Optimizing Orthogonal Edge Routing for Microfluidic Biochip Layouts

Bachelor Thesis of

Ruben Gehring

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dr. Martin Nöllenburg
	Dipl. Inform. Thomas Bläsius

Time Period: 1st February 2014 – 31st May 2014

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 4th June 2014

Abstract

Microfluidic biochips, also called "lab-on-a-chip", are currently rising in popularity. This is mostly because they allow biochemical experiments to be carried out on a single, small chip which has a number of advantages.

However, the design of these chips is still a time-consuming and expensive process, mainly because component layouts for these chips are still done manually. This is because there is no software tool that can provide efficient automatic component placement and edge routing just yet.

Ideally the chip engineer should just have to select which components should be used for the current chip design, and which ones are connected. Then the software should take care of details such as where to place the components and how to route the edges in an efficient manner. But no software can offer this feature without a component placement and edge routing algorithm, the latter of which we want to present in this thesis.

Deutsche Zusammenfassung

Mikrofluidische Biochips, auch "lab-on-a-chip" genannt, steigen momentan in ihrer Beliebtheit. Der Hauptgrund hierfür ist, dass sie biochemische Experimente auf einem einzigen, kleinen Chip erlauben, was einige Vorteile hat.

Allerdings ist der Entwurf dieser Chips noch immer ein zeitaufwändiger und teurer Prozess, hauptsächlich weil das Komponentenlayout noch komplett von Hand gemacht werden muss. Es gibt noch keine Entwicklungswerkzeuge, welche in der Lage wären, die Komponenten und Strecken automatisch und effizient anzuordnen.

Idealerweise sollte ein Entwickler so eines Chips nur angeben müssen welche Komponenten für den aktuellen Entwurf verwendet werden sollen und wie diese verbunden sind. Das Entwicklungsprogramm sollte sich dann darum kümmern, wie die Komponenten angeordnet werden sollten, und was eine effiziente Streckenführung ist. Aber kein Programm kann so eine Funktionalität bereitstellen, ohne einen Algorithmus für die Komponentenanordnung und die Streckenführung zu haben. Letzteres wollen wir in dieser Abhandlung vorstellen.

Contents

1. Introduction

Microfluidic biochips are one kind of what is referred to as “lab-on-a-chip” or “LOC”. The idea behind LOCs is to integrate one or several laboratory functions onto a small chip of a few square centimetres in size. While there are also LOCs that are based on microelectromechanics, the kind of chips that motivated this thesis are the so called microfluidic biochips. These can be used for a variety of purposes, e.g. one can design a chip that, given only 1 μl of blood, can test it for various diseases like HIV and syphilis without the need of a sterile lab with big and expensive equipment [CLC+11]. Other advantages, besides being much easier to use in the field and a low sample consumption, can be high throughput due to massive parallelization, short response and analysis times, better process control and the ability to mass produce those chips making them cheap [RIBT13].

Essentially a microfluidic biochip consists of components that provide the chips functionality and a network of connections. Which components are exactly used depends on the application, the most universal ones being pumps, storages and mixers that are used on almost any kind of chip. These components are usually made by creating specific shapes in the chip material that provide the desired functionality (see also Figure 1.1). Some components, like pumps, also require correctly placed valves to work.

So far, chip designers still have to place the components and route the edges connecting them with minimal software support. Even with a given placement for the components it is difficult to find an efficient, meaning cheap, routing for the connections. It also is very time consuming if done manually.

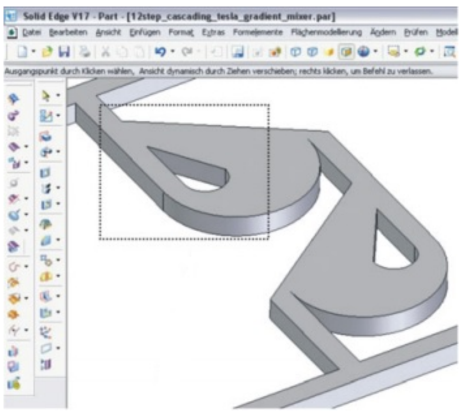

Solid Edge V17 - Part - [12step_cascading_tesla_gradient_mixer.par]
Datei Bearbeiten Ansicht Einfügen Format Extras Formelemente Flächenmodellierung Ändern Prüfen Model
Ausgangspunkt durch Klicken wählen, Ansicht dynamisch durch Ziehen verschieben; rechts klicken, um Befehl zu verlassen.

Figure 1.1.: 3D CAD model of a mixer. [WCH+13]

Right now there are no algorithms for the edge routing problem that we are aware of. Developing one is the first step towards software tools that handle these matters automatically, saving the developers time while also increasing the efficiency of the solution. This thesis addresses the problem of finding an efficient routing to a given initial component placement, as stated above.

When comparing two different, fully functional, routings that both connect the same components of the same initial placement, we assume that the routing that is cheaper to produce is always preferable. The production cost of a routing is essentially the number of valves needed, therefore we want to create routings with as few crossings as possible.

The approach taken for this is similar to the approach of Wybrow et al. [WMS10]. They present an algorithm for minimizing the number of bends and the connector's length in an orthogonal connector drawing for diagram editors. Their intention is to clearify and simplify network diagrams. They do this by creating an Orthogonal Visibility Graph (OVG), use A*-search on this graph to find an ideal route and lastly nudge shared edges apart for clarification.

Tseng et al. [TYLH13] introduce an incremental cluster expansion algorithm for component placement on microfluidic biochips. The idea of this algorithm is to create clusters by taking a storage and placing all components connected to this storage around it. Then they go on to connect the storages, so that every component is connected to every other component via its storage. However, this does not address the issue of how to route within these clusters. It is also limited to applications where the indirect connection via storages is not a problem because of, e.g. route length restrictions.

This thesis presents an algorithm for automated creation of cheap biochip connector routings, given an initial component placement. Together with a component placement algorithm, a software or plug-in can be created to greatly simplify and accelerate the design process of microfluidic biochips. Especially for more complex biochip layouts (as shown in Figure 1.2) such an software aide would also possibly decrease the production cost of the chip, since finding an optimized solution for such layouts would take a human chip designer far too long.

The rest of the thesis is organized as follows: In section 2 we show how we model the problem of finding a cheap routing for a biochip as graph drawing problem. We then go on to analyse the complexity in the following section 3, before explaining our algorithm in detail in section 4. Afterwards we evaluate the algorithms performance in section 5. We then end by drawing our conclusion in the last section 6.

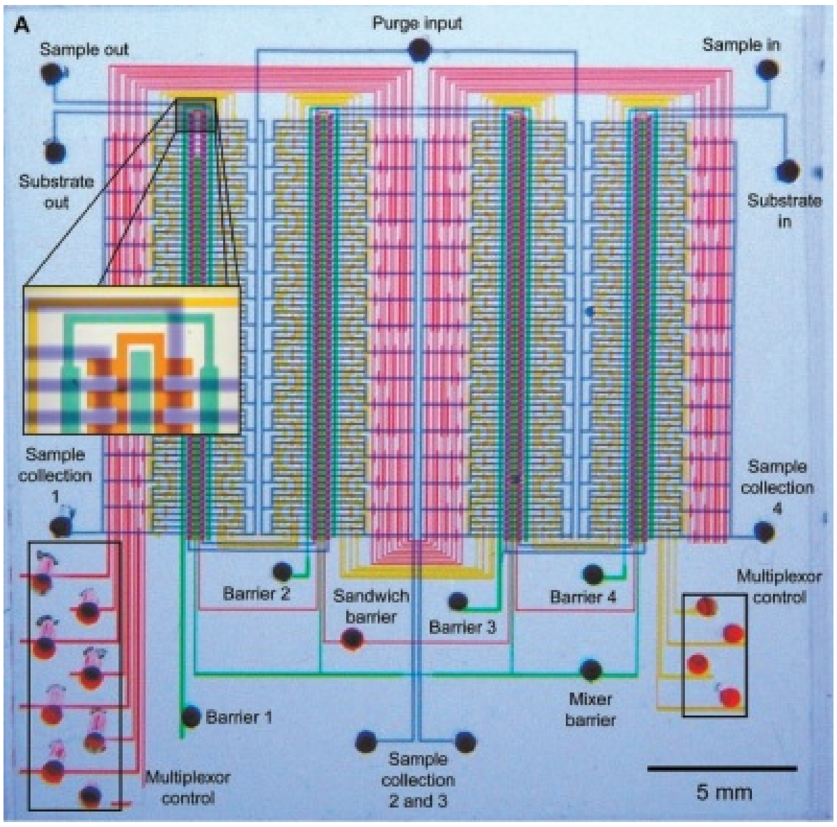


Figure 1.2.: Optical micrograph of a microfluidic comparator chip. [TMQ02]

2. Problem Modelling

While the basis and motivation of this thesis lies in microfluidic biochips, its actual content, from a computer science and graph drawing perspective, is a heuristic for routing edges with few crossings in an orthogonal graph layout. Because of this we first want to take a look at how our physical biochip problem is transformed into a more abstract model, before going on to the algorithm itself. This section contains precise problem definitions for the biochip design problem we address as well as some closely related problems. We will start by describing one problem in detail and then point out the differences to each of the related problems.

Our first problem is called the *port-to-port channel routing problem* and it is defined as follows: Our input consists of an *input graph* $G = (V, E)$ with *nodes* and *edges* representing components and connections, respectively. Each node $v \in V$ is an abstract object that has a corresponding rectangle R_v with a specified position and designated *ports* on its boundary. A node's ports are the designated connection points for incoming and outgoing edges and we allow any number of edges to connect to a single port.

We are looking for a drawing of the input graph in which each node $v \in V$ is represented by it's corresponding rectangle R_v, we call such a drawing an *output layout* (as shown in Figure 2.1). Furthermore a valid output layout must respect the following hard constraints: First of all edges are modelled as port-to-port connections, meaning that every edge $e = (v, w) \in E$ starts at a designated port of R_v and ends at a designated port of R_w. Furthermore e must be orthogonal, which means it may consist of horizontal and vertical segments only and no edge may pass through any rectangle. Lastly edges remain separate and may not join other edges, only cross them or run alongside as parallel but separate edge. There are also two soft constraints that we want to optimize: For one we want the number of crossings in our output layout to be as small as possible, while on the other hand we also try to minimize the accumulated edge length of all edges of the output layout. These goals are often conflicting and how much our algorithm focuses on either can be changed by varying the crossing cost parameter as described later on.

It should be noted, that since we allow any number of edges to connect to a single port, we also allow drawing parallel edges on top of each other in our output layouts. It is possible to separate such edges again using the *nudging step* presented by Wybrow et al. [WMS10]. This is not content of this paper however, since it is not an integral part of our algorithm. It would also be possible to add additional hard or soft constraints, such as minimum and maximum edge lengths as hard constraints, or minimizing the number of bends as soft constraint. Lastly it is important to say, that our modelling of the physical biochip problem ignores certain qualities like for example the width of channels.

A problem that is closely related to the *port-to-port channel routing problem* is the node-to-node channel routing problem. The difference between both is, that the node-to-node version does not specify which ports are to be used for it's connections. So where an edge $e = (v, w)$ of the port-to-port problem always started at a specified port of R_v and ended at a specified port of R_w, the same edge may start at any port of R_v and end at any port of R_w for the node-to-node channel routing problem. To reflect this we now add a so called *node-vertices* v' for each node v. The node-vertex v' is connected to all ports of R_v, which allows us to model edges as node-to-node connections. So a given edge $e = (v, w) \in E$ starts at the node-vertex v' of v and ends at w', the node-vertex of w. This way the algorithm can freely choose which port to use when connecting v and w.

The last two problems we want to define are the *port-to-port chip layout problem* and the *node-to-node chip layout problem*. In both cases the only difference to the corresponding channel routing problem is, that for the port-to-port and node-to-node chip layout problem the positions of the input graph's nodes are not specified. Rather the nodes can be placed freely, which changes the problem significantly. The problem we address in this paper is the node-to-node channel routing problem.

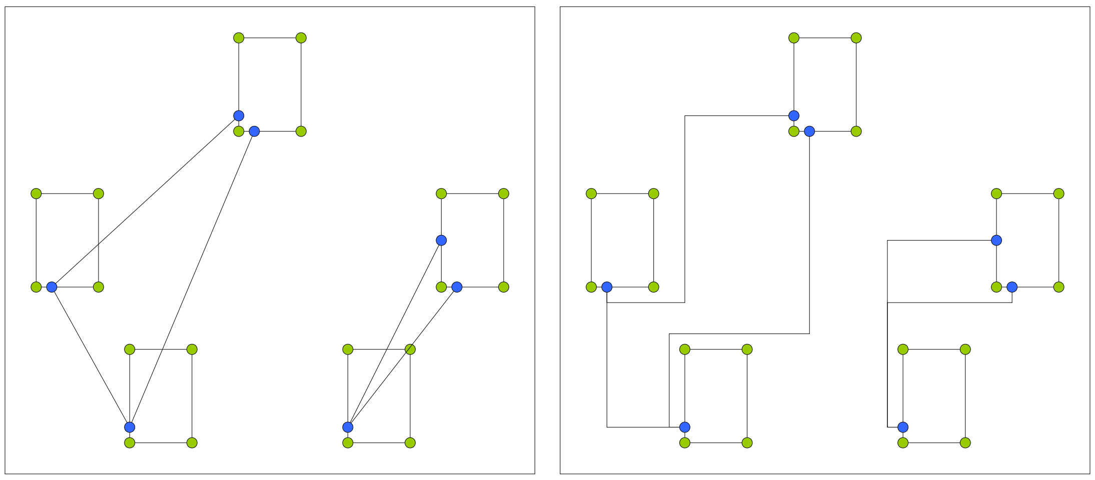


Figure 2.1.: To the right a sample port-to-port problem input graph,
to the left a valid output layout to the given input graph.

3. Complexity

In this section we show, that the node-to-node and port-to-port channel routing problems are NP-hard. It has been shown that finding the crossing number of general graphs is NP-hard [GJ83]. Pelsmajer et al. [PSS08] extended this by showing, that even if the graph has a fixed rotation system it is still NP-hard to find the minimum crossing number. However, in both cases the authors assume their graphs vertices can be placed freely and are not already fixed at specific locations. Since both of our channel routing problems input graphs already specify where the vertices are located, we first have to show that even with fixed vertex positions the graphs crossing number is still the same.

Lemma 3.1. *For a given graph, the minimum crossing number with variable vertex positions is the same as the minimum crossing number with fixed vertex positions.*

Proof. For a given graph G, let us call the minimum crossing number with variable vertex positions k and the minimum crossing number with fixed vertex positions k_f. Our goal is to show, that $k = k_f$. We do this by showing that $k \leq k_f$ and $k \geq k_f$. The first one is rather simple. If k_f is the crossing number with fixed vertex positions, k can be at least as small as k_f by choosing the same vertex positions as in the fixed vertex embedding of k_f. So let us concentrate on $k \geq k_f$. Given an embedding of G with exactly k crossings, we can now add k new intersection-vertices, one at every crossing of this embedding. This gives us the planar graph G', for which we can now find a planar embedding for any given set of fixed vertex locations, as shown by Pach and Wenger [PW01]. So we can choose a set of fixed vertex locations, create a planar embedding of G' with these vertex locations and then remove the intersection-vertices to create a non-planar embedding of G with k crossings. Since we constructed an embedding based on fixed vertex positions (as shown in Figure 3.1), the crossing number k of this embedding must be greater or equal than k_f, the minimum crossing number of embeddings of G with fixed vertex positions. □

Based on this lemma and the papers referenced above we can now show that both microfluidic channel routing problems are NP-hard.

Theorem 3.2. *The node-to-node channel routing problem is NP-hard.*

Proof. The general crossing number problem is known to be NP-hard. Given an instance I, with the input graph $G_I = (V_I, E_I)$, of the general crossing number problem, an instance I_n, with the input graph $G_n = (V_n, E_n)$, of the node-to-node channel routing problem can

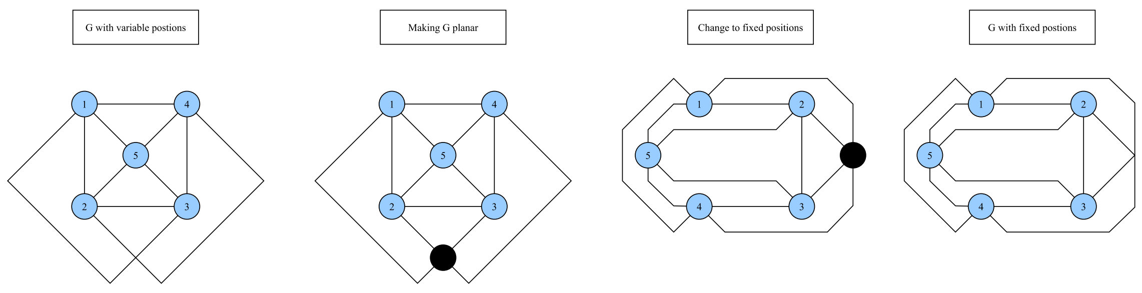


Figure 3.1.: Illustration of finding embedding with fixed position and $k = 1$ crossings.

be created in polynomial time. We do this by identifying each vertex $v \in V_I$ with a node $v_n \in V_n$ and each edge $e = \{v, w\} \in E_I$ with an edge $e_n = \{v_n, w_n\} \in E_n$. The size of the rectangles R_n can be scaled as needed, and we assume there are always enough ports at each side of the rectangle. Also, since the node-to-node channel routing problem requires it's nodes to be placed already, we need to place the nodes in V_n, for example randomly. A deterministic algorithm A that solves the node-to-node channel routing problem in polynomial time creates an output graph with the minimum crossing number for fixed vertex positions k_f of G_n, from which k_f can be extracted easily. As shown in Lemma **??**, k_f is the same as the minimum crossing number k of G_n, which is the same as the minimum crossing number of G_I by construction. Therefore the deterministic algorithm A also solves the general crossing number problem in polynomial time, which makes the node-to-node channel routing problem NP-hard. □

Theorem 3.3. *The port-to-port channel routing problem is NP-hard.*

Proof. An instance I, with the input graph $G_I = (V_I, E_I)$, of the NP-hard problem crossing number for fixed rotation systems can be transformed to an instance I_p, with the input graph $G_p = (V_p, E_p)$, of the port-to-port channel routing problem, similar to the transformation shown in Theorem 3.2. The difference is, that the edges of E are not represented as connections between nodes of V_n but rather between ports of the corresponding nodes. This determines the order of the edges around the node, so the ports used are chosen according to the rotation system of I. The node positions can again be chosen randomly. This way a deterministic algorithm A that solves the port-to-port channel routing problem in polynomial time also solves the crossing number for fixed rotation systems problem in polynomial time, as shown in Theorem 3.2. Therefore the port-to-port channel routing problem is NP-hard, also. □

4. A Graph Drawing Model for Biochip Layouts

In this section we describe our algorithm for the microfluidic node-to-node channel routing problem. As reminder, for an input graph $G = (V, E)$, V is the set of nodes of G. Each node $v \in V$ has a corresponding rectangle R_v with designates ports on its boundary, as described in section 2. We define X and Y as the sets of x- and y-coordinates of all rectangle corners and ports of the input graphs nodes. We can create a graph by placing a vertex at each point in $X \times Y$ and then connecting every vertex with its closest neighbour directly to the north, south, east and west. We call this graph the *complete grid graph* and the graph that we use for our algorithm (the Orthogonal Visibility Graph, or OVG) is a subgraph of the complete grid that we construct as the first step of our algorithm. Both, the complete grid graph and the OVG, are surrounded by a bounding box that represents the outer boundary of the graph. We then go through the input graph's edges e one by one and identify each of them with a route R_e that uses exclusively edges of the OVG, using Dijkstra's Shortest Path algorithm with a metric that depends solely on the accumulated edge length of R_e and the number of crossings with routes of previous edges. Lastly, we create the output layout that corresponds to the routing we found. In the following sections we will describe each of these steps in detail. Algorithm 4.1 shows the corresponding pseudocode for our node-to-node channel routing problem heuristic.

Algorithm 4.1: Node-to-Node Algorithm

```
   Input: inputGraph
   Data: OVG, routing
   Output: outputGraph
1  OVG = createOVG(inputGraph)
2  forall edge $e \in$ inputGraph do
3      route $r$ = computeOptimalRoute(OVG, $e$)
4      OVG = split(OVG, $r$)
5      routing.add($r$)
6  outputGraph = transferRouting(inputGraph, routing)
```

4.1. Creating the Orthogonal Visibility Graph

A fundamental concept we use in our algorithm is the Orthogonal Visibility Graph, first introduced by Wybrow et al. [WMS10]. As noted above it is a subgraph of the complete grid graph that still contains all relevant routes, which means that for every route not in the OVG there is a route in the OVG that is not longer and does not lead to more crossings. We are using the OVG mainly because it limits the number of vertices to consider considerably. Formally the OVG can be defined as follows. To a given an input graph $G = (V, E)$ (see Chapter 2), we define the corresponding Orthogonal Visibility Graph as $\mathrm{OVG}(G) = (V_o, E_o)$ where $V_o = V_R \cup V_p \cup V_{IP}$ with V_R being the set of rectangle corners for every rectangle R_v of a node $v \in V$, V_p being the set of all ports and V_{IP} the set of all *interesting points* of G. A point $p = (x_p, y_p)$ is an interesting point of G if it there are two vertices v_1, $v_2 \in V_R \cup V_p$, with $v_1 = (x_p, y)$ and $v_2 = (x, y_p)$, with v_1, v_2 belonging to different nodes and there is no rectangle between p and v_1, nor between p and v_2. Two vertices $v, w \in V_o$ are connected by an edge e if and only if v is the closest neighbour of w to either the north, south, east or west and there is no rectangle R_n between v and w.

The number of vertices of the OVG $|V_o| \in O(n^2)$ with n being the number of nodes of the input graph G, as shown in Lemma 4.1. Furthermore Lemma 4.2 also shows, that there is a family of OVGs with $|V_o| \in \Theta(n^2)$. Additionally, the number of edges $|E_o|$ is no higher than $4 \cdot |V_o|$ because a vertex may only have up to four neighbours and therefore $|E_o| \in O(n^2)$ as well.
Based on this definition, the OVG can be created using the following steps:

1. Creating vertices of V_R and V_p.
2. Calculating interesting points and creating vertices of V_{IP}.
3. Creating edges of the OVG.

Lemma 4.1. *Let $G = (V, E)$ be a given input graph with $|V| = n$ nodes and* $\mathrm{OVG}(G) = (V_o, E_o)$ *the corresponding OVG. Then the number of vertices of the OVG $|V_o| \in O(n^2)$.*

Proof. Since the OVG is a subgraph of the complete grid graph it is sufficient to show, that the number of vertices of the complete grid graph $|V_g| \in O(n^2)$. Because the complete grid graph is constructed by creating a vertex at every point in $X \times Y$, X and Y defined as above, the number of its vertices $|V_g| = |X| \cdot |Y|$. Therefore it is also sufficient to show that $|X|, |Y| \in O(n)$.
Let p_{max} be the maximum number of ports of any node $v \in V$. Then $k = p_{max} + 2$ is an upper boundary for the number of different x-coordinates of any vertex v, which we define as the number of different x-coordinates of the ports and rectangle corners of the corresponding rectangle R_v of v. This leads to $k \cdot n \geq |X|$ and therefore $|X| \in O(n)$. Showing that $|Y| \in O(n)$ can be done in exactly the same manner, since k is an upper boundary for the number of different y-coordinates of any vertex v also. Because the OVG is a subgraph of the complete grid graph $|V_o| \leq |V_g| = |X| \cdot |Y|$, and with $|X|, |Y| \in O(n)$ it is shown that $|V_o| \in O(n^2)$. □

Lemma 4.2. *Let $G = (V, E)$ be an input graph for which no two node's rectangles share a x- or y-coordinate. Then the corresponding* $\mathrm{OVG} = (V_o, E_o)$ *has $|V_o| \in \Theta(n^2)$ vertices.*

Proof. Since Lemma 4.1 shows that $|V_o| \in O(n^2)$ it is sufficient to show that $|V_o| \in \Omega(n^2)$. We will do so by showing, that the number of interesting points $|V_{IP}| \geq n^2$ for all n. For every node $v \in V$ the corresponding rectangle R_v has four different corner vertices in V_R that share two different x-coordinates and two different y-coordinates, which makes $2 \cdot n$ different x- and y-coordinates in total. For every two rectangles R_v and R_w we can find at least two different interesting points $p_1 = (x_v, y_w)$ and $p_2 = (x_w, y_v)$, for $c_v = (x_v, y_v)$ and $c_w = (x_w, y_w)$ being a random corner point of R_v and R_w, respectively. Since p_1 and p_2 share a x- and y-coordinate with rectangle corners of different rectangles, p_1 and p_2 are interesting points by definition, unless there is a rectangle between either p_1 and c_v, p_1 and c_w, p_2 and c_v or p_2 and c_w. But since no two rectangles of nodes $v \in V$ share a x- or y-coordinate, there can be no rectangle positioned between either or these points. This way we can find at least one different interesting point for each combination of rectangles (also see Figure 4.1), of which there are n^2 different combinations. So we can conclude, that for an input graph G like this there are at least n^2 interesting points, which leads to at least n^2 vertices for OVG(G), from which $|V_o| \in \Omega(n^2)$ and $|V_o| \in \Theta(n^2)$ follow directly. □

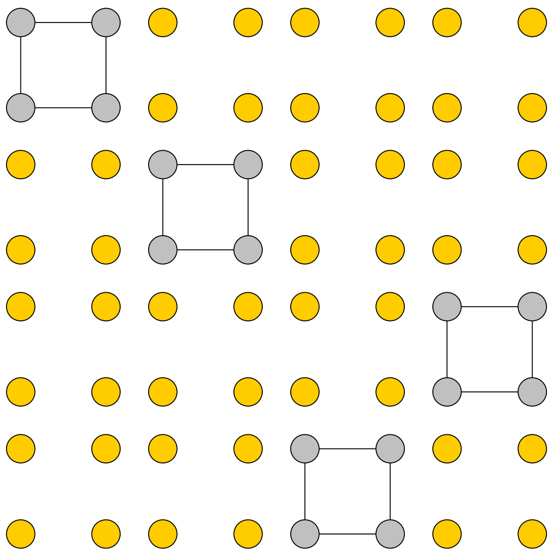


Figure 4.1.: All rectangles and interesting points of a simple sample graph. Rectangle corners in grey, interesting points in yellow

Creating Vertices of V_R and V_p

Transferring the nodes and ports of the input graph into vertices of the OVG is fairly simple. For each node in the input graph we take the coordinates of its corners and add four vertices with these coordinates to the OVG. The ports are just transferred directly by creating a vertex in the OVG with the same coordinates for every port.

Calculating Interesting Points and Creating Vertices of V_{IP}

The remaining vertices are to be created at every point which is an interesting point of the OVG's input graph G. We create them by casting a ray into each of the four orthogonal directions for every vertex in $V_R \cup V_p$. These rays go on until they either hit a rectangle, another vertex or the bounding box of the graph. A vertex may also have less than four rays, which happens when the vertex is situated directly on the border of one of the obstacles mentioned above. For example, a port situated at the side of a node's rectangle may only have up to three rays, since no ray may pass through the rectangle. Lastly, for our set of rays L, we then go through every element $(r_1, r_2) \in L \times L$ with $r_1 \neq r_2$ and intersect r_1 and r_2 if possible. Should both rays intersect we identified an interesting point of G and create a vertex at this position, which we then add to V_{IP}. After identifying all interesting points of G and adding a new vertex to V_{IP} for each of them we add V_{IP} to the OVG.

Creating Edges of the OVG

After we added the vertices of the OVG $= (V_o, E_o)$ we proceed by adding the edges. We create a planar graph by connecting each vertex to its closest neighbour in every orthogonal direction. The only exception to this rule are neighbours on opposite sides of nodes. Since there may be no edges that go through nodes, these vertices remain unconnected. The way we do this is by going through all rays $r \in L$ and then checking for each vertex $v \in V_o$ if it is on the ray. If v is on the current ray r we add it to a list ℓ of vertices on the current ray. After we found all vertices on r we sort ℓ by x- or y-coordinate, for r being a horizontal or vertical ray respectively. We then go on to create edges connecting each vertex $v_\ell \in \ell$ with the previous and the next vertex of ℓ and add these edges to E_o. All edges are created with costs equal to their length.

Theorem 4.3. *To a given input graph $G = (V, E)$ with $|V| = n$ we can construct the* OVG$(G) = (V_o, E_o)$ *in $O(n^3)$ time.*

Proof. The first step we take in constructing the OVG is to create the vertices in V_R and V_p. For p_{max} being the maximum number of ports of any node $v \in V$ and 4 the number of corner vertices for every rectangle R_v, we need to create $4 + p_{max}$ vertices at most per node in V. Therefore creating the vertices in V_R and V_p can be done in $O(n)$ time.

The next part of our algorithm is to find all interesting points of G and then adding a vertex for each of them to the OVG. We do this by creating no more than 4 rays per vertex in $V_R \cup V_p$, therefore the number of rays $|L| \in O(n)$. Because of this, intersecting each pair of different rays to find all interesting points of G is possible $O(n^2)$. Creating the vertex to each interesting point and adding it to the OVG can then be done in constant time. So in total the time needed to find the interesting points of G, create vertices at their positions and add these vertices to the OVG is in $O(n^2)$.

Our last step is to create the edges connecting the vertices of the OVG. For this we first go through the rays found in the last step and then search the vertices of the OVG for all vertices that are positioned on each ray. As shown above there are $O(n)$ rays and we know the number of vertices in the OVG is in $O(n^2)$ (see Lemma 4.1), which means the time needed to go through all vertices once for every ray is in $O(n^3)$. Since every ray can intersect each other ray once at most, the number of vertices on a ray is in $O(n)$, so the time needed for sorting the vertices on each ray once is no longer than $O(n^3)$. Lastly, creating the edges for every ray is in $O(n^2)$, since it is done once per ray, creating an edge can be done in constant time and it is done less than twice per vertex on the ray.

This leads to a total running time of $(O(n) + O(n^2) + O(n^3)) \in O(n^3)$, if adding up the time needed for the first, second and third step. $\square$

Adding and Connecting Node-Vertices

The OVG in its original form is now complete. However, to simplify node-to-node connection routing, we extend the OVG concept by adding another set of vertices, which we call *node-vertices*. There is one node-vertex per rectangle and it is connected to all of the rectangle's ports. This simplifies node-to-node connection routing, since it allows us to represent the input graph's edges as connections between node-vertices instead of connections between ports. This way connections using all ports of the node are considered when choosing the cheapest routes in the OVG to identify the input graph's edges with. Strictly speaking this modified graph is not an Orthogonal Visibility Graph anymore, but we will still reference it as such for convenience.

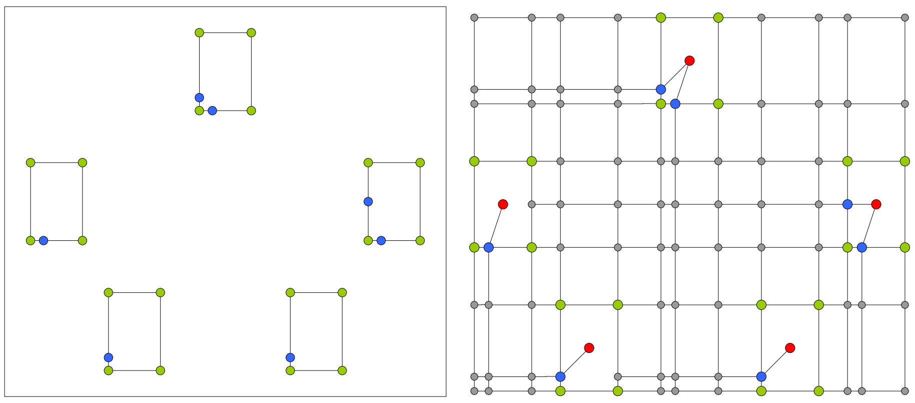


Figure 4.2.:
To the right a simple input graph's rectangles and ports,
to the left the corresponding OVG. Node-vertices in red, ports in blue,
rectangle corners in green and interesting points in grey.

4.2. Routing Edges using the OVG

Based on the OVG constructed as described above we can now start to incrementally compute a channel routing. We do this by modelling the crossing minimization problem as shortest path problem and then use Dijkstra's shortest path algorithm to solve it. Given an edge $e = (v_1, v_2)$ of the input graph, we first identify the node-vertices s and t as corresponding *source* and *target* vertices of the OVG, with s being the node-vertex representing the node v_1 and t being the same for v_2. We then use Dijkstra's algorithm to find the cheapest route R_e connecting s and t in the OVG with a metric that depends on the accumulated length of R_e and the number of crossings with routes of previous edges. There can be no crossings with previous routes for the first edge's route, but after we identified at least one edge with a route in the OVG, we will need to modify the OVG so it retains the quality that the cheapest route causes the minimum number of crossings. So after picking a route, every other route that crosses it should be more expensive. We achieve that by locally rebuilding our graph around the chosen route in a way that makes crossing it expensive, whilst all other costs remain unchanged. For simplicity we will continue to reference the changed graph as OVG.

We call our local rebuilding process *route splitting*, and the resulting graph has the following quality: For every route R there is a set of edges C_R, so that any later route R' uses an edge of C_R if and only if R' crosses R. We call the edges of C_R *tunnel-crossing edges* for every route R. This allows us to easily regulate the cost of crossing another route by assigning our desired crossing costs to these tunnel-crossing edges. For this we modify our graph every time we choose a new route.

Splitting the route works as follows (also shown in Figure 4.3): For a given input graph edge $e = (v_1, v_2)$, the corresponding node-vertices s for v_1 and t for v_2 and the route R_e we define $V_R = (s, w_1, w_2, \ldots, w_k, t)$, the vertices on the route in the order they are passed through. Similarly, we define $E_R = (e_1, e_2, \ldots, e_{k+1})$ the edges of the route in the order they are passed through, so $e_1 = (s, w_1)$, $e_2 = (w_1, w_2)$, $\ldots$, $e_{k+1} = (w_k, t)$. We then mirror R_e twice by creating R' and R'' with $V'_R = (s, w'_1, \ldots, w'_k, t)$ and $V''_R = (s, w''_1, \ldots, w''_k, t)$ with corresponding edge lists E'_R and E''_R. For this we create the new vertices w'_i and w''_i to every vertex $w_i \in V_R$ and also the edges e'_i and e''_i to every edge $e_i \in E_R$. This gives us three different paths from s to t, namely R_e, R' and R''.

The next step is to go through all $w_i \in V_R$ one by one and copy their edges that are not part of R_e. We know that a given vertex $w_i \in V_R$ is part of two edges e_i and $e_{i+1} \in E_R$. So when going around w_i in clockwise order, we call the edges between e_i and e_{i+1} the *top edges* of w_i and the edges between e_{i+1} and e_i the *bottom edges* of w_i. For each edge $e_t = (w_i, w_t)$ in the set of top edges of w_i we then create a new edge $e'_t = (w'_i, w_t)$ and for each edge $e_b = (w_i, w_b)$ of the bottom edges we create a new edge $e''_b = (w''_i, w_b)$. All newly created edge up to this point retain the costs of their original edges. Afterwards we create the tunnel-crossing edges for R_e by connecting w'_i of R' and w''_i of R'' for every $i \in \{1, \ldots, k\}$ with completely new edges that are assigned the desired crossing costs. For example the crossing costs can be chosen, so that the cheapest path from s to t is also a crossing-minimal (as shown in Theorem 4.4). Lastly we delete R_e from the OVG, since it has been replaced by R' and R''.

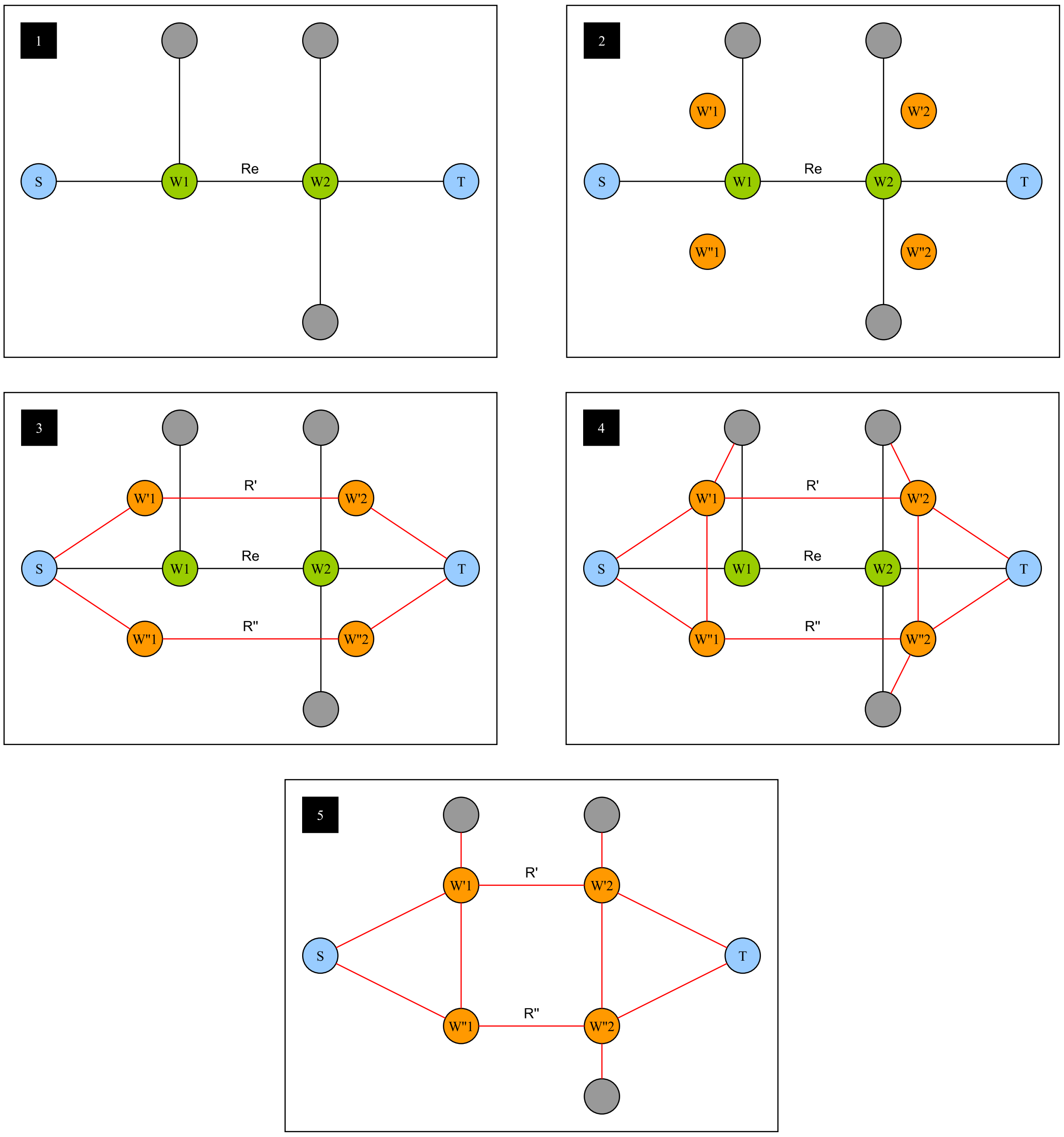


Figure 4.3.: Example of the route splitting step.

Theorem 4.4. *The cheapest path from a given source s to a given target t in the* OVG $= (V_o, E_o)$ *is crossing-minimal regarding the current drawing of the* OVG, *if the crossing costs are set to $m \cdot \ell_{max}$ where $m = |E_o|$ is the number of edges of the OVG and ℓ_{max} the length of the longest edge in E_o.*

Proof. For a given OVG constructed as described above let the costs of all tunnel-crossing edges be equal to $m \cdot \ell_{max}$, with m being the current number of edges in the OVG and ℓ_{max} the longest edge in the graph. Lemma **??** shows that a route R crosses a previous route R' if and only if R uses a tunnel crossing edge of R' as added during the route splitting step as described above. Let us also assume we have already chosen at least one route. We can do this because as long as there are no previous routes chosen every route has the minimum number of crossings. Now, whenever we want to connect two new vertices s and t, Dijkstra's algorithm attempts to find a cheapest route in the graph that connects the current source s to the current target t.

Since any route that contains an edge twice must contain a cycle it cannot be a cheapest path between s and t. This is true, because all edges cost at least 1, which makes the same route just without the cycle cheaper. Also, because no shortest route contains any edge twice, no shortest route contains more edges than the graph.

This means, that the highest cost a shortest 0-crossing route may have is $(m-1) \cdot \ell_{max}$. To elaborate on that, every non-crossing edge costs ℓ_{max} at most and are m edges in the graph, but at least one of them is a tunnel-crossing edge because we have already chosen at least one route and rebuilt the graph around it. Which leaves $m-1$ edges at most that are not tunnel-crossing edges. Now, since every 1-crossing route contains at least one tunnel-crossing edge, its cost are at least $m \cdot \ell_{max}$. Which makes even the most expensive 0-crossing route cheaper than the cheapest 1-crossing route.

This also makes the most expensive n-crossing route cheaper than the cheapest route with $n+1$ crossings. Because given a partial route with n crossings, taking every non-tunnel-crossing edge left in the graph always costs less than $m \cdot \ell_{max}$, the cost of adding an additional crossing edge. Therefore, if there is a way to finish a partial route with n crossings without adding another tunnel-crossing edge, it is always cheaper than to increase the crossing count. $\square$

Lemma 4.5. *For a given OVG, let R be a route of edges of the OVG and C be the set of tunnel-crossing edges of a previous route R_p. Then R', the route representing R before the OVG was rebuilt around R_p, crosses R_p if and only if R contains an edge of C.*

Proof. Let G_p be the OVG containing R_p and G the same OVG after executing the route splitting step around the route R_p. Further let C be the set of tunnel-crossing edges of R_p in G and R' a route that crosses R_p in G_p. We also define $V'_R = (s, v'_1, v'_2, \ldots, v'_k, t)$, the vertices of R', and $E'_R = (e'_1, e'_2, \ldots, e'_{k+2})$, the edges of the R', both in the order they are passed through from the source vertex s of R' to it's target vertex t. Lastly let v'_c, $c \in \{1, \ldots, k\}$, be the vertex that R_p and R' share. Then R, the route that represents R' in G, can be found by substituting v'_c of V'_R with $v_{c'}$ and $v_{c''}$, the vertices that replace v'_c in G. Since $v_{c'}$ and $v_{c''}$ are connected by the tunnel-crossing edge $c \in C$ the route R is contiguous when the sequence $e'_c = (v'_{c-1}, v'_c)$, $e'_{c+1} = (v'_c, v'_{c+1})$ in E'_R is substituted by the sequence $e_{c'} = (v'_{c-1}, v_{c'})$, c, $e_{c+1} = (v_{c''}, v'_{c+1})$. Which proves, that if R' crossed R_p before the route splitting step was executed, then R contains a tunnel-crossing edge $c \in C$.

For the following part let us assume that R' does not cross R_p in G'. Then all vertices and edges of R' still exist in G so $R = R'$ and since R' does not contain a tunnel-crossing edge, neither does R. $\square$

4.3. Transferring the routing

The final step of this algorithm is to transfer the routing on the OVG, found as described above, to a valid output layout as defined in the modelling section 2. This can be done easily for a given input graph $G = (V, E)$ by identifying each edge $e \in E$ with its route R_e. Since R_e exactly specifies the horizontal and vertical segments of e, the drawing of G that is the output layout can be obtained by drawing each edge according to its route.

5. Evaluation

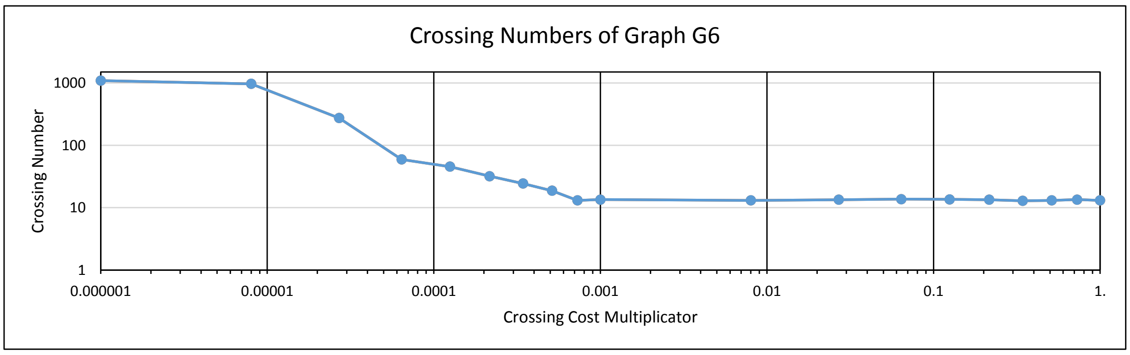
In this section we want to take a look at how our algorithm performs on some realistic test data, kindly provided for us by Prof. Tsung-Yi Ho from National Cheng Kung University, Taiwan. This test data consists of 11 different input graphs, G_1 to G_{11}, which all share the same set of 110 nodes (10 storages and 10 mixers for each of the storages) but differ in their connections and especially the degree of interconnection between mixers that belong to different storages. To be more precise every mixer is connected to its storage in each of the input graphs, while the number of connections between mixers ranges from 3 in G_1 to 90 in G_{11}. A picture of how an output layout may look like for each graph can be found in Appendix A.

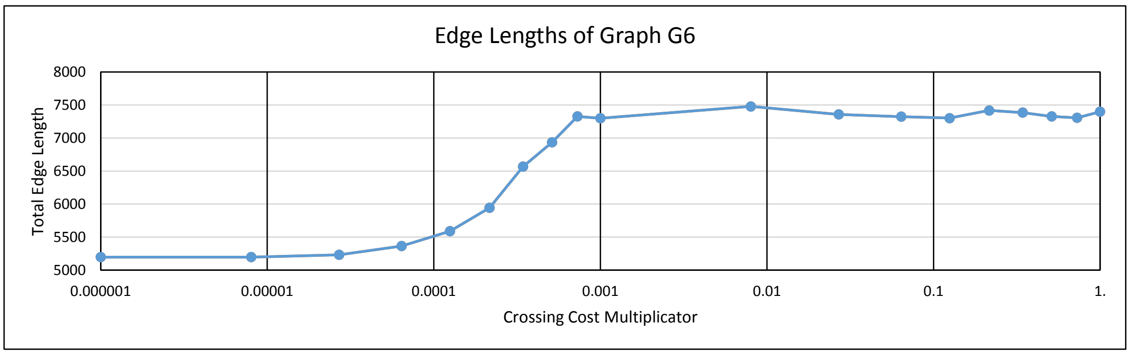
We implemented our algorithm using the Java programming language and Java SE 7u55, JDK 1.7 and the Jung2 framework for working with graphs. All runs were performed on a home computer with an Intel Core i5 CPU @3.00 GHz and 8 GB DDR3 RAM. The operating system was Windows 8.1.

The soft constraints we are trying to optimize are the *crossing number* of our output layout and the *total edge length* of all edges used in our routing. We also track the *execution time* of our algorithm. It should be noted that using the Jung2 framework introduced some randomness into the basically deterministic algorithm, which is why all given crossing numbers and total edge lengths will be averages of 20 runs on the same input graph using the same parameters.

As for parameters, we will present results of the algorithm using different crossing costs, ranging from 1 to a value c_{max} that is set to what we proved to guarantee crossing minimal routes (also see Theorem 4.4 for this input graph. We then multiply the difference between our minimum and maximum value by a *crossing cost multiplicator* and add it to our minimum value of 1. This way a multiplicator of 0 means our crossing cost are set to 1, while with a multiplicator of 1.0 our algorithm uses c_{max} as crossing costs. With this we basically set the importance of crossing minimization versus edge length optimizing, as the crossing costs basically determine how much additional route length is one more crossing is worth. So high crossing costs focus our algorithm primarily on crossing minimization, while lowering them shifts the focus away from crossing minimization and more towards edge length optimizing.

Figure 5.1 shows the typical characteristica of our experimental results (for more see the Appendix A), based on input graph $G6$ (also see Figures 5.2, 5.3). As can be seen, the crossing number begins to drop significantly at first, then slows down and continues to converge against the minimum crossing number. As expected, the graphs for total edge length of the output layout and execution time of the algorithm look almost identical.





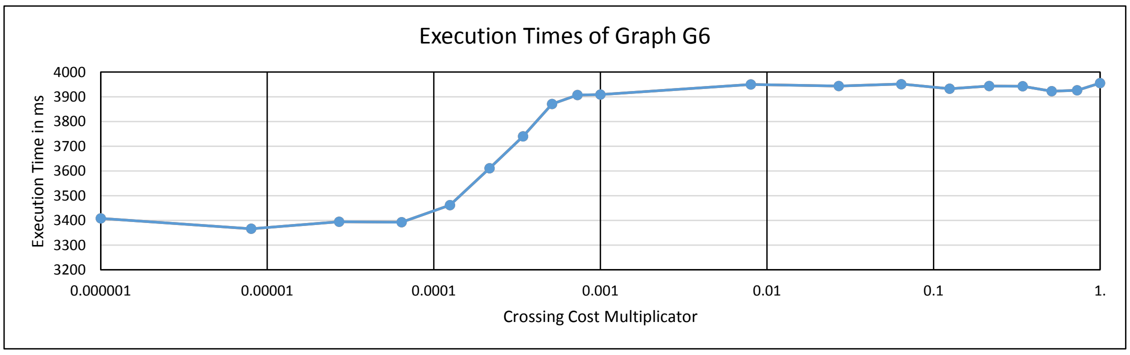
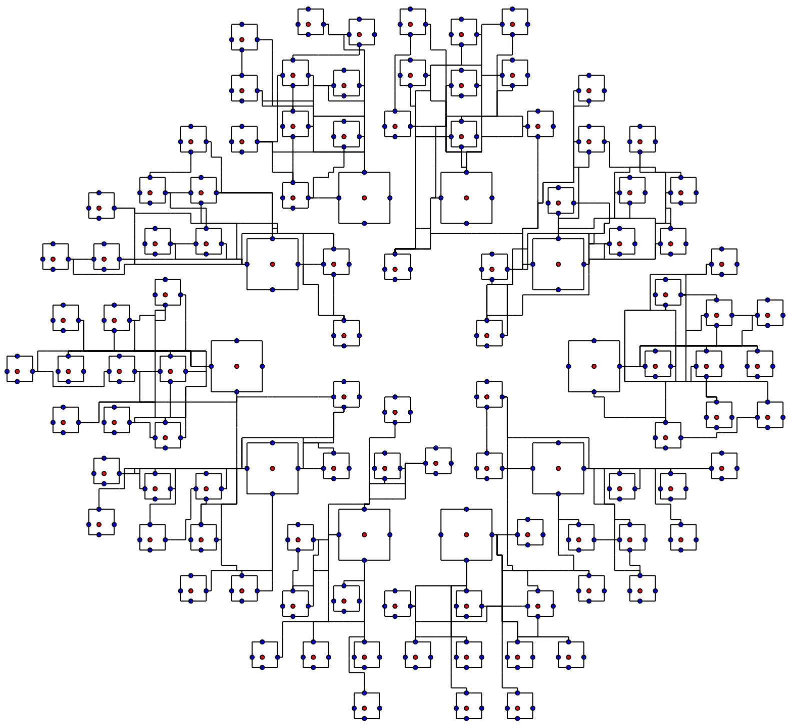


Figure 5.1.: Experimental results for G_6

Both start of about their minimum, starts rising slowly at first, gets faster and then slower again, as they converge against the maximum edge length and maximum execution time, respectively. The reason for both of them looking so similar is, that the most time intensive part of our algorithm is finding optimal routes using Dijkstra's algorithm. And when using Dijktra a longer route usually also takes longer to find. Please note that the crossing number is shown on a logarithmic scale, as is the crossing cost multiplicator but not the total edge length and execution time. This means, that the curve in the crossing numbers graph appears much more shallow than it would on a linear scale. On such a scale the curve would be much steeper.

All in all it can be said, that our results are exactly as expected. A low crossing cost modificator translates into more crossings but less total edge length. The number of crossings drops a lot faster than the total edge length rises at every point, which makes it worth it focusing on the crossing number for most applications of the algorithm, where there are no strict total edge length requirements.



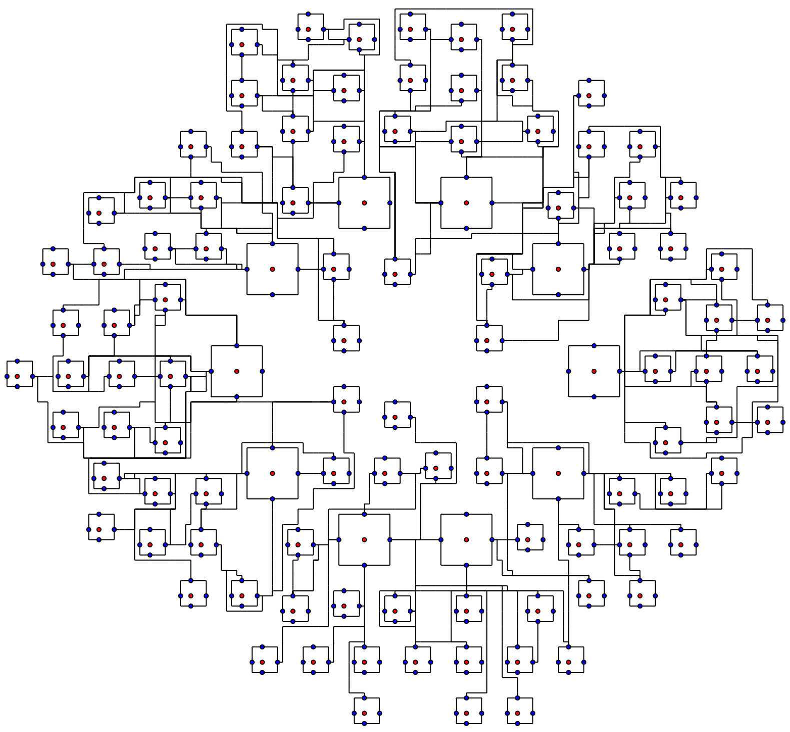
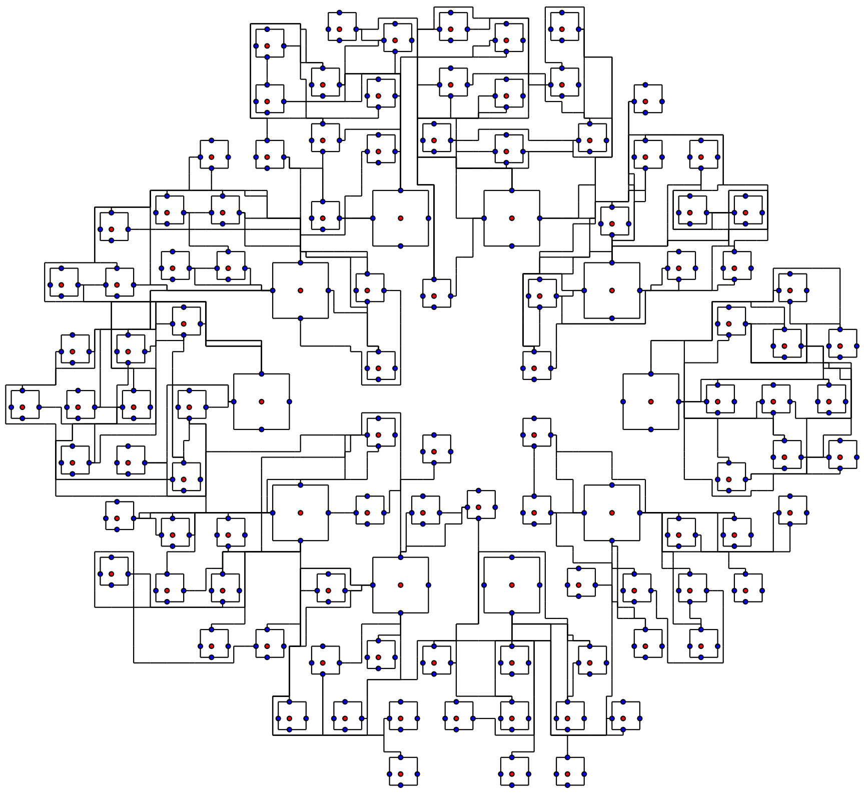


Figure 5.2.: Output layout for G_6 with crossing cost multiplier $(2/100)^3$ at the top and $(6/100)^3$ at the bottom.



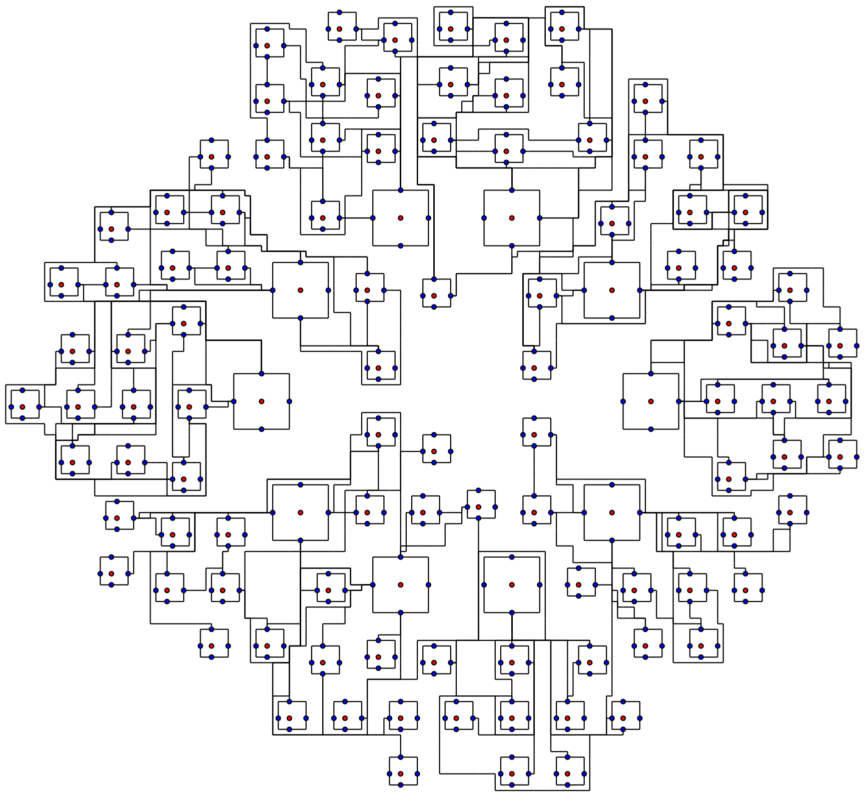


Figure 5.3.: Output layout for G_6 with crossing cost multiplier $(8/100)^3$ at the top and 1 at the bottom.

6. Conclusion

Designs of microfluidic biochips are still done mostly manually. Especially the exact component placement and efficient connector routing could be automated, making the design of new chips cheaper and faster. In this thesis we present an heuristic algorithm that can solve the connector routing problem within reasonable time.

For this our first step is to create the Orthogonal Visibility Graph of our input graph. On the OVG we then calculate a routing with a small number of crossings by modelling the problem as a shortest path problem and solve it using Dijkstra's algorithm. Lastly the routing we computed on the OVG is transferred to an output layout that specifies exactly how each connection should be routed.

This algorithm is a first step to software design aides that can provide automated and efficient component placement and connector routing functionality. It could be extended to cover component placement and then used to improve existing software design tools, or create new ones. Another possible extension of our algorithm would be to add additional costs for bends or long connector routes, weighting the cost of an additional crossing against a more simple output layout that is easier to understand.

Acknowledgements

We would like to express our gratitude to Prof. Tsung-Yi Ho and his associates from National Cheng Kung University, Taiwan, for kindly providing us a lot of information and explanations, including all of our testcases. We would also like to thank Dr.-Ing. Bastian Rapp from KIT for meeting us to discuss microfluidic biochips and proding a lot of insight into this topic. He also kindly provided us with some of the images in this thesis.

Bibliography

[CLC$^+$11] Chin, Laksanasopin, Cheung, Steinmiller, Linder, Parsa, Wang, Moore, Rouse, Umviligihozo, Karita, Mwambarangwe, Braunstein, van de Wijgert, Sahabo, Justman, Sadr, and Sia. Microfluidic-based diagnostics of infectious diseases in the developing world. *Nature Medicine*, 17:1015–1019, 2011.

[GJ83] Michael R. Garey and David S. Johnson. Crossing Number is NP-Complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[PSS08] Pelsmajer, Schaefer, and Stefankovic. Crossing Number of Graphs with Rotation System. In *Graph Drawing*, volume 15 of *Internatinal Symposium GD 2007*, pages 3–12, 2008.

[PW01] Pach and Wenger. Embedding Planar Graphs at Fixed Vertex Locations. *Graphs and Combinatorics*, 17(4):717–728, 2001.

[RIBT13] Ryab, Inglis, Barber, and Taylor. Manifacturing and wetting low-cost microfluidic cell seperation devices. *Biomicrofluidics*, 7(056501), 2013.

[TMQ02] Todd Thorsen, Sebstian J. Maerkl, and Stephen R. Quake. Microfluidic Large-Scale Integration. *Science*, 298:580–584, October 2002.

[TYLH13] Kai-Han Tseng, Sheng-Chi You, Jhe-Yu Liou, and Tsung-Yi Ho. A Top-Down Synthesis Methodology For Flow-Based Microfluidic Biochips Considering Valve-Switching Minimization. In *Cheng-Kok Koh, Cliff C. N. Sze (eds.) International Symposium on Physical Design, ISPD 13*, pages 123–129, Stateline, NV, USA, March 24-27 2013.

[WCH$^+$13] Waldbaur, Carneiro, Hettich, Wilhelm, and Rapp. Computer-aided microfluidic (CAMF): from digital 3D-CAD models to physical structures within a day. *Microfluidics and Nanofluidics*, (20975), 2013.

[WMS10] Wybrow, Marriott, and Stuckey. Orthogonal Connector Routing. In *D. Eppstein and E.R. Gansner (eds.) GD 2009, LNCS*, volume 5849, pages 219–231. Springer, Heidelberg (2006), 2010.

Appendix

A. Example Graph Routings and Plots

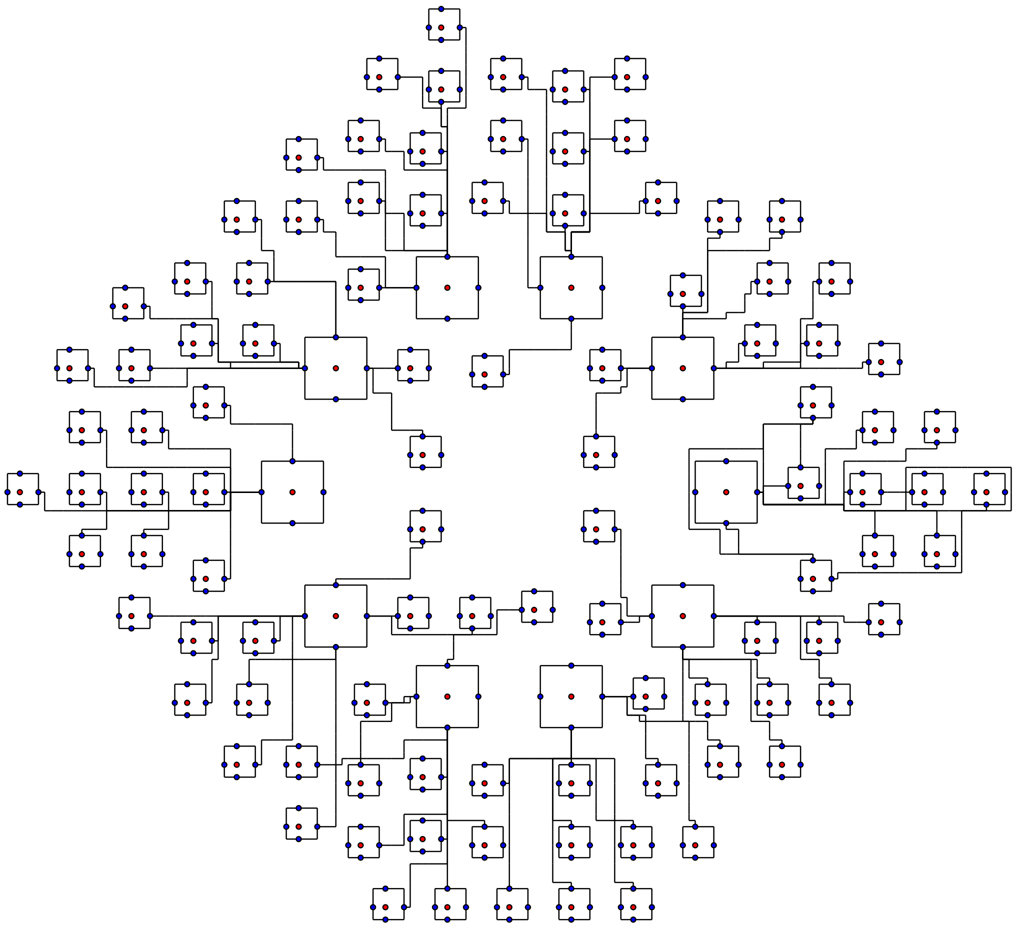


Figure A.1.: Output Layout for Graph $G1$, crossing cost multiplier of 1.
Crossing number = 0, total edge length = 3.257, calculating time = 3.593ms

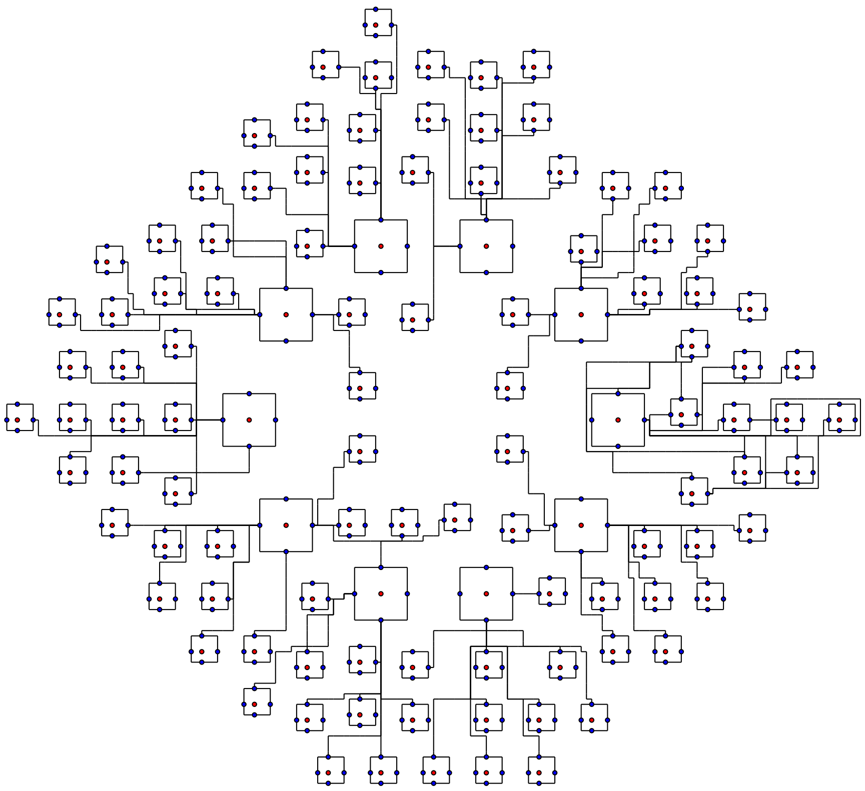


Figure A.2.: Output Layout for Graph $G2$, crossing cost multiplier of 1.
Crossing number = 0, total edge length = 3.414, calculating time = 3.837ms

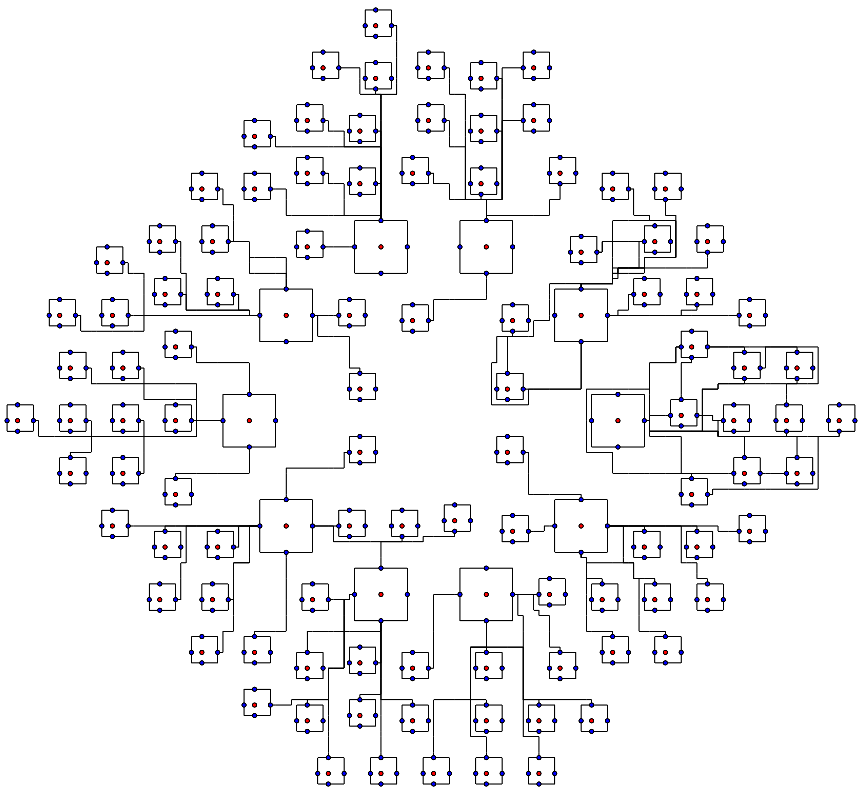


Figure A.3.: Output Layout for Graph $G3$, crossing cost multiplier = 1.
Crossing number = 2, total edge length = 3.534, calculating time = 3.930ms

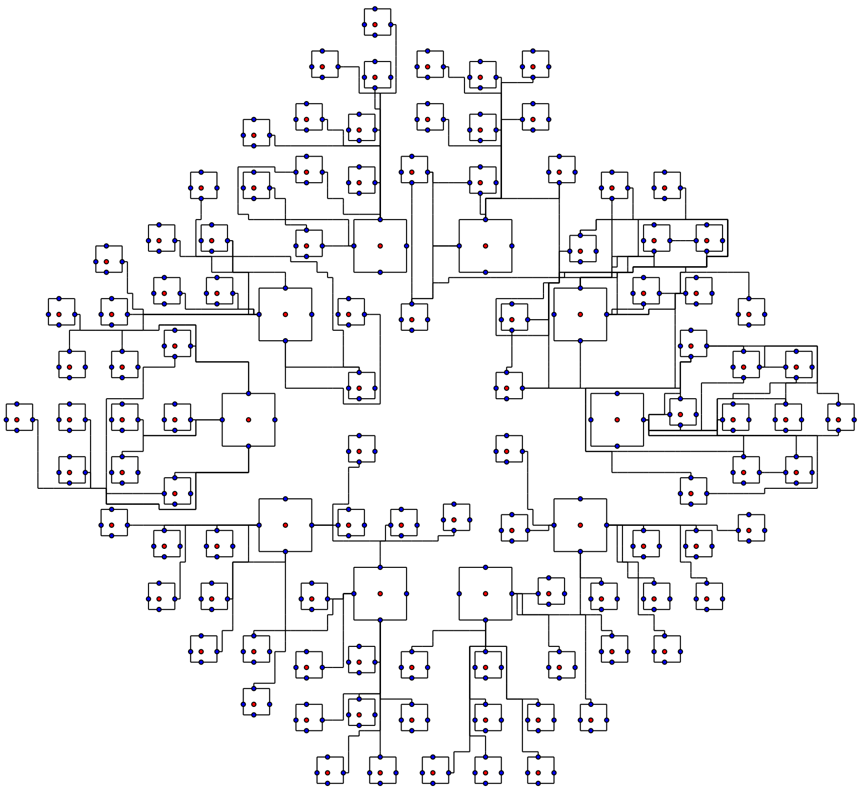


Figure A.4.: Output Layout for Graph $G4$, crossing cost multiplier = 1.
Crossing number = 3, total edge length = 4.415, calculating time = 4.371ms

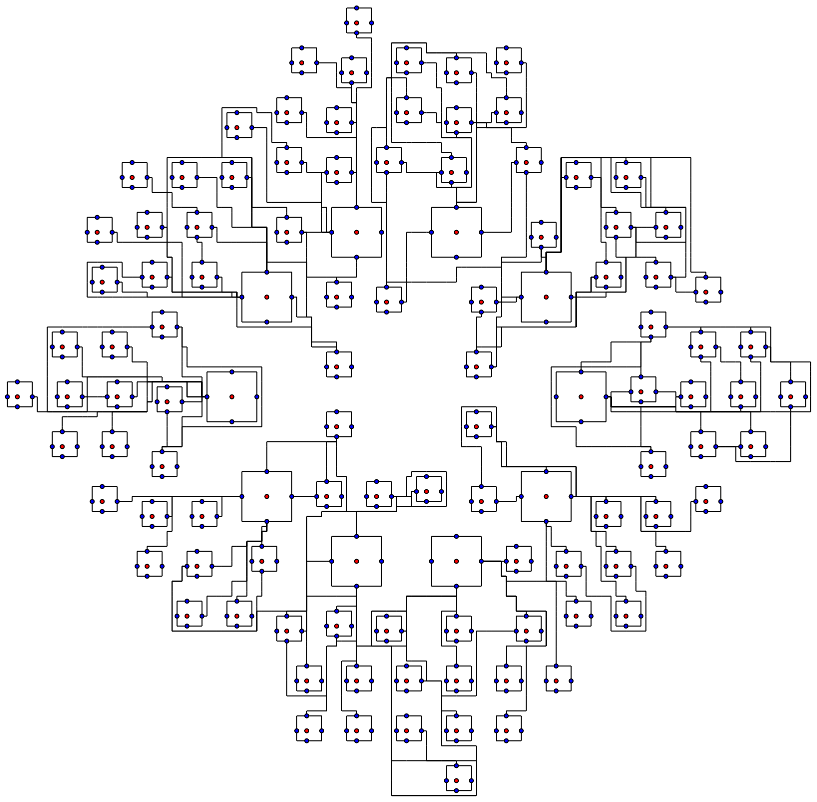


Figure A.5.: Output Layout for Graph $G5$, crossing cost multiplier = 1.
Crossing number = 5, total edge length = 5.394, calculating time = 4.998ms

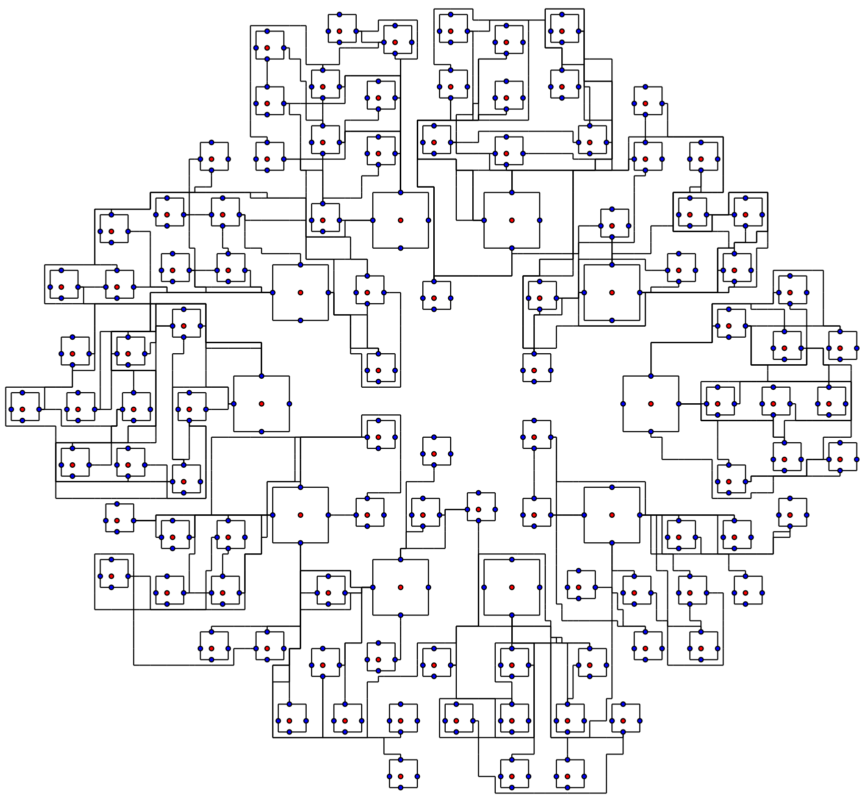


Figure A.6.: Output Layout for Graph $G6$, crossing cost multiplier = 1.
Crossing number = 16, total edge length = 7.165, calculating time = 5.485ms

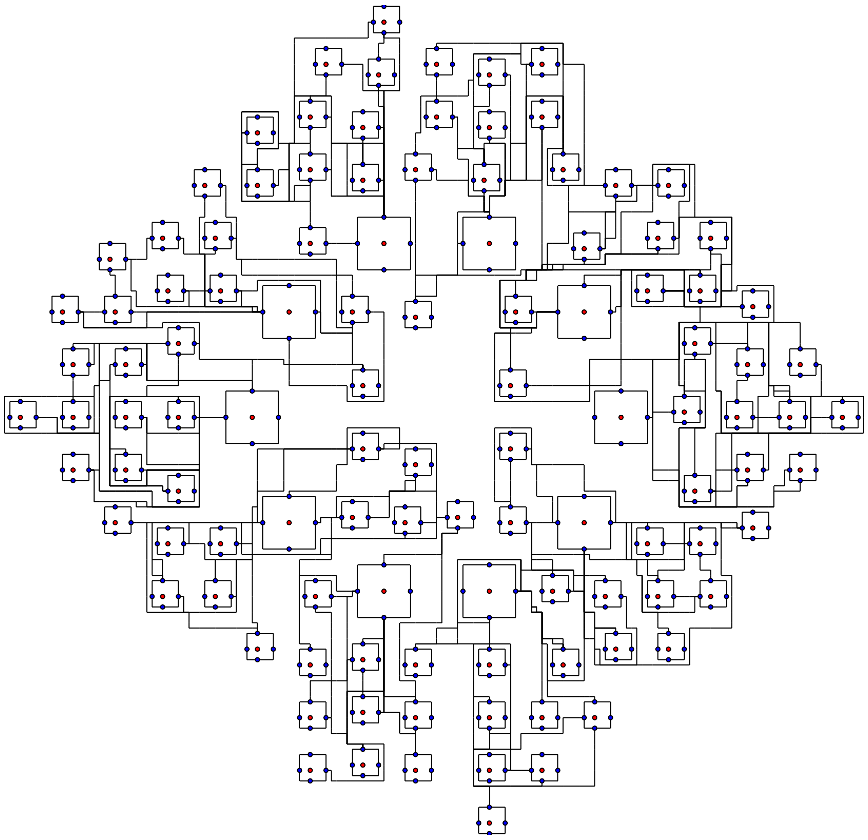


Figure A.7.: Output Layout for Graph $G7$, crossing cost multiplier = 1.
Crossing number = 18, total edge length = 8.136, calculating time = 6.891ms

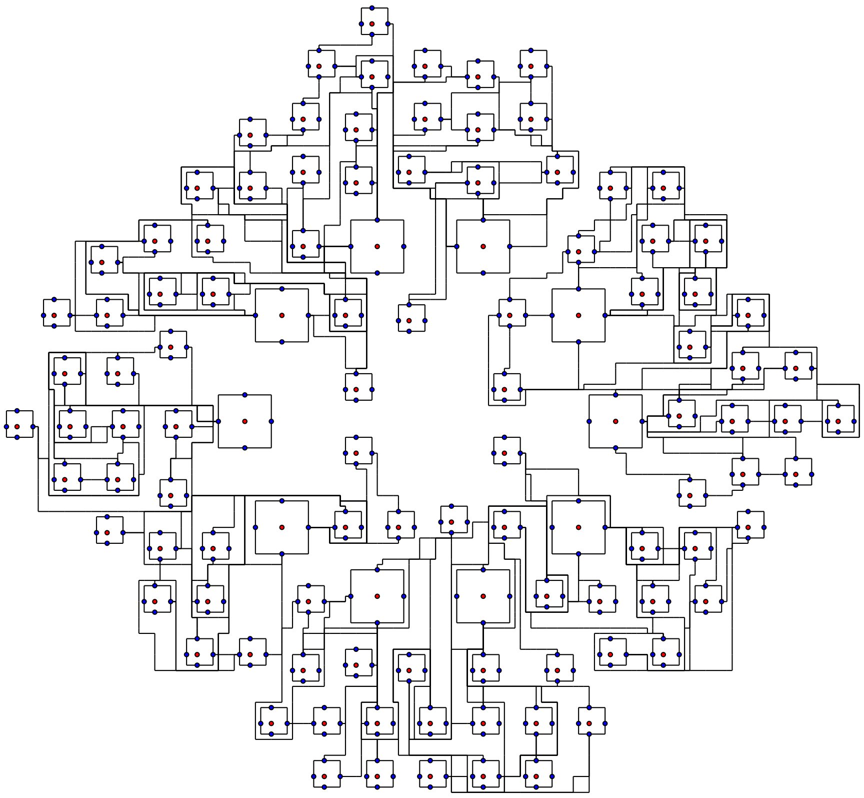


Figure A.8.: Output Layout for Graph $G8$, crossing cost multiplier = 1.
Crossing number = 18, total edge length = 8.775, calculating time = 6.467ms

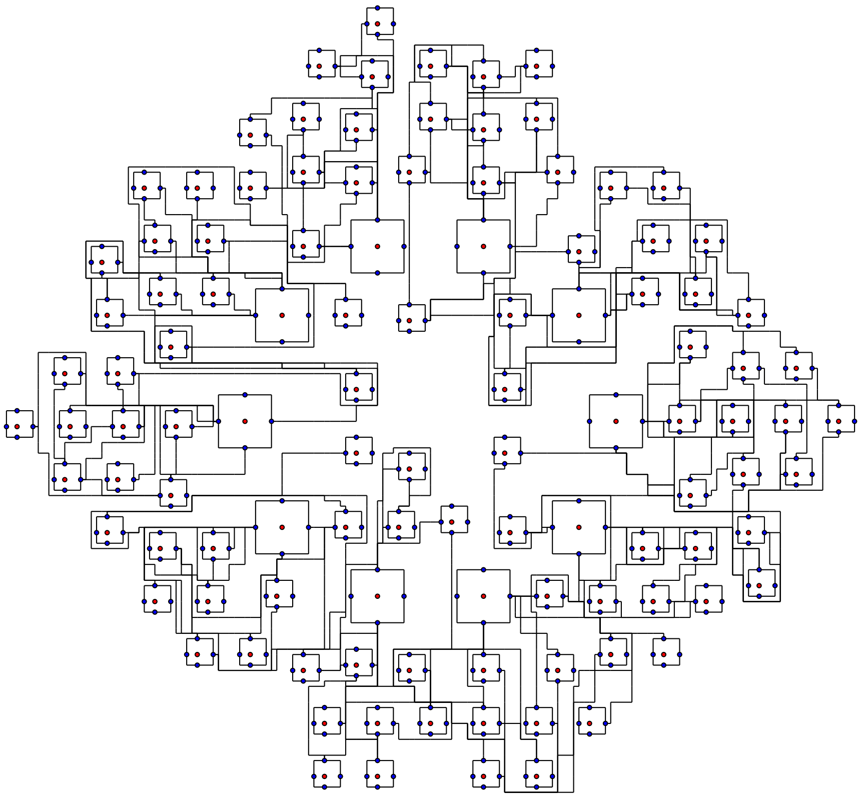


Figure A.9.: Output Layout for Graph $G9$, crossing cost multiplier = 1.
Crossing number = 19, total edge length = 7.600, calculating time = 6.415ms

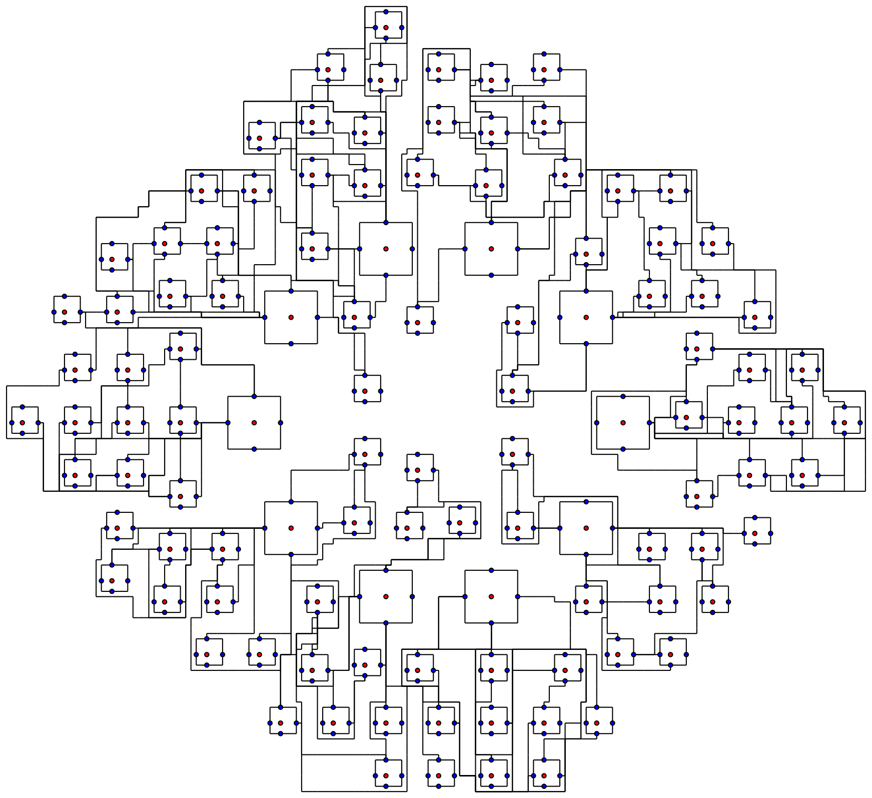


Figure A.10.: Output Layout for Graph $G10$, crossing cost multiplier $= 1$.
Crossing number $= 18$, total edge length $= 7.203$, calculating time $= 5.972$ms

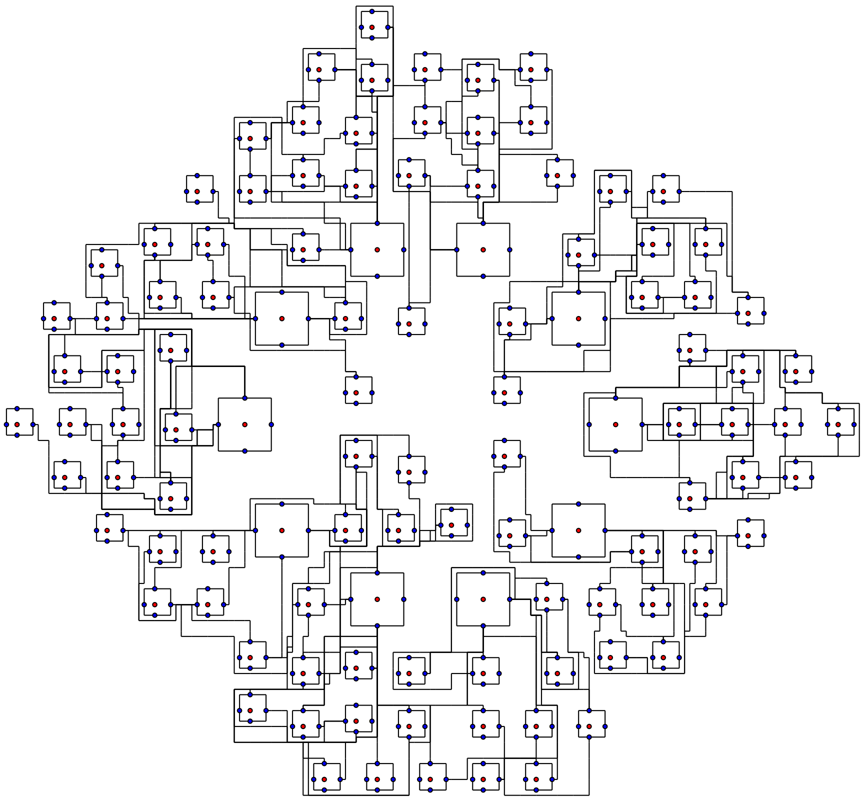
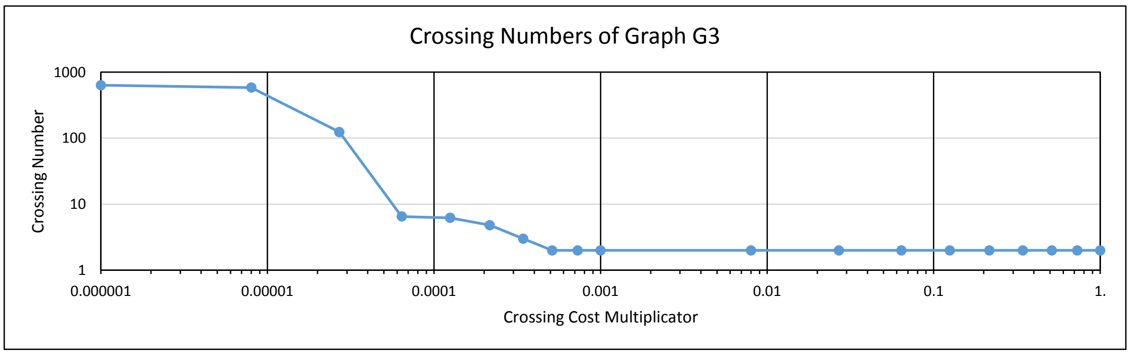
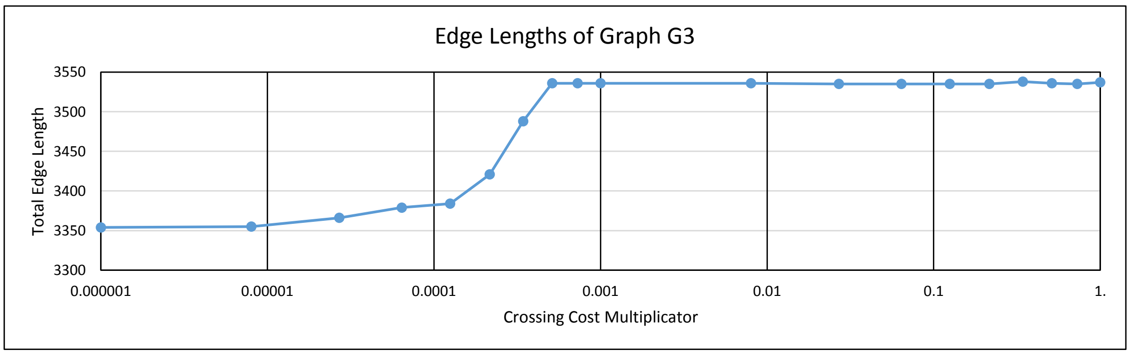


Figure A.11.: Output Layout for Graph $G11$, crossing cost multiplier $= 1$.
Crossing number $= 23$, total edge length $= 7.663$, calculating time $= 6.159$





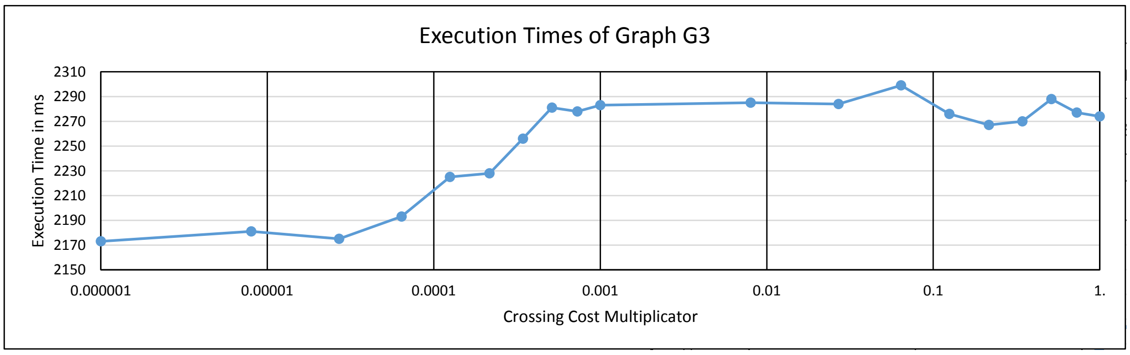
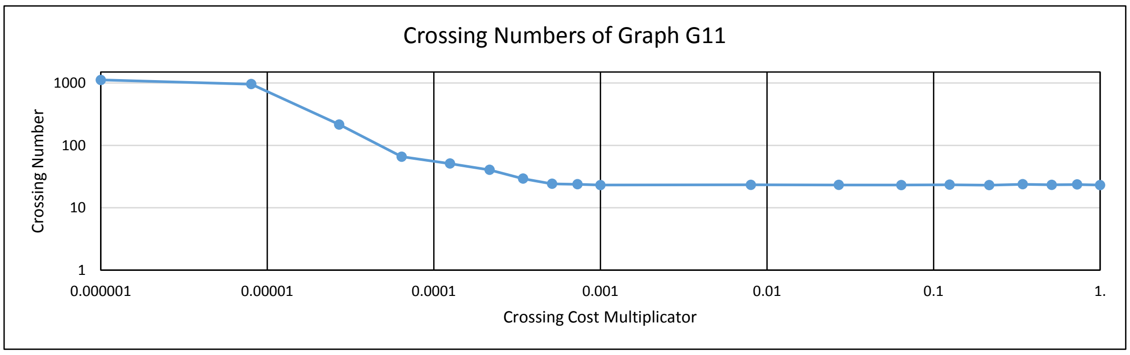
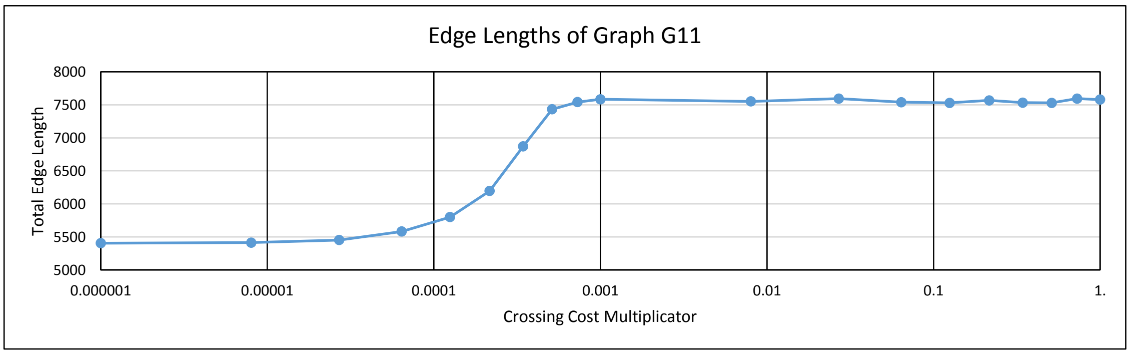


Figure A.12.: Experimental results for G_3





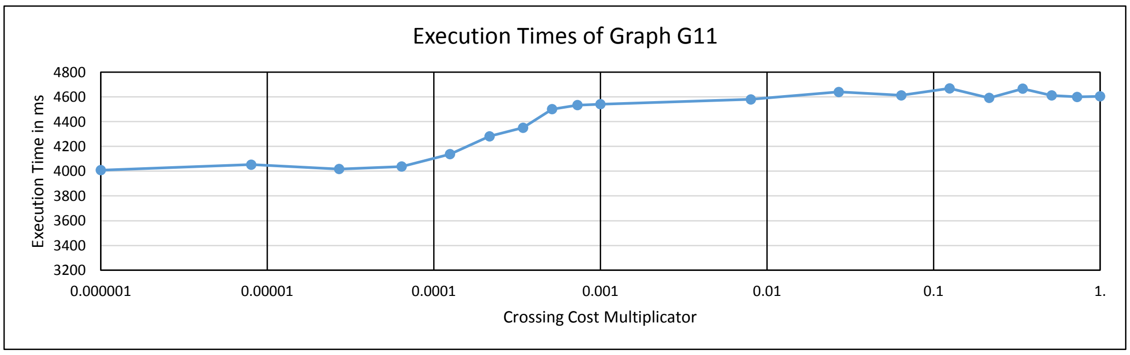


Figure A.13.: Experimental results for G_{11}