Placement Algorithm for Flow-Based Microfluidic Biochips

Michael Raagaard

Kongens Lyngby
B.Sc. June, 2014
Supervisor: Paul Pop

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk

B.Sc.-2014

Summary

Microfluidic biochips revolutionize biology by placing laboratory functionality on a very small chip. The subject of this thesis is a flow-based biochip, which consists of thin channels in which fluids flow. The basic building block of the chip is a valve, which can direct the flow by blocking or unblocking channels. The valves are combined to create components like switches, mixers, and multiplexers.

Biochips are currently designed manually using drawing tools like AutoCAD. The process is both tedious and error prone. Especially as biochips become larger and more complex.

The biochip design process consists of many phases. One of the first things that must be decided, is the placement of the components. The designer seeks the best placement as he places each component on the chip. Components should be packed close together to reduce the chip size. The placement should also attempt to facilitate direct and non-intersecting routes between components. This is to minimize the flow transportation time between components and to avoid introducing additional switches.

This thesis proposes an automated component placement tool, which assists the designer in choosing the best placement. The following chapters describe the algorithmic aspects of analysing and implementing such a tool. Components, connections, routes, and the chip itself are modelled to represent a component placement. A heuristic algorithm is proposed for optimizing the placement, and finally the algorithm is evaluated on several synthetic and real life benchmarks.

Contents

Chapter 1

Introduction

Microfluidic biochips are chips on which one or more laboratory functions are placed. The chips are only a few square centimeters in size and manipulate extremely small volumes of fluid. Based on these properties, biochips are considered a revolutionizing alternative to conventional biochemical laboratories.

Biochips have a number of advantages compared to conventional practises. Small volumes of fluid are used in the chips, which mean small reagent and sample volumes. Furthermore, small fluid volumes enable fast biochemical reactions and fast manipulation like heating, filtering, etc. This result in faster analysis and higher system throughput.

This technology is still new, so the number of applications for biochips is limited. However, many researchers believe that biochips will play an important role in biochemical fields in the near future. In particular, biochips may prove extremely useful in diagnostic testing. The objective is to design biochips that can effectively diagnose deceases, and with minimal fabrication costs [3]. This way, disposable chips can be used to diagnose deceases in clinics, which do not have access to expensive laboratory equipment. Such a chip has recently been designed to test for HIV and syphilis [2].

This thesis is based on the type of biochips called flow-based microfluidic biochips. Figure 1.1 shows a flow-based biochip. On this kind of chip, fluids flow in a con-

tinuous flow. The fluids enter the chip at several input ports, and flow on the chip in thin channels until leaving the chip at output ports. Along the way the fluids are manipulated in different ways. The next section describes the structure of the flow-based biochip in detail.

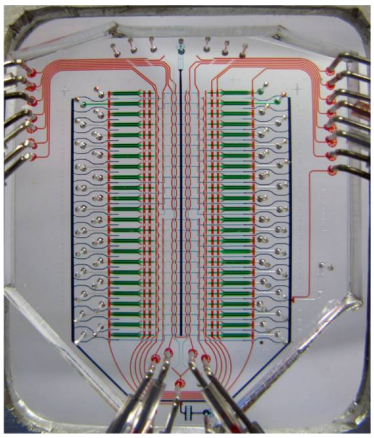


Figure 1.1: Picture of a flow-based microfluidic biochip. The thin lines are either flow channels or control channels to manipulate the direction of the flow. Borrowed from [12].

1.1 Flow-Based Microfluidic Biochips

The smallest unit in a flow-based biochip is a valve. When closed, fluids cannot pass the valve. The valves are combined to obtain various operations such as switching, mixing, and multiplexing. This resembles electronic components, which is why the process is referred to as microfluidic Very Large Scale Integration (mVLSI).

The chip is made up of two layers, the control layer and the flow layer. Figure 1.2 shows the two layers. The flow layer contains thin flow channels in which the fluids flow. The control layer contains empty channels, which are connected to an external air pressure source. A valve is constructed by letting a control channel intersect with a flow channel like in figure 1.2. If pressure is applied to the control channel it will expand and block the flow channel. This corresponds to a closed valve. If no pressure is applied the flow can pass the valve. This corresponds to an open valve.

The flow in a biochip is directed by opening and closing valves in a certain

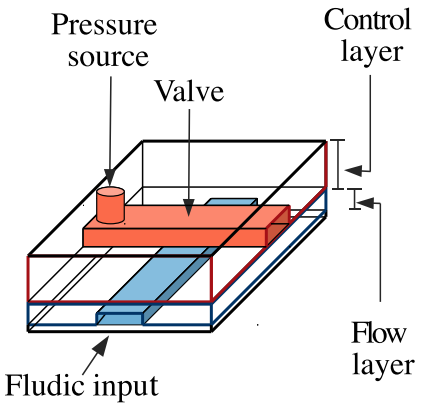


Figure 1.2: Illustration of the control layer and the flow layer. The control layer contains empty channels, which can block flow if pressure is applied. This is how a valve is constructed. Adapted from [7].

order. Figure 1.3 shows how three different switch configurations are build using valves. Other components, like multiplexers and mixers, are build in similar ways. Unlike electric circuits, two flow channels are allowed to intersect. However, if two different flows use the same channel simultaneously both fluids are contaminated. To avoid this, a switch of four valves is introduced to make sure that both routes cannot pass the intersection point simultaneously. Figure 1.3c shows such a switch.

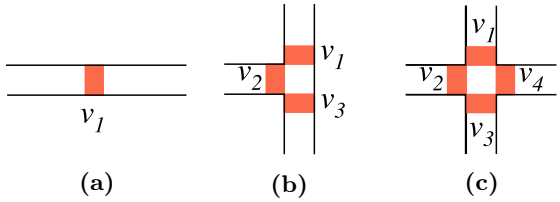


Figure 1.3: Three different switch component configurations to direct the flow. Adapted from [7].

1.2 Biochip Design Process

The biochip design process goes through a number of different design phases before the chip is fabricated. The design phases are mentioned in the following subsections.

1.2.1 Schematic Design

The first phase is designing the schematic of the chip. A number of different components are allocated to perform the operations required by the applications that should run on the chip. The connectivity of the allocated components is described in a netlist. See figure 1.4. A connection from one component to another represents that fluids can flow out of the first component and into the second component.

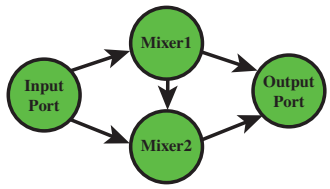


Figure 1.4: Illustration of a netlist.

1.2.2 Component Placement

Component placement is the first part of the physical synthesis of the chip. Based on the netlist from the schematic design phase, the physical positions and orientations of the components are determined. The placement is done based on the physical dimensions of the components and a set of design rules that ensure fabrication of the chip. The result is a chip architecture with all components placed, like shown in figure 1.5.

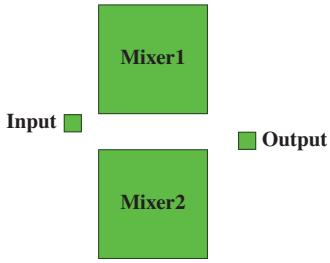


Figure 1.5: Illustration of a component placement.

1.2.3 Flow Channel Routing

Flow channel routing is the second part of the physical synthesis. The purpose of the phase is to design the flow channel routes between components. The routing is done based on the netlist and the physical placement of the components, while complying with a set of design rules. See figure 1.6.

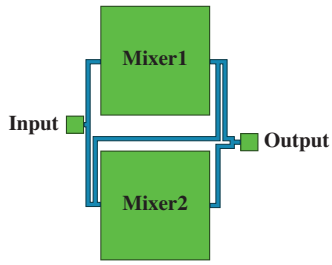


Figure 1.6: Illustration of a flow channel routing.

1.2.4 Application Mapping

The purpose of the application mapping phase is to map the application onto the chip by scheduling which components are used when and for what operation. The mapping must take into account the time it takes for fluids to pass a component and the time spend transporting fluids from one component to another. The time to transport fluids from one component to another depends on the length of the route. This is why the application mapping phase must be performed after the flow channel routing phase.

1.2.5 Control Channel Routing

Control channel routing is the third and last part of the physical synthesis. With the application mapping phase completed it is known which valves to open and close at what time. Based on this information the control channels are routed. The control channel routing phase must know the internal structure of each component in order to route a channel to each valve. See figure 1.7.

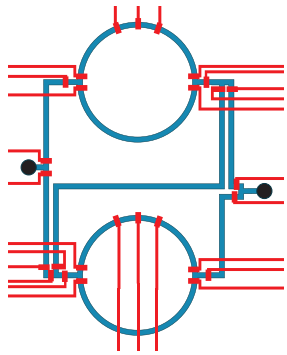


Figure 1.7: Illustration of a control channel routing. Small black circles indicate ports and the large circular channels are the internal structure of a mixer.

1.2.6 Fabrication

When all design phases are completed the chip is ready for fabrication. The biochips are fabricated using a transparent, inexpensive, rubber-like material, polydimethylsiloxane (PDMS). One layer is fabricated for the flow layer and one layer is fabricated for the control layer. These two layers are aligned and sealed on a flat plate, typically made of glass. Channels are made by removing PDMS material where the channels should be. Thus, channels are effectively absence of PDMS material.

1.3 Motivation

Currently, biochips are designed manually. The designer must have a deep understanding of all aspects of the design process, from application, to component sizes and internal structure, to design rules.

The placement and routing is done using drawing tools like AutoCAD. Components are drawn within the chip area and lines are drawn between components to represent flow channel routes. This design process is very labor intensive and error prone, which makes it expensive to develop new biochip designs. Especially as the complexity of the chips continue to increase. Thus, tools that assist and automate part of the design process could greatly help the designers and contribute to the advancements in the field.

This thesis explores techniques and algorithms for automating the component placement. The purpose is to design and implement a tool to assist the designer in the component placement design phase. Implementing such a tool requires a biochip architecture model, heuristics for placing components, as well as a way to judge the quality of a placement. The remaining chapters elaborate these topics, formulate the exact problem, and evaluate the tool on a set of benchmarks.

Chapter 2

Biochip Architecture Model

Performing an automated component placement requires a biochip architecture model and a netlist model. The biochip architecture models the essential parts of the physical architecture and the netlist models the components on the chip and connections between components.

Simplifying assumptions are made to explicitly define the problem and to reduce the problem size. In the following sections models are proposed for the component placement phase of the physical synthesis. In addition, a flow channel route model is proposed. The route model proves useful when judging the quality of placements.

2.1 Netlist

The purpose of the netlist is to define which components are to be placed on the chip, and their interconnections. Thus, it models the functionality of the chip but not the physical layout. The netlist is modelled as a directed graph, $\mathcal{N} = (V, E)$, where V is the set of components and E is the set of connections between components. An edge going from vertex, v_i, to vertex, v_j, represents that fluids can flow out of component, i, and into component, j. Figure 2.1

shows a simple netlist example.

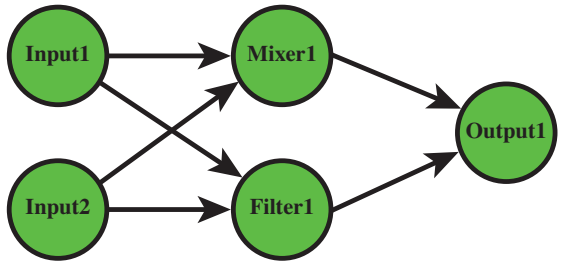


Figure 2.1: A netlist, $\mathcal{N}$, with vertices representing components and edges representing connections. Each component is given a unique identifier.

2.2 Architecture

The architecture model is a combination of four models. The first three are: Component model, connection model, and grid model. These three models are necessary to guarantee that the architecture model represents valid placements. A valid placement has no overlapping components and satisfies design rules that ensure the biochip fabrication. The fourth model, the route model, is used to judge the quality of the placement in terms of how well the flow channels are routed.

2.2.1 Component Model

The component model represents properties of the physical component, which are needed to perform the placement. The internal structure of the components is not important to the placement. Thus, it is sufficient to represent components only by a surrounding rectangle and the points where flow enters (entry point) and leaves (exit point) the component. The internal flow channels, control channels, and valves are left out. Figure 2.2a shows the internal design of a mixer component, and figure 2.2b shows the simplified component model for the same component.

In general, it is throughout the thesis assumed that the internal component design is a local problem that will not affect the global placement problem. It is also assumed that there is only one entry point and one exit point for

(a) (b)

Figure 2.2: A mixer component with internal flow channels, control channels, and valves (a), and the corresponding component model (b). indicates entry point and indicates exit point.

each component. The input and output ports of the chip are also modelled as components. However, flow can only leave the input port and only enter the output port, so the input port has no entry point and the output port has no exit point.

2.2.1.1 Component Library

Based on the physical dimensions of the different component types, and the location of the entry and exit points, a component library, $\mathcal{L}$, is constructed. See table 2.1. In addition, the number of valves that make up each component is listed. The valves are used for experimental evaluations in chapter 7.

The components listed in $\mathcal{L}$ are in the default orientation. This means that rectangular components are wider than they are high, the entry point is on the left side of the component, and the exit point is on the right side of the component. A placement might involve rotating a component either 90°, 180°, or 270° degrees, so each component type can be placed in four different orientations.

2.2.1.2 Design Rules

A number of design rules have been suggested to ensure that a chip is fabricatable. These design rules involve all aspects of the physical synthesis. The ones relevant to placement are listed in table 2.2.

Adding a border of 5 units (1 unit = $150\mu m$) around all components ensures that the spacing between any two components is at least 10 units. This is effectively done by increasing the size of all components by 10 units in both dimensions.

Component Type	**Dimensions** $(w \times h)$	**Entry Point**	**Exit Point**	**Valves**
Input	5×5	-	$(5, 2)$	1
Output	5×5	$(-1, 2)$	-	1
Mixer	30×30	$(-1, 15)$	$(30, 15)$	9
Filter	120×30	$(-1, 15)$	$(120, 15)$	2
Detector	20×20	$(-1, 10)$	$(20, 10)$	2
Separator	70×20	$(-1, 10)$	$(70, 10)$	2
Heater	40×15	$(-1, 7)$	$(40, 7)$	2
Metering	30×15	$(-1, 7)$	$(30, 7)$	6
Multiplexer	30×10	$(-1, 5)$	$(30, 5)$	2
Storage	90×30	$(-1, 15)$	$(90, 15)$	28

Table 2.1: Component Library, $\mathcal{L}$. Entry and exit points are coordinates relative to the upper left corner of the component. A scaling unit of $150\mu m$ is used. Dimensions and valve information are extracted from [7, 14], and entry and exit points are based on own assumptions.

Rule	**Minimum Value**
Spacing between external ports (input and output)	10
Spacing between other components	10

Table 2.2: Placement design rules suggested ensuring fabrication. A scaling unit of $150\mu m$ is used. The design rules are extracted from [1].

2.2.2 Connection Model

From the netlist, $\mathcal{N}$, the component interconnections are retrieved. An edge in $\mathcal{N}$ between components, i and j, represents a connection from the exit point of i to the entry point of j. The connection model is constructed from the netlist, $\mathcal{N}$, and the component library, $\mathcal{L}$.

The connections are modelled as a bipartite graph, $\mathcal{C} = (U, V, E)$, where U is the set of exit points, V is the set of entry points, and E is the set of connections between entry and exit points. The properties for a bipartite graph ensure that no connections exist between two entry points or two exit points. Figure 2.3a shows a component placement and the edges illustrate component interconnections. Figure 2.3b shows the corresponding connection model, where a connection is an edge between exit and entry points.

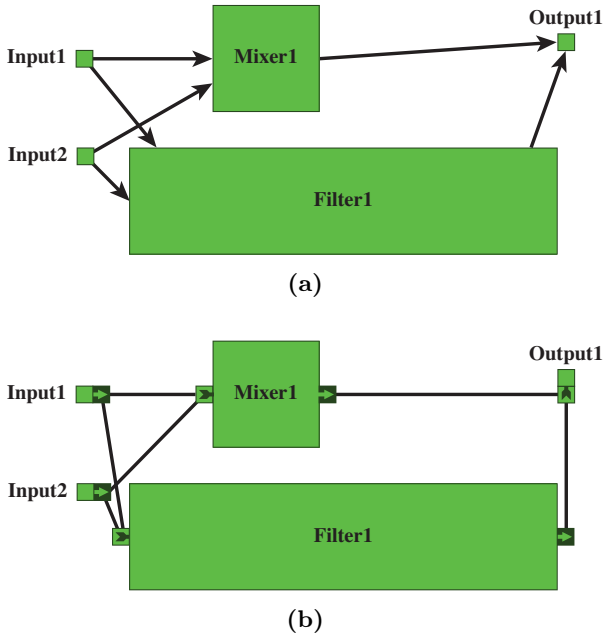


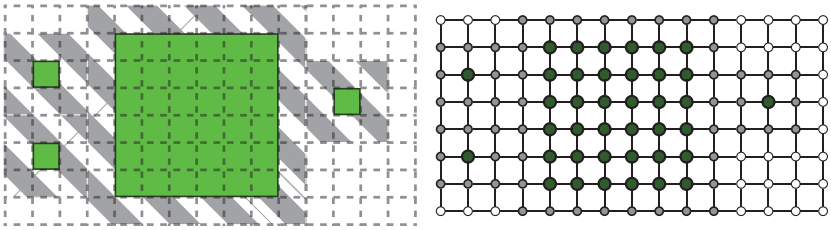
Figure 2.3: An architecture with placed components and connections (a), and the corresponding connection model, with entry and exit points (b).

2.2.3 Grid Graph Model

The chip area is divided into square cells such that it is represented by a $w \times h$ grid. w is the number of cells in the horizontal direction, and h is the number of cells in the vertical direction. The grid is modelled as a grid graph, $\mathcal{G} = (V, E)$, where V is the set of cells in the grid, and E is the set of edges defining adjacent cells. An edge between vertex, v_i, and vertex, v_j, represents that v_i and v_j are adjacent in the grid. Figure 2.4a shows a chip divided into cells, and figure 2.4b shows the corresponding grid graph.

When components are placed on the chip, cells that are within the area of the component are allocated to that component. A cell is only allocated to one component at a time. This prevents overlapping of components. Furthermore the component border ensures that the placement design rules are satisfied.

The size of the individual cells determines the total number of cells. Choosing a large cell size decreases the granularity of the grid, hence also decreases the problem size. If the cell size is too large accuracy is lost and chip area might



(a) (b)

Figure 2.4: Chip area divided into cells (a), and the corresponding grid, $\mathcal{G}$, (b). Notice that vertices are occupied for components as well as component border. The shaded area indicates component border.

be wasted. Unless otherwise state, the cell size is 5 units throughout the thesis. All component sizes in $\mathcal{L}$ and placement design rule values are multiplums of 5, which means that a cell size of 5 will not compromise placement accuracy.

2.2.4 Route Model

A route is modelled as the set of cells, which the route overlaps. A route belongs to the connection, which it routes. The route always begins in an entry point and ends in an exit point, as defined by the connection model. Let $r(e)$ be the set of cells for the route corresponding to connection, e. Furthermore, let $|r(e)|$ denote the number of cells in $r(e)$, and thus the length of the route.

As for placement, there are a number of design rules for the flow channels. These design rules are relevant when doing the flow channel routing. They are presented in table 2.3.

Rule	**Value**
Minimum spacing between flow channels	0.27
Flow channel width	0.67
Valve width	0.67

Table 2.3: Flow channel design rules suggested to ensure fabrication. A scaling unit of $150\mu m$ is used. The design rules are extracted from [1].

Note that the total width of the flow channel when adding the minimum spacing is 0.94 units. Thus, with a grid size of one unit, routes have a width that is approximately the same as the physical width.

Routes cannot occupy cells that are already occupied by components. However,

routes are allowed to occupy cells that are occupied by component border. Furthermore, multiple routes are allowed to occupy the same cell. This is the case when two routes intersect or if two routes overlap, which means that they share a route segment.

To meet these requirements a layer model, $\mathcal{Z}$, is introduced. The layer model associates a set of occupants for a given cell in $\mathcal{G}$. There are three different types of occupants: Components, component border, and routes. A component is always the only occupant of a cell. Let $\mathcal{Z}(c)$ be the set of occupants at cell, c. Similarly let $\mathcal{Z}_R(c)$ be the set of routes at cell, c. A notation for the set of routes at a cell proves useful when basing the placement on routing quality. This is discussed in chapter 6. Figure 2.5 illustrates the layer model.

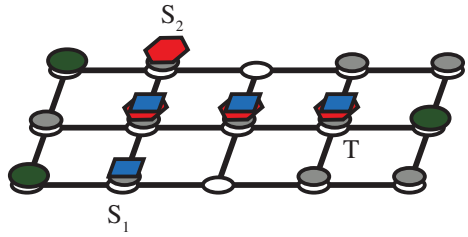


Figure 2.5: 3D-visualization of the different layers in the layer model. Associated with each cell is the set of occupants at the cell. Occupants are either components (green circles), component border (small gray circles), or routes (blue squares for route (S_1, T) and red polygons for route (S_2, T).

The figure shows two routes beginning in cell S_1 and S_2, respectively, and both ending in T. The set of cells associated with route (S_1, T) is visualized with blue squares while the set of cells associated with route (S_2, T) is visualized with red polygons. Both routes are associated with the cells where the two routes overlap.

Chapter 3

Problem Formulation

This thesis deals with the problem of designing a component placement algorithm for flow-based microfluidic biochips. The placement is based on interconnections between components and the sizes of individual component types. Each component must be placed so the design rules from table 2.2 are satisfied. The objective of the placement is to provide the best possible conditions for the subsequent design phases.

For the flow channel routing phase, this means enabling short channel routes with few intersections. Short routes minimize the time spend transporting fluids from one component to another, thus reducing the overall application completion time. Short routes also imply that the components are packed closely together which optimizes the chip area. Avoiding intersections allows liquids to flow in multiple channels simultaneously, which increases the parallelizability of the application and also reduces completion time.

The number of intersections also affects the control channel routing phase. Each intersection is handled by introducing a switch which consists of up to four valves. Minimizing the number of intersections reduces the number of valves. A small number of valves means a smaller problem size for the control channel routing phase.

Finally, the application mapping phase depends on the placement algorithm.

The application mapping phase might require storage facilities to temporarily store the output of a component, so the component can be reused for another operation. At the time of the placement, the storage requirements cannot be determined, so the placement algorithm must ensure sufficient storage facilities.

3.1 Formalization

The algorithm takes two inputs, a netlist, $\mathcal{N}$, and a component library, $\mathcal{L}$. $\mathcal{N}$ contains all components and their interconnections. $\mathcal{L}$ contains the physical dimensions of all components along with the location of the entry and exit points.

The components are placed on a biochip architecture grid, $\mathcal{G}$, such that the design constraints are satisfied, and the length of the flow channel routes and the number of intersections are minimized. An unambiguous way to judge the quality of a placement must be defined in order to find a placement that minimizes route length and intersections.

Furthermore, it must be ensured that all component exit points are connected to a storage entry point, and all component entry points are connected to a storage exit point. This ensures that all components have storage facilities. Finding such a placement involves adding a storage component to $\mathcal{N}$ if one is not already present. The storage component is bi-directionally connected to all other components, excluding inputs and outputs.

The output of the algorithm is the biochip architecture represented as a grid graph, $\mathcal{G}$. It contains all the components at their respective positions. The minimum chip size needed to contain that particular placement can be extracted from $\mathcal{G}$.

Chapter 4

Simulated Annealing

The solution space for component placements exponentially increases with the number of components and the chip size. This makes the problem presented in chapter 3 NP-complete. Thus, it is not possible to design an algorithm that computes the optimal solution in polynomial time. Instead, a suitable heuristic is needed to find a good solution in polynomial time.

Simulated annealing has previously been successfully used to place electronic components on Very Large Scale Integration (VLSI) chips [6]. The placement problem for VLSI is very similar to the placement problem for microfluidic biochips, which makes it reasonable to believe that simulated annealing is also well suited for our problem. This thesis explores the possibilities of using simulated annealing for the component placement phase of the biochip design process. This chapter explains the general concept of simulated annealing and describes an implementation adapted to the component placement problem.

4.1 Concept

The basic concept of simulated annealing is inspired by the slow cooling of solid material in a heat bath. This process is known as annealing. The cooling rate

affects the properties of the cooled material. It turns out that a simulation of the annealing process converges towards the optimal solution [9]. This property makes it suitable for minimization problems including the component placement problem.

The idea is to randomly create an initial solution, s_0, and choose an initial temperature, t_0. A neighboring solution of s_0 is explored to see if that solution is an improvement. The quality of the solutions is based on a *cost function*. If the neighbouring solution is better, that solution is accepted as the current one, and the temperature is reduced according to a *cooling schedule*. But even if the explored solution is worse it is accepted with some probability. Otherwise, the algorithm would only converge towards a local optimum and not the global optimum.

The intuition is that when the temperature is high a large part of the solution space is explored. Consequently, at high temperatures almost any solution is accepted, regardless of the quality. As the temperature decreases the simulation gradually becomes more and more reluctant to accept inferior solutions. At some point it will only accept improving solutions, causing it to become stable at a local optimum. If the cooling is slow this local optimum is a good approximation on the global optimum. For a minimization problem the result is a reduction of the cost over time, as shown in figure 4.1.

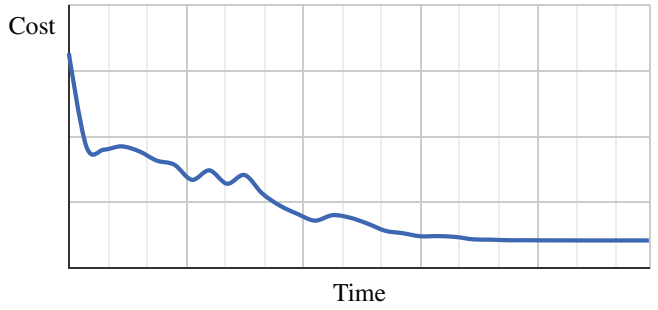


Figure 4.1: The simulated annealing algorithm minimizes the cost over time until a stable non-improving solution is found. The graph shows the cost as a function of time for the placement algorithm performed on a benchmark netlist.

The probability, $p(s)$, that an inferior solution, s, is accepted, is given by the exponential expression:

$$p(s) = e^{-\delta/t} \tag{4.1}$$

where δ is the cost increase from the previous solution, and t is the temperature. The probability is inspired by the laws of thermodynamics.

4.2 Implementation

Algorithm 1 shows a simulated annealing implementation adapted to the placement problem. The algorithm takes as input a netlist, $\mathcal{N}$, and a component library, $\mathcal{L}$, and outputs a biochip architecture in the form of a grid graph, $\mathcal{G}$, containing the placed components.

Algorithm 1 Component placement using simulated annealing.

```
1:  function PLACEMENT(𝒩, ℒ)
2:      𝒢 ← INITIALPLACEMENT(𝒩, ℒ)
3:      t ← t_0                                  ▷ Temperature is initialized, t_0 > 0.
4:      repeat
5:          for i ← 0 to n_rep do
6:              𝒢' ← RANDOMNEIGHBOUR(𝒢)
7:              δ ← COST(𝒢') − COST(𝒢)
8:              if δ < 0 ∨ RANDOM(0, 1) < e^(−δ/t) then 𝒢 ← 𝒢'
9:          t ← α(t)                             ▷ Temperature is reduced.
10:     until t ≤ t_termination
11:     return 𝒢
```

Whether to accept a solution candidate, $\mathcal{G}'$, or keep the current one, $\mathcal{G}$, is decided on line 8. A superior solution ($\delta < 0$) is always accepted, but an inferior solution is accepted with the probability from equation 4.1. This criterion is met by generating a uniformly random number between 0 and 1, which must be less than the acceptance probability, $p(s)$.

Before the annealing process can start certain parameters must be decided:

- How to find an initial solution (line 2).
- The initial temperature, t_0 (line 3), and the termination temperature, $t_{termination}$ (line 10), which defines when the simulation stops.
- The number of iterations at each temperature, n_{rep} (line 5).
- The temperature reduction function, $\alpha(t)$ (line 9). It defines the rate of cooling.
- The definition of neighbouring solutions. This is necessary in order to select a random neighbour (line 6).
- The cost function that defines the quality of a solution (line 7). The simulation converges towards the lowest cost function value.

The considerations and decisions regarding the parameters are discussed in the following subsections. In general, the parameters must be adjusted experimentally to obtain the best results.

4.2.1 Cooling Schedule

The temperature cooling schedule consists of four parameters: Initial value, reduction function, termination value, and repetitions at each temperature.

Initially, almost all solutions should be accepted in order to explore the entire solutions space. Thus, the initial value should be high enough to lead to an acceptance rate of close to 100%. The specific value varies with the problem size and the cost function definition, and must be decided experimentally.

The reduction function determines how fast the temperature cools down. Many different reduction functions have been proposed [9]. One of the simplest and most successful is of the form:

$$\alpha(t) = k \cdot t \tag{4.2}$$

where $k < 1$. This reduction function is used throughout the thesis. To ensure slow cool-down values between 0.98 and 0.999 have been used for k.

Doing many repetitions at each temperature explores more of the solution space. Since superior solutions are accepted unconditionally, spending more time at each temperature tends to lead to better final solutions. The number of repetitions at each temperature, n_{rep}, can be used to scale the running time of the algorithm to ensure a thorough exploration of the solution space. Values between 1 and 500 have been used to obtain results.

In theory, the cooling should continue until the temperature is zero. However, in practice the heuristic often settles for a non-improving stable solution long before the temperature reaches zero. The temperature at which this happens varies with the problem size and cost function. Values below 5 have provided good results for the termination temperature, $t_{termination}$.

4.2.2 Initial Placement

It has been reported [13] that the initial solution of simulated annealing has a negligible effect on the final solution, provided that the initial temperature

is high enough and the cool-down is slow. Consequently, any feasible initial placement is adequate.

The chosen initial placement approach is as follows: The components are listed in an arbitrary order. The first component, c_1, is placed in the upper left corner of the biochip. The second component, c_2, is placed such that the left border of c_2 is aligned with the right border of c_1, and the top border of c_2 is aligned with the bottom border of c_1. The remaining components are placed in the same way. Figure 4.2 shows the resulting placement.

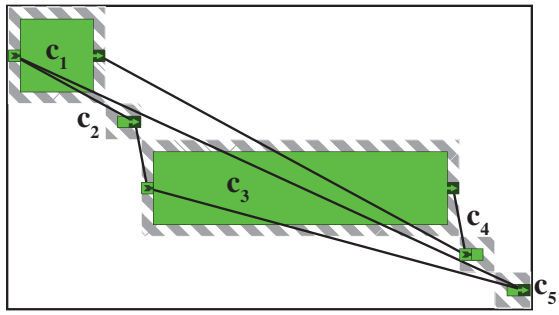


Figure 4.2: Initial placement used with simulated annealing. The shaded area indicates component border, which is not allowed to overlap.

There are several advantages of using this initial placement compared to a random initial placement. The solution is by definition feasible, because no components overlap, and the border ensures that the design rules are satisfied. Furthermore, the minimum size of the chip is given by the sum of the widths and the sum of the heights of all components. The chip size can be determined before the components are placed, so it is guaranteed that all components will fit on the chip. The chip size is also large enough to allow components to move freely on the chip.

4.2.3 Neighbourhood

The neighbourhood, $H(\mathcal{G})$, defines the set of solutions that are candidates to replace $\mathcal{G}$. The size of $H(\mathcal{G})$ must be limited to facilitate that a random solution, $\mathcal{G}'$, is quickly chosen from $H(\mathcal{G})$. $\mathcal{G}'$ is constructed by randomly performing one of three operations: Move of one component, rotation of one component, or swapping of two components. The operations are randomly chosen according to a discrete probability distribution. The most successful distribution has been found to be $(60\%, 20\%, 20\%)$ for moves, rotations, and swaps, respectively.

The components that are affected by a given operation are randomly picked among all components in $\mathcal{N}$. $H(\mathcal{G})$ only contains feasible solutions. Thus, it must be checked that the affected components are inside the chip and not overlapping other components. If an operation results in an infeasible solution it is discarded and a new one is chosen.

4.2.3.1 Move

The move operation is performed by randomly choosing a direction and how much to move in that direction. The operation is only feasible if all the cells in $\mathcal{G}$, which correspond to the new placement of the component, are either free or occupied by the component itself.

The maximum translation, T_{max}, determines how far a component can be moved from its original position. A large T_{max} explores a large part of the solution space in few moves, but also results in more infeasible and unaccepted moves. If T_{max} is large and the temperature is low, it is very likely that the simulated annealing algorithm rejects the operation. To minimize this problem, T_{max} is reduced at the same rate as the temperature until $T_{max} = 1$. Initially, T_{max} is set to one fourth of the chip size. This value is found to be a reasonable compromise between exploring much of the solution space and avoiding a lot of infeasible moves.

4.2.3.2 Rotation

Rotation is done according to figure 4.3. The new orientation is chosen randomly and rotation is done around the center of the component. Like the move operation, the rotation operation is only feasible if the affected cells are free or occupied by the component itself.

4.2.3.3 Swap

Swapping is performed by interchanging the positions of the two affected components. The orientations of the components are unchanged. The operation is only feasible if the cells, which correspond to the new positions of the two components, are free or occupied by either one of the affected components.

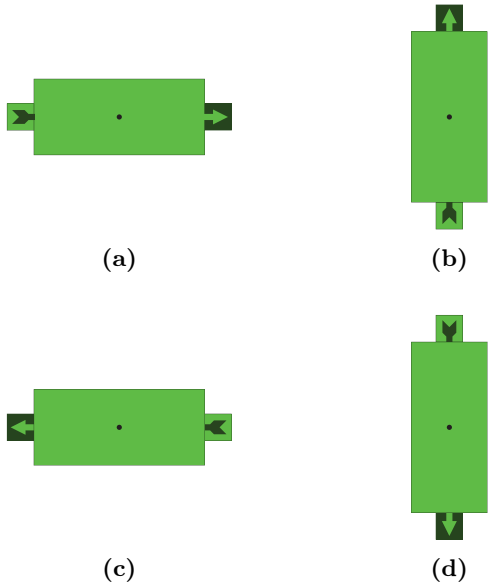


Figure 4.3: Illustration of the four orientations. (a) is the default orientation. Rotation is done around the center point.

4.2.4 Cost Function

The cost function must reflect the quality of the placement. As mentioned in section 3.1, the objective of the placement is to minimize the length of the flow channel routes and the number of intersections.

An obvious cost function is obtained by routing all the channels and calculating the route length and intersections. But due to the complexity of the routing problem such a cost function is not feasible. Instead, simplifying assumptions and estimations are made in order to minimize the number of computations. Chapter 5 and 6 describe two different approaches to estimating the route length and intersections.

Chapter 5

Approximated Cost Function

The purpose of this cost function is to make a qualified approximation of the channel length and number of intersections without doing any actual routing. The idea is to define the cost function by metrics that can easily be calculated from the entry and exit points of the components. This way, no computation time is spend doing the actual routing.

5.1 Metrics

From $\mathcal{N}$ and $\mathcal{L}$, the connection model is constructed as a bipartite graph, $\mathcal{C} = (U, V, E)$. An edge, e_{uv} in $\mathcal{C}$ represents a connection from exit point, u, to entry point, v. See section 2.2.2. $\mathcal{C}$ enables the introduction of three metrics: The total length of all edges, the total squared length of all edges, and the number of intersections between edges. The metrics are described in detail in the following subsections. The choice of metrics is inspired by [5].

5.1.1 Approximated Total Length

Under ideal conditions, the length of the route between an exit point, u, and an entry point, v, is given by the Manhattan distance between u and v. Thus, the Manhattan distance is a lower bound on the route length between u and v. The majority of routing algorithms seek to minimize the total route length. Based on this, the Manhattan distance is expected to be a good approximation for the route length. Figure 5.1 shows an example where the Manhattan distance is equal to the route length.

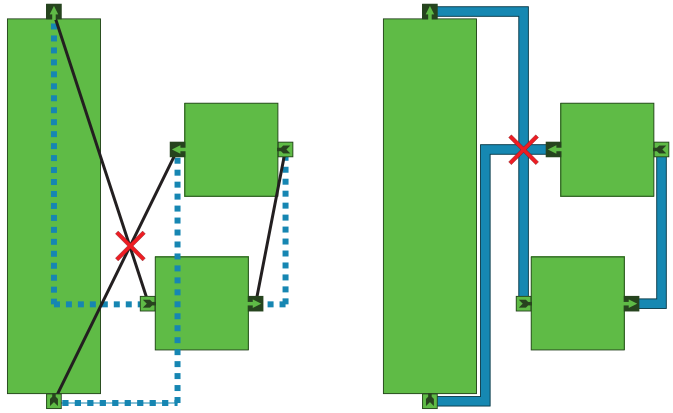


Figure 5.1: Illustration of $\mathcal{C}$ with Manhattan distances, and line segment intersection (a), and the corresponding routing (b). The dashed lines are the Manhattan distances, and the red crosses indicate intersection.

The Manhattan length for an edge, e_{uv}, from u to v is:

$$||e_{uv}|| = |u_x - v_x| + |u_y - v_y| \tag{5.1}$$

The approximated total length of the routes is found as the sum of the Manhattan lengths of all edges:

$$L_A = \sum_{e_{uv} \in E} ||e_{uv}|| \tag{5.2}$$

5.1.2 Approximated Total Squared Length

The approximated total squared route length is found in a similar way:

$$S_A = \sum_{e_{uv} \in E} ||e_{uv}||^2 \tag{5.3}$$

The motivation for this metric is to make all routes roughly the same length by penalizing long routes. This evens out the routing latencies, making all operations on the chip take roughly the same amount of time.

5.1.3 Approximated Number of Intersections

An edge, e_{uv}, in $\mathcal{C}$ defines a connection between u, and v. Thus, it also represents the line segment that goes from u to v. An intersection between such two line segments often leads to an intersection between the two corresponding channel routes. This is illustrated in figure 5.1, where an intersection between line segments in figure 5.1a results in an intersection between routes in figure 5.1b. Consequently, the total number of intersections between line segments is used to estimate the total number of route intersections. First, we define:

$$I_A(e_{uv}, e_{pq}) = \begin{cases} 1 & \text{If } e_{uv} \text{ and } e_{pq} \text{ intersect} \\ 0 & \text{Otherwise} \end{cases} \tag{5.4}$$

Then, the total number of intersections is calculated as:

$$N_A = \sum_{e_{uv} \in E} \sum_{e_{pq} \in E \setminus e_{uv}} I_A(e_{uv}, e_{pq}) \tag{5.5}$$

5.1.3.1 Determine Intersection between Two Line Segments

Consider two line segments, s_{uv} and s_{pq}. Let l_{uv} and l_{pq} be lines of infinite length that go through the end points of s_{uv} and s_{pq}, respectively. The lines are mathematically expressed as:

$$l_{uv} : F(x, y) = 0 \qquad\qquad l_{pq} : G(x, y) = 0 \tag{5.6}$$

where $F(x, y)$ and $G(x, y)$ are of the form:

$$a \cdot x + b \cdot y + c = 0 \tag{5.7}$$

For two points, (x_1, y_1) and (x_2, y_2), not on l_{uv} we have $F(x_1, y_1) \neq 0$ and $F(x_2, y_2) \neq 0$. Furthermore, if the two points are on opposite sides of l_{uv} they have different signs [8], thus $F(x_1, y_1) = -F(x_2, y_2)$.

The strategy is to use this property to determine if s_{uv} and s_{pq} intersect. The two segments only intersect in the case that (1) the end points, u and v, are on opposite sides of l_{pq}, and (2) p and q are on opposite sides of l_{uv}.

5.2 Computing the Cost Function

The cost function is defined by all three metrics mentioned in the previous section. The cost function is computed as:

$$Cost_A(\mathcal{G}) = \alpha N_A + \beta L_A + \gamma S_A \tag{5.8}$$

where α, β, and γ are constant weights. The exact values of the weights depend on which metrics should be given highest priority. This is decided by the biochip designer based on parameters like which applications should run on the chip, available chip area, etc.

Equation 5.8 unambiguously expresses the quality of a placement with respect to the described metrics and for the chosen values of α, β, and γ. This means that the simulated annealing algorithm can compare two solutions and choose the better one, or with some probability, choose the inferior one.

5.2.1 Analysis

The cost function is computed for each iteration of the simulated annealing algorithm and has great impact on the running time. This is also the reason why it is necessary to approximate the quality of the routes instead of doing the actual routing.

Let n denote the number of connections, $|E|$, in $\mathcal{C}$. According to equation 5.2 and 5.3, L_A and S_A are computed by summing over all n connections and calculating the Manhattan distance between the exit and entry points. The Manhattan distance is calculated in $O(1)$ time so the time to calculate L and S is $O(n)$.

According to equation 5.5, N_A is computed by summing over all pairs of line segments and calculating whether they intersect or not. Intersection is checked

in $O(1)$ time. There is one line segment for each connection, thus, there are $O(n^2)$ pairs of line segments. This results in $O(n^2)$ computations to calculate the cost function.

5.2.2 Incremental Update

The neighbour, $\mathcal{G}'$, (section 4.2.3) to a solution, $\mathcal{G}$, is identical to $\mathcal{G}$ except that one component is moved or rotated, or two components are swapped. Consequently, it is needless to recalculate all intersections and all lengths for every iteration of the algorithm. Instead, the cost function is computed for the first iteration and updated incrementally from that point on. The cost function is then calculated as:

$$Cost_A(\mathcal{G}') = \alpha(N_A + \Delta N_A) + \beta(L_A + \Delta L_A) + \gamma(S_A + \Delta S_A) \tag{5.9}$$

where N_A, L_A, and S_A are the values for $\mathcal{G}$, and ΔN_A, ΔL_A, and ΔS_A, denote the difference between $\mathcal{G}'$ and $\mathcal{G}$.

ΔN_A, ΔL_A, and ΔS_A are only affected by the connections going into or out of the changed components. Let $\mathcal{E}$ denote the subset of edges that are connected to components that change from $\mathcal{G}$ to $\mathcal{G}'$. Furthermore, let $N(\mathcal{E})$ and $N'(\mathcal{E})$ be the number of intersections for edges $\mathcal{E}$ in solution $\mathcal{G}$ and $\mathcal{G}'$, respectively. Then ΔN_A is given by:

$$\Delta N_A = N'_A(\mathcal{E}) - N_A(\mathcal{E}) \tag{5.10}$$

The same notation can be used to describe ΔL_A and ΔS_A:

$$\Delta L_A = L'_A(\mathcal{E}) - L_A(\mathcal{E}) \tag{5.11}$$

$$\Delta S_A = S'_A(\mathcal{E}) - S_A(\mathcal{E}) \tag{5.12}$$

ΔL_A and ΔS_A are straightforward to compute using the approach in equations 5.2 and 5.3. However, ΔN_A, requires an auxiliary data structure, D. $D(e)$ associates each edge, e, with a list of edges, which intersect e. Using D, $N_A(\mathcal{E})$ is computed without recalculating the intersections:

$$N_A(\mathcal{E}) = \sum_{e_{uv} \in \mathcal{E}} |D(e_{uv})| \tag{5.13}$$

where $|D(e_{uv})|$ denotes the length of the intersection list associated with e_{uv}.

The number of intersections in $N'_A(\mathcal{E})$ is computed as:

$$N'_A(\mathcal{E}) = \sum_{e_{uv} \in \mathcal{E}} \left(\sum_{e_{pq} \in E \setminus \mathcal{E}} I_A(e_{uv}, e_{pq}) + \sum_{e_{pq} \in \mathcal{E} \setminus e_{uv}} \frac{I_A(e_{uv}, e_{pq})}{2} \right) \tag{5.14}$$

It is necessary to distinguish between the edges in $\mathcal{E}$ and the rest of the edges. An intersection between an edge in $\mathcal{E}$ and an edge not in $\mathcal{E}$ only occurs once in the sum. But an intersection between a pair of edges, e_i and e_j, in $\mathcal{E}$ occurs in the sum both as (e_i, e_j) and (e_j, e_i). Thus, the contribution from such pairs is divided by two.

Incremental update decreases the number of computations by only recomputing the cost function for modified edges. The number of changed components is either one or two, with a total of $|\mathcal{E}|$ connections. In the worst case $|\mathcal{E}| = n$. Each of these connections might intersect with any other connection so the total time to compute ΔN_A is $O(n^2)$. Thus, in the worst case there is no asymptotic improvement, but in a typical case the speedup is significant. Section 7.3.2 indicates that incremental updates are 5 to 30 times faster than non-incremental updates.

Chapter 6

Routed Cost Function

The cost function described in this chapter uses actual routing to judge the quality of the placement. Due to the complexity of the routing problem the routing algorithm must be fast and simple. This is necessary to ensure feasible computation time.

The routed cost function is merely a tool to do the component placement. The actual channel routing is done in the flow channel routing phase of the biochip design process. Thus, the final routing might differ from the routing done by the cost function. However, the route length and the number of intersections calculated by the routed cost function is a good upper bound on what is achievable in the flow channel routing phase.

6.1 Routing Algorithm

The initial step is to decide on a routing algorithm. Many algorithms have been proposed in the literature and have proved successful for routing VLSI architectures [10]. In the following sections we describe two related algorithms. Lee's algorithm, which is inefficient but always finds the shortest route, and Soukup's algorithm, which is faster but does not guarantee to find the shortest route. Both algorithms take as input a grid graph of vertices that are either

occupied or available. This is in accordance with the proposed architecture model, thus the algorithms can be directly applied to the grid graph, $\mathcal{G}$. The algorithms also take as input a source, s, and a target, t. The output of the algorithm is a route going from s to t.

6.1.1 Lee's Algorithm

Lee's algorithm [10] is basically a breadth-first search. A wave front starts in s, and expands until t is discovered. See figure 6.1. Due to properties of the breadth first search, the route found by Lee's algorithm is guaranteed to be the shortest. In fact, Lee's algorithm will find the shortest distance from s to any cell it discovers. This means that Lee's algorithm can route from s to multiply targets at once. Due to its simplicity, Lee's algorithm is widely used.

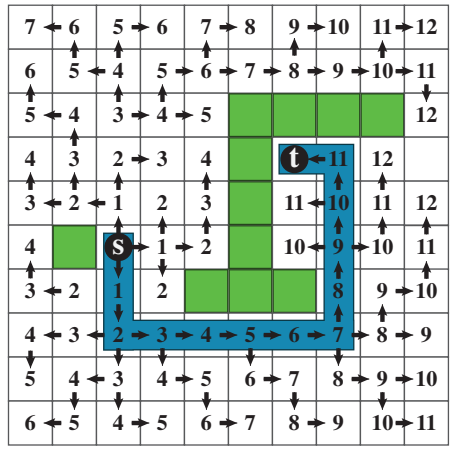


Figure 6.1: Visualization of propagating waves in Lee's algorithm. The numbers indicate the wave front number, which is also the distance to s. The arrows indicate the direction from which a cell is discovered. The route is found by retracing the arrows from t to s.

Algorithm 2 describes an implementation of Lee's algorithm. Two lists, w and w', are maintained (lines 2-3). w is the current wave front, which initially only contains s, and w' is the next wave front, which is initially empty.

All the neighbours of cells in w, that have not yet been visited, are explored (lines 7-12). Neighbours are assumed to be explored in a counter-clockwise

order starting from the top, and cells are retrieved from the end of w. As the neighbours are explored they are added to w' (line 10).

If t is found among the neighbours, the route is returned and the algorithm terminates (lines 11-12). Otherwise, additional wave fronts are needed to find t. The cells in w' are moved into w and the process is repeated (lines 13-15). The algorithm repeats until t is found or until there are no more cells to explore.

Algorithm 2 Routing using Lee's Algorithm.

```
1:  function LEES(G, s, t)
2:      w ← s                                          ▷ Current wave front
3:      w' ← ∅                                         ▷ Next wave front
4:      repeat
5:          for each cell, c, in w do
6:              neighbours ← UNVISITEDNEIGHBOURS(G, c)
7:              for each cell, c', in neighbours do
8:                  d ← DIRECTION(c, c')
9:                  DISCOVERED(c', d)
10:                 APPEND(c', w')
11:                 if c' = t then                     ▷ Target found
12:                     return RETRACEROUTE(G, t)
13:         w = w'                                     ▷ Move on to next wave front
14:         w' = ∅
15:     until w ≠ ∅
16:     return ∅                                       ▷ Entire chip searched, no route found
```

6.1.1.1 Finding the Actual Route

Algorithm 2 only finds the target, it does not find the actual route. An auxiliary function, *RetraceRoute*$(\mathcal{G}, t)$, is introduced to retrace the route from t to s. As a cell, c, is discovered by algorithm 2 it is recorded from which direction c is discovered (lines 8-9). Because a cell is only discovered once this leaves a trail that can be followed from t.

The directions of discovery is indicated by the arrows in figure 6.1. *RetraceRoute*$(\mathcal{G}, t)$ simply starts at t and follows the arrows backwards until s is found. The list of cells from s to t is returned.

6.1.1.2 Analysis

In the worst case Lee's algorithm will search the entire search space before t is found. The architecture grid graph, $\mathcal{G}$, has dimensions $h \times w$, so the time for finding a route between s and t is $O(h \times w)$. The direction from which a cell is discovered must be stored for each cell. This yields a total space usage of $O(h \times w)$.

6.1.2 Soukup's Algorithm

The main drawback of Lee's algorithm is the large number of cells searched for the average case. The wave front propagates equally in all directions, which means that a large fraction of the computation time is spend on exploring cells that are not directed towards t.

Soukup's Algorithm [11] is an extension of Lee's algorithm. If a cell, c, has neighbours that are closer to the target, a line search is conducted in those directions. See figure 6.2. The line search stops when reaching an obstacle or

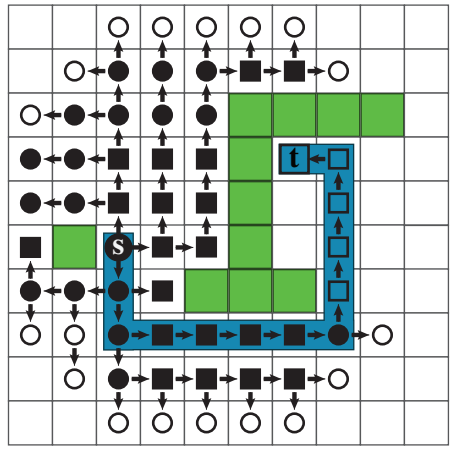


Figure 6.2: Visualization of Soukup's algorithm. Circles represent discovery with Lee expansion, squares represent discovery with line search. Filled symbols means that all neighbours of that cell have been explored. The arrows indicate from which direction a cell is discovered.

when not further approaching t. From this point a new line search might be

conducted. If no neighbour is closer to t then Lee's algorithm is applied until another cell is found, from which a line search can be conducted. Effectively, this means that Lee's algorithm is used to search around obstacles. The result is that the search space is minimized by first trying to move in the direction of t and when t cannot be approached any further, Lee's algorithm is applied to get around any obstacles.

Algorithm 3 describes an implementation of Soukup's algorithm. The structure is similar to that of Lee's algorithm. Soukup's algorithm also maintains two wave front lists, w and w' (lines 2-3), although the wave fronts are not as easily visualized.

Like Lee's algorithm, the neighbours of cells in w are explored (lines 8-20). If a neighbour, n, is closer to t than the original cell, a line search is conducted in this direction (lines 12-17). As the cells are discovered on the line search they are added to w. The effect is, that the next neighbours to be explored are those belonging to the last cell in the line search. So far, this cell is the closest to t, thus we wish to continue the search from here.

If a neighbour it not closer to t it is instead inserted at the beginning of w' (lines 18-20). When w is empty, all the neighbours of cells in w have been explored. This means that no more cells can be discovered with line search. Consequently, a new wavefront is initiated by moving w' into w. Like Lee's algorithm, Soukup's algorithm terminates when there are no more cells to explore, or when t has been found. Because t is always closer to itself than any other cell it is always discovered with line search. Thus, it is sufficient to only check if t is found when line searching (lines 14-15).

6.1.2.1 Finding the Actual Route

Cells discovered by line search and cells discovered by Lee expansion must be distinguished (line 13 and 19). At a given cell, a line search is allowed to replace a Lee expansion but not the other way around. This requires that an *unvisited* neighbour in algorithm 3 refers to cells that are either available or discovered by Lee expansion (line 7 and 17).

Furthermore, only unmarked cells can be marked as discovered by Lee expansion, and each cell marks itself as discovered with line search (line 6). This is done so that the marking cannot be overwritten at a later time. It ensures that each cell is only discovered by one cell, and thus preserves a trail from t to s. The trail is followed in the same way as the retrace function for Lee's algorithm (section 6.1.1.1)

Algorithm 3 Routing using Soukup's Algorithm.

```
 1: function SOUKUPS(G, s, t)
 2:     w ← s                                          ▷ Current wave front
 3:     w' ← ∅                                         ▷ Next wave front
 4:     repeat
 5:         for each cell, c, in w do
 6:             MARKWITHLINESEARCH(c)
 7:             neighbours ← UNVISITEDNEIGHBOURS(G, c)
 8:             for each cell, c', in neighbours do
 9:                 d ← DIRECTION(c, c')
10:                 if c' is closer than c to t then
11:                     n ← c'
12:                     while n is approaching t do          ▷ Conduct line search
13:                         DISCOVERED(n, d, LineSearch)
14:                         if n = t then                          ▷ Target found
15:                             return RETRACEROUTE(G, t)
16:                         APPEND(n, w)
17:                         n ← UNVISITEDNEIGHBOURINDIRECTION(G, n, d)
18:                 else if c' not already discovered then
19:                     DISCOVERED(c', d, LeeExpansion)
20:                     INSERT(c', w')
21:         w = w'                                 ▷ Move on to next wave front
22:         w' = ∅
23:     until w ≠ ∅
24:     return ∅                          ▷ Entire chip searched, no route found
```

6.1.2.2 Analysis

In the worst case Soukup's algorithm will search the entire search space just like Lee's algorithm and it must also store the retrace direction for each cell. This gives a worst case time and space usage of $O(h \times w)$ for an architecture grid graph, $\mathcal{G}$, with dimensions, $h \times w$.

In practice, however, Soukup's algorithm performs much better. Especially for the biochip routing problem, where component borders ensure that s and t are always routable. Soukup [11] states that his algorithm is typically 10-50 times faster than Lee's algorithm. Our own experiments suggest that it is 5-130 times faster for instances of the biochip routing problem (section 7.3.3).

The routing cost function only serves as an estimation of the final routing. For estimation purposes, Soukup's algorithm is preferable to Lee's algorithm due

to its faster routing time. It is also acceptable that Soukup's algorithm does not guarantee to find the shortest route. Based on these observations Soukup's algorithm is used as the primary routing algorithm for the routed cost function.

6.2 Metrics

The route found by the routing algorithm enables the metrics of section 5.1 to be estimated more accurately. The three metrics are: The total length of all routes, the total squared length of all routes, and the number of route intersections. The routing also introduces another metric: The total amount of overlap. The following subsections describe the four metrics in detail.

6.2.1 Total Length and Total Squared Length

The length of a route is given by the number of cells in the route. Recall from section 2.2.4 that $r(e)$ denotes the set of cells in the route corresponding to edge, e. Furthermore, $|r(e)|$ denotes the number of cells in $r(e)$ and thus the length of the route. Let E be the set of all edges in the connection model, $\mathcal{C}$. Then, the total length for all routes is found as:

$$L_R = \sum_{e \in E} |r(e)| \tag{6.1}$$

The total squared length of all routes is calculated in a similar way:

$$S_R = \sum_{e \in E} |r(e)|^2 \tag{6.2}$$

Except for the modified length definition, the two sums are identical to L_A and S_A for the approximated cost function (section 5.1.1 and 5.1.2).

6.2.2 Number of Intersections

Where routes intersect, a switch is needed in order to direct the flow. If many routes intersect at the same cell, still only one switch is needed. To let the number of intersections reflect the number of introduced switches, this should also only count as one intersection. Thus, the total number of intersections is the number of cells where two or more routes intersect. Figure 6.3 shows an example with intersecting routes.

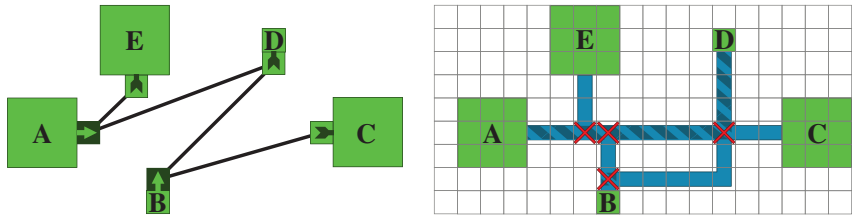


Figure 6.3: Example of a connection model, $\mathcal{C}$ (a), and the corresponding routes (b). Notice how routes intersect and overlap. Intersections are visualized by red crosses, and the shaded area indicates that the flow channel is used by multiple routes.

Intersections occur if the set of routes at one cell, c_1, differs from the set of routes at another, adjacent cell, c_2. Intersections are introduced in three cases:

1. One or more routes at cell, c_1, are not at cell, c_2. This means that routes have stopped or changed direction at c_1. Thus, an intersection is introduced at c_1.

2. Identical to case one, but with c_1 and c_2 interchanged. An intersection is introduced at c_2.

3. One or more routes at cell, c_1, have been replaced by a number of other routes at cell, c_2. This means that routes have stopped, started, or changed directions both at c_1 and c_2. Consequently intersections are introduced both at c_1 and c_2.

Figure 6.4 visualizes the routing lines of the example from figure 6.3b. The routing lines illustrate the flow channels that the individual routes use, and show which routes cause intersections.

The two adjacent intersections in figure 6.4 are both instances of case three. The route between A and E changes direction in the left cell and the route between B and C appears in the right cell. The two other intersections are instances of case one or two depending on, from which cell they are discovered.

Recall from section 2.2.4 that the layer model, $\mathcal{Z}$, associates a cell with the occupants of that cell. In particular, $\mathcal{Z}_R(c)$, is the set of routes at cell, c. Based on the three cases the set of intersection cells, $I_R(c_1, c_2)$, between adjacent cells,

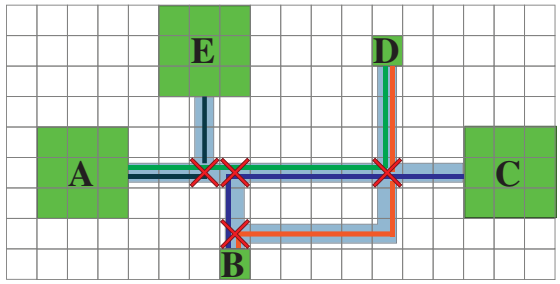


Figure 6.4: Example with flow channel intersections. Each route is visualized with a thin line.

c_1 and c_2, is formalized as:

$$I_R(c_1, c_2) = \begin{cases} c_1 & \mathcal{Z}_R(c_1) \supset \mathcal{Z}_R(c_2) \\ c_2 & \mathcal{Z}_R(c_1) \subset \mathcal{Z}_R(c_2) \\ c_1, c_2 & \mathcal{Z}_R(c_1) \neq \mathcal{Z}_R(c_2) \\ \varnothing & \text{Otherwise} \end{cases} \tag{6.3}$$

Let C_R be the set of cells, which are occupied by one or more routes. Intersections only occur at cells that are occupied by routes, so it is sufficient to only check the cells of C_R for intersections. $I_R(c_i, c_j)$ is computed for all pairs of adjacent cells, c_i and c_j, in C_R. The total set of intersection cells, I, is retrieved by taking the union of all the found intersections. The total number of intersections, N_R, is defined as $N_R = |I|$.

6.2.3 Amount of Overlap

Performing the actual routing means that the cells of all routes are known. This makes it possible to identify if routes overlap. Overlapping routes share part of their flow channel, and thus two overlapping routes cannot be in use at the same time. On the other hand, shared flow channels imply reduced total channel length. Depending on the channel length requirements and the parallelism of the biochip application, it might be desirable or undesirable to have overlapping routes. Figure 6.3b illustrates an example with overlap.

The amount of overlap at a cell, c is defined as the number of routes at c, which have another route below it. This means that cells occupied by zero or one route has no overlap. For cells with more than one route the bottommost route does not overlap any routes, but all the remaining routes overlap the route directly

below them. Figure 6.5 shows the amount of overlap at each cell for the example in figure 6.3b.

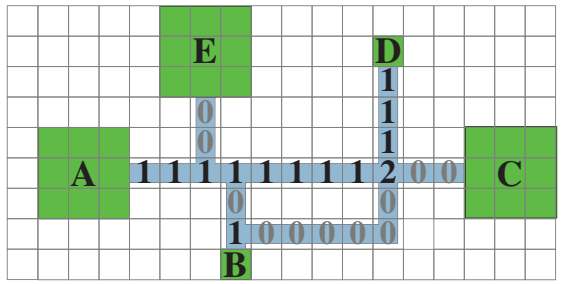


Figure 6.5: Illustration of the number of overlapping routes at each cell.

Notice that cells with only one route do not contribute to the total overlap. If two cells overlap the contribution is one, and if three cells overlap the contribution is two, etc. Also notice that there is always overlap at an intersection cell.

We define the total overlap as the sum of the overlap at each cell. The total overlap is calculated as:

$$V_R = \sum_{c \in C_R} (|\mathcal{Z}_R(c)| - 1) \tag{6.4}$$

where $\mathcal{Z}_R(c)$ is the number of routes at cell c, and C_R is the set of cells, which are occupied by one or more routes.

6.3 Computing the Cost Function

The cost function is defined by all four metrics mentioned in the previous section. It takes the same form as in section 5.2, but with the addition of the fourth metric, the amount of overlap, V_R. The cost function is computed as:

$$Cost_R(\mathcal{G}) = \alpha N_R + \beta L_R + \gamma S_R + \omega V_R \tag{6.5}$$

As mentioned in section 6.2.3, it might be desirable to minimize the total channel length. By the definition of L_R and V_R the total channel length is $L_R - V_R$. Setting $\omega = -\beta$ rewards overlapping and finds solutions with a short total channel length.

6.3.1 Analysis

According to section 6.1.2.2 the worst case time for routing between two points is $O(h \times w)$ for an architecture grid graph, $\mathcal{G}$, with dimensions $h \times w$. There are n connections, each with one route. This gives a total routing time of $O(n \cdot h \times w)$.

L_R and S_R are calculated by summing over the number of cells in all routes (equation 6.1 and 6.2). It is assumed that the number of cells in a route is stored at the time of routing, which means that the number of cells in a single route is retrieved in $O(1)$ time. Consequently, the time to calculate L_R and S_R for n routes is $O(n)$.

N_R and V_R are both calculated by iterating through all cells of all routes. If the cell has previously been visited it will be skipped. If not, N_R and V_R are computed according to section 6.2.2 and 6.2.3. An upper bound for the total time for N_R and V_R is $O(n \cdot |r(e_{max})|)$, where e_{max} is the edge, whose corresponding route has the maximum number of cells.

Routing all n routes is the bottleneck. This results in $O(n \cdot h \times w)$ computations to calculate the cost function. But as argued in section 6.1.2.2, Soukup's algorithm often performs much better.

6.3.2 Incremental Update

Like the approximated cost function (section 5.2.2) it is not necessary to recalculate all routes for each iteration of the simulated annealing algorithm. In fact, only one or two components change from solution, $\mathcal{G}$, to a neighbour solution, $\mathcal{G}'$. Thus, only the routes that are affected by the changed components need rerouting. However, the problem of determining the set of routes to reroute is more complex than determining the set of modified edges for the approximated cost function.

Once the cost function is initially computed it is incrementally updated according to:

$$Cost_R(\mathcal{G}') = \alpha(N_R + \Delta N_R) + \beta(L_R + \Delta L_R) + \gamma(S_R + \Delta S_R) + \omega(V_R + \Delta V_R) \quad (6.6)$$

where ΔN_R, ΔL_R, ΔS_R, and ΔV_R denote the change from solution $\mathcal{G}$ to solution $\mathcal{G}'$.

6.3.2.1 Routes to Reroute

Let $\mathcal{E}$ denote the subset of edges that change from solution $\mathcal{G}$ to solution $\mathcal{G}'$, and let $R(\mathcal{E})$ and $R'(\mathcal{E})$ denote the set of routes corresponding to the edges in $\mathcal{E}$ for solution $\mathcal{G}$ and $\mathcal{G}'$, respectively. These are the routes that must be rerouted from $\mathcal{G}$ to $\mathcal{G}'$.

Like the approximated cost function all edges that are connected to changed components are in $\mathcal{E}$. Furthermore, two other types of routes must be in $R(\mathcal{E})$: Routes, which are obstructed in $\mathcal{G}'$ because a changed component is blocking, and routes, which propagation was blocked in $\mathcal{G}$ by a changed component. The two types are illustrated in figure 6.6.

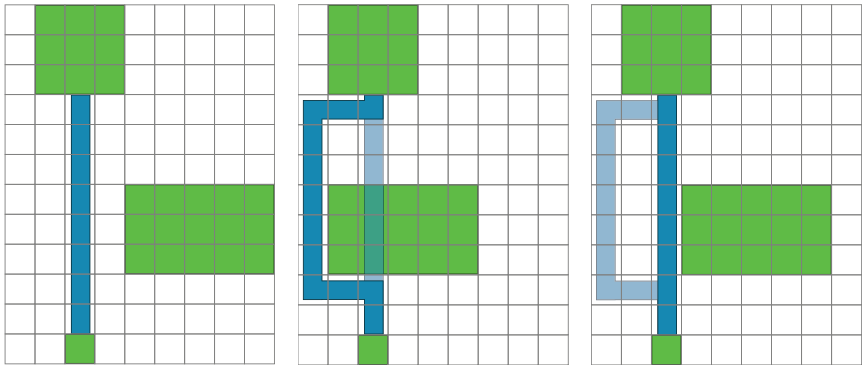


Figure 6.6: Illustration of an initial route (a), a reroute due to obstruction (b), and a reroute due to removed obstruction (c). Notice that the reroute in (c) is required in order to make the incremental update match an initial routing.

Routes that are obstructed by a changed component in $\mathcal{G}'$ must be rerouted to ensure that all routes are feasible. This situation is shown in figure 6.6b. Obstruction is checked when the component is occupying the cells of its new position.

Routes, which in the propagation phase in $\mathcal{G}$ were blocked by a changed component, might have chosen an inferior route because of the position of the changed component. See figure 6.6c. The inferior route is still valid in $\mathcal{G}'$, but the route should be rerouted to find the best route. In addition, the incremental routing should always correspond to a non-incremental routing, which requires that the inferior route is rerouted. To implement this, each component maintains a list of the routes that reached the component during the wave propagation. When

the component is changed, all the routes in the list are rerouted.

6.3.2.2 Length and Squared Length

Let $L_R(\mathcal{E})$ denote the total length of routes in $R(\mathcal{E})$. Similarly, let $L'_R(\mathcal{E})$ denote the total length of routes in $R'(\mathcal{E})$. Then the change of the total length is given by:

$$\Delta L_R = L'_R(\mathcal{E}) - L_R(\mathcal{E}) \tag{6.7}$$

Using similar notation ΔS_R is expressed as:

$$\Delta S_R = S'_R(\mathcal{E}) - S_R(\mathcal{E}) \tag{6.8}$$

ΔL_R and ΔS_R are calculated using equations 6.1 and 6.2.

6.3.2.3 Intersections

The change in the number of intersections is defined as:

$$\Delta N_R = N'_R(\mathcal{E}) - N_R(\mathcal{E}) \tag{6.9}$$

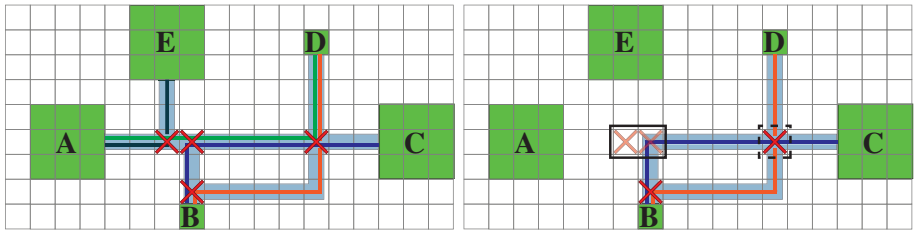
$N_R(\mathcal{E})$ corresponds to the number of intersections that are removed when routes in $R(\mathcal{E})$ are removed from $\mathcal{G}$. That is, intersection cells that no longer have an intersection after $R(\mathcal{E})$ are removed.

Let $C_{R(\mathcal{E})}$ be the set of cells which are occupied by routes in $R(\mathcal{E})$. For each cell, c, in $C_{R(\mathcal{E})}$, there are three cases to consider:

1. There is no intersection at cell, c. Removing a route can never introduce new intersections, so nothing should be done.

2. There is an intersection at cell, c. Furthermore, there is at least one pair of intersecting routes at c, where both of the routes are not in $R(\mathcal{E})$. This means that removing the routes in $R(\mathcal{E})$ will not remove the intersection, because there is still at least one intersecting pair at c.

3. There is an intersection at cell, c, and all pairs of intersecting routes at c have one route in $R(\mathcal{E})$. This means that no routes intersect at c when $R(\mathcal{E})$ are removed. In this case the intersection at c is removed.



(a) (b)

Figure 6.7: Solution, $\mathcal{G}$, with all routes (a), and with routes in $R(\mathcal{E})$ removed (b). Each route is illustrated by a line that characterizes the channel it uses. Grayed out intersections are removed with $R(\mathcal{E})$.

Figure 6.7 shows how $N_R(\mathcal{E})$ is calculated. In the example component, A, is about to be moved to the left, and thus $\mathcal{E}$ is the two edges going from A to E, and A to D.

The initial situation is showed in figure 6.7a, and figure 6.7b shows the situation where the routes in $R(\mathcal{E})$ have been removed. The intersection outlined by a dashed rectangle corresponds to case two. After removing $R(\mathcal{E})$ there is still an intersection at that cell, namely the one between routes (B, D) and (B, C).

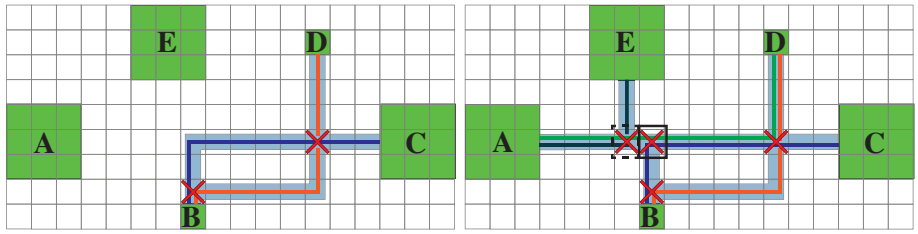
The intersections outlined by the solid rectangle correspond to case three. When removing $R(\mathcal{E})$ no route intersections remain, which means that the two intersections are removed. It is noted that in total two intersections are removed, so $N_R(\mathcal{E}) = 2$.

$N'_R(\mathcal{E})$ corresponds to the number of intersections that are introduced when the routes in $R'(\mathcal{E})$ are added to $\mathcal{G}'$. Let $C'_{R(\mathcal{E})}$ be the set of cells which are occupied by routes in $R'(\mathcal{E})$. For each cell in $C'_{R(\mathcal{E})}$ there are two cases where intersections are introduced:

1. At least one pair of routes in $R'(\mathcal{E})$ intersect at cell, c, and no intersection already exists at c.

2. At least one route in $R'(\mathcal{E})$ intersects at c with routes not in $R'(\mathcal{E})$. An intersection does not already exist at c.

Whether two routes intersect or not is detected using equation 6.3. Figure 6.8 shows how to calculate $N'_R(\mathcal{E})$. Compared to figure 6.7, component A has been moved one cell to the left.



(a) (b)

Figure 6.8: Solution, $\mathcal{G}'$, with routes in $R(\mathcal{E})$ removed (a), and with routes in $R'(\mathcal{E})$ added (b). Each route is illustrated by a line that characterizes the channel it uses. Intersections outlined with a rectangle are added with $R'(\mathcal{E})$.

The intersection outlined with a dashed rectangle corresponds to case one. No intersections exist at the cell before the routes in $R'(\mathcal{E})$ are added (figure 6.8a). The intersection is caused by routes (A, E) and (A, D), which are both in $R'(\mathcal{E})$. Thus, case one contributes with one intersection.

The intersection outlined with a solid rectangle corresponds to case two. An intersection does not already exist at the cell. (A, D), which is in $R'(\mathcal{E})$, intersects with (B, C), which is *not* in $R'(\mathcal{E})$. Case two also contributes with one intersection.

In total two intersections are introduced by adding routes in $R'(\mathcal{E})$, so $N'_R(\mathcal{E}) = 2$. It follows that $\Delta N_R = 2 - 2 = 0$ for this example, which means that moving component, A, one cell to the left does not change the number of intersections. The same procedure can be followed in general to compute ΔN_R for any solution, $\mathcal{G}$, and a neighbouring solution, $\mathcal{G}'$.

6.3.2.4 Overlap

The change in the amount of overlap is described using notation similar to the previous sections:

$$\Delta V_R = V'_R(\mathcal{E}) - V_R(\mathcal{E}) \tag{6.10}$$

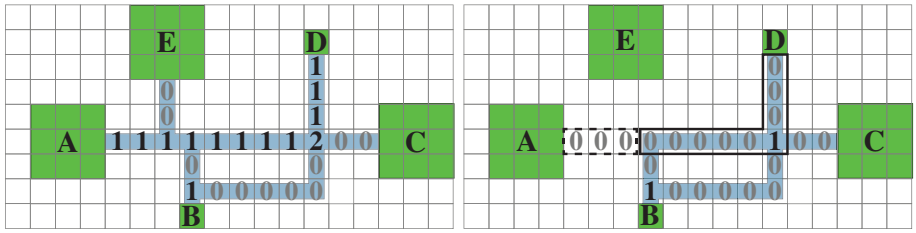
$V_R(\mathcal{E})$ corresponds to the amount of overlap that routes in $R(\mathcal{E})$ contribute to the total amount of overlay for solution, $\mathcal{G}$. Put another way, $V_R(\mathcal{E})$ is the decrease in overlap if the routes in $R(\mathcal{E})$ are removed.

For each cell, c, in $C_{R(\mathcal{E})}$, there are two cases to consider:

1. All routes at cell, c, are in $R(\mathcal{E})$. Consequently, if the routes in $R(\mathcal{E})$ are removed, no routes remain at cell, c. This means that all routes at c, except the bottommost, contribute to the total overlay.

2. Some routes at cell, c, are not in $R(\mathcal{E})$. In this case, there are routes that will remain at c, if the routes in $R(\mathcal{E})$ are removed. Thus, all routes that are at c and in $R(\mathcal{E})$ contribute to the total overlay.

Figure 6.9 illustrates how $V_R(\mathcal{E})$ is calculated, when component, A, is about to be moved. The situation where the routes, $R(\mathcal{E})$, have been removed is showed



(a) **(b)**

Figure 6.9: Solution, $\mathcal{G}$ with all routes (a), and with routes in $R(\mathcal{E})$ removed (b). The numbers indicate the overlap for each cell.

in figure 6.9b. The cells outlined by a dashed rectangle correspond to case one. The only two routes at these cells are the ones in $R(\mathcal{E})$. Removing the routes at the cells decreases the overlap by three.

The cells outlined by a solid rectangle correspond to case two. Routes still remain at these cells when $R(\mathcal{E})$ are removed. Only route (A, D) is both in $R(\mathcal{E})$ and in the solid rectangle. There are nine cells in the solid rectangle, where (A, D) contributes with one overlap for each. Thus, case two decreases the overlap by nine. In this example we have $V_R(\mathcal{E}) = 3 + 9 = 12$.

Let $\mathcal{Z}_{\in R(\mathcal{E})}(c)$ denote the set of routes at cell, c, that are in $R(\mathcal{E})$. Based on the two cases, the overlap contribution for each cell in $C_{R(\mathcal{E})}$ is formalized as:

$$O_{\mathcal{E}}(c) = \begin{cases} |\mathcal{Z}_{\in R(\mathcal{E})}(c)| - 1 & \text{If } \mathcal{Z}_{\in R(\mathcal{E})}(c) = \mathcal{Z}_R(c) \\ |\mathcal{Z}_{\in R(\mathcal{E})}(c)| & \text{Otherwise, } \mathcal{Z}_{\in R(\mathcal{E})}(c) \subset \mathcal{Z}_R(c) \end{cases} \tag{6.11}$$

$V_R(\mathcal{E})$ is calculated by summing over all contributions:

$$V_R(\mathcal{E}) = \sum_{c \in C_{R(\mathcal{E})}} O_{\mathcal{E}}(c) \tag{6.12}$$

$V'_R(\mathcal{E})$, on the other hand, corresponds to the amount of overlap introduced from solution, $\mathcal{G}$, to solution, $\mathcal{G}'$. This equals the overlap introduced when the routes in $R'(\mathcal{E})$ are added to $\mathcal{G}'$.

Let $C'_{R(\mathcal{E})}$ be the set of cells which are occupied by routes in $R'(\mathcal{E})$. Then, for each cell in $C'_{R(\mathcal{E})}$ there are two cases where overlap is introduced:

1. No other routes are at cell, c. This means that when adding the routes in $R'(\mathcal{E})$, only overlap between these routes needs to be considered. All the routes except the bottommost will contribute to the total overlap.

2. Some other routes are already at cell, c. Consequently, all routes in $R'(\mathcal{E})$ that occupy c, contribute to the total overlay, when added to $\mathcal{G}'$.

Figure 6.10 visualizes how $V'_R(\mathcal{E})$ is calculated. The routes in $R'(\mathcal{E})$ are added to $\mathcal{G}'$ in figure 6.10b.

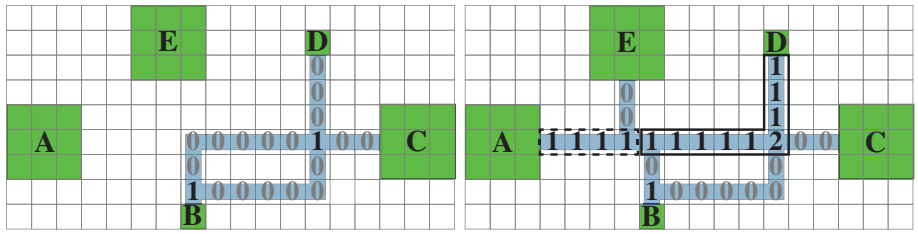


Figure 6.10: Solution, $\mathcal{G}'$, with routes in $R(\mathcal{E})$ removed (a), and with routes in $R'(\mathcal{E})$ added (b). The numbers indicate the overlap for each cell.

The cells outlined by a dashed rectangle correspond to case one. The two routes, (A, E) and (A, D), are the only routes at the four cells, and they are both in $R'(\mathcal{E})$. The bottommost route does not contribute to the overlay which means that the routes contribute with four to the total overlap.

The cells outlined by a solid rectangle correspond to case two. Other routes are already at the cells, so any added route will contribute to the overlay. Only route (A, D) is added to these cells and with nine cells the total contribution is nine. This gives $V'_R(\mathcal{E}) = 4 + 9 = 13$ for this example. It follows that $\Delta V_R = 13 - 12 = 1$, which means moving component, A, one cell to the left increases the overlay by one. $V'_R(\mathcal{E})$ can be formalized in a way similar to $V_R(\mathcal{E})$.

6.3.2.5 Analysis

In the typical case incremental updates reduce the number of computations. However, in the worst case all connections are connected to the changed components, yielding $|\mathcal{E}| = |E| = n$. This means that all connections must be rerouted, which is the same as calculating the cost function non-incrementally. From section 6.3.1 the complexity is thus $O(n \cdot h \times w)$.

Experimental evaluations in section 7.3.2 indicate that incremental updates are 1 to 7 times faster than non-incremental updates for the routing cost function. This is significantly less than for the approximated cost function. The difference is caused by two factors: The set of edges, $\mathcal{E}$, is larger for the routed cost function due to routes obstructed by components, and maintaining data structures for all routes adds an extra overhead.

Chapter 7

Experimental Evaluation

The placement algorithm is experimentally evaluated by applying it to a number of benchmark netlists. Evaluations are conducted in terms of placement quality and performance.

The algorithm is implemented in Python 3.3. To perform multiple placements at once, the algorithm was run on DTU High Performance Computing Clusters, with 512 MB of RAM dedicated for each job. Performance oriented evaluations were conducted on a MacBook Pro, 2.5 GHz Intel Core i5 with 16 GB of RAM.

All placement experiments use a cell size of 5 units. This is to reduce the number of cells in $\mathcal{G}$, which reduces the problem size for component placement and, in particular, for the routed cost function.

7.1 Benchmarks

The benchmark netlists are adapted from existing architectures in [7]. There are two types of benchmark netlists: Synthetic netlists of five different sizes, and real-life netlists that are generally smaller than the synthetic ones.

A selection of the benchmark netlists are presented in table 7.1. The table shows the components that make up each netlist and the total number of connections in a netlist. The complete set of benchmark netlists is found in appendix A.

Name	Type	Components						Connections
		In.	Out.	Mix.	Fil.	Hea.	Stor.	
10-2	Synthetic	1	1	4	2	4	1*	33
30-2	Synthetic	12	1	17	7	6	1*	102
40-2	Synthetic	15	1	21	10	9	1*	135
50-2	Synthetic	17	1	26	12	12	1*	167
IVD-1	Real Life	2	2	2	2	-	1	40
IVD-2	Real Life	6	6	6	6	-	1*	42
PCR-2	Real Life	3	3	3	-	-	1	24
PCR-3	Real Life	4	4	4	-	-	1	40

Table 7.1: Selected benchmark netlists.

The asterisk symbol (*) in the storage column indicates that a storage component has been introduced by the placement algorithm. All netlists must have a storage unit that is connected to all components to ensure successful scheduling in the application mapping phase.

Storage facilities are used to store the output of a component, so the component can be reused for another operation in the application. Note that most real-life netlists already have such a storage unit. Only *IVD-2* does not. In fact it does not need one, but the algorithm cannot distinguish the ones that need a storage unit from the ones that do not. Whether a storage unit is needed or not, is not decided until the application mapping phase.

7.2 Placement Quality

The parameters on which to evaluate the quality of placement depends on many factors. An important factor is the applications that are to run on the chip. In this thesis the applications are not known, so general assumptions are made about the quality of a placement.

A placement must provide the best possible conditions for the subsequent design phases while minimizing the size of the chip. The flow channel routing phase has optimal conditions when the routes are short, and intersection are few. The application mapping phase also prefers short routes as that implies small routing latencies. Furthermore, the problem size of the control channel routing phase is minimized if the number of valves is minimized. Finally, the fabrication success

rate depends on the amount of chip area dedicated to flow channels. Long flow channels means much PDMS material is removed from the chip, which might cause the flow channel layer to collapse. Thus, the fabrication phase prefers short flow channels.

Based on the observations above, the following parameters have been chosen for placement quality evaluation: Total length of all routes, the total channel length, total number of intersections, total number of valves, and the chip size. Note that the total length of all routes and the total channel length are different metrics because multiple routes can use the same channel.

Soukup's routing algorithm is used on a grid with a cell size of 1 unit to estimate the final routing. See figure 7.1. Based on the routing the following parameters

Figure 7.1: Example of a component placement achieved with simulated annealing. Rectangles are components, lines are routes found with Soukup's algorithm. The cell size is 1.

are determined: The total length of all routes, the total channel length, and the number of intersections. The number of valves for all components is retrieved from the component library (section 2.2.1.1). A switch introduced by route intersections has two, three, or four valves, depending on the switch configuration. The configurations are shown in figure 1.3. The total number of valves is then

the sum of the valves for all components and the valves introduced by route intersections. The chip size is given by the dimensions of the smallest rectangle that contains all components in $\mathcal{G}$.

For comparison, each netlist is placed using four different cost functions. The four cost functions are:

Manhattan A simple cost function. It minimizes the Manhattan distance between components. It is defined as:

$$Cost_M(\mathcal{G}) = L_A \tag{7.1}$$

where L_A is the total Manhattan distance between components. There are reports in the literature that this cost function has been used with success for both VLSI [4] and mVLSI [7] placement. Because of this, the Manhattan cost function is used as a reference.

Approximated The approximated cost function as described in chapter 5. It is defined as:

$$Cost_A(\mathcal{G}) = 500 \cdot N_A + 100 \cdot L_A + S_A \tag{7.2}$$

where N_A is the number of connection edge intersections, L_A is the Manhattan distance between components, and S_A is the squared Manhattan distance between components. This cost function has also previously been used for mVLSI placement [5]. The weights are chosen such that minimizing intersections is given highest priority. S_A is typically much larger than the two other metrics, so the weight for S_A is set to one to reduce the impact of this metric on the cost function.

Routed The routed cost function as described in chapter 6. It is defined as:

$$Cost_R(\mathcal{G}) = 500 \cdot N_R + 100 \cdot L_R + S_R + 100 \cdot V_R \tag{7.3}$$

where N_R is the number of route intersections, L_R is the total route length, S_R is the total squared route length, and V_R is the amount of route overlap. The placement seeks to minimize both intersections and overlap while finding the shortest routes.

Channel Minimization Another routed cost function defined as:

$$Cost_R(\mathcal{G}) = 500 \cdot N_R + 100 \cdot L_R + S_R - 100 \cdot V_R \tag{7.4}$$

Using a negative weight for V_R maximizes the amount of overlap, which effectively minimizes the total channel length.

Name	Cost Func.	Chan.	Rout.	Inters.	Valves	Chip Size
10-2	Manhattan	1272	2366	42	197	170×220
	Approx.	1352	2529	35	176	285×170
	Routed	1625	2311	38	185	195×195
	Channel Min.	1197	2893	31	170	260×170
30-2	Manhattan	4841	9794	171	749	450×305
	Approx.	4258	10222	146	661	365×405
	Routed	4909	10000	132	638	460×400
	Channel Min.	3984	12205	113	564	415×465
40-2	Manhattan	5765	14886	189	834	400×450
	Approx.	6310	15273	204	895	410×565
	Routed	6477	14817	172	800	470×570
	Channel Min.	5097	18299	169	800	460×495
50-2	Manhattan	7396	20288	272	1177	600×450
	Approx.	7545	21337	244	1076	460×630
	Routed	8081	20742	237	1065	525×575
	Channel Min.	7087	27824	194	914	480×680

Table 7.2: Cost function quality comparison for selected synthetic benchmark architectures. The comparison is based on total channel length, total route length, number of intersections, number of valves, and chip size.

Table 7.2 and 7.3 show the results for selected benchmark netlists. All results are presented in appendix B. In terms of channel length, number of intersections, and number of valves, the channel minimizing cost function performs best on most netlists. On average the total channel length is reduced by 20% compared to the Manhattan reference cost function. Similarly, the number of intersections is reduced by 35%, which results in a valve reduction of 25%. However, the reductions is at the expense of an increased total route length. In fact, the route length is on average increased by 25% compared to the reference cost function.

The routed cost function, on the other hand, achieves route and channel lengths close to that of the reference cost function. Additionally, it improves the number of intersections and valves by 20% and 15%, respectively. In terms of number of intersections that is not quite as good as the channel minimizing cost function, because overlap often results in shared intersections.

The positive effect of the approximated cost function is negligible. The number of intersections and valves is almost the same as for the Manhattan cost function, and the channel and route lengths are both increased by 10% on average.

There is no clear connection between chip size and choice of cost function. Since

Name	Cost Func.	Chan.	Rout.	Inters.	Valves	Chip Size
IVD-1	Manhattan	1116	3592	47	204	210 × 120
	Approx.	1368	3764	43	182	120 × 180
	Routed	1244	3408	28	134	170 × 135
	Channel Min.	678	3922	19	106	165 × 170
IVD-2	Manhattan	1761	3230	57	266	200 × 470
	Approx.	1568	3147	47	238	205 × 470
	Routed	2274	3181	43	231	395 × 235
	Channel Min.	1445	4104	39	216	310 × 260
PCR-2	Manhattan	615	1304	24	124	120 × 100
	Approx.	800	1401	29	140	125 × 120
	Routed	505	1291	15	103	115 × 120
	Channel Min.	448	1437	14	106	115 × 120
PCR-3	Manhattan	1184	2514	56	248	120 × 135
	Approx.	1453	2714	72	303	125 × 145
	Routed	762	2432	27	144	105 × 160
	Channel Min.	463	2592	16	116	110 × 140

Table 7.3: Cost function quality comparison for selected real life benchmark architectures. The comparison is based on total channel length, total route length, number of intersections, number of valves, and chip size.

chip size is not a metric in either cost function, this is expected.

All percentages are based on the netlists in appendix B, and are compared to the Manhattan reference cost function. Based on the observations above the routed and the channel minimizing cost function provide good results with respect to the chosen parameters. Which one to use depends on which is considered more important: The total channel length or the total length of the routes.

The placement of the netlists took between 30 minutes and 24 hours to complete, depending on the netlist size. Because of the exponentially increasing search space, large netlists require considerably more repetitions at each temperature to achieve good results.

7.3 Performance

In the following sections different variations of the placement algorithm is evaluated in terms of performance. Introducing more complex optimizations might result in improved placement quality but also reduces performance, which increases the running time of the algorithm.

Because of the random nature of simulated annealing, the presented results are only estimations. The results are based on a limited number of samples, and the results will in general vary depending on how the randomized part of the algorithm evolves.

7.3.1 Cost Functions

Throughout the thesis two cost functions are distinguished: The routed cost function, which relies on actual routing, and the approximated cost function, which estimates the routing based on certain assumptions.

The routed cost function produced better placement results in section 7.2, but the number of computations is also much greater. Table 7.4 shows a comparison of the running times for the routed and the approximated cost function.

	10-2	**30-2**	**40-2**	**50-2**	**IVD-1**	**PCR-3**
Routed	140	270	250	540	160	100
Approximated	25	50	70	110	30	20
Ratio	5.6	5.4	3.6	4.9	5.3	5.0

Table 7.4: Computation time comparison between the routed cost function and the approximated cost function. The unit is computation time for the simple Manhattan cost function. The comparison is based on selected netlists.

The used unit is the computation time for the Manhattan cost function. The table shows that the routed cost function is 100 to 540 times slower than the Manhattan cost function for the selected netlists. It is also approximately 5 times slower than the approximated cost function.

Note that both the routed and approximated cost function are significantly slower for large netlists. This is due to the fact that large netlists have more connections and both cost functions must maintain data structures for the intersections of the connections.

7.3.2 Incremental Speedup

Incremental update of the cost functions is introduced to save computations and increase performance by only recomputing what changed from one solutions to another. Table 7.5 shows experimental results for the speedup.

	10-2	30-2	40-2	50-2	IVD-1	PCR-3
Routed	1.1	3.5	6.4	6.7	1.2	1.5
Approximated	5.3	20.4	24.3	28.0	3.7	5.0

Table 7.5: Incremental speedup for routed and approximated cost function for selected netlists.

The speedup is greater for large netlists for both cost functions. For larger netlists a smaller fraction of the connections need to be updated. For smaller netlists the overhead of maintaining the incremental data structure is significant and the effect is reduced.

The results indicate that the approximated cost function is more ideal for incremental update than the routed cost function. This is expected because the routed cost function must update the obstructed routes and the routes that were previously obstructed in addition to the routes connected to changed components (section 6.3.2.1). The routed cost function also has more internal data structures to maintain for both intersections and overlapping.

7.3.3 Routing Algorithms

Two different routing algorithms are considered in section 6.1.1 and section 6.1.2. Soukup's algorithm is chosen over Lee's algorithm because it primarily searches in the direction of the target, while Lee's algorithm searches evenly in all directions.

We wish to compare the two routing algorithms to see the effect of the directed search of Soukup's algorithm. Placements are conducted for six netlists where all routes are routed with both Soukup's algorithm and Lee's algorithm. The total search spaces of the two algorithms are compared and the reduction factor of Soukup's algorithm is presented in table 7.6.

10-2	30-2	40-2	50-2	IVD-1	PCR-3
10 to 30	20 to 85	25 to 115	30 to 130	5 to 20	5 to 25

Table 7.6: Search space reduction of Soukup's algorithm compared to Lee's algorithm. The comparison is based on selected netlists.

In the initial solution (see section 4.2.2) of any netlist all routes can be routed by going directly to the target first in the horizontal direction and then in the vertical direction. These are ideal conditions for Soukup's algorithm, which

results in a huge reduction in the search space. As the placement approaches its final solution the search space becomes congested which is bad for Soukup's algorithm. The result is a huge reduction in the initial phase of the placement, which gradually reduces. In table 7.6 this is represented by the interval at each netlist.

Chapter 8

Computer-Aided Design Tool

The placement algorithm described in the previous chapters has been implemented as part of a Computer-Aided Design (CAD) tool. The purpose of the tool is to assist the designer in the component placement phase of the biochip design process. The CAD tool consists of a placement engine that executes the placement algorithm and a Graphical User Interface (GUI) that visualizes the simulation as it evolves. When a placement is complete the result can be saved in an XML file. The result can then be restored in the GUI by loading the XML file, or the XML file can be used in other tools for the subsequent phases of the biochip design process. Furthermore, the GUI assists in choosing the correct parameters for the simulated annealing by providing relevant information.

The simulation engine can also run as a stand-alone application. This is convenient when doing time consuming simulations that don't require a visualization. The following sections describe the main features of the CAD tool.

8.1 Visualization

Finding the best configuration for the simulated annealing is, due to the many parameters, a difficult task. The solution found by the simulated annealing should correspond to our perception of a good solution. Seeing the placement evolve, as it is modified by the simulated annealing algorithm, will help identify what parameters should be changed. This is the motivation behind the visualization of the placement algorithm. Figure 8.1 shows a GUI screenshot during a placement.

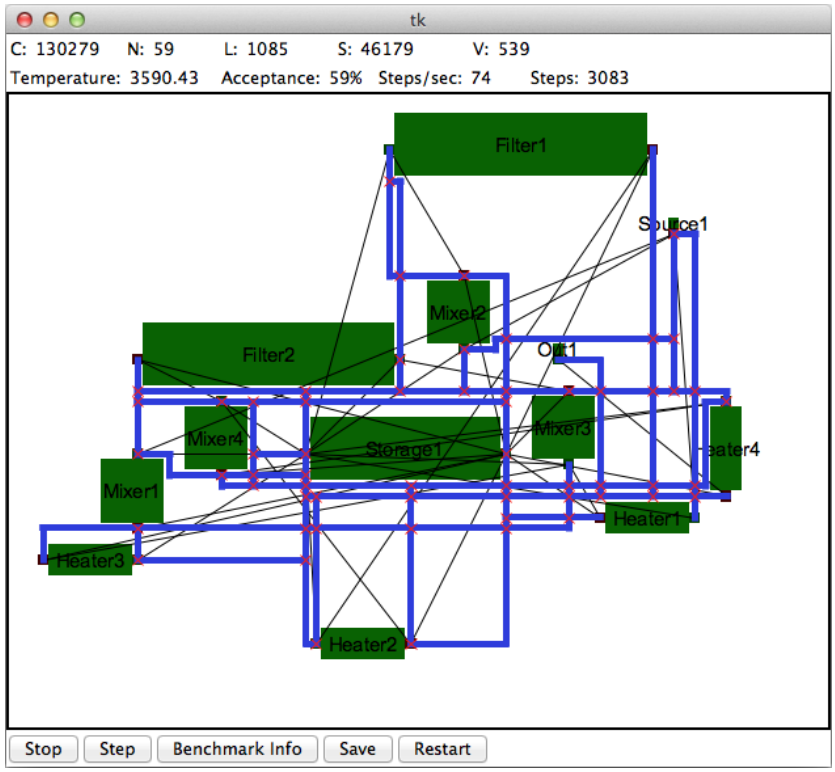


Figure 8.1: Screenshot of the GUI as the tool is performing the placement of architecture 10-2. Green rectangles are components, thick blue lines are routes, red crosses are intersections, and thin black lines indicate a connection between two connection points. The top bar provides data about the current placement.

In addition to showing the visualization of the placement, the GUI also provides the following information:

- The current cost function value, C, along with the metric values, N, L, S, and V.
- The current temperature.
- The current acceptance rate, which is the rate of neighbouring solutions that are accepted since the last GUI update.
- The total number of steps calculated. This corresponds to iterations of the simulated annealing algorithm. Furthermore, the current computation speed in terms of steps per second is shown.

Along with the provided information, the visualization can help determine if the parameters have the correct value. In particular, the acceptance rate is helpful in determining the initial temperature. Initially the acceptance rate should be close to 100% and the initial temperature must be high enough to facilitate that.

At any time during the placement, the user can choose to stop the placement, step through n_{rep} iterations at a time, print benchmark information, save the current placement in an XML file, or restart the placement. These options are helpful to obtain and save results, but they were also of great use in the debugging process. The benchmark information include routing lengths, number of intersections, number of valves, and chip size. This information was used to generate the benchmark results in section 7.2.

In general the GUI is well suited for ensuring that the placement behaves as indented. Either in the debugging phase, to test that the model works as expected, or when obtaining results, to test that the parameters have the desired effect.

8.2 Configuration

Many parameters can be set for the simulated annealing. All configurations are collected in a JSON file. The configuration file is passed to the program upon start-up. This enables having multiple predefined configurations, where a particular one can quickly be applied when needed. The most important configurations are outlined in the following descriptions.

Cell Size Defines height and width of a single cell. The unit is $150\mu m$. Large cell sizes decrease the grid granularity and thus reduces the problem size. A cell size of five was mainly used for conducting the placement, and a cell size of one was used to retrieve routing benchmark information.

Component Library Sets the path to the component library. The component library is an XML file that defines the sizes, connection points, and number of valves for all component types. Input and output ports are also defined as components.

Component Border Defines the thickness of the component border. It must comply with the placement design rules (table 2.2). A border of five units ensures that the spacing between any two components is at least 10 units ($150\mu m$).

Initial Temperature Sets the temperature, from which the simulated annealing cool-down starts.

Reduction Rate Decides how fast the temperature is reduced. Typical values are between 0.98 and 0.999.

Nrep Sets the number of repetitions at each temperature. This is useful to scale the placement time by a constant factor. More repetitions at each temperature generally provides better results but also takes longer.

Operation Probabilities Defines the probabilities that the three different operations, move, rotate, and swap, are performed at each iteration. The probabilities should sum to 100%. $(60\%, 20\%, 20\%)$ for move, rotate, and swap provide good results.

Cost Function The cost function must be defined both in terms of which cost function to use and the weights of the metrics. The CAD tool supports the routed and the approximated cost function. The Manhattan and the channel minimizing cost functions are obtained by tweaking the metric weights of the approximated and the routed cost function, respectively.

Termination Defines the termination conditions. A termination temperature is set, and it is stated if an XML-file should be saved automatically upon completion. The tool can also be configured to perform additional iterations at temperature zero to ensure that the solution is at a local optimum.

The configurations above can be combined in many different ways. Knowing which values to use is the biggest challenge when using simulated annealing. The purpose of the CAD Tool is to assist in experimentally determining the configuration to use, so that the achieved placements meet the expectations for a good placement. When a good configuration has been chosen the GUI is no longer particularly useful. Instead, the placement engine might as well run independently on a high performance computer, until the results are ready.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This thesis describes and analysis problems involved in implementing an algorithm for automated biochip component placement. A biochip architecture model is proposed. Components and flow channel routes are placed on a grid, representing the biochip area. The purpose of the algorithm is to find the best placement for any given biochip netlist. A placement must comply with certain design rules in order to be feasible.

Due to the complexity of the problem, the optimal solution cannot be computed in polynomial time. Instead, a common heuristic for minimization problems, simulated annealing, is used to optimize the component placement. An important property of the simulated annealing algorithm, is that the solution converges towards the optimal solution. This is used to find a solution in polynomial time, that is considered good enough.

An important problem is to decide and define what a good solution is. In this thesis, a placement is good, if it provides good conditions for the subsequent phases of the biochip design process. A good solution is thus a placement with short routes between components, a short total channel length, and few route

intersections. The simulated annealing algorithm must optimize the placement with respect to these metrics.

Two different cost functions are proposed for the simulated annealing algorithm: An approximated cost function, which tries to make qualified estimations of the metric values, and a routed cost function, which uses the actual routes between components to determine the metric values. The approximated cost function is fast to compute but is based on estimations. On the other hand, the routed cost function takes longer to compute but the metric values are accurate.

The two cost functions are evaluated on several benchmark netlists. They are compared to a reference cost function, which minimizes the Manhattan distance between components. All benchmarks are routed using the same simple routing algorithm as the routed cost function. The benchmarks indicate that the approximated cost function does not improve the number of route intersections and increases the route and channel length, compared to the reference cost function. In contrast, the routed cost function provides good results. In one version it decreases the number of intersections by 35%. However, the few intersections require longer routes, so the total route length is increased by 25%. In another version, intersections are decreased by 20%, and route lengths are similar to the reference cost function. The downside is that it needs significantly longer time to finish, than the approximated cost function.

Simulated annealing relies on many different parameters. The values of these parameters affect the outcome of the algorithm. Among other things the parameters decide how long the algorithm runs and the exact form of the cost function. It is the general consensus that these parameters are best decided experimentally [9]. In order to help the biochip designer decide the parameters, a CAD tool has been implemented which visualizes the simulated annealing algorithm along with relevant information. The CAD tool is also useful to verify that the placement is acceptable or if the algorithm should be restarted.

9.2 Future Work

The component placement algorithm can be extended and improved in many ways. Either to improve the quality of the placements or to improve algorithm performance. Some interesting extensions for future work are:

- Introduction of a new metric, the chip size. Often the biochip must be as small as possible or the placement must fit on a standard sized chip.

Both situations could be solved with a chip size metric. One possible implementation is to let the metric be defined by the height and width of the smallest rectangle, r, that fits around all components. Then the simulated annealing algorithm is used to minimize the height and width. Another approach is to give a chip size as input to the algorithm. If the bounding rectangle, r, extends beyond the chip size a significant penalty is added to the cost function. This encourages the simulated annealing algorithm to find solutions that fit inside the chip area. At the same time, if r is smaller than the chip size, no penalty is added, and other metrics like number of intersection can decide the placement within the chip area.

- The input and output ports of the chip are connected to off-chip pumps that pump fluids into the flow channels. Restricting the sections on which input and output ports can be placed, will ease the process of connecting the pumps. Such sections could be defined as rectangles on the chip area. Ideally, the sections are defined relative to the bounding rectangle of the components. This way, the sections will adapt as the components pack together.

- The experimental evaluations reveal that the routed cost function generally provides better results than the approximated cost function, but it is significantly slower. It would be interesting to investigate if the two cost functions could be combined to achieve fast high quality placements. The approximated cost function would be adequate at high temperatures where the components move freely. When the temperature gets to a certain point, the routed cost function is applied to fine-tune the placement based on actual routes. It is hard to say which temperature is ideal for switching cost functions, and this adds another parameter which must be experimentally obtained.

- Soukup's routing algorithm was chosen as the primary routing algorithm for the routed cost function. This decision was based on the fact that Soukup's algorithm directs it search towards the target, and thus minimizes the search space. Lee's algorithm on the other hand searches equally in all directions. The advantage of Lee's algorithm is that it can route from a source to multiple targets. Due to its directed search Soukup's can only route to one target. If a netlist contains many connections per component it might be faster to use Lee's routing algorithm to route multiple routes simultaneously.

Appendix A

Benchmark Netlists

The table below lists the benchmark netlists used for experimental evaluation. They are adapted from [7].

Name	**Type**	**Components**						**Connections**
		In.	Out.	Mix.	Fil.	Hea.	Stor.	
10-1	Synthetic	2	1	4	2	4	1*	36
10-2	Synthetic	1	1	4	2	4	1*	33
20-1	Synthetic	2	1	12	4	4	1*	76
20-2	Synthetic	1	1	12	4	4	1*	68
30-1	Synthetic	2	1	17	7	6	1*	124
30-2	Synthetic	12	1	17	7	6	1*	102
40-1	Synthetic	2	1	21	10	9	1*	140
40-2	Synthetic	15	1	21	10	9	1*	135
50-1	Synthetic	2	1	26	12	12	1*	167
50-2	Synthetic	17	1	26	12	12	1*	167
IVD-1	Real Life	2	2	2	2	-	1	40
IVD-2	Real Life	6	6	6	6	-	-	18
PCR-1	Real Life	2	2	2	-	-	1	12
PCR-2	Real Life	3	3	3	-	-	1	24
PCR-3	Real Life	4	4	4	-	-	1	40

Appendix B

Cost Function Comparison

The following sections provide the placement results for all benchmark netlists.

B.1 Synthetic Architectures

The table below shows the cost function quality comparison for synthetic benchmark architectures. The comparison is based on total channel length, total route length, number of intersections, number of valves, and chip size.

Name	Cost Func.	Chan.	Rout.	Inters.	Valves	Chip Size
10-1	Manhattan	1375	2346	43	205	185×205
	Approx.	1464	2670	47	221	285×170
	Routed	1596	2428	37	185	180×235
	Channel Min.	1142	3056	41	202	170×230
10-2	Manhattan	1272	2366	42	197	170×220
	Approx.	1352	2529	35	176	285×170
	Routed	1625	2311	38	185	195×195
	Channel Min.	1197	2893	31	170	260×170

Name	Cost Func.	Chan.	Rout.	Inters.	Valves	Chip Size
20-1	Manhattan	2895	6395	107	475	265×270
	Approx.	3112	6699	106	481	325×265
	Routed	3181	6418	81	397	285×335
	Channel Min.	1955	7949	54	320	295×300
20-2	Manhattan	2876	6241	109	492	295×245
	Approx.	2520	6099	85	410	245×310
	Routed	3207	5966	84	411	295×290
	Channel Min.	2027	6981	49	298	295×275
30-1	Manhattan	4481	12218	164	713	345×445
	Approx.	4850	13621	161	703	515×345
	Routed	4370	12053	147	680	335×365
	Channel Min.	3391	14512	113	557	335×330
30-2	Manhattan	4841	9794	171	749	450×305
	Approx.	4258	10222	146	661	365×405
	Routed	4909	10000	132	638	460×400
	Channel Min.	3984	12205	113	564	415×465
40-1	Manhattan	5704	15543	221	944	395×435
	Approx.	5992	17412	203	877	465×500
	Routed	7071	16236	203	899	445×420
	Channel Min.	4839	19095	149	731	335×530
40-2	Manhattan	5765	14886	189	834	400×450
	Approx.	6310	15273	204	895	410×565
	Routed	6477	14817	172	800	470×570
	Channel Min.	5097	18299	169	800	460×495
50-1	Manhattan	7286	20724	264	1139	560×485
	Approx.	7865	21565	254	1097	410×585
	Routed	7810	21276	232	1038	435×490
	Channel Min.	6310	25489	176	847	520×560
50-2	Manhattan	7396	20288	272	1177	600×450
	Approx.	7545	21337	244	1076	460×630
	Routed	8081	20742	237	1065	525×575
	Channel Min.	7087	27824	194	914	480×680

B.2 Real Life Architectures

The table below shows the cost function quality comparison for real life benchmark architectures. The comparison is based on total channel length, total route length, number of intersections, number of valves, and chip size.

Name	Cost Func.	Chan.	Rout.	Inters.	Valves	Chip Size
IVD-1	Manhattan	1116	3592	47	204	210×120
	Approx.	1368	3764	43	182	120×180
	Routed	1244	3408	28	134	170×135
	Channel Min.	678	3922	19	106	165×170
IVD-2	Manhattan	1761	3230	57	266	200×470
	Approx.	1568	3147	47	238	205×470
	Routed	2274	3181	43	231	395×235
	Channel Min.	1445	4104	39	216	310×260
PCR-1	Manhattan	447	670	18	101	80×125
	Approx.	512	676	17	98	125×95
	Routed	418	632	8	73	100×120
	Channel Min.	300	672	8	74	120×95
PCR-2	Manhattan	615	1304	24	124	120×100
	Approx.	800	1401	29	140	125×120
	Routed	505	1291	15	103	115×120
	Channel Min.	448	1437	14	106	115×120
PCR-3	Manhattan	1184	2514	56	248	120×135
	Approx.	1453	2714	72	303	125×145
	Routed	762	2432	27	144	105×160
	Channel Min.	463	2592	16	116	110×140

Bibliography

[1] Stanford microfluidic foundry. `http://http://stanford.edu/group/foundry/`.

[2] C. D. Chin, T. Laksanasopin, Y. K. Cheung, et al. Microfluidics-based diagnostics of infectious diseases in the developing world. *Nature Medicine*, 17:1015–1019, 2013.

[3] D. W. Inglis, R. S. Pawell, T. J. Barber, and R. A. Tayler. Manufacturing and wetting low-cost microfluidic cell separation devices. *Biomicrofluidics*, 7(5), 2013.

[4] A. B. Kahng, J. Lienig, I. Markov, and J. Hu. Vlsi physical design: From graph partitioning to timing closure. In *Global and Detailed Placement*. Springer, 2011.

[5] B. Karker, J. McDaniel, and P. Brisk. Simulated annealing-based placement for microfluidic large scale integration (mlsi) chips. University of California, Riverside.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. 1983.

[7] W. H. Minhass. *System-Level Modeling and Synthesis Techniques for Flow-Based Microfluidic Very Large Scale Integration Biochips*. PhD thesis, Technical University of Denmark, 2012.

[8] M. Prasad. Intersection of line segments. In *Graphics Gems II*. AP Professional, 1995.

[9] C. R. Reeves. Simulated annealing. In *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, 1993.

[10] N. A. Sherwani. Global routing. In *Algorithms for VLSI Physical Design Automaton*. Kluwer Academic Publishers, 2002.

[11] J. Soukup. Fast maze router. 1978. Bell-Northern Research, Ottawa, Ontario.

[12] A. M. Streets and Y. Huang. Chip in a lab: Microfluidics for next generation life science research. *Biomicrofluidics*, 7(011302), 2013.

[13] F. Su and K. Chakrabarty. p. 690. In *Module Placement for Fault-Tolerant Microfluidics-Based Biochips*. Duke University, 2006.

[14] B. Theis. Programmable microfluidics. `http://groups.csail.mit.edu/cag/biostream/talks/microfluidics-berkeley-07.pdf`, 2007.