TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

**TAPIO KATAJISTO**

# AUTOMATIC CONNECTOR ROUTING AS A WEB APPLICATION

Master's thesis

Examiner: Professor Kari Systä

The examiner and topic of the thesis
were approved on 3 January 2018

# ABSTRACT

**TAPIO KATAJISTO**: Automatic connector routing as a web application
Tampere University of Technology
Master of Science Thesis, 43 pages
July 2018
Master's Degree Programme in Information Technology
Major: Pervasive Computing
Examiner: Professor Kari Systä

Keywords: diagrams, routing, orthogonal routing, web, TypeScript

Diagrams are used in multiple fields as an aid in the design process. These diagrams can be automatically generated or manually drawn. In manually drawn diagrams, the editor software can assist the drawing process by offering automatic routing of connectors to speed up the drawing process. In automatically generated diagrams automatic routing is a must. We go through an orthogonal connector routing algorithm and compare its design decisions to other alternatives. A web based prototype of the algorithm was implemented in TypeScript and evaluated. It was considered to produce good quality routes and to be efficient enough for interactive editing of diagrams.

# TIIVISTELMÄ

**TAPIO KATAJISTO**: Automatic connector routing as a web application
Tampereen teknillinen yliopisto
Diplomityö, 43 sivua
Heinäkuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Computing
Tarkastaja: professori Kari Systä

Avainsanat: diagrammit, reititys, suorakulmainen reititys, web, TypeScript

Diagrammeja käytetään monella alalla tutkimustyön apuna. Käytetyt diagrammit ovat joko automaattisesti generoituja tai käsin piirrettyjä. Manuaalisessa piirtämisessä editointiohjelmisto voi auttaa käyttäjää viivojen piirrossa reitittämällä ne automaattisesti ja nopeuttaa siten piirrostyötä. Automaattisessa diagrammin generoinnissa reititys on välttämätöntä. Työssä käydään läpi algoritmi suorakulmaisten viivojen reititykseen ja pohditaan algoritmissä käytettyjä ratkaisuja. Työn ohessa on toteutettiin ja evaluoitiin web-pohjainen prototyyppi tästä algoritmista käyttäen TypeScript ohjelmointikieltä. Prototyypin todettiin tuottavan hyvälaatuisia reittejä ja olevan tarpeeksi tehokas interaktiiviseen editointiin.

# PREFACE

This thesis work was made for Valmet Automation to investigate the viability of automatic routing. During the work, I have been able to deepen my knowledge on a broad spectrum of technologies and work on a very interesting project.

I want to thank Valmet and Ari-Pekka Pura for coming up with this interesting problem to work on and allowing me to write my thesis about it. I would also like to thank my professor Kari Systa for guidance and encouragement on getting the thesis done. In addition I would like to thank everyone at Valmet, my family and friends who supported me when writing the thesis and given valuable feedback. Additional thanks to Inari Frank who proofread the thesis.

In Tampere, Finland, on 31 July 2018

Tapio Katajisto

# CONTENTS

1. INTRODUCTION ..... 1
2. BACKGROUND AND THEORY ..... 3
   2.1 Automatic routing ..... 3
   2.2 Requirements ..... 3
   2.3 Existing solutions ..... 4
   2.4 Terminology ..... 5
   2.5 Diagram quality ..... 6
   2.6 Minimizing length, bends and crossings ..... 6
3. ORTHOGONAL CONNECTOR ROUTING ..... 10
   3.1 Step 1: Orthogonal visibility graph ..... 12
   3.2 Step 2: Routing connectors ..... 16
   3.3 Step 3: Ordering and nudging ..... 18
4. IMPLEMENTATION ..... 24
   4.1 TypeScript ..... 25
   4.2 Router ..... 26
   4.3 Challenges in implementation ..... 28
5. EVALUATION ..... 30
   5.1 Routing quality ..... 30
   5.2 Shortcomings ..... 30
   5.3 Performance ..... 33
      5.3.1 Impact of diagram size ..... 33
      5.3.2 Impact of different layouts ..... 34
6. FURTHER DEVELOPMENT ..... 37
   6.1 Extensibility ..... 37
   6.2 Combining manual routing with automatic routing ..... 37
   6.3 Improved connection networks with rectilinear steiner tree heuristic ..... 37
   6.4 One-bend graph ..... 38
7. CONCLUSION ..... 40
REFERENCES ..... 42

# LIST OF FIGURES

***Figure 2.1.*** *Minimum remaining bends.* ..... 7
***Figure 2.2.*** *Ordering overlapping connectors* ..... 8

***Figure 3.1.*** *Orthogonal Visibility Graph* ..... 12
***Figure 3.2.*** *Visibility in OVG* ..... 13
***Figure 3.3.*** *Hanan's grid and Minimum Rectilinear Steiner Tree* ..... 13
***Figure 3.4.*** *OVG line sweep* ..... 14
***Figure 3.5.*** *Routed connector in OVG* ..... 17
***Figure 3.6.*** *Two overlapping connectors routed in OVG* ..... 17
***Figure 3.7.*** *Connection network* ..... 19
***Figure 3.8.*** *Line sweep for generating nudge limits* ..... 19
***Figure 3.9.*** *Segments without nudging* ..... 20
***Figure 3.10.*** *Nudged segments* ..... 21
***Figure 3.11.*** *Nudging* ..... 22

***Figure 4.1.*** *Modules* ..... 24

***Figure 5.1.*** *grid25x50* ..... 30
***Figure 5.2.*** *Segment overlap* ..... 31
***Figure 5.3.*** *Block overlap* ..... 32
***Figure 5.4.*** *Related connectors* ..... 32
***Figure 5.5.*** *compact25x50* ..... 34
***Figure 5.6.*** *networks25x50* ..... 35

***Figure 6.1.*** *Two same cost routes* ..... 38

# LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| *OVG* | Orthogonal Visibility Graph |
| *UML* | Universal Modeling Language |
| *CAD* | Computer Aided Design |
| *HTML* | Hypertext Markup Language |
| *EDA* | Electronic design automation |
| *MCP* | Minimum Cost Path |
| *MRST* | Minimum Rectilinear Steiner Tree |
| *GPL* | General Public License |
| *VLSI* | Very Large Scale Integration |
| *VPSC* | Variable Placement with Separation Constraints |
| *V* | Set of vertices |
| *E* | Set of edges |

# 1. INTRODUCTION

Diagramming is an integral part of many design tools, including UML diagram tools, CAD tools, and graphics software. These tools are extensively used to aid in design work in numerous fields of application including software design and development. In software design, diagrams are used to represent various concepts, including system architecture, class diagrams, program flow, software dependencies, and visual programming languages. Some of these visualizations are automatically generated and are used to visualize existing data. However, many are hand drawn as an actual design input and require every object and connection to be drawn manually. This can be tedious [16], especially when you need to reorganize an existing diagram with complex connections to make room for additional objects. Usually one ends up needing to draw every connection all over again. People also have a tendency to spend huge amounts of time in fine-tuning connection lines. In all, these will add up to a significant amount of time that could be spent more productively.

To reduce the time spent on tedious tasks related to diagram drawing, automated routing of connections between diagram items can be utilized. Most diagram editors have some kind of automatic connector routing capabilities but they usually have limited features, produce unpredictable routes and do not allow fully interactive editing [16]. This thesis reviews various existing pieces of software with connection routing capabilities, goes through theory and implementation of an algorithm for routing connectors in diagrams and evaluates the achieved results.

The routing algorithm implemented in this thesis is based on the paper Orthogonal Connector Routing [17]. The paper introduces an algorithm in three steps to produce predictable orthogonal object-avoiding routes in presence of obstacles. In this thesis, a prototype of the algorithm introduced by Wybrow's paper was implemented using TypeScript. The prototype is supposed to be used in web applications.

Evaluation of the implemented prototype shows that TypeScript implementation is feasible in terms of performance, showing acceptable performance even in larger diagrams. Lastly, before the conclusion, we go through possible topics for further developing the prototype of which one is already partly implemented.

In this chapter we introduced the topic of this thesis. In the second chapter, we cover the background and theory. The third chapter introduces an algorithm for routing connectors in diagrams.

Next, we look into the Typescript implementation of the connector routing algorithm that was done during the thesis work and evaluate its routing quality and performance.

The fourth chapter describes various ideas for further development of the implemented software to further improve its performance.

Lastly, we conclude with a summary of what was done and the main results of this thesis.

# 2. BACKGROUND AND THEORY

In this chapter we define the terminology used in this thesis and then go through the basic idea behind automatic routing and review a collection of different criteria by which we can evaluate visual aesthetics and readability of diagrams.

## 2.1 Automatic routing

When authoring any drawings or diagrams with logical connections between objects, e.g. circuit diagrams, CAD drawings, UML diagrams; where it is important to clearly identify which items are connected; it usually needs extensive fine-tuning of the connectors. The user needs to manually adjust them to avoid overlaps, bundle related connections, arrange connector segments to introduce symmetry, and so on. Usually, all of this needs to be repeated multiple times when making modifications to the diagram, for example adding or removing objects or rearranging objects into new hierarchies.

Requirements for connection layout can be divided into three cases. Firstly, in some cases the layout should be freely drawn by user. General drawing applications can be placed in this category. In this case, no automatic routing is needed. In the second case, one is not interested in the layout except that it should be easily readable, i.e. aesthetics are the most important factor. And lastly, the layout needs to be optimized by some other constraints. These constraints can include overall area, gap between lines, length of connectors. In this thesis we concentrate on the second category where we are only interested in readability of the diagram.

## 2.2 Requirements

This thesis had only a few requirements. This was because it started as a investigation on possibilities to utilize automatic routing in interactive editor tools. The basic idea was to try to implement a good enough automatic routing prototype as a web application.

**Web as the platform** restricted the choice of implementation language to JavaScript or another language that transpiles to JavaScript, such as TypeScript, but offers many advantages such as a web browser being available on almost any device today. This is not true any more as web browser vendors have started to ship WebAssembly [3] implementation which expands the language selection with languages that have it as a compilation target.

**No restrictive licenses** can be included through third party components such as proprietary or copyleft licenses.

**Support diagrams** that consist of rectangular obstacles and rectilinear connectors or rectilinear connection networks (connectors that connect more than two connection points). Furthermore, obstacles are not overlapping in most cases but because interactive editing must be supported, there may be intermediate situations where overlapping obstacles are created and we have to handle them in a sensible and predictable way. Connector segments cannot the share same path unless they belong to the same connection network. The gap between neighbouring connectors and obstacles must be configurable. Connection points reside on either the left, right, top, or bottom side of the obstacles. Connectors can approach connection points only from predefined directions. At this phase it is defined that connection points on the left side of obstacles can be approached only from the right of the connection point, points on the right side from the left of the connection point, and so on.

**Routing quality** has to be good enough so that manual connector drawing can be forfeit and have automatic routing as the only option. The requirements did not specify how the quality should be measured but stated only that it must be subjectively good enough. However, the next chapter explores various criteria of how the quality could be compared objectively.

**Performance** must be good enough to allow relatively smooth interactive editing in diagrams with up to 100 obstacles and 200 connectors.

Only the first two of these requirements are strict and objective. The last two regarding the quality and performance were only loosely defined as ”as good as possible” and are thus not actually requirements but rather results of this work.

## 2.3 Existing solutions

**Libavoid** seems to be the most promising one of the reviewed solutions. It is part of the adaptagrams library, which offers extensive tools for layouts and routing with adaquate performance. On top of this, it is completely open source. However, the library is distributed under GPL license making it hard to use in a JavaScript environment without infecting the whole codebase with the license. The library is implemented in C++, which in itself wouldn't be a problem as it could be compiled to JavaScript or WebAssembly. However, the resulting binary would be quite large and GPL licensing causes issues.

**yWorks** is a commercial proprietary library implementing much of the same features that adaptagrams is offering. Its quality and performance seems to be good enough but being Java-based means that it cannot be integrated natively in a web browser. Also the proprietary license means that yWorks cannot be used.

**WebCola** is a TypeScript based library distributed under a non-restricting MIT license. The library implements only a handful of constraint-based layout algorithms and a simple grid routing algorithm. This means that it is not good enough for our use because we are not interested in the layout of the whole diagram but only in routing of the connectors

between blocks. The grid routing provided was also not useful because it requires very coarse snap-to-grid to be fast enough for interactive editing. However, some parts of it are very useful in implementing a routing library.

In this thesis we implemented a routing library with TypeScript, using parts of WebCola as a library. This choice was made partly because WebCola contains a few critical parts that were needed for the implementation, namely the constraint solver, that is somewhat hard to implement efficiently and without bugs. In addition, it was first planned to implement the prototype as a part of the WebCola library but that idea was later abandoned. Another reason was that static typing was preferred to dynamic typing and TypeScript was the most reliable language applicable being created and maintained by Microsoft.

The implemented algorithm is using a paper by Wybrow et al. called orthogonal connector routing [17] as the basis. Libavoid is based on the same work. The rest of the thesis concentrates on the theory and the implementation of the routing algorithm.

## 2.4 Terminology

**Graph** is a data structure consisting of a set of nodes connected together by a set of edges. An edge is a pair of nodes marking them as connected. An edge can also contain additional data such as cost value for traversal. A node usually contains some useful data, for example coordinates.

**Visibility graph** is a special graph representing each vertex's visibility to each other. It consists of vertices, edges, and obstacles. In a visibility graph, an edge can only be defined between two vertices if there are no obstacles in between.

**Obstacle** is a rectangular area in the diagram which we want connectors to avoid in routing. An obstacle can contain any number of connection points which can be located on any side of the obstacle.

**Connection point** is a point residing on a side of an obstacle to which connectors can be connected to.

**Connector** represents a connection between two connection points and is represented in the diagram as a rectilinear **route**. A connector consists of list of segments. These two terms can be used interchangeably. Connectors are calculated by finding the minimum cost paths in the graph and first consists of list of graph edges. However, they are later "simplified" by unifying all consecutive edges between bends.

**Rectilinear** connector or route is a route consisting of only rectangular bends, i.e. it only has horizontal and vertical segments. Synonymous with **orthogonal**.

**Segment** is defined as a straight line between two points. In this thesis, we are only interested in rectilinear routes, which means that all segments we are dealing with are either

horizontal or vertical. In the former case, both points share their y-coordinates; and in the latter case, their x-coordinates.

**Connection network** is a connector which has more than two endpoints, thus forming a tree between the endpoints.

**Junction** is a branch point in connection network visualization.

**Bend** is a corner in the connector's route, i.e. point between two segments of different alignment.

## 2.5 Diagram quality

Diagram quality can be evaluated using various criteria. Purchase et al. shows that the most important criteria are minimizing **route length**, number of **bends**, number of **route crossings**, using only horizontal text, joining of inheritance connectors, narrower diagrams, and **orthogonality** of the layout, i.e. objects are aligned on the same horizontal and vertical lines[14]. These criteria are based on UML diagrams but can probably be also applied to more general cases. Some diagrams representing concrete real world concepts might benefit from mimicking the real world parallel which is also impossible to automate without user input [14]. An earlier paper by Purchase et al. shows that increasing symmetry doesn't have as large an effect on graph aesthetics as minimizing crossings, but still has some effect [13].

Out of these criteria we try to focus on route length, number of bends, number of crossings and symmetry. These criteria were picked because they can easily be affected by adjusting the routes while leaving the obstacle layout as is.

In case of symmetry, we can only adjust the symmetry of the routes but not the symmetry of the whole layout. This decreases the effect of it even further and may not be very important at all. However, route symmetry is tried to be addressed by trying to place the route bends in the middle of the available space and having equal length gaps between adjacent routes.

Another way of evaluating automatic routing quality is the predictability of the result. Routes should behave predictably and produce consistent results when the user makes modifications to the diagram layout or connections. In other words, making small changes to the layout should not cause large changes in routes. [17]

The term orthogonality used here is referring to the whole diagram, describing the orthogonality between diagram blocks, and should not be confused with orthogonality of connectors.

## 2.6 Minimizing length, bends and crossings

Dijkstra's algorithm [6] solves the criteria of minimizing the path length, i.e. the shortest path problem. It works by traversing a graph, keeping track of the minimum cost to every

traversed node. Usually some kind of distance is used as a cost function, such as Euclidean distance, i.e. the actual distance between two points, or Manhattan distance, i.e., the sum of horizontal and vertical distance between two points. A* algorithm [5] is a variant of Dijkstra's algorithm. It applies a heuristic function estimating the remaining cost to our goal node, thus allowing it to discard certainly suboptimal routes. The heuristic function needs to be admissible, meaning that it never overestimates the remaining cost, to guarantee that the minimum cost path is found [5]. A* is therefore much more efficient when finding a single route from point A to B because it needs to traverse much fewer graph nodes. Dijkstra, on the other hand, is more efficient when we need the shortest path from some node A to every other nodes. In this case, A* is the preferred algorithm as we are only interested in individual paths.

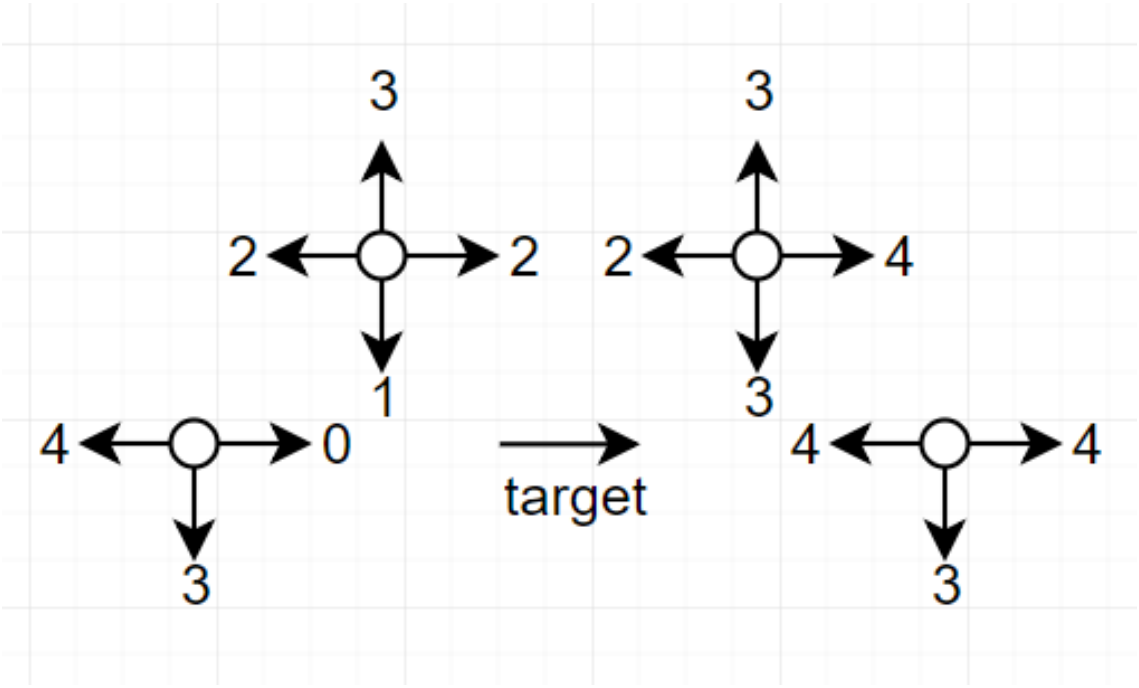


***Figure 2.1***. *Minimum remaining bends where the arrow marked as the target is the connection point and the angle which it can be entered from. Circles are possible current positions and arrows indicate all possible directions for the next position. The numbers tell the minimum amount of bends that need to be taken if that direction is chosen.*

A* provides an efficient way of calculating a minimum cost path in a graph. In addition to minimizing the distance, thus providing the shortest path, one can add an additional cost for bends in the cost function. If one gives large enough penalty to bends we can get the algorithm to always choose the shortest path available when considering only paths with the minimum bends possible.

On the other hand, instead of adding the bend penalty to the cost function, the bend penalty can be pre-calculated to the edge costs of the graph. This can be achieved by separating the graph into two parts, the other part contains the horizontal segments, and the other contains the vertical segments. The parts are linked by adding additional edges representing the

bends between the horizontal and vertical graph nodes which are at in the same position. This has the benefit that the cost function does not need to know about the previously traversed edges. For this implementation, the cost function approach was used as it seemed more simple to implement at the time.

However, if one applies a bend penalty, explicitly or implicitly, to the cost function one needs to add heuristic of it to the A* heuristic function as well. Fortunately, Wybrow et al. introduced a simple heuristic to estimate the remaining bends in their paper [17]. It works by observing the direction where A* is currently traversing and the directions from where one can approach the goal and deducing the remaining bends from this information according to the figure (Figure 2.1). If we start at a position indicated as a circle in the figure and the target is at the end of the ”target” arrow and must be approached in the direction indicated by the arrow, the minimum remaining bends for each possible direction can be read from the number next to each arrow.

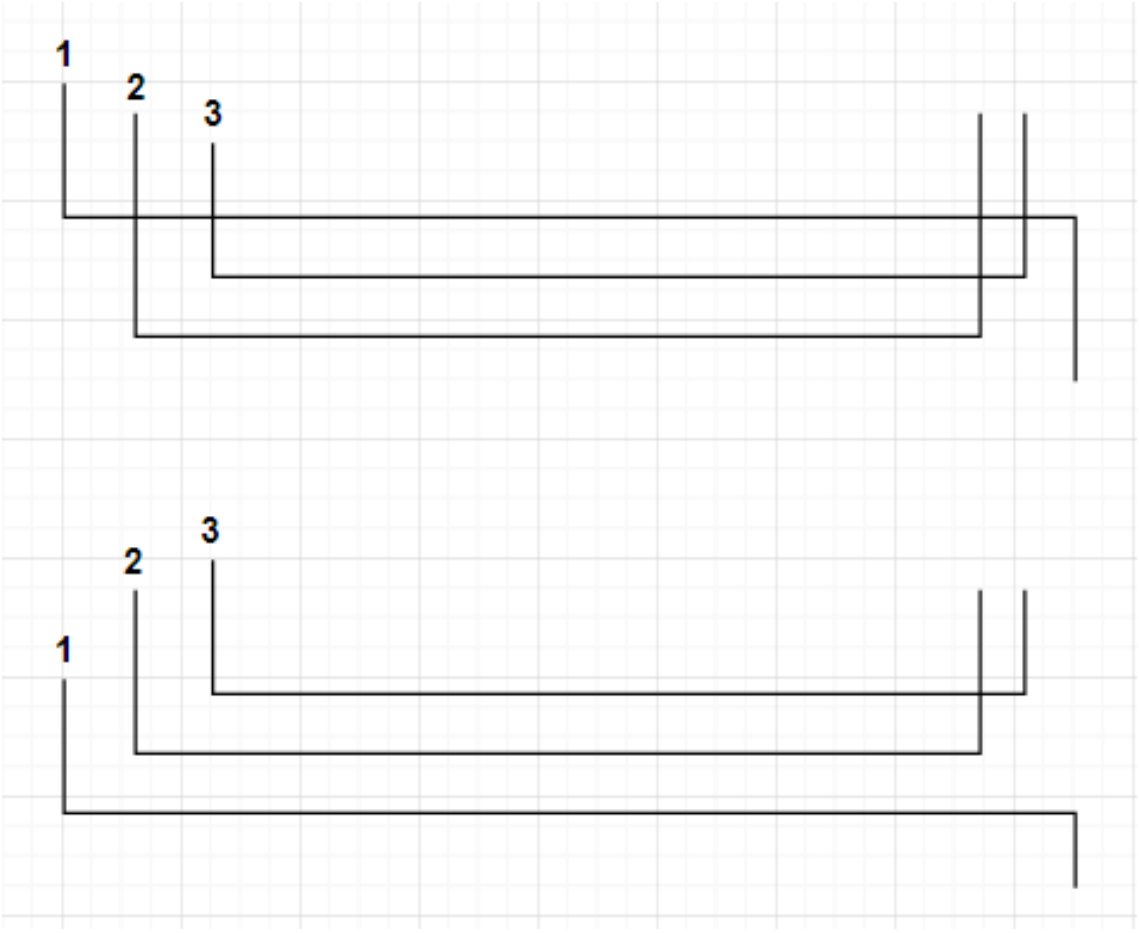


***Figure 2.2***. *Demonstration of importance of ordering nudged segments. On the top we have the segments in an arbitrary order. On the bottom we have ordered the segments so that connector crossings are minimized.*

Crossings minimization can be addressed by identifying parallel connector segments and ordering them in such a way that the first one to bend in either direction is in the outermost position (Figure 2.2). Wybrow et al. has come up with a solution that utilizes the longest

common subsequence and the direction of the convergence and divergence which can decide the optimal order which minimizes crossings between these parallel routes [17].

In the above sections we have described solutions to minimize properties of connectors, which produces quality diagrams. In the next chapter we will go through the specifics of the Orthogonal Connector Routing as described by Wybrow et al.

# 3. ORTHOGONAL CONNECTOR ROUTING

This chapter introduces the algorithm that was implemented in the thesis and explains its details. We start with the overall view of the three step algorithm and then go through the details of each step in subsequent sections.

A suitable data structure has to be found for representing the diagram. A graph is a widely used structure for optimizing paths and has been used to solve similar problems in various ways [8] [5] [15] [12] [9] [17]. A graph is also the basis for the algorithms introduced in the earlier chapter.

In the paper Orthogonal Connector Routing, published in 2010 [17], Wybrow et al. introduced an algorithm for routing predictable orthogonal object-avoiding point-to-point connections in block diagrams. The paper also walks through the some of the other aforementioned papers in the paper Orthogonal Connector routing and points their weaknesses.

Lee [8] used an uniform grid graph which would cause the graph to be massive when fine-grained block placement is required. Wu uses a so-called track graph which is only concerned with length minimization [15], Miryala uses rectangulation which is similar to their own representation but does not model vertical connections well [12].

In this thesis, it was decided to implement the algorithm described in Wybrow's paper. The decision was made by considering the similarity of the problem at hand and the following benefits of the approach.

- Documented clearly with the published paper and reference implementation.
- The algorithm is efficient enough to allow fully interactive editing even in large diagrams and the performance could be verified with the reference implementation.
- Implementation is easily modularized as the three-step model clearly separates the concerns of each step. The algorithm also introduces the concept of nudging in which connectors can be routed on top of each other and they are separated from each other at later stage. This allows routes to be routed separately, which in turn opens up the possibility of routing them in parallel. It also makes the A* routing easier because you do not need to avoid other routes.
- It was the only source where the implementation details were explained on a practical level.

Wybrow's algorithm is divided to three steps, each with different concerns. The algorithm is derived from earlier work, taking its three-stage model from [1] and showing that the

three-stage model is also applicable to orthogonal routes. Wybrow's algorithm is also similar to work introduced by Miriyala et al. [12] but with different design decisions and the contribution of concept of nudging to the original work.

The three-stage model used in Wybrow's algorithm consists of following steps.

1. Construct a graph representation of the diagram.
2. Route connectors in the graph, without taking overlapping routes into consideration.
3. Nudge overlapping segments apart from each other and adjust route visualization (this step can be composed of arbitrary visual adjustments, fitting different use-cases).

The three-stage algorithm's idea is to utilize existing research on path and tree minimization done in graph theory and VLSI [6][5][9][17] by representing the diagram as a graph, using existing graph algorithms to minimize connectors and then trying to overcome the visual shortcomings of those algorithms in the last step including overlapping segments, non-uniform gaps between routes, and large amount of crossing routes. The last step is very important in terms of diagram quality. Next we present a high level overview of the whole algorithm in very general terms.

1. Build orthogonal visibility graph
   - Generate horizontal graph segments with vertical line sweep.
   - Generate vertical graph segments with horizontal line sweep.
   - Intersect the segments to produce complete OVG.
2. Route connectors using A*
3. Final adjustments and nudging
   - Simplify routes by removing intermediate vertices
   - For vertical and horizontal segments separately
     - Calculate nudging limits for each segment
     - Divide segments to bundles based on their nudging limits and placement
     - Try to place as many segments in the same position to improve ordering results
   - Nudge vertical bundles
     - Create separation constraints between segments in the bundle
     - Solve constraints using the VPSC (variable placement with separation constraints) algorithm from WebCola repeatedly until satisfied.
   - Recalculate nudging limits because vertical nudging may have changed limits for horizontal segments.
   - Nudge horizontal bundles.

The following sections describe the details of each step of the algorithm, and provide the chosen implementation details.

## 3.1 Step 1: Orthogonal visibility graph

The paper Orthogonal Connector Routing [17] describes the so-called orthogonal visibility graph (OVG) to be used for A* routing. This is built in the first step of the three-step model described in the section above. The last two steps do not enforce a particular graph structure so a different choice could be made if it would serve our use case better. An improvement to the OVG was introduced in a later paper by Marriot et al. called Seeing Around Corners: Fast Orthogonal Connector Routing [10]. This graph structure makes an improvement to the OVG by trying to remove topologically equivalent routes. This was, however, not chosen to be implemented in this thesis because the algorithm was more complicated and there were errors in the documentation.

The definition of the Orthogonal Visibility Graph as follows: Let $I$ be set of interesting points in the diagram, i.e. corners and connection points of objects [17]. These points can be considered interesting because connection points are where we start and finish connectors and corners allow us to go around obstacles. If we draw a line from each interesting point in $I$ to each cardinal direction; stopping to first obstacle met; we can define vertices $V$ as a union of interesting points $I$, intersection points between aforementioned lines, and points where lines stopped at an obstacle [17]. Now we can define the orthogonal visibility graph as $OVG = (V, E)$ where edges $E$ are defined between each orthogonally adjacent vertices, i.e. if the vertices have rectilinear visibility to each other [17].

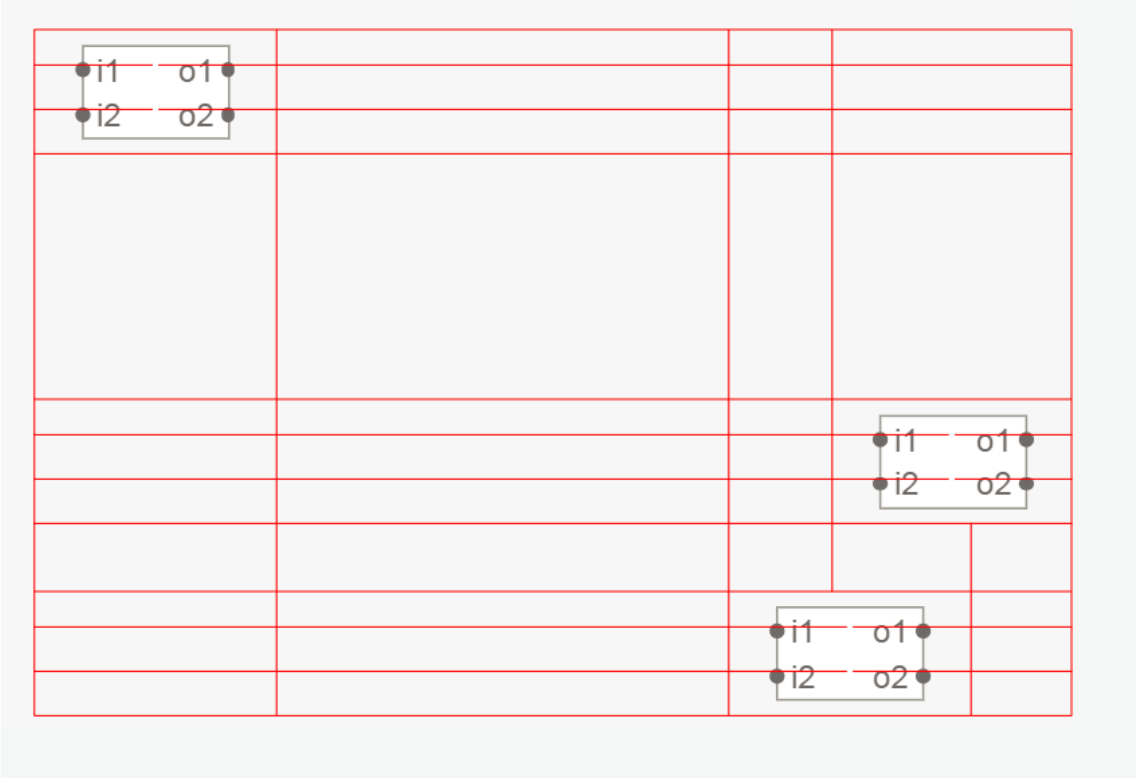


***Figure 3.1***. *Visualization of Orthogonal Visibility Graph where the red lines are edges and intersection points are vertices.*

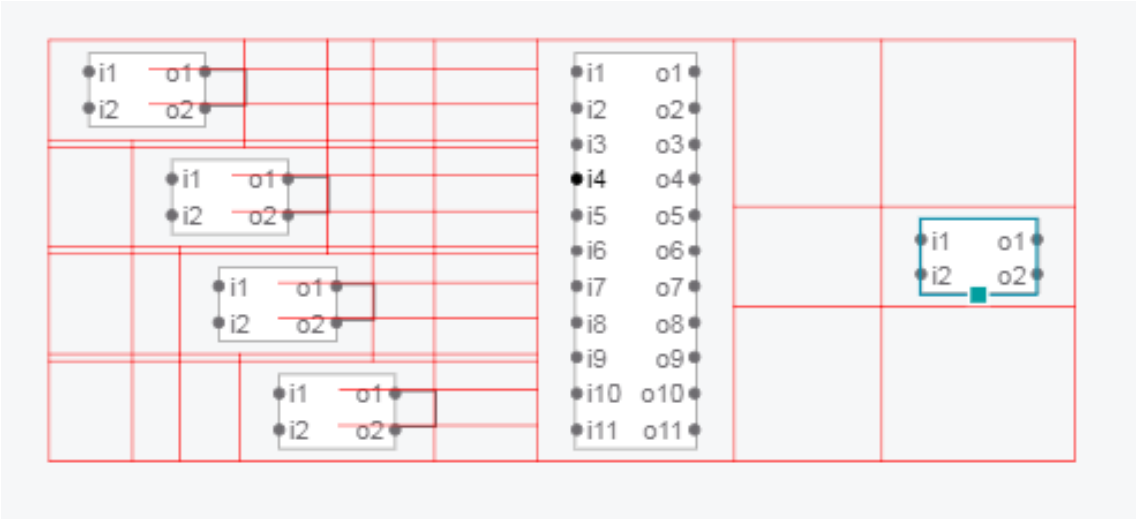


***Figure 3.2***. *Orthogonal visibility graph demonstrating the visibility part of it. The large block on the right blocks the visibility lines of the smaller blocks. In this figure we have removed all non-connected connection points from the graph to reduce complexity.*

By using OVG as the underlying graph, we get the guarantee that it contains the shortest paths between the connection points that avoid obstacles [17] while still having a relatively small amount of vertices and edges. However, routes and connection networks in the OVG might not be the most visually pleasing, because of overlapping routes. Correcting these visual shortcomings are done in the last step of the algorithm which addresses the problem by trying to order, center, and nudge route segments to be more visually pleasing.

OVG is also interesting because it is very similar to the Hanan grid (shown in Figure 3.3), which has been studied extensively because it is known to contain a minimum rectilinear steiner tree (MRST) for its vertices [4]. The Figure 3.3 has one possible MRST solution shown with blue lines on the right. Remember mind that there can be multiple correct solutions. MRST, or Minimum Rectilinear Steiner Tree, is an optimal solution to the

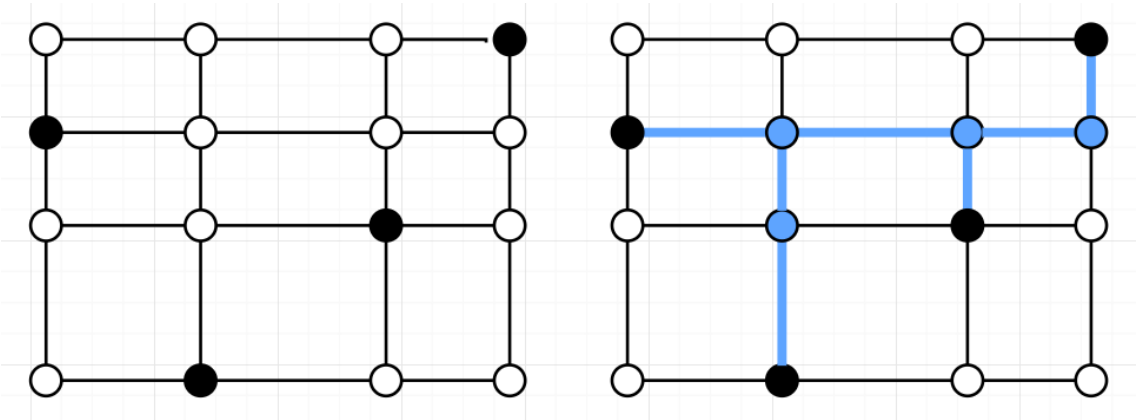

***Figure 3.3***. *Left: Hanan Grid with four terminals. The black dots are terminals (connection points), and the white dots intersection points. Right: Minimum rectilinear steiner tree in the same grid.*

following problem. Connect n vertices in a plane with the shortest graph, using only vertical and horizontal lines. The resulting graph can be shown to be a tree, with the initial vertices as terminals with some vertices added. These added vertices are called Steiner points. Unfortunately, calculating exact MRST is NP-hard [4] so using it in the routing phase is not feasible in our case because we have requirements for interactive editing and ability to support diagrams with 100 obstacles and 200 connectors, however the property of having the most optimal connection network by total length included in the graph is still useful as it opens the possibility of approximating MRST. However, MRST approximation was not done in this work but is included in the further development chapter.

Line sweep is the core of the OVG building algorithm. Line sweep operates on nodes and events. Nodes are used to identify and order obstacles and connection points along the scanline. Events are positions in the sweeping direction where the scanline is ”stopped”. These points include points where obstacles start and end. In a vertical sweep, as in figure 3.4 we have a horizontal scanline, nodes represent x-position of obstacles and events represent the y-position of interesting points of obstacles. The algorithm goes through every obstacle and creates a $node(id, r, p, left, right)$ where $id$ is a unique id, $r$ is the reference to the obstacle, $p$ is the rectangle's centre position, and $left$ and $right$ are ids of neighbouring nodes. Then for each node an $event(t, n, p)$ for both the start and end of the rectangle is created where $t$ is the event type, $n$ is a reference to a node, and $p$ the is position of the event in the processed dimension. An event is a position in the diagram that contains interesting properties, such as start or end boundary of a block or a connection point. In addition to obstacle events, a node and an event are also created for each connection point. In Figure 3.4, we can see an example of event and node positions with a vertical line sweep.

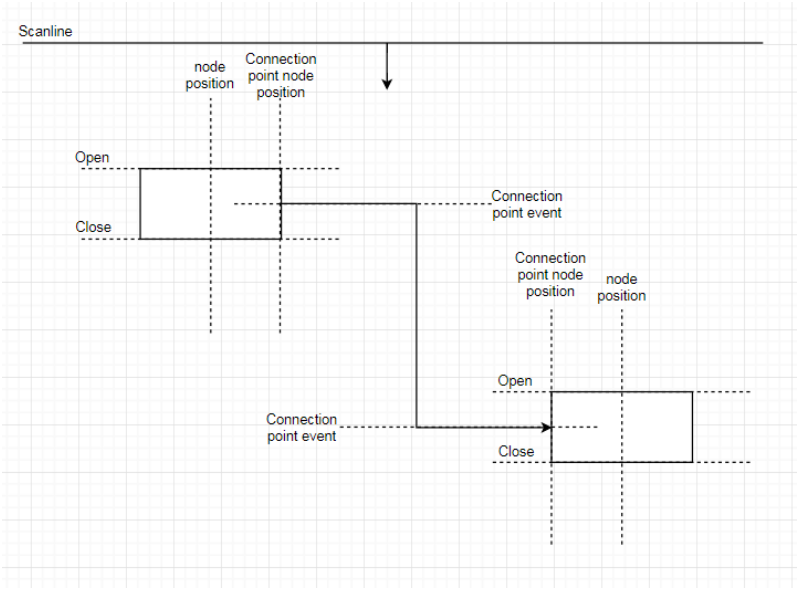


***Figure 3.4***. *Vertical line sweep producing horizontal segments*

Events are then sorted primarily by their position p and secondarily by their type where following order is used: *Open* > *Connection Point* > *Close*.

We then iterate through these events in two passes. The event processing function `processEvent` is provided as pseudo-code in the code listing below. The pseudo-code is somewhat simplified as it drops certain details including overlap handling. In the first pass we add the nodes from open events to a scan line object. The scan line is a self-balancing binary search tree which keeps the inserted nodes in order by their position and allow us to query neighbour nodes. When adding nodes to the scan line, we also iterate the scan line to both direction from the inserted node and update the left and right directions in the node objects.

```
1  Segment {
2    position: number
3    type: Open | ConnectionPoint | Close
4  }
5
6  Node {
7    position: number
8    left: Node
9    right: Node
10 }
11
12 function findNearest(node, position, forward):
13   left = MIN_NUMBER
14   right = MAX_NUMBER
15   next = node[forward]
16   while (next != null):
17     atSameRegion = isBetween(position, next.obstacle.top, next.obstacle.bottom)
18     if atSameRegion AND next.obstacle.right <= node.obstacle.left:
19       left = max(next.obstacle.right, left)
20     if atSameRegion AND next.obstacle.left >= node.obstacle.right:
21       left = min(next.obstacle.left, right)
22     next = next[forward]
23   return left, right
24
25 function processEvent(event, scanline, pass, direction):
26   node = event.node
27   obstacle = node.obstacle
28   position = event.position
29   if (pass == 1 AND event.type == Open)
30       OR (pass == 2 AND event.type == ConnectionPoint):
31     scanline.insert(node)
32     prev = scanline.prevOf(node)
33     next = scanline.nextOf(node)
34     if prev:
35       prev.right = node
36       node.left = prev
37     if next:
38       next.left = node
39       node.right = next
40
41   if pass == 2 AND (event.type == Open OR event.type == Close):
42     left, right = findNearest(node, position)
```

```
43
44      segments.insert((left, obstacle.left, position))
45      segments.insert((obstacle.left, obstacle.right, position))
46      segments.insert((obstacle.right, right, position))
47
48    if pass == 2 && event.type == ConnectionPoint:
49      left, right = findNearest(node, position)
50      if (direction == vertical):
51        if left connection point:
52          segments.insert((left, node.connectionPoint.x, position))
53        if right connection point:
54          segments.insert((node.connectionPoint.x, right, position))
```

After the first pass, we have successfully inserted all obstacle nodes to the scan line and have set up references to neighbouring nodes to them.

In the second pass we reiterate through the events. On encountering a connection point event, we add the node to the scan line similarly as in the first pass. On open and close events, we go through their neighbour nodes (left nodes in case of open event and right nodes in case of close events) starting from the closest one and check if their rectangle would block the line drawn from the originating point. Depending on the outcome, we create a segment starting from the node being processed and end it to the blocking rectangle boundary, if there are no blocking nodes we set the other end point to drawing area maximum. Also, in case of open events, the matching close event is searched and a segment is created between them and marking the end points as corner points.

These segments are then merged into continuous lines with the end point information still preserved for each segment as they are later used as graph vertices.

This process is then repeated vertically so that the notion of left and right becomes top and bottom.

## 3.2 Step 2: Routing connectors

In the second step, we utilize the graph structure constructed in the first phase to route the connectors. Any graph shortest path algorithm can be utilized as long as it minimizes the given cost function and does not have any special assumptions about the graph shape.

The solution used in Wybrow's paper [17] is to route each connector independently from each other in the orthogonal visibility graph using the A* algorithm with Manhattan distance and the number of bends with heavy weight as the penalty function. Because we are using additional penalty for bends, we need to acknowledge it in the heuristic function to keep it admissible. The minimum amount of bends is estimated as described in Figure 2.1. Routing the connectors in this manner can produce routes with shared segments which will be addressed in the final step.

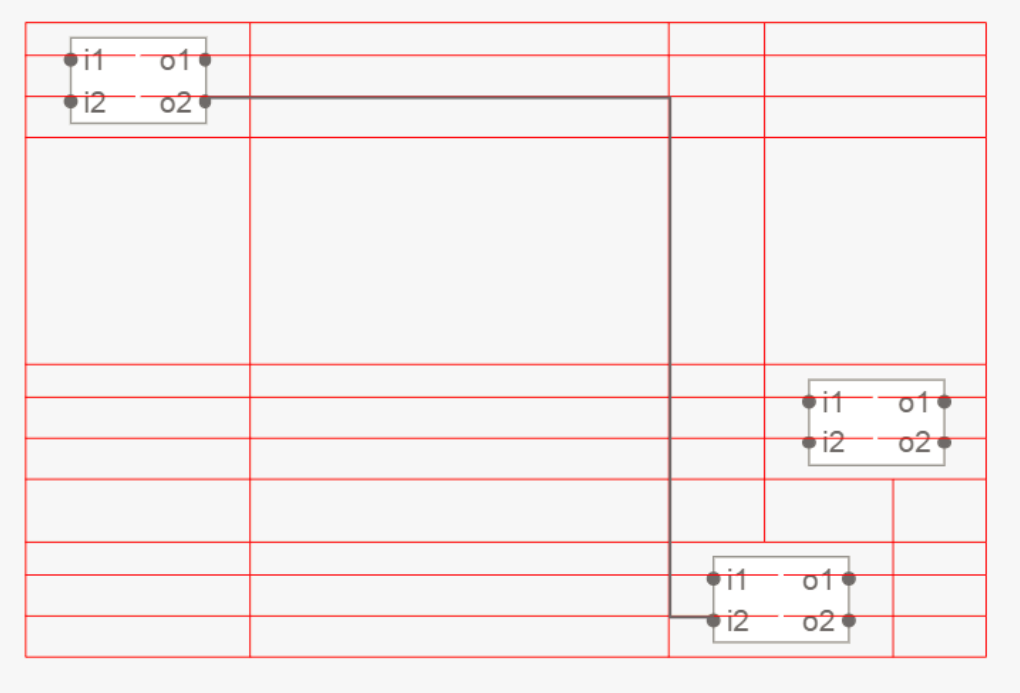


***Figure 3.5***. *Visualization of Orthogonal Visibility Graph where the red lines are edges and intersection points are vertices. The gray line is a routed connector.*

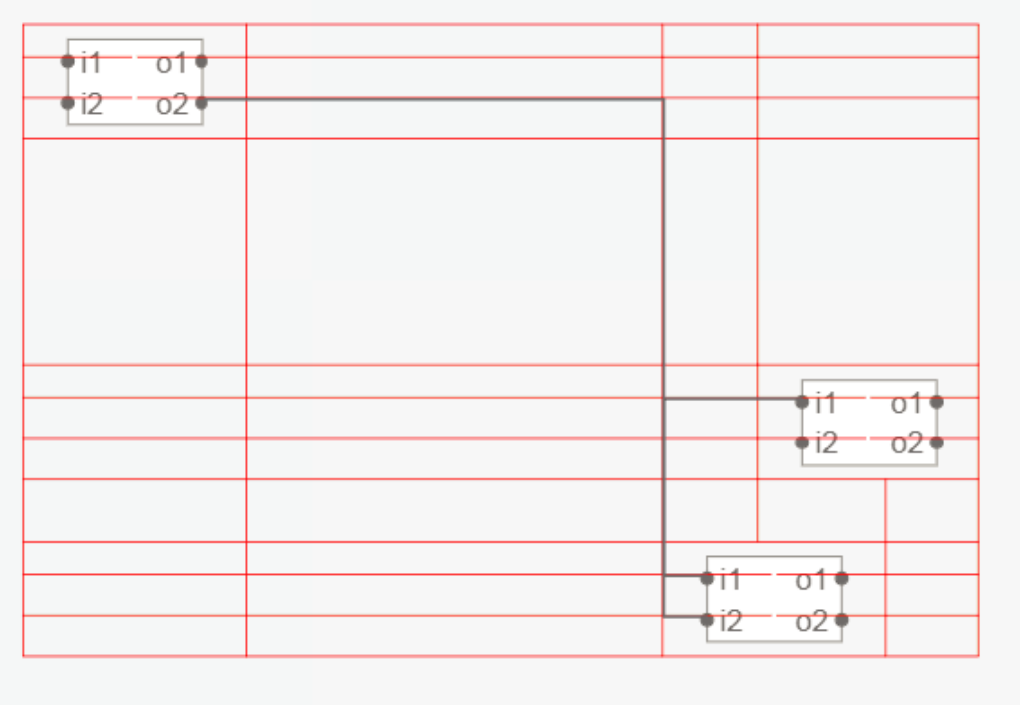


***Figure 3.6***. *Visualization of Orthogonal Visibility Graph where the red lines are edges and intersection points are vertices. Gray line is routed connector.*

In Figure 3.5, one route is routed with A* in the OVG from the top left block to the bottom right block. By adding another route in Figure 3.6 we can demonstrate route overlapping. This is normal behaviour in this step of the algorithm and is fixed in the last step.

Using a shortest path algorithm to minimize a single connector cannot, however, produce optimal results for connection networks, i.e. connectors with multiple terminals. In these cases, we either try to post-process these in the final step, or use multiple routing algorithms, one for point-to-point connections and one for connection networks.

## 3.3 Step 3: Ordering and nudging

The routing step leaves us with overlaps in routes. To fix this they need to be nudged away from each other. However, we need to figure out in which order they should be after nudging. The nudging algorithm from Wybrow's paper Orthogonal Connector Routing uses an ordering which minimizes the connector crossings between the nudged routes.

Figure 2.2 demonstrates the necessity of ordering. On the left side we have the nudged-apart segments in an arbitrary order, which leaves us with five crossings between connectors. On the right side we have calculated the optimal order which reduces the connector crossings to one. The specifics of the ordering algorithm are not explained here as we did not implement it ourselves but used a ready-made implementation.

However, all overlapping segments are not nudged apart. In Figure 3.7 we have two connectors which have same connection point as the other end. We want this to be represented as a connection network so that it is shown as a single tree structure in the diagram instead of two separate connectors. In these cases, we want to make sure that the overlapping segments remain overlapping.

Before nudging the routed connectors, the nearest obstacle boundaries have to be calculated first for each segment (in Figure 3.8 the right side of the left block and the left side of the right block act as boundaries for the vertical middle segment). These boundaries act as limits for segments when nudging them. A modified version of the line sweep algorithm used in building the OVG is applied in the boundary calculation. Vertical sweep is used in the description, the same applies to horizontal sweep but the x and y dimensions are swapped. With vertical sweep, horizontal limits for vertical segments are calculated. See Figure 3.8.

Boundary calculation works in the same way as finding the nearest blocks in the OVG building except now connector segments are also taken into account. Thus we have *Open, Open Segment, Close Segment, and Close* as possible event types which have priority in the aforementioned order. Again, events know their vertical position and have a reference to a node object. Obstacle nodes are the same as earlier and segment nodes have a reference to the segment object and have the segment's horizontal position information. Also, four passes are now used and are explained below.

**Pass one** updates limits for segments when encountering *Close Segment* by going through the node's left and right neighbours, and for *Close* events it iterates through segments on

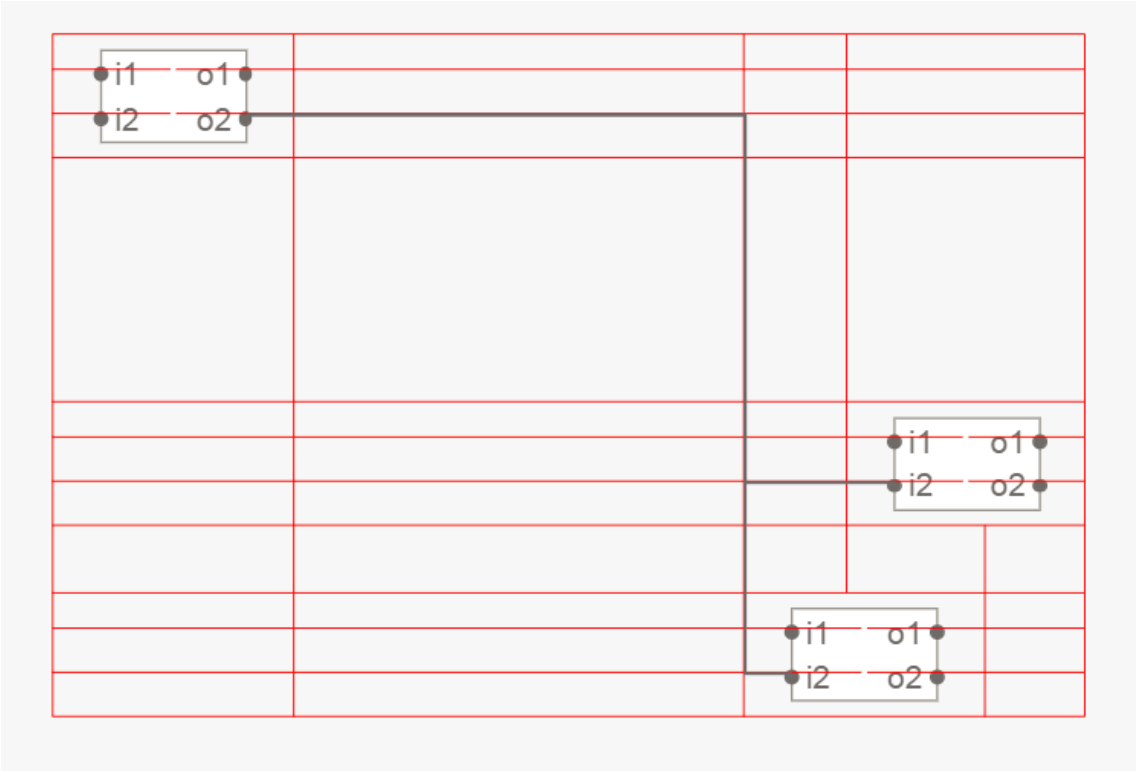


***Figure 3.7***. *Two routes which have same connection point form a connection network.*

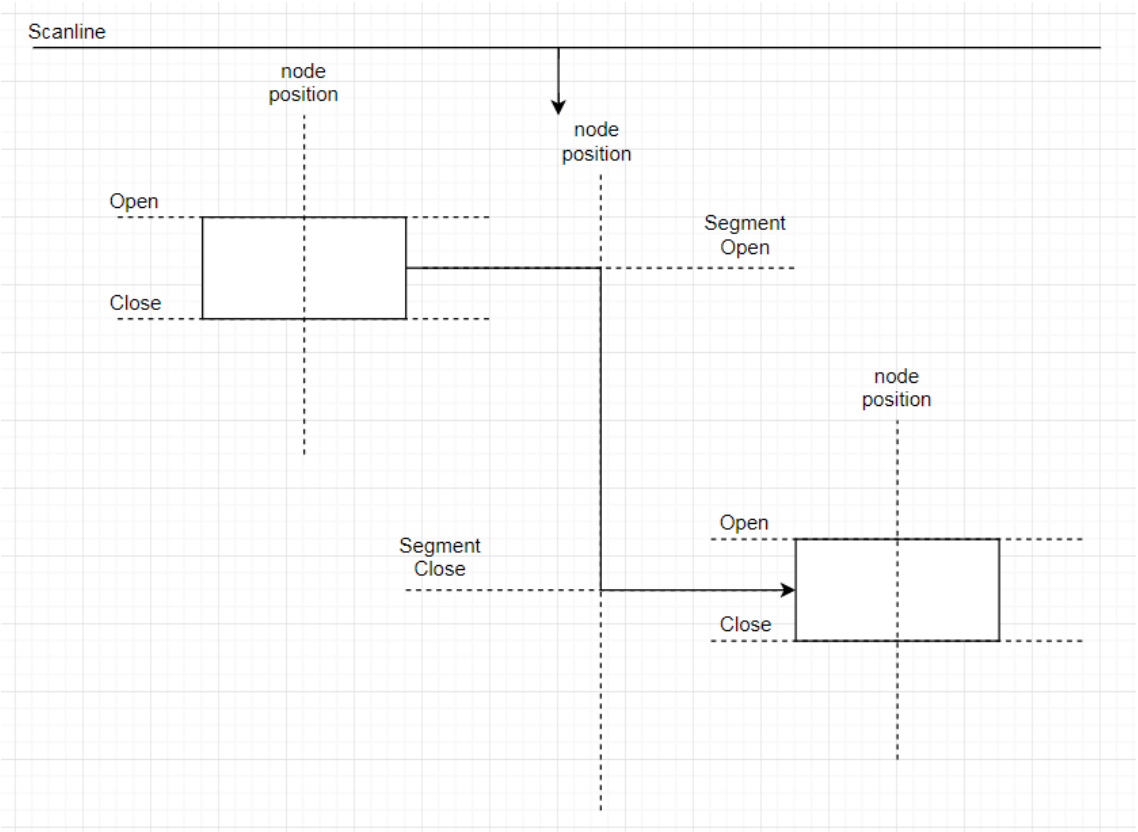


***Figure 3.8***. *Vertical scan line sweep producing horizontal limits.*

its left and right sides and adds itself as the segment's limit if it is closer than the limit the segment already has.

**Pass two** removes neighbour references from the node and removes it from the scanline. This is only done on *Close and Close Segment* events.

**Pass three** adds a node from the *Open and Open Segment* event to the scanline and then iterates through the scanline to both directions, updating the neighbour references.

**Pass four** does the same thing as pass one but for *Open and Open Segment* events.

The algorithm applies each pass to all events in the same position, one pass at the time for each event. When encountering an event in a new position it starts the passes from one again, repeating the process.

Intuitively this means that we move a horizontal scanline from top to bottom, checking which events (obstacles and segments) are still open in that particular position and adding the closest rectangle borders as limits to the segments. In addition, first and last segments always have both boundaries set to its own position because we do not allow them to be nudged.

Next, after having calculated boundaries for each segment, we try to place as many segments as possible in the same position. This is necessary for the order calculation as it depends on segments having the same positions. Then we calculate an optimal order which minimizes connector crossings using a method described in the Wybrow's paper [17]. After the order of segments are calculated, we need to nudge the overlapping segments apart.

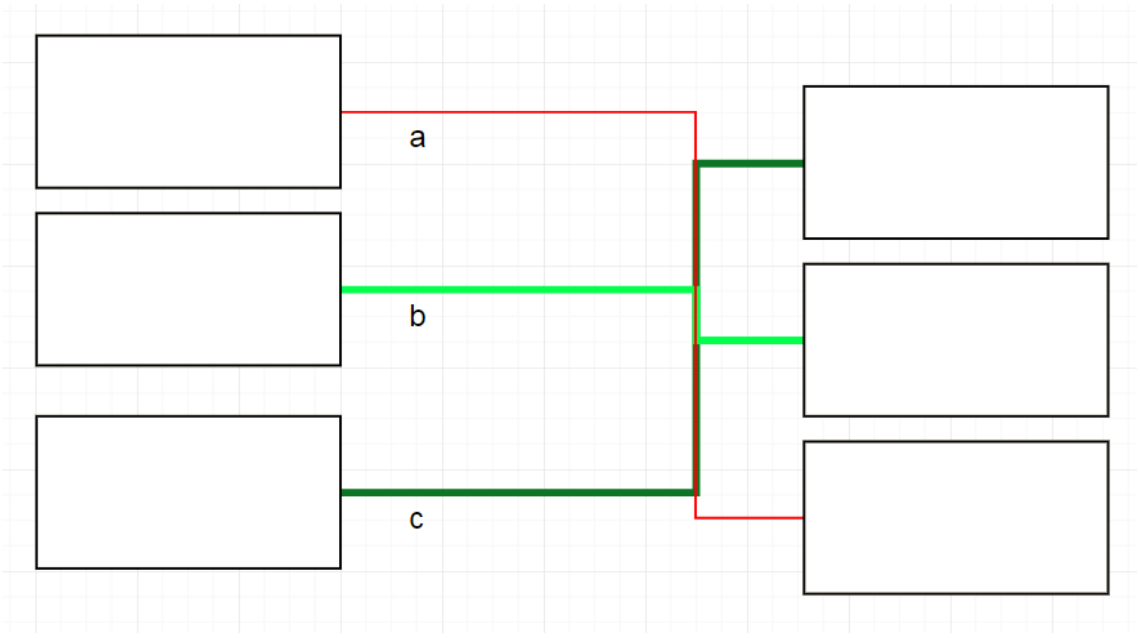


***Figure 3.9***. *Three routes a, b and c with segments 1 to 3 from left to right.*

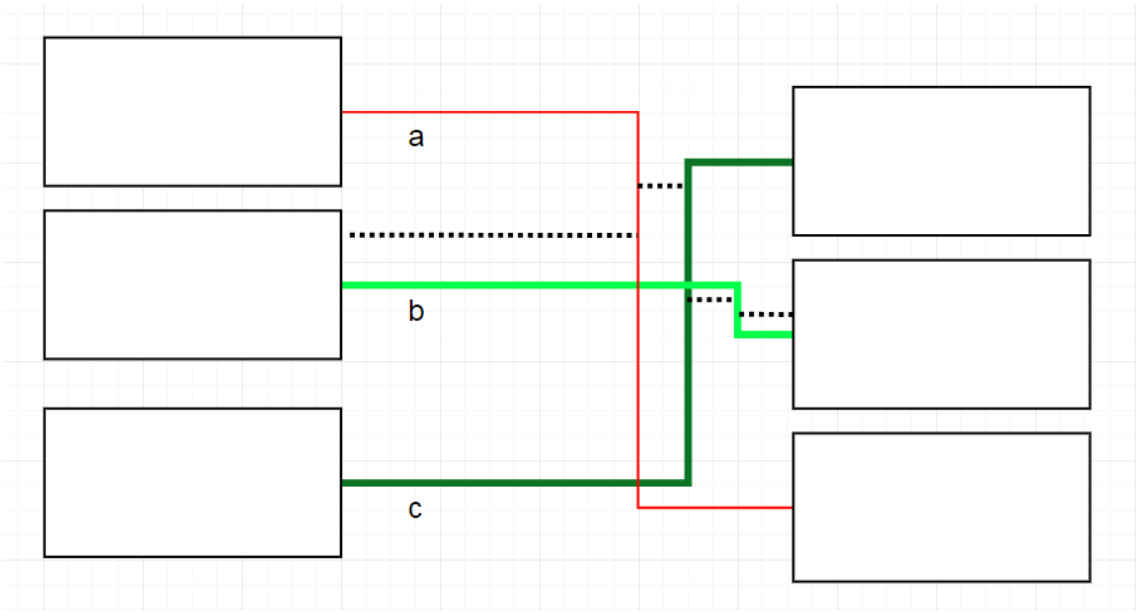


***Figure 3.10***. *Routes a, b and c nudged apart with the constraints between them visualized by dashed lines.*

The nudging is done separately for vertical and horizontal segments so we start by separating them. We also need to recalculate nudge limits between the vertical and horizontal nudging because the nudging may affect them. The actual nudging process is done by placing so-called constraints between segments and then solving the minimum-stress state for the segment placements. By constraint, we mean a rule that some numerical values $a$ and $b$ are constrained in some way. In our case, we want segments to have a minimum gap between them so we create inequality constraints between the segment coordinate values. The values that are constrained are called variables and have a weight in addition to the value. The weight is used in constraint solving. In Figure 3.9 we have three routes labeled $a$, $b$ and $c$. We now refer to each segment as $r_i$, where $r$ is the route label and $i$ the segment number from left to right, and to the x-coordinate of a segment as $r_i.x$. In the figure, let's say that we want the vertical segment $a$ to be at least 10 pixels left of the vertical segment $c$, we create the following inequality constraint:

$$a_2.x + 10 \leq c_2.x$$

We do not want to place constraints between every segment because that would result in a huge amount of constraints which in turn would slow down the algorithm in larger diagrams. Instead, we divide the segments in so-called bundles. A bundle contains all segments that can end up overlapping as a result of nudging. This is done by comparing each segment's boundaries and adding them to the same bundle if their boundaries overlap with the widest boundary in the bundle. This way of dividing the segments is not optimal as it usually results in an overly large bundle with segments that would not need nudging. This is one place where we could need some improvement. In Figure 3.9 we can put all the three vertical segments in the same bundle because all of them have the same boundaries.

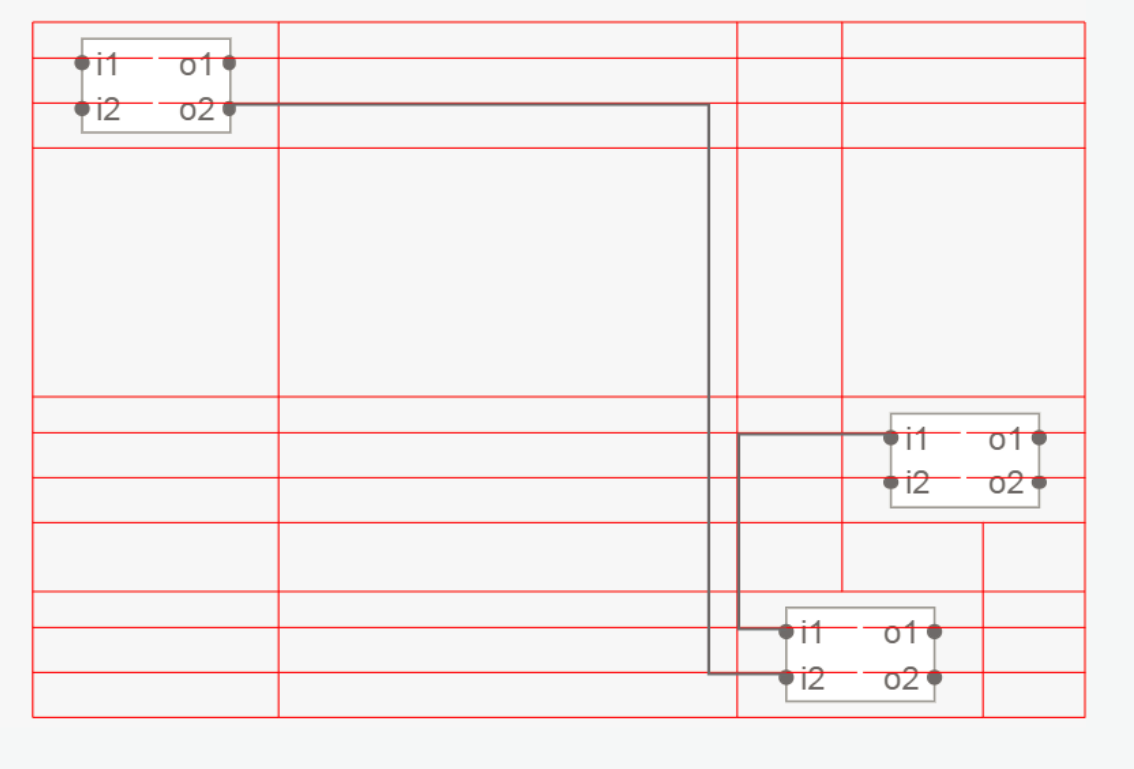


***Figure 3.11***. *Two routes where overlapping segments have been nudged apart.*

After the segments have been divided in bundles, the segments in each bundle are ordered using the nudge order calculated earlier. Then constraints are placed between the neighbouring segments in the bundle. Additionally, the leftmost segment in the bundle is constrained with the left boundary of that segment and the rightmost one with the right boundary. The boundaries are given a weight that is an order of magnitude greater than the normal weight. This way we can ensure that segments are nudged apart in the right order and no segments are nudged inside an obstacle. We also create an equality constraint between overlapping segments that belong to the same connection network. The first and last segments of a route are given a strong weight in order to keep them in place instead of them moving when some other segments are constrained to them. In the example Figure 3.9, let's say we have ordered the routes as follows: $a$ is left of $c$ is left of $b$. Then the following constraints would be created (visualized in Figure 3.10):

$$\begin{aligned} a.leftBoundary + gap &\leq a_2.x \\ a_2.x + gap &\leq c_2.x \\ c_2.x + gap &\leq b_2.x \\ b_2.x + gap &\leq b.rightBoundary \end{aligned}$$

No constraints are created for horizontal segments because all of them are first or last segments in the example and thus are not nudged at all. After the constraints for the bundle have been placed, they are solved using a constraint solver based on the algorithm by Dwyer et al. [2] and implemented in WebCola. The result is then checked if any of the strongly

weighted segments or boundaries have moved. If so, we lower the gap in the constraints that results in their movement and try to solve the constraints again. Do this until satisfied or the minimum gap size has been reached. This whole process of placing constraints and solving them is then done for each remaining segment bundle. Figure 3.11 demonstrates the end result of nudging two overlapping routes. The original route without nudging can be seen in Figure 3.6.

# 4. IMPLEMENTATION

During the thesis work, a TypeScript implementation of the connector routing algorithm was made. TypeScript was chosen as the implementation language as it was originally the plan to implement the algorithm as a part of WebCola library. However, in the end, the implementation ended up using only two parts of the library: edge ordering and the constraint solver. No other libraries were used in the implementation. The implementation consists of eight different modules: *Router, OVG, Scanline, Route, ShortestPath, Ordering, Nudging and Direction.* Their dependencies are shown in Figure 4.1.

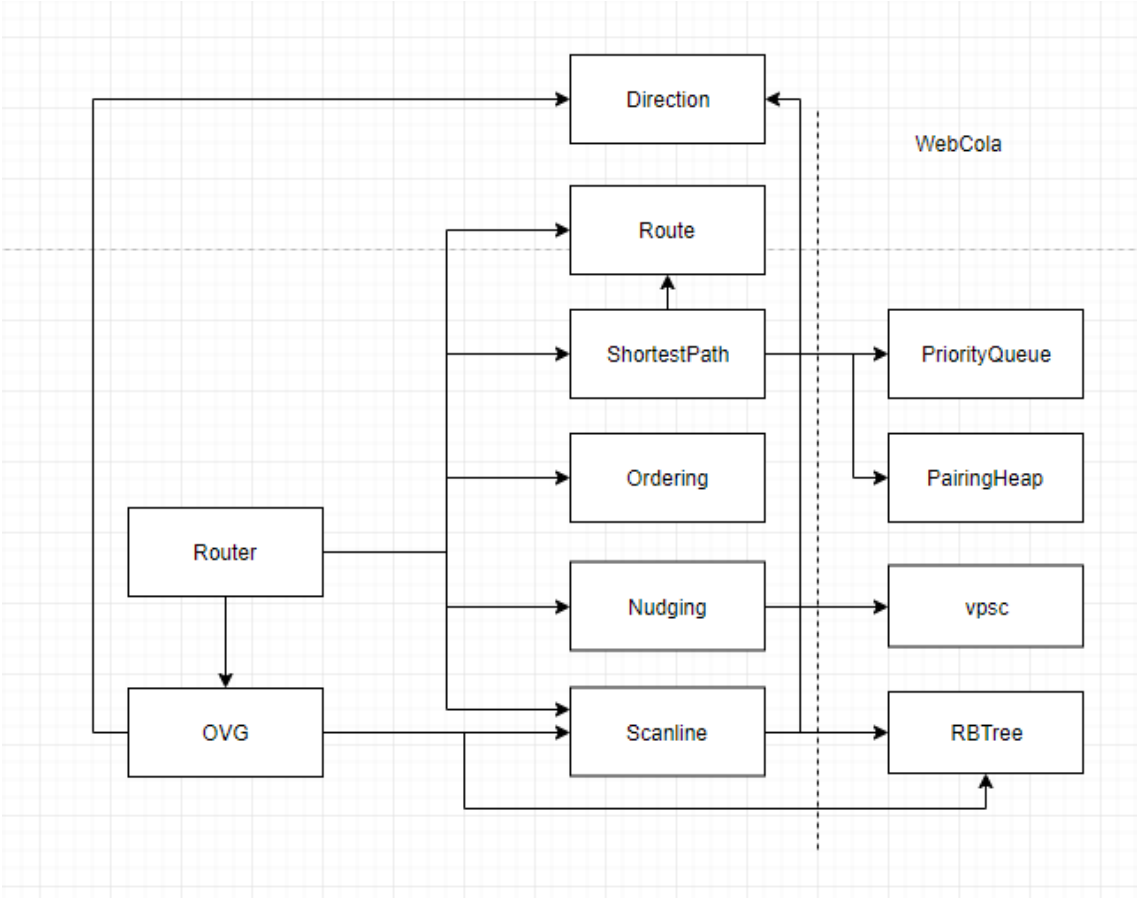


***Figure 4.1***. *Modules that comprise the implementation and their dependencies. The ordering module is shown in the prototype side because it was copied and modified.*

The whole routing application is packaged as an npm module but is currently only used internally and not published to the npm registry. After installing the npm module with `npm install connector-router` it can be imported and intialized to your application as an EcmaScript module as follows:

```
import { Router } from "connector-router";
const router = new Router(options);
```

After initializing the router object, you can start routing connectors by adding obstacles and connectors and then calling the routing methods.

```
router.addObstacles(myObstacles);
router.addRoutes(myRoutes);
router.buildOvg();
router.routeConnectors();
router.makeRoutes();
router.orderEdges();
router.nudgeRoutes();
const routes = router.getRoutes();
// Do something with the routes here
```

## 4.1 TypeScript

TypeScript was chosen as the implementation language. The decision was first made because the original plan was to implement the algorithm as part of the WebCola library. This idea was later abandoned but TypeScript still remained the language of choice. TypeScript is an open-source programming language developed by Microsoft and is used mostly for web programming. TypeScript as a language is a superset of JavaScript which means that all valid JavaScript is also valid TypeScript. As the name suggests, TypeScript adds (optional) static typing to the language. One of the goals of TypeScript's development is that it aims to follow the standardization process of JavaScript and try to keep the language's features as close to the current or upcoming standard's version as possible. TypeScript is an ahead-of-time compiled language and it is compiled to JavaScript so no additional runtime support is needed. [11]

TypeScript's type system is static and all of its type checking is done at compile time. The type system has support for all the basic features found in JavaScript, including primitives, objects, inheritance, functions and generics. It also includes the *any* type as a fallback which tells the compiler to opt-out of any type checking for that variable. However, using the *any* type is risky because all the possible type errors are now left to runtime. It is thus better to use either generics or object subtyping whenever polymorphism is needed.

TypeScript was found to be extremely helpful in finding bugs and errors as early as possible. Having all your functions and data structures typed statically gives a tremendous advantage when refactoring code. However, one of the most useful features of TypeScript's type system has been the quite recent compiler option strictNullChecks. It forces the programmer to add checks for null or undefined if a value can possibly be null or undefined. This is done by having it differentiate between types, $T$ and a union type $T|undefined$. Without strictNullChecks $T$ and $T|undefined$ are considered to be equivalent.

Another helpful feature in TypeScript's type system is its type inference capabilities. The programmer can omit type annotations from the code if the compiler can infer the types from the expressions. This removes much of the tediousness of static typing by not having to write obvious type annotations and having the compiler do it instead. However, when compared to type inference features of languages like OCaml or Haskell, TypeScript's type inference is still rather primitive.

## 4.2 Router

Router provides an interface for adding, removing, and manipulating obstacles and routes and triggering the routing. This class only holds the routing options and data structures and is only mostly glue code, delegating the calls to other modules.

Currently the following routing options can be configured. Property *bendCost* is a number value for bend penalty in A* routing. *margin* is a pair of numbers *(x, y)* indicating how much space should be left between the obstacle and connector. *margin* can be separately configured for horizontal and vertical sides of obstacles. *gap* is a number indicating how much space should be left between connector segments. The property $fixedWeight$ can be used to configure weight for boundary constraints in nudge constraint solving. Property *pruneNonConnected* is a boolean value. If true, connection points are not added to the OVG if they do not have connector connected to them. *proneNonConnected* is used for performance reasons but it is useful to be able to disable it for visualization purposes.

The router module implements the following interface and is the only public facing interface. The reason why the routing is split between *buildOVG*, *routeConnectors*, *orderEdges*, *makeRoutes* and *nudgeRoutes* is because it allows visualizing the OVG and intermediate routes after each step. It also allows easier canceling of routing in the middle of the process if obstacles or connectors have been modified. In order to use the router, one has to first add obstacles and connectors with *addObstacles* and *addConnectors* and then call the following functions in the specified order: *buildOvg, routeWires, makeRoutes, orderEdges, nudgeRoutes* and finally to get the results by calling *getRoutes*.

```
1  interface Router {
2      constructor(options: RoutingOptions): Router
3      addObstacles(obstacles: Obstacle[]): void
4      removeObstacle(obstacleIds: string[]): void
5      addConnectors(connectors: Connector[]): void
6      getObstacles(): Obstacle[]
7      getConnectors(): Connector[]
8      removeConnectors(connectorIds: string[]): void
9      getOVG(): OVG
10     buildOvg(): void
11     routeConnectors(): void
12     orderEdges(): void
13     makeRoutes(): void
14     nudgeRoutes(): void
```

```
15      getRouteById(id: string): Route
16      getRoutes(): Route[]
17  }
```

Functions *addObstacles, removeObstacles, addConnectors, removeConnectors* are used for adding and removing obstacles and connectors. Structure of the Obstacle and Connector objects are defined below. Adding an obstacle or connector that has the same id as an existing one can be used to modify that obstacle or connector.

```
1   interface Obstacle {
2       id: string;
3       x: number;
4       X: number;
5       y: number;
6       Y: number;
7       points: ConnectionPoint;
8   }
9   interface Connector {
10      id: string;
11      source: string;
12      target: string;
13  }
```

An obstacle can have connection points on its left and right side. We decided to include them in the *Obstacle* definitions because in our demo application they are always located on sides of obstacles. The structure of the *ConnectionPoint* objects is defined below. They have their own id that is referenced by the *Connector* source and target properties. The side property indicates on which side of the block the connection point is located and the offset tells the location in y-dimension in number of pixels.

```
1   interface ConnectionPoint {
2       id: string;
3       side: "left" | "right";
4       offset: number;
5   }
```

Function *buildOVG* creates an OVG using the added obstacles. If *pruneNonConnected* is enabled all connection points that are not referenced by any connector are excluded from the OVG. The OVG is generated using the procedure explained in detail in Section 3.1

The function *routeConnectors* requires buildOVG to be called first so that the OVG is built. BuildOVG runs the A* algorithm using the bendPenalty defined in options. The A* is implemented using a pairing heap priority queue to store open neighbours. The priority queue data structure is taken from the WebCola library.

The function *makeRoutes* is called after routeConnectors. It creates segments from list of points created by routeConnectors. Routes created by routeConnectors contain vertices of OVG and this method simplifies these routes by eliminating all the intermediate vertices from each segment, only retaining the start and end vertices of each segment. After simplifying, boundaries as explained in section 3.3 are calculated and then segments are placed in the middle of the available space between the boundaries. This ensures that as many segments as possible can be taken into account in the next phase where routes are ordered. This is because it relies on overlapping segments. Lastly, points where overlapping routes diverge from each other are added to each route pairs as the Longest Common Subsequence algorithm used by the ordering needs the routes to contain all the common points.

The function *orderEdges* requires *makeRoutes* to be called first. The ordering algorithm is provided by the WebCola library with a little modification that allows a custom comparison function for the Longest Common Subsequence algorithm that is used in the implementation.

The function *nudgeRoutes* follows the approach described earlier in section 3.3. All segments in all routes are split to vertical and horizontal segments which are processed separately. As the last step in nudgeRoutes, all junction points are searched from the routes that form connection networks.

The algorithm needs a way of storing rectilinear direction information in many places. The direction also needs to be manipulated efficiently, such as reversing and rotating it. In some cases, multiple directions are needed at the same time,for example representing all connectible directions of a connection point.

```
1 const enum Dir {
2         NONE  = 0b0000,
3         NORTH = 0b1000,
4         EAST  = 0b0100,
5         SOUTH = 0b0010,
6         WEST  = 0b0001
7 }
```

A four-bit field was chosen for this, represented as number in TypeScript as there is no better type for it. The choice of bit field gives us an interesting way of defining operations on it. Rotating the direction left or right becomes right or left bit rotation; and reversing becomes swapping high and low bits.

## 4.3 Challenges in implementation

Performance was a concern as JavaScript, which TypeScript is compiled to, has limited performance, especially as it is critical for it to be fast enough to be used interactively. TypeScript also lacks many basic data structures as it doesn't have a standard library. Thankfully,

most of the basic algorithms and data structures that were needed were implemented in the WebCola library that was used; including red-black tree, priority queue, pairing heap, vpcs (variable placement with separation constraints) and the longest common subsequence. On the other hand, vpsc and longest common subsequence are very specific algorithms that are hardly available in any language's standard libraries. In this perspective, TypeScript was a good language choice in terms of having most of the needed algorithms available in just one library.

The algorithm that was implemented was also somewhat hard to understand from the research paper alone and many details required reading the reference implementation called libavoid. This was especially true with the corner cases and small details in getting the nudging step working correctly.

# 5. EVALUATION

## 5.1 Routing quality

Evaluating the produced routings with the defined criteria was difficult. To be able to get any meaningful comparison of produced routes, you would need to compare the results to another implementation for meaningful results. This is, however, not relevant for this thesis, as the only reference implementation which would be somewhat feasible to compare to is libavoid because it uses the same algorithm as our implementation but has more optimizations. Also because of licensing and platform requirements there were no comparable implementations. Because of this, we are satisfied with a subjective analysis that the produced routing quality is good enough for this purpose.

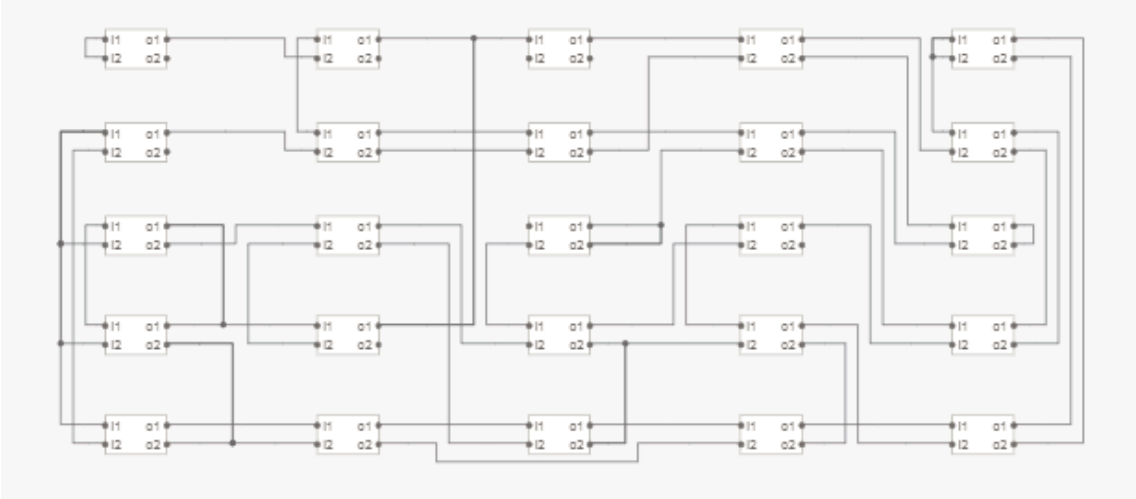

***Figure 5.1***. *Grid layout and routed connectors used in measurements.*

## 5.2 Shortcomings

Our implementation has several shortcomings in different areas. The connection network implementation depends on multiple equality constraints to get multiple point to point routes to same positions. This creates a large amount of additional constraints which in turn increases the execution duration. It also doesn't optimize the connection network globally, which results in redundant segments and loops in larger networks. This is a temporary solution that was easy to implement in the given timeframe and will be replaced by a more sophisticated algorithm in future. The algorithm planned for this is shortly explained in Section 6.3.

Overlaps between segments are created in situations where we would need to nudge the first or last segments of a route with other segments demonstrated in Figure 5.2. This is

because the first and last segments are never moved from their initial positions to keep them in the original connection point. To fix this problem, the first and last segments could be split with a joint segment between them. This, however, has not been done in the current implementation.

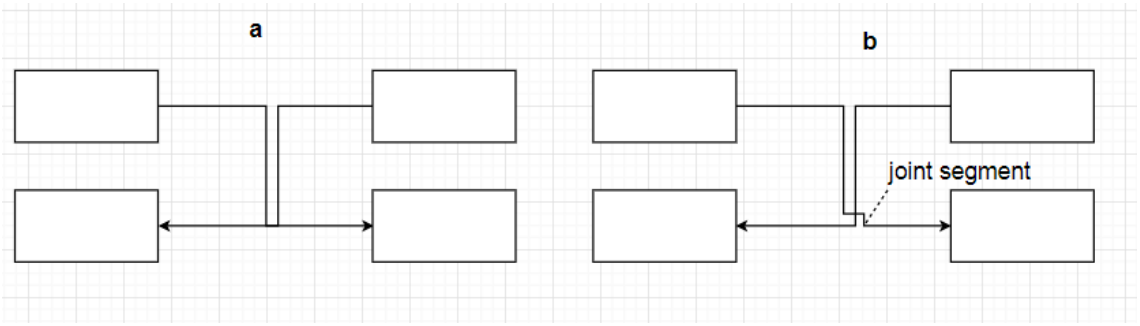


***Figure 5.2***. *(a) Two fixed segments overlapping. (b) Proposed fix to split the first and last segment with a joint segment to allow nudging the segments.*

The current implementation does not properly handle overlapping blocks. This is only a problem if the routing is used in an environment where the user can freely place the blocks. Even in that situation, overlapping blocks are probably mostly created while rearranging blocks, thus temporarily creating an overlap situation. Because of this, we need to handle overlaps in a predictable manner so that the user does not notice that something abnormal is happening while he or she is moving the blocks. Figure 5.3 demonstrates the resulting orthogonal visibility graph (OVG) in the current implementation. In short, edges are not drawn over the intersection where the blocks overlap, making a stack of overlapping blocks one large obstacle from the OVG point of view. This is favourable in cases where you don't want for connectors to go through a column of stacked blocks but can create situations where all possible routes from another connection point are blocked. One way to tackle this problem would be to add an additional cost penalty in the graph depending on how much nudging space that route uses. It would favour routes that have a large amount of nudging space and would decrease the amount of routes which would only have a minimum amount of nudging space.

This would also mitigate the next problem where route quality starts to suffer when there is not enough nudging space available. This problem is shown in Figure 5.5.

Another problem, similar to the connection network shortcoming, is the routing related connections. As demonstrated in Figure 5.4, you can see two problems with the current implementation. Firstly, connectors that are related to each other, in this case by having the same blocks as start and end, are routed with a completely different topology. And secondly, unrelated connections are nudged in the same bundles, which makes it hard to differentiate them.

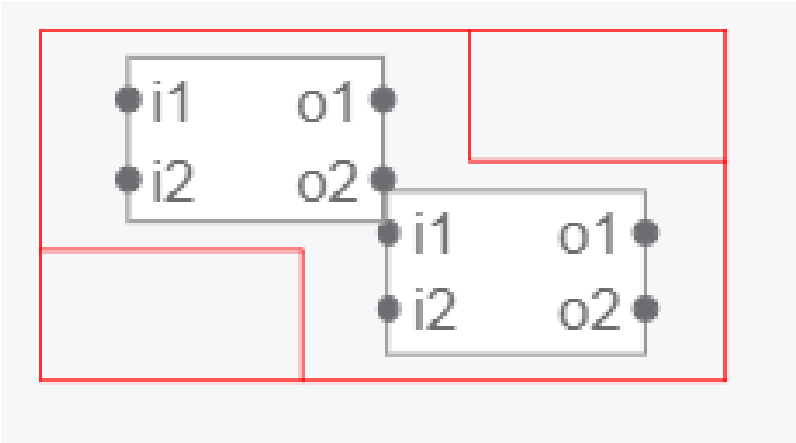


***Figure 5.3***. *OVG of two overlapping blocks. Connection points are not included in this OVG because the connection points are not connected.*

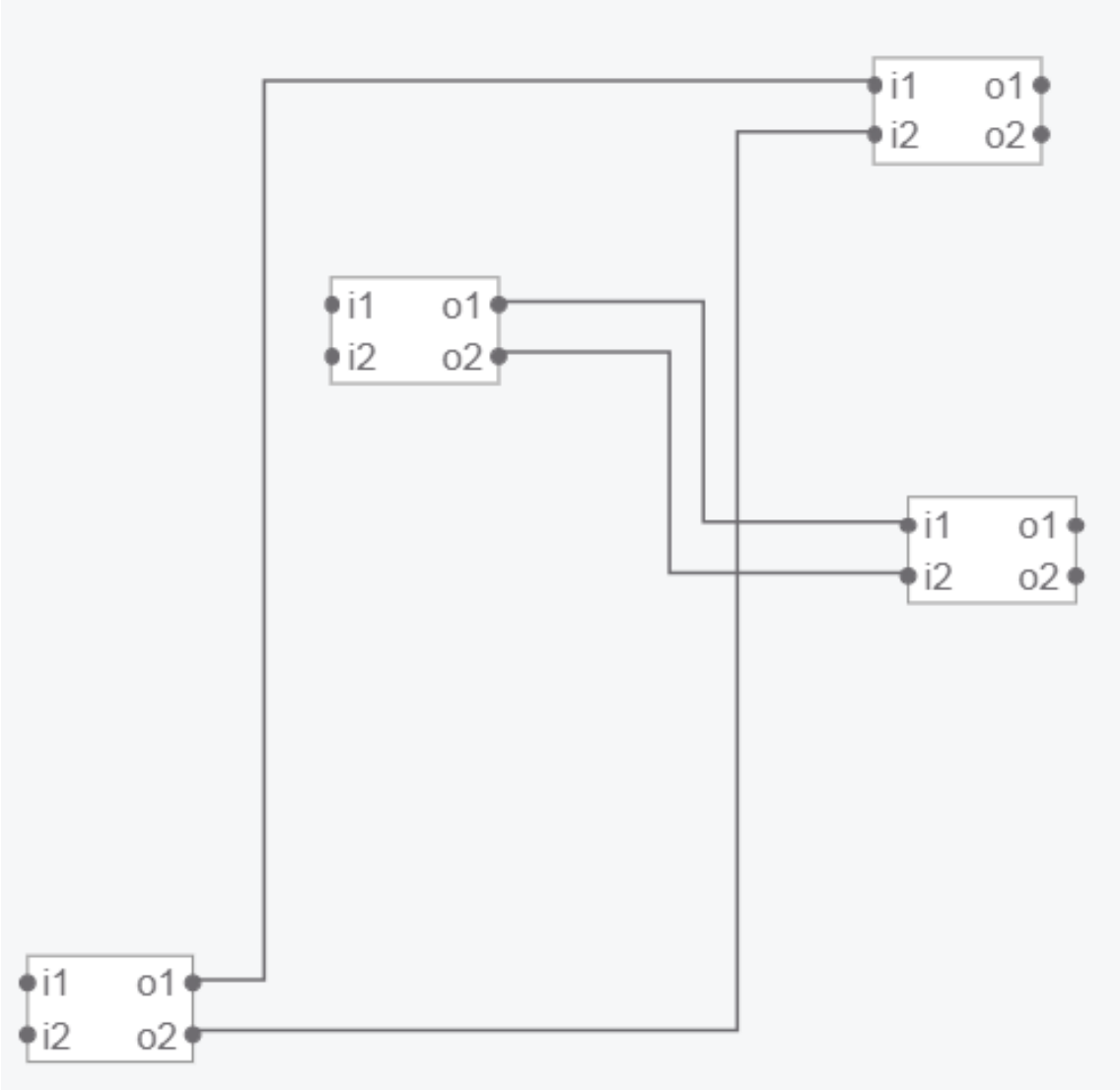


***Figure 5.4***. *The related connectors shortcoming: both connectors from the lower left block should be routed close to each other.*

## 5.3 Performance

Performance was measured by preparing five diagrams with a different number of blocks, connections, and layout. The hardware used in the measurement was a Lenovo W541 laptop with 2.8GHz Intel i7-4810MQ and 16GB of memory. Measurements were done with Chrome browser performance API and Google Chrome version 65.0.3325.146. Because of recent security vulnerabilities in CPUs, browser vendors had to lower the resolution of these performance measurement APIs to mitigate timing based attacks [7]. Because of this, all measurements have the maximum resolution of 20 microseconds.

The measurement procedure was as follows: the duration of each step of the routing was measured and stored in an array. After the diagram was loaded, one block from the diagram was moved repeatedly causing connector routing to recalculate the routes. This was repeated until 100 measurements were completed. Finally, the total duration for each measurement was calculated after which we calculate minimum, maximum, mean, and average for each duration array.

### 5.3.1 Impact of diagram size

The impact of the increased diagram size was measured by creating a diagram with a symmetrical grid layout consisting of 25 blocks and 50 connections shown in the figure 5.1. From now on we use the notation of <amount of blocks>x<amount of connections> to describe diagram sizes. For the larger versions of the same layout, the same diagram was copy-pasted side by side two times for the medium size and four times for the large size. It is important to use the exact same layout in the measurements because different layouts perform differently as can be seen from the layout comparisons in the next section.

| Layout | Blocks | Connections | | Graph | Routing | Ordering | Nudging | Total |
|---|---|---|---|---|---|---|---|---|
| Grid | 25 | 50 | min | 1.3 ms | 1.1 ms | 1.0 ms | 1.0 ms | 4.6 ms |
| | | | max | 9.8 ms | 5.7 ms | 5.9 ms | 10.8 ms | 25.3 ms |
| | | | mean | 1.5 ms | 1.5 ms | 1.1 ms | 1.2 ms | 6.3 ms |
| | | | avg | 2.0 ms | 2.0 ms | 1.0 ms | 2.0 ms | **7.0** ms |
| Grid | 50 | 100 | min | 4.0 ms | 4.7 ms | 4.0 ms | 2.4 ms | 15.4 ms |
| | | | max | 13.3 ms | 14.9 ms | 15.0 ms | 32.5 ms | 55.1 ms |
| | | | mean | 4.4 ms | 5.1 ms | 4.3 ms | 2.8 ms | 18.5 ms |
| | | | avg | 5.0 ms | 5.0 ms | 5.0 ms | 4.0 ms | **20.0** ms |
| Grid | 100 | 200 | min | 14.3 ms | 18.6 ms | 17.3 ms | 7.4 ms | 60.7 ms |
| | | | max | 59.1 ms | 35.9 ms | 38.2 ms | 24.8 ms | 121.5 ms |
| | | | mean | 16.8 ms | 20.5 ms | 19.9 ms | 8.5 ms | 65.6 ms |
| | | | avg | 18.0 ms | 22.0 ms | 20.0 ms | 10.0 ms | **70.0** ms |

From these measurements we can see that the average total duration seems to behave polynomially. A large deviation between the minimum and the maximum can also be observed, but this can be explained by garbage collector triggering. These also seems to be quite rare by because the average is much closer to the minimum than to the maximum.

From these results we can safely say that the performance is quite adequate for interactive editing especially in diagrams that are 25x50 or smaller. If we set 60 frames per second as

our target, 20 milliseconds take 20% of the frame budget, which is still a large amount, but depending on the application, might still be feasible.

### 5.3.2 Impact of different layouts

In addition to measuring the effect of an increased diagram size, different layouts with a fixed number of blocks and connectors were used in measurements. The following three layouts were used.

A simple layout (Figure5.1) has mostly point to point connections and connections that flow from the left to the right.

A compact layout (Figure5.5) is mostly the same but more compact, thus having less space for the connections.

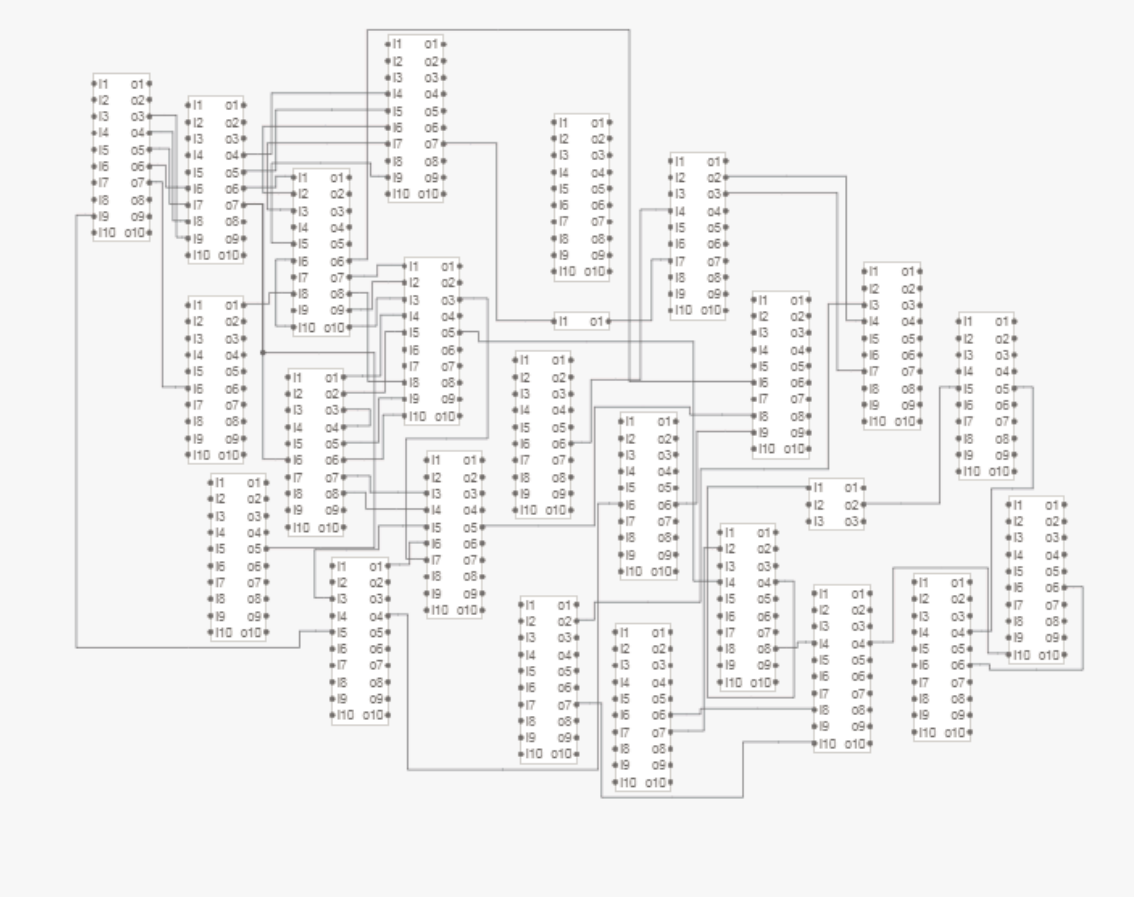

***Figure 5.5***. *Compact layout used in measurements*

Lastly, the networks layout (Figure 5.6) has mostly connection networks instead of point-to-point connections, which allows measuring the impact of network constraints.

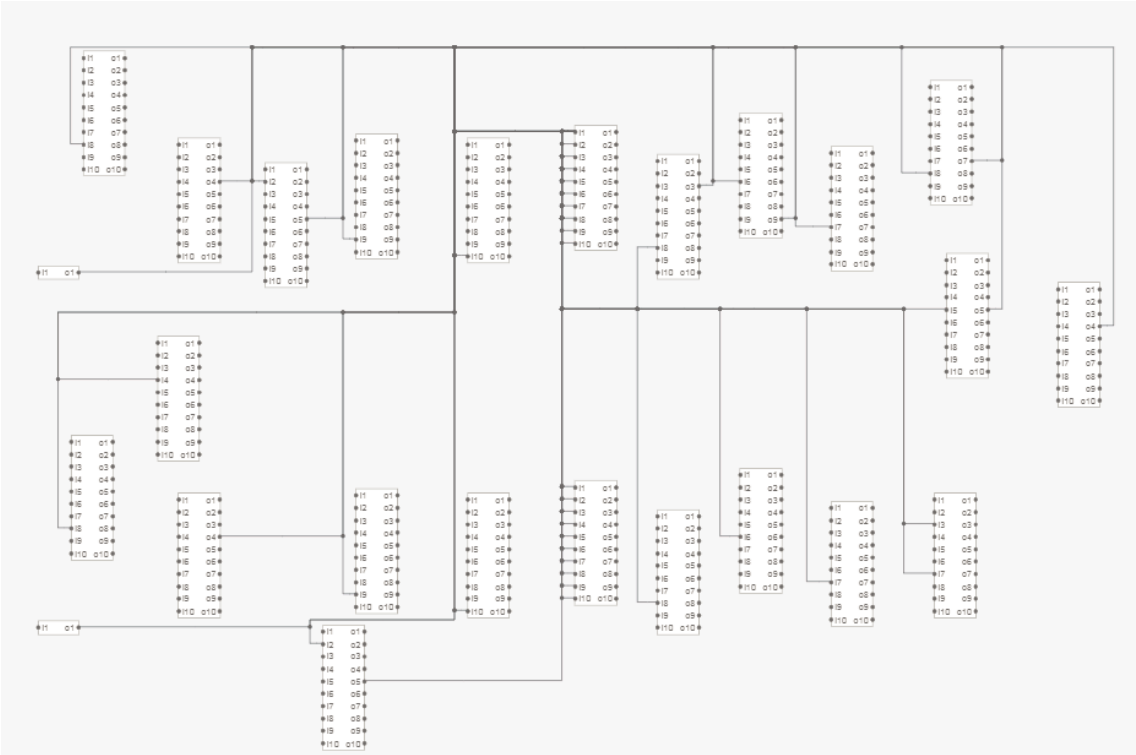

***Figure 5.6***. *Networks layout used in measurements*

| Layout | Blocks | Connections | | Graph | Routing | Ordering | Nudging | Total |
|---|---|---|---|---|---|---|---|---|
| Grid | 25 | 50 | min | 1.3 ms | 1.1 ms | 1.0 ms | 1.0 ms | 4.6 ms |
| | | | max | 9.8 ms | 5.7 ms | 5.9 ms | 10.8 ms | 25.3 ms |
| | | | mean | 1.5 ms | 1.5 ms | 1.1 ms | 1.2 ms | 6.3 ms |
| | | | avg | 2.0 ms | 2.0 ms | 1.0 ms | 2.0 ms | **7.0** ms |
| Compact | 25 | 50 | min | 1.9 ms | 3.1 ms | 2.0 ms | 2.5 ms | 9.7 ms |
| | | | max | 12.7 ms | 19.4 ms | 35.1 ms | 12.2 ms | 54.2 ms |
| | | | mean | 2.2 ms | 3.7 ms | 2.1 ms | 3.2 ms | 13.1 ms |
| | | | avg | 3.0 ms | 5.0 ms | 3.0 ms | 4.0 ms | **14.0** ms |
| Networks | 25 | 1 network<br>50 terminals | min | 1.2 ms | 6.2 ms | 4.8 ms | 10.9 ms | 24.8 ms |
| | | | max | 8.8 ms | 26.4 ms | 13.9 ms | 39.5 ms | 73.4 ms |
| | | | mean | 1.4 ms | 8.0 ms | 5.1 ms | 12.6 ms | 28.3 ms |
| | | | avg | 2.0 ms | 9.0 ms | 6.0 ms | 13.0 ms | **30.0** ms |

The difference in the execution time between the grid and compact layout in each step of the algorithm is between *0.5x to 3x* with a difference of *2x* in the total duration. This can be explained by the larger visibility graph in the compact layout caused by the more dense layout and asymmetric block placement. This, in turn, affects routing by having a larger search space in the A* phase. Increase in ordering can be explained by observing the diagram and noticing that it has a larger number of overlapping connector segments, which in turn is caused by having smaller gaps between obstacles. This means that we have to calculate more orderings between connectors. Lastly, we have the increase in the nudging step: the increase in this can also be tracked down to same reason as in the ordering phase – more overlapping segments that need nudging.

Grid and network layouts, on the other hand, have no meaningful difference in graph

building duration but have a noticeable difference in routing, ordering and nudging. When you compare the compact and networks diagrams, you can notice that the routes in the networks diagram are much longer, which affects the duration of A* routing. The **6x** increase in the ordering duration is most likely caused by the network diagram mostly having overlapping segments. And lastly, the increase in nudging can be explained by the same reason. Additionally, when nudging overlapping segments that belong to the same connection network, an equality constraint is placed between the overlapping segments keeping them in the same location. These additional constraints also explain the increased duration. In total, we can see a **6x** increase between grid and network layouts.

From these results we can see that both the layout and diagram size is a factor in the total routing duration of the connectors, with a difference over **4x** between the simplest and most complex layout that have the same number of blocks and connectors.

# 6. FURTHER DEVELOPMENT

## 6.1 Extensibility

The algorithm's three phase structure makes it easy change parts of it to another implementation if its output satisfies the next step's expected input. Also, an arbitrary number of post-processing steps can be added. However, no API for extensions were made in the prototype as it was not needed nor was seen feasible for the scope of this work. TypeScript's type system could be leveraged more extensively to bring benefits of static typing. This would allow it to be used as a library more easily in different use cases.

## 6.2 Combining manual routing with automatic routing

Currently the algorithm only allows for fully automatically routed connectors as it was originally intended. In some situations, however, it could be beneficial for the user to have some control over how the connector is routed, e.g. manually correcting some shortcomings of the automatic routing. The user could, for example, add checkpoints for a connector without bringing too many new concepts into the algorithm.

When building OVG, you could add a list of checkpoints which are then processed by the line sweep to generate horizontal and vertical segments of the OVG. The checkpoint would contain the id, coordinates, and the id of the connector it is related to.

Routing connectors in the OVG needs to route from the start through each checkpoint to the end. However, adding checkpoints modifies the graph and will affect routing other connectors indirectly even if routing through other connectors checkpoints is disallowed. This may mean that the checkpoint should be added to the graph in another way. The steps that come after the routing do not need to take the checkpoints in to account.

## 6.3 Improved connection networks with rectilinear steiner tree heuristic

The algorithm used does not model connection networks very well. We have a local optimization which gives us acceptable route quality in simple cases. Orthogonal hyperedge routing from Wybrow's 2012 paper [18] could be utilized to improve the connector network quality.

The algorithm works by starting to build minimum cost trees from each network terminal. This is done by traversing the graph by Dijkstra's algorithm. When two of these trees reach

each other, the connecting edge is marked as the bridge edge and tree building continues without traversing anymore in to the other tree's area. This process continues until the new tree edge cost is more than double of the bridge cost. At this point we commit to the bridge and connect these two terminals by taking the shortest path to the bridge edge from both trees. After that, we forget other branches from the tree and start this process again by starting the tree-growing from every point from the committed path. This process is repeated until all terminals are connected at which point we have the complete network.

To be able to use this approach, we would need to represent connectors as a set of terminals instead of a tuple of start and end. We would also need to adapt our edge ordering and nudging to work on tree structures instead of paths. This on top of the work of implementing the above-mentioned algorithm, would need large amounts of work so it was left out from this implementation.

Most parts of this algorithm were implemented as of June 10, 2018 but not yet integrated to the rest of the prototype. The more recently implemented parts have not been evaluated at the time of writing.

## 6.4 One-bend graph

The OVG is not optimal in every way to represent the diagram. One of its weaknesses is that it contains multiple routes between two connection points that have an equal cost as in Figure 6.1. If we could remove all these kind of equal cost routes from the OVG, we could achieve much better performance in the routing step.

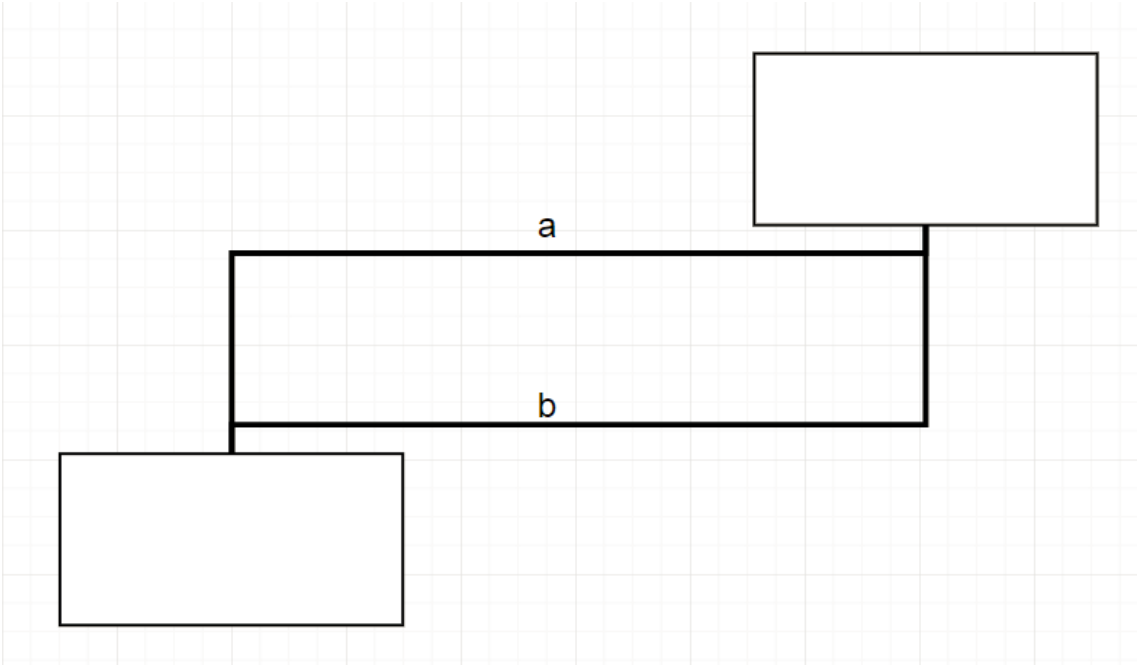


***Figure 6.1***. *Two routes, a and b, that have equivalent cost.*

Fortunately Marriot et al. have published an algorithm in a paper Seeing Around Corners: Fast Orthogonal Connector Routing [10], which tries to solve this problem with a so-called 1-bend graph which is an optimization of the OVG. The one-bend graph tries to remove all *topologically* equivalent routes from the OVG, thus potentially removing considerable of edges from the graph. The paper notes that the same effect could be achieved by adding additional route pruning to A*.

# 7. CONCLUSION

Diagram editors usually require users to draw connectors lines manually or provide limited automatic routing capabilities. We reviewed a few solutions that provide better routing quality but were unable to utilize them due to restrictive licensing and not being able to use them in a web-based application. However, one of the reviewed solutions was based on a promising algorithm that had enough research material on it so it was decided to implement it using TypeScript.

The orthogonal connector routing algorithm has many benefits. It produces predictable routes, which means that the user can reason its behavior and it is not sensitive to the order in which objects and connectors are placed [17]. The routing quality is sufficient by many of the criteria mentioned earlier. It minimizes both connector length and bend count, does some local crossing minimization, tries to produce symmetric routes and does not allow routes to overlap other routes or obstacles. However, it does not minimize global connector crossings, the algorithm only makes sure that it does not introduce any unnecessary additional crossings while nudging routes. Some local connector crossing reduction could be achieved by adding a penalty to the A* cost function [17]. However, minimizing crossings globally would be hard. The algorithm is also efficient enough for interactive editing even in larger diagrams [17].

One drawback of the approach is that the algorithm does not model connection networks, i.e., connections with more than two endpoints, very well because it can only minimize the cost of a single route at a time. This will lead to a potentially very suboptimal outcome if a diagram contains multiple connection networks. A solution to this problem could be to allow the user-defined junction points, i.e., points in the connection network where the network forks, which could be placed freely in the diagram. This would introduce some of the problems which we tried to solve in the first place but could improve the quality of routes in difficult cases. Another solution to this is proposed by Wybrow et al. in Hyperedge Routing [18], which introduces a similar algorithm to minimize the connection network's overall cost. This algorithm was also explained in Chapter 6 Further development and is already partially implemented in the prototype.

The algorithm was implemented using TypeScript as the language. The language choice was considered to be an improvement over JavaScript in terms of easier maintenance and refactoring. The implementation was not completely faithful to the original algorithm in all areas. Especially nudging was not described with sufficient details for me to be able to implement it in the same manner. Some other details were also done differently or were not implemented. However, the prototype implemented has sufficient performance for interactive editing with the required 100 blocks and 200 connectors and has received

mostly positive feedback from test users. The negative points that users have reported were covered in the Section 5.2 Performance.

# REFERENCES


[1] T. Biedl, B. Madden, I. Tollis, The three-phase method: A unified approach to orthogonal graph drawing, in: Graph Drawing, Springer, 1997, pp. 391–402.

[2] T. Dwyer, K. Marriott, P.J. Stuckey, Fast node overlap removal, in: International Symposium on Graph Drawing, Springer, 2005, pp. 153–164.

[3] A. Haas, A. Rossberg, D.L. Schuff, B.L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien, Bringing the web up to speed with webassembly, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2017, pp. 185–200.

[4] M. Hanan, On steiner's problem with rectilinear distance, SIAM Journal on Applied Mathematics, Vol. 14, Iss. 2, 1966, pp. 255–265.

[5] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, Systems Science and Cybernetics, IEEE Transactions on, Vol. 4, Iss. 2, 1968, pp. 100–107.

[6] D.E. Knuth, A generalization of dijkstra's algorithm, Information Processing Letters, Vol. 6, Iss. 1, 1977, pp. 1–5.

[7] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, arXiv preprint arXiv:1801.01203, 2018.

[8] C.Y. Lee, An algorithm for path connections and its applications, IRE transactions on electronic computers, Iss. 3, 1961, pp. 346–365.

[9] D. Lee, C.D. Yang, C. Wong, Rectilinear paths among rectilinear obstacles, Discrete Applied Mathematics, Vol. 70, Iss. 3, 1996, pp. 185–215.

[10] K. Marriott, P.J. Stuckey, M. Wybrow, Seeing around corners: Fast orthogonal connector routing, in: Diagrammatic Representation and Inference, Springer, 2014, pp. 31–37.

[11] Microsoft, TypeScript language web page, 2018. Available (accessed on 27.7.2018): https://www.typescriptlang.org/

[12] K. Miriyala, S.W. Hornick, R. Tamassia, An incremental approach to aesthetic graph layout, in: Computer-Aided Software Engineering, 1993. CASE'93., Proceeding of the Sixth International Workshop on, IEEE, 1993, pp. 297–308.

[13] H. Purchase, Which aesthetic has the greatest effect on human understanding?, in: International Symposium on Graph Drawing, Springer, 1997, pp. 248–261.

[14] H.C. Purchase, J.A. Allder, D. Carrington, User preference of graph layout aesthetics: A uml study, in: Graph Drawing, Springer, 2001, pp. 5–18.

[15] Y.F. Wu *et al.*, Rectilinear shortest paths and minimum spanning trees in the presence of rectilinear obstacles, IEEE Transactions on Computers, Vol. 100, Iss. 3, 1987, pp. 321–331.

[16] M. Wybrow, Using semi-automatic layout to improve the usability of diagramming software, dissertation, Monash University, 2008.

[17] M. Wybrow, K. Marriott, P.J. Stuckey, Orthogonal connector routing, in: Graph Drawing, Springer, 2010, pp. 219–231.

[18] M. Wybrow, K. Marriott, P.J. Stuckey, Orthogonal hyperedge routing, in: Diagrammatic Representation and Inference, Springer, 2012, pp. 51–64.