175. };
177. **if** (diagonals) {
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. };
190. **if** (**this**.CordsOnBoard(Cell.x - 1, Cell.y - 1)) {
191. neighbours.push(**this**.grid[Cell.y - 1][Cell.x - 1]);
192. };
193. }
195. **return** neighbours
196. }
198. /\*\*
199. \* Returns a list of the Neighbours of a Cell, if the
200. \* cell is valid it will return the cell's neighbours object
201. \* @param {Cell} Cell
202. \*/
203. Board.prototype.getValidNeighbours = **function** (Cell){
205. let neighbours = **new** Array;
207. **if** (**this**.CordsOnBoard(Cell.x + 1, Cell.y)
208. && **this**.grid[Cell.y + 1][Cell.x].routeable) {
210. neighbours.push(**this**.grid[Cell.y + 1][Cell.x]);
211. };
213. **if** (**this**.CordsOnBoard(Cell.x, Cell.y + 1)
214. && **this**.grid[Cell.y][Cell.x + 1].routeable) {
216. neighbours.push(**this**.grid[Cell.y][Cell.x + 1]);
217. };
219. **if** (**this**.CordsOnBoard(Cell.x, Cell.y -1)
220. && **this**.grid[Cell.y - 1][Cell.x].routeable) {
222. neighbours.push(**this**.grid[Cell.y - 1][Cell.x]);
223. };
225. **if** (**this**.CordsOnBoard(Cell.x, Cell.y - 1)
226. && **this**.grid[Cell.y][Cell.x - 1].routeable) {
228. neighbours.push(**this**.grid[Cell.y][Cell.x - 1]);
229. };
231. **return** neighbours;
232. }
234. /\*\*
235. \* Gets the neighbours of a cell including diagonals
236. \* @param {Cell} Cell
237. \*/
238. Board.prototype.getCellAndAllNeighbours = **function**(Cell) {
239. let cells = **new** Array
241. **for** (let x = -1; x <= 1; x++) {
242. **for** (let y = -1; y <= 1; y++) {
244. **if** (**this**.CordsOnBoard(Cell.x + x,Cell.y + y)
245. && !(x == 0 && y == 0)) {
246. cells.push(**this**.grid[Cell.y + y][Cell.x + x]);
247. }
248. }
249. }
251. **return** cells;
252. }
254. /\*\*
255. \* Gets the manhattan distance between two cells
256. \* @param {BoardObject.Cell} cell1 Cells for distance to be found between
257. \* @param {BoardObject.Cell} cell2 Cells for distance to be found between
258. \*/
259. Board.prototype.getManhattan = **function**(cell1, cell2) {
260. //Pythagouses theorem to get the Manhattan distance
261. **return** (Math.abs(cell1.x-cell2.x) + Math.abs(cell1.y-cell2.y));
262. }
264. /\*\*
265. \*
266. \* @param {BoardObject.Cell} cell1
267. \* @param {BoardObject.Cell} cell2
268. \*/
269. Board.prototype.getEuclidean = **function**(cell1, cell2) {
270. **return** Math.sqrt(Math.pow(cell1.x-cell2.x, 2) + Math.pow(cell1.y-cell2.y, 2));
271. }
273. /\*\*
274. \* Finds all of the cells which are neighbours (that are also on the)
275. \* board and marks them and all of their neighours as not routeable.
276. \* It then specifies the reason why they are not routeable and the level
277. \* of overide required to make it routebale
278. \*/
279. Board.prototype.markNeighboursAsUnrouteable = **function**(Cell, diagonals=**false**, NetID=**null**, overide=0) {
280. **this**.getNeighbours(Cell, diagonals).forEach((neighbour) => {
281. **this**.markCellAsUnrouteable(neighbour,NetID,overide);
282. });
283. }
285. /\*\*
286. \* @param {\*} Cell
287. \* @param {\*} diagonals
288. \* @param {\*} ID
289. \*/
290. Board.prototype.markNeighboursAsRouteable = **function**(Cell, diagonals=**false**, ID, overide=0) {
291. **this**.getNeighbours(Cell, diagonals).forEach(neighbour => {
292. **this**.markCellAsRouteable(neighbour,ID,overide)
293. },**this**);
294. };
296. Board.prototype.markCellAsRouteable = **function**(cell, ID=**null**, overide) {
297. /\*\*Need to check weather this is the only net that
298. \* Controlls this cell, if not then we don't want to
299. \* remove it from this nets trace
300. \*
301. \* If it is the only one we remove it from the controlling array
302. \*
303. \* If the controlling array is then empty then we the demark it
304. \*/
305. **for** (let i = 0; i < cell.controllingNet.length; i++) {
306. **if** (cell.controllingNet[i].controllingNetID == ID) {
307. **if** (overide >= cell.controllingNet[i].overide) {
308. cell.controllingNet.splice(i, 1);
310. **if** (cell.controllingNet.length == 0) {
311. cell.routeable = **true**;
312. };
313. };
315. **break**; //No point to continue itteration
316. }
317. }
318. }
320. Board.prototype.markCellAsUnrouteable = **function**(cell, ID=**null**, overide=0) {
321. //Convert to set
322. **for** (let i = 0; i < cell.controllingNet.length; i++) {
323. **if** (cell.controllingNet[i].controllingNetID == ID)   {
324. **if** (cell.controllingNet[i].controllingNetID < overide) {
325. cell.controllingNet[i].controllingNetID = overide;
326. cell.routeable = **false**;
327. }
328. **return**;
329. }
330. }
332. //Not in list
333. cell.controllingNet.push({
334. controllingNetID: ID,
335. overide: overide
336. })
337. cell.routeable = **false**;
338. }
340. //!This should be removed but needs to have uses removed first
341. Board.prototype.markCordsAsUnrouteable = **function**(x,y, ID=**null**, overide=0) {
342. **this**.markCellAsUnrouteable(**this**.getCell(x,y),ID, overide);
343. }

346. Board.prototype.markCordsAsTracked = **function**(x,y) {
347. **this**.grid[y][x].tracked = **true**;
348. }
350. Board.prototype.markCellAsUntracked = **function**(cell) {
351. cell.tracked = **false**;
352. }
354. Board.prototype.getCell = **function**(x,y) {
355. **try** {
356. **return** **this**.grid[y][x];
357. } **catch** (err) {
358. **if** (err.message == "TypeError") {
359. **throw** **new** err ("Cords not on Board")
360. } **else** {
361. **throw** err
362. }
363. }
364. }

367. 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"**>**
32. <!--Work out why the cells need a container/padder-->
33. **</div>**
35. **<div** id="flood-container"**></div>**
37. **<div** id="keepout-container"**></div>**
39. **</div>**
41. **<script** type="text/javascript" src="indexScript.js"**></script>**
43. **</body>**
44. **</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. //Does cellsize work up here?
3. **function** getRoundedMouseX (event, roundToNearest = cellSize) {
4. **return** Math.ceil(event.clientX / roundToNearest) \* roundToNearest;
5. }
7. **function** getRoundedMouseY (event, roundToNearest = cellSize) {
8. **return** Math.ceil(event.clientY / roundToNearest) \* roundToNearest;
9. }
11. //Make this just a data strucuture
12. **class** clientSideCell {
13. constructor(x,y){
14. //Initalize parameters
15. **this**.x = x;
16. **this**.y = y;
17. }
19. buildDOM (elementID, ...classes) {
20. **var** cellPadder = document.createElement("div");
21. cellPadder.className = "PointPadder";
22. cellPadder.id = elementID + "Padding";
24. classes.forEach(DOMclass => {
25. cellPadder.classList.add(DOMclass);
26. });
28. **var** cell = document.createElement("div");
29. cell.className = "Point";
30. cell.id = elementID;
32. cell.appendChild(cellPadder);
34. **this**.el = cell;
35. **this**.elementID = elementID;
37. **return** cell;
38. }
39. }
41. //Make this just a data strucuture
42. **class** clientSideNet {
43. constructor(start, end, id) {
44. **this**.start = start;
45. **this**.end = end;
46. **this**.id = id;
47. }
48. }
50. //Maybe make an abstract class for these two
51. **class** clientsideKeepout {
52. constructor(start, end, id) {
53. **this**.start = start;
54. **this**.end = end;
55. }
56. }
58. //The names of the local varibles in this class are messy, this.grid describes the board etc.
59. **class** Grid {
60. /\*\*
61. @pram {Number} width the width cell units
62. @pram {Number} width the height in cell units
63. @pram {Number} cellSize the size in cell units
64. \*/
65. constructor (gridID, endPointContainerID, floodContainerID ,keepoutContainerID, width, height, cellSize) {
66. **this**.grid = document.getElementById(gridID);
67. **this**.endPointContainer = document.getElementById(endPointContainerID);
68. **this**.floodContainer = document.getElementById(floodContainerID);
69. **this**.keepoutContainer = document.getElementById(keepoutContainerID);
70. **this**.width = width;
71. **this**.height = height;
72. **this**.cellSize = cellSize;
73. **this**.netList = [];
74. **this**.floodList = [];
75. **this**.keepoutList = [];
76. **this**.cellCounter = 0;
77. **this**.svg = **null**;
78. }
80. makeInteractable(cell, deleteCallBack) {
81. **var** board = **this**;
82. **var** startx = cell.x \* **this**.cellSize;
83. **var** starty = cell.y \* **this**.cellSize;
84. **var** newXPos = 0;
85. **var** newYPos = 0;
86. **var** oldXPos = 0;
87. **var** oldYPos = 0;
88. **var** elmnt = cell.el; //el is standard shorthand for element in web frameoworks
89. **var** mouseDownPos = {
90. cellx: cell.x,
91. celly: cell.y,
92. mousex: startx,
93. mousey: starty
94. }

97. //Overide default method
98. document.getElementById(cell.elementID + "Padding").onmousedown = dragMouseDown ;
100. //Offset by the correct amount given css grid layout
101. elmnt.style.top = (starty)
102. + board.grid.getBoundingClientRect().top
103. + "px";
105. elmnt.style.left = (startx)
106. + board.grid.getBoundingClientRect().left
107. + "px";
109. **function** dragMouseDown(e) {
110. e.preventDefault();   //Prevents the default method from running

113. // get the mouse cursor position at startup:
114. oldXPos = getRoundedMouseX(e);
115. oldYPos = getRoundedMouseY(e);
117. mouseDownPos.mousex = elmnt.offsetLeft;
118. mouseDownPos.mousey = elmnt.offsetTop;
119. mouseDownPos.cellx = cell.x;
120. mouseDownPos.celly = cell.y;
122. /\*\*
123. \* If the control key is down then the user wants to delete this item
124. \*/
125. **if** (e.ctrlKey) {
126. deleteCallBack()
127. board.update();
128. } **else** {
129. document.onmouseup = stopDragging;  //Override the onmouseupmethod
130. document.onmousemove = elementDrag;     //Assign the on mousemove elementDrag method to elementDrag
131. }
132. }
134. **function** elementDrag(e) {
135. e.preventDefault();
137. // calculate the new cursor position:
138. newXPos = oldXPos - getRoundedMouseX(e);
139. newYPos = oldYPos - getRoundedMouseY(e);
141. // set the element's new position:
142. elmnt.style.top = (elmnt.offsetTop - newYPos) + "px";
143. elmnt.style.left = (elmnt.offsetLeft - newXPos) + "px";
145. //Update the old positions
146. oldXPos = getRoundedMouseX(e);
147. oldYPos = getRoundedMouseY(e);
148. }

151. **function** stopDragging() {
152. /\* stop moving when mouse button is released:\*/
153. document.onmouseup = **null**;
154. document.onmousemove = **null**;
156. //find new the cells position in grid terms.
157. let newCellX = Math.ceil((elmnt.offsetLeft - board.grid.getBoundingClientRect().left)/board.cellSize);
158. let newCellY = Math.ceil((elmnt.offsetTop - board.grid.getBoundingClientRect().top)/board.cellSize);
160. /\*\*Test if the new position for the cell is where any other cells are, or
161. \* if it is adjacent to any other cells
162. \* !NB will not work with floods as they would be element.x
163. \*/
164. let isCellInvalid = (cell) => {
165. //Check weather it was the orginal cell
166. **if** ((mouseDownPos.celly) != cell.y || (mouseDownPos.cellx) != cell.x) {
167. **for** (let x = - 1; x <= 1; x++) {
168. **for** (let y = - 1; y <= 1; y++) {
169. **if** ((cell.x - x == newCellX
170. && cell.y - y == newCellY)
171. || (cell.x - x == newCellX
172. && cell.y - y == newCellY))
173. {
174. **return** **true**
175. }
177. }
178. }
179. }
180. **return** **false**
181. };
183. **if** (
184. (board.netList.some(net => {
185. **return** isCellInvalid(net.start)
186. || isCellInvalid(net.end)
187. } ))
189. || (board.floodList.some(isCellInvalid))
191. || (board.keepoutList.forEach(keepout => {
192. **return** isCellInvalid(keepout.start)
193. || isCellInvalid(keepout.end)
194. } ))
196. ){
197. //!This interaction with the user possibly shouldn't be in this function
198. document.getElementById('warning').innerHTML = "Nodes too close!";
199. setTimeout(() => document.getElementById('warning').innerHTML = "", 2500);
201. elmnt.style.top = (mouseDownPos.mousey) + "px";
202. elmnt.style.left = (mouseDownPos.mousex) + "px";
203. } **else** {
204. cell.x = newCellX;
205. cell.y = newCellY;
206. }
208. board.update();
209. }
210. }
212. /\*\*
213. \* Inalizes a net
214. \*/
215. createNet(startx,starty,endx,endy) {
216. let start = **new** clientSideCell(startx, starty);
217. let end = **new** clientSideCell(endx, endy);
218. let startID = "start" + **this**.cellCounter;
219. let endID = "end" + **this**.cellCounter;
220. **this**.cellCounter++;
222. **this**.endPointContainer.appendChild(
223. start.buildDOM(startID)
224. );
226. **this**.endPointContainer.appendChild(
227. end.buildDOM(endID)
228. );
230. let net = **new** clientSideNet(start,end);
232. **this**.makeInteractable(start,deleteCallBack);
233. **this**.makeInteractable(end,deleteCallBack);
235. let netList = **this**.netList
236. netList.push(net);
238. **function** deleteCallBack() {
239. start.el.remove(); //Remove the Dom
240. end.el.remove();
241. **var** index = netList.indexOf(net);
242. **if** (index > -1) {
243. netList.splice(index, 1);
244. }
245. }

248. }
250. createFlood(x,y) {
251. let cell = **new** clientSideCell(x,y);
253. **this**.floodContainer.appendChild(
254. cell.buildDOM("flood","flood")
255. )
257. **this**.makeInteractable(cell, deleteCallBack);
259. **this**.floodList.push(cell);
261. let floodList = **this**.floodList;
263. **function** deleteCallBack() {
264. cell.el.remove(); //Remove the Dom
265. document.getElementById("addFlood").disabled = **false**;
266. floodList.pop();
267. }
268. }
270. createKeepout(startx, starty, endx, endy) {
271. let start = **new** clientSideCell(startx, starty);
272. let end = **new** clientSideCell(endx, endy);
274. **this**.keepoutContainer.appendChild(
275. start.buildDOM("keepoutStart" + **this**.cellCounter,"keepout")
276. );
278. **this**.keepoutContainer.appendChild(
279. end.buildDOM("keepoutEnd" + **this**.cellCounter,"keepout")
280. );
282. let keepout = **new** clientsideKeepout(start,end,**this**.cellCounter);
284. **this**.cellCounter++;
286. **this**.makeInteractable(start,deleteCallBack);
287. **this**.makeInteractable(end,deleteCallBack);
289. **this**.keepoutList.push(keepout);
291. let keepoutList = **this**.keepoutList;
293. **function** deleteCallBack() {
294. start.el.remove(); //Remove the Dom
295. end.el.remove();
296. **var** index = keepoutList.indexOf(keepout);
297. **if** (index > -1) {
298. keepoutList.splice(index, 1);
299. }
300. }
301. }
303. addNet(net) {
304. **this**.netList.push(net)
305. **return** **this**.netList.length - 1;
306. }
308. //This is quite hacky and not very OOP should fix at somepoint
309. net(id, contents) {
310. **this**.netList[id] = contents;
311. }
313. async update() {
314. //Fetch the SVG DOM from the server but store the value as a promise
315. let requestContent = {
316. netList: **this**.netList,
317. floodList: **this**.floodList,
318. keepoutList: **this**.keepoutList
319. }
321. let resGrid = fetch('/route?cellSize=' + **this**.cellSize, {
323. method: 'POST',
324. headers: {
325. 'Content-Type': 'application/json'
326. },
327. body: JSON.stringify(requestContent)
329. })
331. //Await for the promise to be fullied when it is set the value of the dom to be that
332. let response = await resGrid.then(response => response.text());
333. response = JSON.parse(response)
334. **this**.grid.innerHTML = response.board;
335. **this**.svg = response.board;
336. //!Example of reduce
337. **if** (response.errors.length > 0) {
338. let errorHeader = "Warning: <br/>"
339. let groupedErrors = errorHeader + response.errors.reduce((accumulator, current) => accumulator + "<br/>" + current);
340. document.getElementById('errors').innerHTML = groupedErrors;
341. } **else** {
342. document.getElementById('errors').innerHTML = "";
343. }
344. }
345. }
347. **function** addNetButtonListener() {
348. grid.createNet(2,2,4,4)
349. grid.update();
350. }
352. **function** addFloodButtonListener() {
353. document.getElementById("addFlood").disabled = **true**;
354. grid.createFlood(5,5);
355. grid.update();
356. }
358. **function** addKeepoutButtonListener() {
359. grid.createKeepout(15,15,20,20);
360. grid.update();
361. }
363. **function** downloadButtonListener() {
364. **var** a = window.document.createElement('a');
365. console.log(grid.grid.innerHTML)
366. a.href = window.URL.createObjectURL(**new** Blob([grid.grid.innerHTML], {type: 'text/svg'}));
367. a.download = 'board.svg';
369. // Append anchor to body.
370. document.body.appendChild(a);
371. a.click();
373. // Remove anchor from body
374. document.body.removeChild(a);
375. }
376. //------------MAIN------------
378. **const** cellSize = 10;
379. **const** gridWidth = 10;
380. **const** gridHeight = 10;
381. **const** nodeContainerID = "node-container";
382. **const** floodContainerID = "flood-container";
383. **const** keepoutContainerID = "keepout-container"
385. **const** gridID = "board";
387. **var** grid = **new** Grid(gridID,
388. nodeContainerID,
389. floodContainerID,
390. keepoutContainerID,
391. gridWidth,
392. gridHeight,
393. cellSize);
395. grid.createNet(2,2,3,3)
397. grid.update();

# Bibliography

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

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

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

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

1. A *Net* is a collection of wires that connects all of the points or pins in a single circuit [↑](#footnote-ref-2)
2. I have chosen to export in SVG form as it is easy to describe using a programing language which will make the export stage easier. [↑](#footnote-ref-3)
3. A Thermal Sink is an area of the PCB which is entirely copper and is used to sink heat from high current devices [↑](#footnote-ref-4)
4. Isometric routing is where multiple routes are routed to be approximately the same length to reduce skew in parallel data transition. [↑](#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. Most breadth first algorithms have a complexity similar to [↑](#footnote-ref-8)