Global Gate Sizing and Buffering

Lucas Slot

Born 9 January 1996 in Amsterdam

September 7, 2018

Master's Thesis Mathematics

Advisor: Prof. Dr. Stephan Held

Second Advisor: Prof. Dr. Stefan Hougardy

Research Institute for Discrete Mathematics

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Chapter 1

Introduction

As the prevalence and complexity of computer chips increases, so does the need to optimize their design. Of particular importance for modern circuits is the ability to meet strict timing requirements while keeping power consumption at a minimum. Careful selection of the physical implementation of logical gates and the insertion of repeaters in long segments of wire connecting those gates are crucial steps towards that goal.
In this thesis, we consider algorithmic solutions to the interconnected problems of gate sizing and buffering and attempt to develop methods for dealing with them simultaneously. We will do so in part within the context of BonnTools, a software suit for automated chip design developed by the Institute for Discrete Mathematics in Bonn. We will need to use ideas from several areas of mathematical optimization, both discrete and continuous. A particular focus will be on various forms of convex programming.
The basic outline is as follows: In chapter two we review some basic notions of convexity, generalized convexity and convex optimization. We consider several special types of (convex) programs that will be used in later chapters. In chapter three we introduce chip design. We consider in particular the problems of gate sizing and buffering, and the interplay between them. In chapter four we study gate sizing with movement on repeater chains, attempting to take a step towards a unified algorithm for gate sizing and buffering. In chapter five we present some practical additions made to BonnTools.

Acknowledgements

I would like to thank my advisors Prof. Dr. Stephan Held and Prof. Dr. Stefan Hougardy as well as Prof. Dr. Jens Vygen and Siad Daboul for many useful comments and suggestions. I am also grateful to my colleagues at the Institute for Discrete Mathematics in Bonn, who were very patient with my many questions when I first started working there. Lastly, I wish to thank my parents and sister for their continued support and encouragement throughout my studies.

Chapter 2

Convexity and Convex Programming

In this chapter we will review some basic properties of convex functions and convex optimization. Afterwards, we shall consider some generalized notions of convexity. Finally, we review three specific cases of convex programming which we will use directly in later chapters.

2.1 Convexity

We begin by introducing three notations which will be very convenient to use in what follows.

Definition 2.1. For any function f of a single variable, we shall interpret $f(x)$ for a vector $x \in \mathbb{R}^n$ as the pointwise application of f to x whenever this is well-defined, i.e. $f(x) \in \mathbb{R}^n$ is the vector having entries $f(x)_i = f(x_i)$. Important examples are $e^x, \log x$ and $\sqrt{x}$, the last two being applicable only when $x \geq 0$.

Definition 2.2. For $x, y \in \mathbb{R}^n$ we write $x \circ y \in \mathbb{R}^n$ for the vector having entries $(x \circ y)_i = x_i \cdot y_i$.

Definition 2.3. For any two points $x, y \in \mathbb{R}^n$, we write $[x, y]$ for the closed line segment connecting x and y, i.e.

$$[x, y] = \{tx + (1-t)y : t \in [0, 1]\}.$$

Using the final notation, we get the following elegant definition of a *convex* set.

Definition 2.4. A set $U \subseteq \mathbb{R}^n$ is *convex* if $[x, y] \subseteq U$ for all $x, y \in U$.

Definition 2.5. Let $U \subseteq \mathbb{R}^n$ convex. We say a function $f : U \to \mathbb{R}$ is *convex* if

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

for all $t \in [0, 1]$ and $x, y \in U$.

The following proposition motivates the use of the term convex above.

Proposition 2.6. *Let $f : U \to \mathbb{R}$. Then the* epigraph *of f, defined as*

$$\mathrm{Epi}(f) := \{(x, t) \in U \times \mathbb{R} : f(x) \leq t\},$$

is convex if and only if f is convex.

Convexity of functions is preserved under several operations.

Proposition 2.7. *Let $f_1, f_2 : U \to \mathbb{R}$ be convex, $A : U \to U$ affine. Then the following functions are convex:*

1. *$\alpha f_1 + \beta f_2$ for $\alpha, \beta \geq 0$,*
2. *$f_1 \circ A$,*
3. *$f : U \to \mathbb{R}$ given by $f(x) = \max\{f_1(x), f_2(x)\}$.*

A useful property of convex functions is that their local minima are always global minima.

Proposition 2.8. *Let $f : U \to \mathbb{R}$ convex, and let $x \in U$ be a local minimum of f. Then x is a global minimum of f.*

Proof. For a contradiction assume that there exists a $y \in U$ with $f(y) < f(x)$. Then by convexity of f we have

$$f(x + \epsilon(y - x)) \leq (1 - \epsilon)f(x) + \epsilon f(y) < f(x)$$

for any $\epsilon \in (0, 1)$ and so x is not a local minimum of f. □

If f is (twice) differentiable, it is possible to give a characterization of convexity of f in terms of its (second) derivative. These so-called first and second order conditions are usually much easier to work with than with definition 2.5. For their proofs we follow [2]. We will first need a lemma.

Lemma 2.9. *(Convexity on lines) Let $f : U \to \mathbb{R}$ be a function on a convex domain $U \subseteq \mathbb{R}^n$. Then f is convex iff for each $x_0 \in U, y \in \mathbb{R}^n$ the function $g : L \to \mathbb{R}$ given by $g(\theta) = f(x_0 + \theta y)$ is convex, where $L := \{\theta \in \mathbb{R} : x_0 + \theta y \in U\}$.*

Theorem 2.10. *(First-Order Condition) Let $f : U \to \mathbb{R}$ be a differentiable function on a convex domain $U \subseteq \mathbb{R}^n$. Then f is convex iff for all $x, y \in U$*

$$\nabla f(x)^\top (y - x) \leq f(y) - f(x).$$

That is, f is convex iff its first-order Taylor approximation at any point is a global underestimator.

Proof. For the first implication, assume f is convex and let $x, y \in U$. Then

$$\begin{aligned}
\nabla f(x)^\top (y - x) &= \lim_{t \downarrow 0} \frac{f(x + t(y - x)) - f(x)}{t} \\
&\leq \lim_{t \downarrow 0} \frac{(1 - t)f(x) + tf(y) - f(x)}{t} \\
&= \lim_{t \downarrow 0} \frac{tf(y) - tf(x)}{t} \\
&= f(y) - f(x).
\end{aligned}$$

For the other direction, assume the first-order condition holds for f, and let $x, y \in U, t \in [0,1]$. Writing $m = tx + (1-t)y$, we have

$$\begin{aligned} f(x) &\geq f(m) + \nabla f(m)^\top (x - m), \\ f(y) &\geq f(m) + \nabla f(m)^\top (y - m), \end{aligned}$$

and so using linearity of $\nabla f(m)^\top(\,\cdot\,)$ we get

$$\begin{aligned} tf(x) + (1-t)f(y) &\geq t\big(f(m) + \nabla f(m)^\top (x-m)\big) + \\ &\quad (1-t)\big(f(m) + \nabla f(m)^\top (y-m)\big) \\ &= f(m) + \nabla f(m)^\top (tx + (1-t)y - m) \\ &= f(m) + \nabla f(m)^\top (0) \\ &= f(tx + (1-t)y). \end{aligned}$$

□

Theorem 2.11. *(Second-Order Condition) Let $f : U \to \mathbb{R}$ be a twice differentiable function on a convex domain $U \subseteq \mathbb{R}^n$. Then f is convex iff for all $x \in U$*

$$\nabla^2 f(x) \succeq 0.$$

Proof. We prove first for $n = 1$. If f is convex, we have $f'(x)(y-x) \leq f(y) - f(x)$ and $f'(y)(y-x) \geq f(y) - f(x)$ for all $x < y \in U$ by the first-order condition, and so for each $x \in U$ we have

$$\begin{aligned} f''(x) &:= \lim_{y \downarrow x} \frac{f'(y) - f'(x)}{y - x} \\ &= \lim_{y \downarrow x} \big(f'(y)(y-x) - f'(x)(y-x)\big)(y-x)^{-2} \\ &\geq 0. \end{aligned}$$

For the other direction, let $x < y \in U$. By Taylor's Theorem there exists an $m \in [x, y]$ s.t.

$$f(y) = f(x) + f'(x)(y-x) + \frac{1}{2} f''(m)(y-x)^2$$

and so as $f''(m) \geq 0$ we obtain $f(y) \geq f(x) + f'(x)(y-x)$ which is the first-order condition.

For $n > 1$, observe that for any line $\{x_0 + \theta y\} \subseteq U$ the second derivative g'' of the function $g(\theta) := f(x_0 + \theta y)$ is given by $g''(\theta) = y^\top \nabla^2 f(x_0 + \theta y) y$ and so $g''(\theta) \geq 0$ for all θ iff $\nabla^2 f(x) \succeq 0$ for all $x \in U$. Applying Lemma 2.9 finishes the proof. □

Example 2.1. The *log-sum-exponential* function $\mathrm{LSE} : \mathbb{R}^n \to \mathbb{R}$ given by

$$\mathrm{LSE}(x) := \log\left(1^\top e^x\right) = \log\Big(\sum_{j=1}^{n} e^{x_j}\Big)$$

is convex.

Proof. We compute the Hessian and show $\nabla^2 \mathrm{LSE}(x) \succeq 0$. Writing $y = e^x \geq 0$ we have

$$\begin{aligned}
\frac{\partial \,\mathrm{LSE}(x)}{\partial x_i} &= \frac{\delta \,\mathrm{LSE}(x)}{\delta y_i}\frac{\delta y_i}{\delta x_i} \\
&= \frac{\delta \log(\sum_{k=1}^n y_k)}{\delta y_i} y_i \\
&= \frac{y_i}{\sum_{k=1}^n y_k}
\end{aligned}$$

and so

$$\begin{aligned}
\frac{\partial^2 \,\mathrm{LSE}(x)}{\partial x_i \partial x_j} &= \frac{\partial\big(y_i(\sum_{k=1}^n y_k)^{-1}\big)}{\partial x_j} \\
&= \frac{\partial y_i}{\partial x_j}\frac{1}{\sum_{k=1}^n y_k} + \frac{\partial(\sum_{k=1}^n y_k)^{-1}}{\partial x_j} y_i \\
&= \frac{\delta_{ij} y_i}{\sum_{k=1}^n y_k} - \frac{y_i y_j}{(\sum_{k=1}^n y_k)^2}.
\end{aligned}$$

In matrix form, this gives

$$\nabla^2 \,\mathrm{LSE}(x) = \frac{\mathrm{diag}(y)}{1^\top y} - \frac{yy^\top}{(1^\top y)^2} = \frac{(1^\top y)\,\mathrm{diag}(y) - yy^\top}{(1^\top y)^2}.$$

Now, using Cauchy-Schwarz, we have

$$\begin{aligned}(1^\top y)^2 \cdot u^\top \nabla^2 \operatorname{LSE}(x) u &= u^\top \big((1^\top y)\operatorname{diag}(y) - yy^\top\big) u \\ &= (1^\top y)\sum_{k=1}^{n} y_k u_k^2 - (y^\top u)^2 \\ &= \sum_{k=1}^{n} y_k \sum_{k=1}^{n} y_k u_k^2 - \big((\sqrt{y})^\top(\sqrt{y}\circ u)\big)^2 \\ &= ||\sqrt{y}||^2 ||\sqrt{y}\circ u||^2 - \big((\sqrt{y})^\top(\sqrt{y}\circ u)\big)^2 \\ &\geq 0\end{aligned}$$

for any $u \in \mathbb{R}^n$ and so $\nabla^2 \operatorname{LSE}(x) \succeq 0$. □

2.2 Convex Programming

The goal of this section is to give a brief overview of basic notions from convex programming that we shall refer to later. We use [23] as a basis, further details and proofs of some of the statements made below can be found there.

A *convex program* P is a program of the form

$$\begin{aligned}\min \quad & f_0(x) \\ s.t. \quad & f_i(x) \leq 0, \quad i = 1, \ldots, I \\ & g_j(x) = 0, \quad j = 1, \ldots, J \\ & x \in \mathbb{R}^n\end{aligned}$$

where the f_i are convex functions and the g_j are affine. Define the convex set $\mathcal{D} := \{x \in \mathbb{R}^n : f_i(x) \leq 0, g_j(x) = 0\}$. A point $x \in \mathcal{D}$ is called a *feasible solution* of P. If $f_0(x) \leq f_0(y)$ for all $y \in \mathcal{D}$ with $||x - y|| \leq \epsilon$ for some $\epsilon > 0$, x is called a locally optimal solution. If $f_0(x) \leq f_0(y)$ for *all* $y \in \mathcal{D}$, x is called a globally optimal solution. From our earlier discussion, it follows that any locally optimal solution to a convex program must indeed be globally optimal. We shall write $p^* := \inf_{x\in\mathcal{D}} f_0(x)$.

Lagrangian Duality

The Lagrangian $L : \mathcal{D} \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R}$ of P is defined as

$$L(x, \lambda, \nu) = f_0(x) + \sum_{i=i}^{I} \lambda_i f_i(x) + \sum_{j=1}^{J} \nu_j g_j(x).$$

The variables λ, ν are referd to as *Lagrangian multipliers* or *dual variables*. The *Lagrange dual function* $\mathcal{L} : \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R} \cup \{-\infty\}$ is defined as

$$\mathcal{L}(\lambda, \nu) := \inf_{x \in \mathcal{D}} L(x, \lambda, \nu).$$

If $x \in \mathcal{D}$ and $\lambda \geq 0$, we have

$$L(x, \lambda, \nu) = f_0(x) + \sum_{i=i}^{I} \lambda_i f_i(x) + \sum_{j=1}^{J} \nu_j g_j(x) \leq f_0(x),$$

and so $\mathcal{L}(\lambda, \nu) \leq p^*$ for all $\lambda \geq 0$, that is, $\mathcal{L}(\lambda, \nu)$ provides a lower bound for P. This observation leads to the definition of the so-called *Lagrange dual problem* D of P given by

$$\begin{aligned} \max \quad & \mathcal{L}(\lambda, \nu) \\ s.t. \quad & \lambda \geq 0, \\ & \nu \in \mathbb{R}^J. \end{aligned}$$

This program is again convex. Writing d^* for the value of an optimal solution to D, we find that $d^* \leq p^*$, a property known as *weak duality*. It can be shown under minor additional assumptions on P that in fact $d^* = q^*$, in which case we say that *strong duality* holds.

Although we shall not discuss any algorithms for solving convex programs here, we would like to note here that if strong duality holds it is usually possible to find an optimum solution to P by instead solving the dual program D. This fact can be very useful if D admits some sort of special structure which can be exploited, as will be the case in one of the problems we consider in a later chapter.

2.3 Generalized Convexity

We shall now consider some notions that generalize convexity and explore the relations between these notions.

Definition 2.12. (Quasiconvexity) A function $f : U \to \mathbb{R}$ on a convex set $U \subseteq \mathbb{R}^n$ is called *quasiconvex* if for all $x, y \in U, m \in [x, y]$ we have

$$f(m) \leq \max\{f(x), f(y)\}.$$

If the inequality is strict for all $x \neq m \neq y$, f is called *strictly quasiconvex*.

Proposition 2.13. *A function $f : U \to \mathbb{R}$ on a convex set $U \subseteq \mathbb{R}^n$ is quasiconvex iff its level sets*

$$L_s(f) := \{x \in U : f(x) \leq s\}$$

are convex for each $s \in \mathbb{R}$.

The two following proposition give alternative characterizations of quasiconvexity, for proofs see for instance [23].

Proposition 2.14. *A differentiable function $f : U \to \mathbb{R}$ on an open convex set U is quasiconvex iff for all $x, y \in U$*

$$\nabla f(x)^\top (y - x) > 0 \implies f(y) > f(x).$$

Definition 2.15. (Pseudoconvexity) A differentiable function $f : U \to \mathbb{R}$ on an open convex set $U \subseteq \mathbb{R}^n$ is called *pseudoconvex* if for all $x, y \in U$ we have

$$\nabla f(x)^\top (y - x) \geq 0 \implies f(y) \geq f(x).$$

Another generalization of convex functions is the class of functions for which stationary points are global minima. These where called *invex* by Craven [6], and a first exploration of the relationship between invexity and other notions of generalized convexity was given by Ben-Israel and Mond [1].

Definition 2.16. (Invexity) A differentiable function $f : U \to \mathbb{R}$ on a convex set $U \subseteq \mathbb{R}^n$ is called (η-)*invex* if there exists a function $\eta : U \times U \to \mathbb{R}^n$ such that

$$\nabla f(x)^\top \eta(y, x) \leq f(y) - f(x)$$

for all $x, y \in U$. Analogously, we say f is *pseudoinvex* if

$$\nabla f(x)^\top \eta(y, x) \geq 0 \implies f(y) \geq f(x),$$

and f is *quasi-invex* if

$$\nabla f(x)^\top \eta(y, x) > 0 \implies f(y) > f(x).$$

Differentiable convex functions are invex as can be seen by taking $\eta(y, x) = (y - x)$ and using the first-order condition.

Proposition 2.17. *A differentiable function $f : U \to \mathbb{R}$ on a convex set $U \subseteq \mathbb{R}^n$ is invex iff all of its stationary points are global minima.*

Proof. Suppose f is invex w.r.t. η. Let $x^* \in U$ such that $\nabla f(x^*) = 0$. Then for all $y \in U$ we have

$$f(y) - f(x^*) \geq \nabla f(x^*)^\top \eta(y, x^*) = 0, \tag{2.3.1}$$

and so x^* is a global minimum of f. Now suppose that all stationary points of f are global minima. Define η by

$$\eta(y, x) = \begin{cases} \frac{f(y)-f(x)}{||\nabla f(x)||^2} \cdot \nabla f(x) & \text{if } \nabla f(x) \neq 0, \\ 0 & \text{else.} \end{cases}$$

Let $x^* \in X$. If $\nabla f(x^*) = 0$ then x^* is a global minimum of f, and so for each $y \in Y$

$$\nabla f(x^*)^\top \eta(y, x^*) = 0 \geq f(y) - f(x^*).$$

If instead x^* is not a stationary point, we get

$$\nabla f(x^*)^\top \eta(y, x^*) = \nabla f(x^*)^\top \frac{f(y) - f(x^*)}{||\nabla f(x^*)||^2} \cdot \nabla f(x^*) = f(y) - f(x^*).$$

□

Note that the inequality (2.3.1) can also be obtained if f is only pseudoinvex. It follows that there is no difference between pseudoinvex and invex functions (different from the case of regular convexity).

Definition 2.18. A function $f : U \to \mathbb{R}^n$ is called *local-global* if all its local minima are global minima. If this implication only holds for strict local minima, f is called *weakly local-global*.

Proposition 2.19. *If $f : U \to \mathbb{R}$ is a quasiconvex function, then f is weakly local-global.*

Proof. Assume x' is a local but not a global minimum, i.e. there exists some $x^* \in U$ with $f(x^*) < f(x')$. Since x' is a strict local minimum, there is an $m \in (x', x^*)$ with $f(x') < f(m)$. But this means f is not quasiconvex, as $f(m) > \max\{f(x'), f(x^*)\}$. □

Note that strictly quasiconvex functions are local-global (by a similar proof) but quasiconvex functions in general are not, as can be observed from $f(x) = \max\{0, -x\}$ on $\mathbb{R}$ (taking e.g. $x' = -1$).

Proposition 2.20. *If $f : U \to \mathbb{R}$ is pseudoconvex, then f is invex.*

Proof. Let $x^* \in U$ be a stationary point, and $y \in X$ arbitrary. As $\nabla f(x^*) = 0$, we have $\nabla f(x^*)^\top (y - x^*) = 0$, and so by pseudoconvexity of f, $f(y) \geq f(x^*)$. So f is invex by Prop 2.17. □

Proposition 2.21. *If $f : U \to \mathbb{R}$ is invex on $U \subseteq \mathbb{R}^n$, then f is pseudoconvex if $n = 1$.*

Proof. Suppose f is not pseudoconvex, i.e. there are $x, y \in U$ such that $f'(x)(y - x) \geq 0$ but $f(x) > f(y)$. W.l.o.g. assume $f'(x) \geq 0$ and $y > x$. If $f'(x) = 0$, x is a global minimum of f which is a contradiction. So $f'(x) > 0$. By continuity of f, there is an $m \in (x, y)$ such that $f(m) = f(x)$ and $f(z) > f(x)$ for all $z \in (x, m)$. By Rolle's theorem, there is an $m' \in (x, m)$ with $f'(m') = 0$. But then m' is a global minimum of f which is a contradiction as $f(m') > f(x)$. □

Note that this proposition does not hold in general for $n > 1$.

Example 2.2. Let $U = (-3, 3) \times (1, 3)$ and define $f : U \to \mathbb{R}$ by $f(x_1, x_2) = x_2 - x_1^2$. As f does not have stationary points on U, it is certainly invex. However, if we take $x^* = (-1, 2)$ and $y = (2, 2)$ we have

$$f(y) - f(x^*) = -2 - 1 < 0,$$

but

$$\nabla f(x^*)^\top (y - x^*) = \frac{\partial f}{\partial x_1}(x^*) \cdot x_1^* > 0$$

and so f is not pseudoconvex. In fact, if we take $m = (0, 2) \in [x^*, y]$, we have $f(m) = 2 > \max\{f(x^*), f(y)\}$ and so f is not even quasiconvex.

Proposition 2.22. *Let $f : U \to \mathbb{R}$ be a differentiable function on a convex set $U \subseteq \mathbb{R}^n$. The following statements hold:*

1. *If f is convex, f is pseudoconvex,*
2. *If f is pseudoconvex, f is strictly quasiconvex.*

Proof. For the first statement, assume f is convex. Then by the first order condition $\nabla f(x)(y - x) \leq f(y) - f(x)$ for all $x, y \in U$ and so

$$\nabla f(x)^\top (y - x) \geq 0 \implies f(y) - f(x) \geq 0$$

as required.

For the second statement assume that f is pseudoconvex but not strictly quasiconvex, i.e. for some $x, y \in U$, $m \in (x, y)$ we have that $f(m) \geq \max\{f(x), f(y)\}$. If $\nabla f(m)^\top (y - m) \geq 0$, we have a contradiction since $f(y) < f(m)$. If instead $\nabla f(m)^\top (y - m) < 0$, we have $\nabla f(m)^\top (x - m) \geq 0$, which is a contradiction again as $f(x) < f(m)$.

□

For each of the statements in Proposition 2.22 the converse does not hold: the logistic function $f_{log}(x) = \frac{1}{2} + \frac{1}{2}\tanh(\frac{x}{2})$ is pseudoconvex but not convex on $\mathbb{R}$, the polynomial x^3 is strictly quasiconvex but not pseudoconvex on $\mathbb{R}$ (it has a stationary point at $x = 0$ which is not a local minimum).

2.3.1 Convexifiability

Even if a function is not convex, it may still be the case that it can be made convex by first applying some coordinate transform. For example the function[1] $f(x_1, x_2) = x_1 x_2$ is not convex on $\mathbb{R}^2_{>0}$, but it can be made convex by first applying the diffeomorphism $\phi : x \mapsto e^x$. Now the original non-convex function f can be minimized by minimizing the convex function $f \circ \phi$. Functions which can be made convex in this way were first named *convexifiable* by Hanson and Mond in [18].

Definition 2.23. Let $U \subseteq \mathbb{R}^n, V \subseteq \mathbb{R}^m$ be convex sets. Let $f : U \to \mathbb{R}$ be a differentiable function and $\phi : V \to U$ a diffeomorphism. We say f is (ϕ-)convexifiable if $f \circ \phi : V \to \mathbb{R}$ is convex.

Convexifiable functions are invex, as can be seen by showing that invexity is preserved under diffeomorphisms.

Proposition 2.24. *Let $U \subseteq \mathbb{R}^n, V \subseteq \mathbb{R}^m$ be convex sets. Let $f : U \to \mathbb{R}$ be an invex function and $\phi : V \to U$ a diffeomorphism. Then $f \circ \phi : V \to \mathbb{R}$ is invex.*

Proof. For each $x \in V$, we have

$$\nabla(f \circ \phi)(x) = D\phi(x)^\top \nabla f(\phi x)$$

by the chain rule. As $D\phi(x)$ is invertible, this means that $\nabla(f \circ \phi)(x) = 0$ iff $\nabla f(\phi x) = 0$. In other words, $x \in V$ is a stationary point of $(f \circ \phi)$ iff ϕx is a stationary point of f. Clearly, x is a global minimum of $(f \circ \phi)$ iff ϕx is a global minimum of f. So invexity of f implies invexity of $f \circ \phi$ by Proposition 2.17. □

Corollary 2.25. *Convexifiable functions are invex.*

[1] This function is a so-called monomial, which will be studied in section 2.4.

If $f \circ \phi$ is η-invex, we can explicitely compute a function $\theta : U \times U \to \mathbb{R}^n$ depending only on η and ϕ such that f is θ-invex: For any $x, y \in V$ we have

$$\begin{aligned} f(y) - f(x) &= (f \circ \phi)(\phi^{-1}y) - (f \circ \phi)(\phi^{-1}x) \\ &\geq \nabla(f \circ \phi)(\phi^{-1}x)^\top \eta(\phi^{-1}y, \phi^{-1}x) \\ &= \nabla f(\phi\phi^{-1}x)^\top D(\phi)(\phi^{-1}x)\eta(\phi^{-1}y, \phi^{-1}x) \\ &= \nabla f(x)^\top D(\phi)(\phi^{-1}x)\eta(\phi^{-1}y, \phi^{-1}x), \end{aligned}$$

and so we may choose $\theta(y, x) = D(\phi)(\phi^{-1}x)\eta(\phi^{-1}y, \phi^{-1}x)$. In particular, if f is ϕ-convexifiable, it is θ-invex for $\theta(y, x) = D(\phi)(\phi^{-1}x)(\phi^{-1}y - \phi^{-1}x)$ which only depends on ϕ, as was already noted in [18]. Unsurprisingly, not all invex functions are convexifiable.

Example 2.3. Consider again the invex function $f(x_1, x_2) = x_2 - x_1^2$ on $U = (-3, 3) \times (1, 3)$ from Example 2.2. The level set

$$L_1(f) = \{(x_1, x_2) \in U : x_2 - x_1^2 \leq 1\}$$

is not connected. Now if $\phi : V \to U$ is a diffeomorphism, we have

$$L_1(f \circ \phi) = \phi^{-1}(L_1(f))$$

which is also not connected as ϕ is continuous. In particular, $L_1(f \circ \phi)$ is not convex, and so $f \circ \phi$ is not convex by Proposition 2.13 (in fact, it is not even quasiconvex).

2.3.2 Biconvexity

We shall now study *biconvexity*, following the structure and definitions of the survey paper [15] in part. Let $X \subseteq \mathbb{R}^n, Y \subseteq \mathbb{R}^m$ be non-empty convex sets and let $B \subseteq X \times Y \subseteq \mathbb{R}^n \times \mathbb{R}^m$. For $x \in X, y \in Y$ write

$$B_x := \{y \in Y : (x, y) \in B\} \text{ and } B_y := \{x \in X : (x, y) \in B\}.$$

Definition 2.26. We say that B is *biconvex* on $X \times Y$ if B_x, B_y are convex for all $x \in X, y \in Y$.

Clearly, any convex set $C \subseteq X \times Y$ is biconvex on $X \times Y$, and so if $C_1 \subseteq X$ and $C_2 \subseteq Y$ are both convex, their product $C_1 \times C_2 \subseteq X \times Y$ is biconvex. For an example of a biconvex set that is not convex, take for instance a plus sign '+' $\subseteq \mathbb{R} \times \mathbb{R}$. A biconvex set does not need to be connected, consider for instance

$$B = \{(-1,-1),(1,1)\} \subseteq \mathbb{R} \times \mathbb{R}.$$

Lemma 2.27 (Goh et al. [16])**.** *A set $B \subseteq X \times Y$ is biconvex iff $[x_1, x_2] \times [y_1, y_2] \subseteq B$ for all $x_1, x_2 \in X$ and $y_1, y_2 \in Y$ with $(x_i, y_j) \in B$ for $i, j = 1, 2$.*

From this lemma, it can be seen that as in the convex case, intersections of biconvex sets are again biconvex.

Definition 2.28. We say that a function $f : B \to \mathbb{R}$ on a biconvex set $B \subseteq X \times Y$ is *biconvex* if for all (fixed) $x \in X, y \in Y$ the functions $f_x : B_x \to \mathbb{R}$ and $f_y : B_y \to \mathbb{R}$ given by

$$f_x(\star) := f(x, \star) \text{ and } f_y(\star) = f(\star, y)$$

are convex. In the same way, we define *biconcave*, *biaffine* and *bilinear*, *bi-invex*, *bipseudoconvex* etc. functions.

Just as in the case of regular convexity, a function f is biconvex iff $-f$ is biconcave, and biaffine iff it is both biconvex and biconcave. We can give several equivalent direct definitions of biconvexity.

Theorem 2.29 (Goh et al. [16])**.** *Let $f : B \to \mathbb{R}$ be any function on a biconvex set $B \subseteq X \times Y$. Then the following are equivalent:*

1. *f is biconvex on $X \times Y$,*
2. *For all $x_1, x_2 \in X, y_1, y_2 \in Y$ and $t, s \in [0, 1]$ we have*

$$\begin{aligned} f(m_x, m_y) \leq\ & tsf(x_1, y_1) + t(1-s)f(x_1, y_2) + \\ & (1-t)sf(x_2, y_1) + (1-t)(1-s)f(x_2, y_2), \end{aligned}$$

   where $m_x = tx_1 + (1-t)x_2$, $m_y = sy_1 + (1-s)y_2$.

3. *The epigraph* $\mathrm{Epi}(f)$ *is biconvex on $X \times Y$.*

Lemma 2.30 (Gorski et al. [15]). *If $f : B \to \mathbb{R}$ is biconvex, then its level sets*

$$L_s(f) := \{(x, y) \in X \times Y : f(x, y) \leq s)$$

are biconvex for all $s \in \mathbb{R}$.

A biconvex program is defined as any program of the form

$$\min\{f(x, y) : (x, y) \in B \subseteq X \times Y\}$$

where B is a biconvex set on $X \times Y$ and $f : B \to \mathbb{R}$ is a biconvex funtion. From Lemma 2.30 it follows that any program of the form

$$\begin{array}{ll} \min & f(x, y) \\ s.t. & g_i(x, y) \leq 0, \quad i = 1, \ldots, I \\ & h_j(x, y) = 0, \quad j = 1, \ldots, J \\ & x, y \in X \times Y, \end{array}$$

where f and the g_i are biconvex, and the h_j are biaffine is a biconvex program.

Definition 2.31. Let $f : B \to \mathbb{R}$ be some function. We say that $(x^*, y^*) \in B$ is a *partial minimum* of f if

$$f(x^*, y^*) \leq f(x, y^*) \text{ for all } x \in B_{y^*} \text{ and } f(x^*, y^*) \leq f(x^*, y) \text{ for all } y \in B_{x^*}.$$

That is, a point (x^*, y^*) is a partial minimum if the value of f cannot be further reduced by *either* changing x^* *or* y^*. Partial minima need not be local minima, even for biconvex functions. Take for instance $f(x, y) = xy$. Then $(0, 0)$ is a partial minimum, but $f(\epsilon, -\epsilon) < 0$ for all $\epsilon > 0$.

Proposition 2.32 (Gorski et al. [15]). *If $f : B \to \mathbb{R}$ is (partial) differentiable at $z^* = (x^*, y^*) \in B$, and z^* is a partial minimum, then $\nabla f(z^*) = 0$.*

Corollary 2.33. *If $f : B \to \mathbb{R}$ is differentiable, bi-pseudoconvex and invex then $z^* \in B$ is a partial minimum of f if and only if it is a global minimum of f on B.*

The converse of Proposition 2.32 does not hold in general, take for example $f(x, y) = x^2 - y^2$ on $\mathbb{R}^2$. Then $\nabla f(0, 0) = 0$, but $f(0, 1) = -1 < f(0, 0)$. For biconvex f, however, it does.

Proposition 2.34 (Gorski et al. [15])**.** *If $f : B \to \mathbb{R}$ is differentiable and bipseudoconvex, then for all $z^* = (x^*, y^*) \in B$ with $\nabla f(z^*) = 0$, z^* is a partial minimum.*

Proof. Suppose z^* is not a partial minimum. Then we may asume w.l.o.g. that there exists some $x \in B_{y^*}$ such that $f(x, y^*) < f(x^*, y^*)$. As $f_{y^*}(x) := f(x, y^*)$ is a pseudoconvex function, we have

$$\nabla f_{y^*}(x^*)^\top (x^* - x) < 0.$$

In particular $\nabla f_{y^*}(x^*) \neq 0$ and so $\nabla f(x^*, y^*) \neq 0$, a contradiction. □

We have shown that for bi(pseudo)convex functions, partial minima are exactly stationary points.

Alternate Convex Search

We will now describe a fairly straightforward algorithm for solving a biconvex program of the form

$$\min\{f(x, y) : (x, y) \in B\} \tag{2.3.2}$$

which is a special case of the block-relaxation methods by de Leeuw [17]. A survey on this algorithm can be found in [31].

Algorithm 2.35. (Aternate Convex Search)

1. Choose an initial point $(x_0, y_0) \in B$, set $i = 0$.

2a. Compute an optimum solution x^* of

$$\min\{f(x, y_i) : x \in B_{y_i}\}. \tag{2.3.3}$$

If such a solution does not exist, return (x_i, y_i).

2b. If $f(x^*, y_i) < f(x_i, y_i)$ for the solution x^* found in 2a, set $x_{i+1} = x^*$. Otherwise, set $x_{i+1} = x_i$.

3a. Compute an optimum solution y^* of

$$\min\{f(x_{i+1}, y) : y \in B_{x_{i+1}}\}. \tag{2.3.4}$$

If such a solution does not exist, return (x_{i+1}, y_i).

3b. If $f(x_{i+1}, y^*) < f(x_{i+1}, y_i)$ for the solution y^* found in 3a, set $y_{i+1} = y^*$. Otherwise, set $y_{i+1} = y_i$.

4. If some stopping criterion is met, return (x_i, y_i). Otherwise, increment i and repeat step 2.

The programs occuring in steps 2 and 3 of the algorithm are convex, and so can usually be solved efficiently.

Of course, this algorithm is not guaranteed to converge to a minimum solution of (2.3.2). Consider the sequences (z_i) and (f_i) given by $z_i = (x_i, y_i)$ and $f_i = f(x_i, y_i)$ produced by Algorithm 2.35. Clearly, (f_i) is a monotonically decreasing sequence, and so if f is bounded from below it is guaranteed to converge to some value as long as the programs (2.3.3) and (2.3.4) are solvable in each iteration. The sequence (z_i) however, might diverge.

The following proposition gives a simple guarantee on the solutions found by ACS. The statement given here is adapted from [15] and a different proof is given.

Proposition 2.36 (Gorski et al. [15])**.** *Let $X \subseteq \mathbb{R}^n, Y \subseteq \mathbb{R}^m$ be closed sets and let $f : X \times Y \to \mathbb{R}$ be a continous function that is bounded from below. If the sequence (z_i) produced by Algorithm 2.35 has an accumulation point $z^* \in \mathbb{R}^n \times \mathbb{R}^m$, then $z^* \in X \times Y$ and z^* is a partial minimum of f. Furthermore, all accumulation points of (z_i) have the same function value.*

Proof. Let z^* be any accumulation point. Since $X \times Y$ is closed, we have $z^* \in X \times Y$. As we noted before, $\lim_{i\to\infty} f(z_i)$ exists. Now let $(z_{i_j}) \subseteq (z_i)$ be a subsequence such that $z^* = \lim_{j\to\infty} z_{i_j}$. By continuity of f we get

$$f(z^*) = f(\lim_{j\to\infty} z_{i_j}) = \lim_{j\to\infty} f(z_{i_j}) = \lim_{i\to\infty} f(z_i),$$

showing all accumulation points have the same function value. Now suppose that $z^* = (x^*, y^*)$ is not a partial optimum of f. Without loss of generality we may then assume there is a $y' \in Y$ and an $\epsilon > 0$ such that

$$f(x^*, y') \leq f(x^*, y^*) - \epsilon.$$

By continuity of f, there exists a $\delta > 0$ such that

$$||z - z'|| < \delta \implies |f(z) - f(z')| < \epsilon.$$

Now consider a $k \in \mathbb{N}$ such that $|x_k - x^*| < \delta$. Then $||(x_k, y') - (x^*, y')|| < \delta$ and so by definition of y_k we get

$$f(x_k, y_k) \leq f(x_k, y') < f(x^*, y') + \epsilon \leq f(x^*, y^*) = \lim_{i \to \infty} f(x_i, y_i),$$

which is a contradiction as (f_i) is a monotonically decreasing sequence. □

Note that we may drop the assumption that X, Y are closed from the above proposition and assume instead that $z^* \in X \times Y$. The following result is immediate:

Proposition 2.37. *Let $X \subseteq \mathbb{R}^n, Y \subseteq \mathbb{R}^m$ be compact sets and let $f : X \times Y \to \mathbb{R}$ be a continous function. Then the sequence (z_i) has at least one accumulation point.*

Combining Propositions 2.36 and 2.37 we see that for compact $X \times Y$ and continuous $f : X \times Y \to \mathbb{R}$ we may find a partial optimum of f on $X \times Y$ using ACS as long as the programs (2.3.3) and (2.3.4) are solvable in each iteration.

2.4 Posynomials and Geometric Programming

Definition 2.38. A function $f : U \to \mathbb{R}$ of the form

$$f(x) = f(x_1, x_2, \ldots, x_n) = c x_1^{a_1} x_2^{a_2} \ldots x_n^{a_n},$$

where $c \in \mathbb{R}_{>0}$, $a_i \in \mathbb{R}$ arbitrary and $U \subseteq \mathbb{R}^n_{>0}$, is called a *monomial*. A finite sum of monomials is called a *posynomial*.

Monomials in general are not convex functions, even when the a_i are positive. Take for example $\sqrt{x}$ on $U = \mathbb{R}^n_{>0}$, which is strictly concave. In fact, they are not even quasiconvex. For an example here, take $f(x, y) = xy$ and observe that

$$f(1, 2) = 2 = f(2, 1) < f(1.5, 1.5) = 2.25.$$

However for any monomial f on a convex set U, the function $F(y) := \log f(e^y)$ is affine for $y \in \log U$, which is a convex set as e^y is a convex function. Indeed, if $f(x) = cx_1^{a_1}x_2^{a_2}\dots x_n^{a_n}$ is a monomial, then

$$\begin{aligned} F(y) &= \log f(e^y) \\ &= \log[ce^{a_1y_1}e^{a_2y_2}\dots e^{a_ny_n}] \\ &= \sum_{i=1}^n a_iy_i + \log c. \end{aligned}$$

If $f(x)$ is a posynomial $F(y)$ is not affine in general, but will be convex.

Proposition 2.39. *For any posynomial $f : U \to \mathbb{R}$ on a convex set $U \subseteq \mathbb{R}^n_{>0}$ the function $F : \log U \to \mathbb{R}$ is convex.*

Proof. We have $f = \sum_{i=1}^k f_i$, where f_i are monomials. For each i, let $a^i \in \mathbb{R}^n, c_i \in \mathbb{R}$ such that $f_i(x) = c_i \prod_{j=1}^n (x_j)^{a^i_j}$ and so

$$f_i(e^x) = \exp(\log c_i + \sum_{j=1}^n a^i_j x_j) = \exp(\log c_i + (a^i)^\top x).$$

Writing $A \in \mathbb{R}^{k\times n}$ for the matrix having rows a^i, and $b \in \mathbb{R}^k$ for the vector with entries $b_i = \log c_i$ we get that

$$F(x) = \log\big(\sum_{i=1}^k \exp(\log c_i + \sum_{j=1}^n a^i_j x_j)\big) = \mathrm{LSE}(Ax + b),$$

showing that F is the composition of a convex function with an affine function, and therefore convex. □

It will be convenient later on to define a term to refer to functions that become convex after an exponential variable substitution.

Definition 2.40. We say a function $f : U \to \mathbb{R}$ on a convex domain $U \subseteq \mathbb{R}^n_{>0}$ is *exponentially* convex if $u \mapsto f(e^u)$ is convex for $u \in \log U$. As an example, we have just proven that posynomials are exponentially log-convex.

Lemma 2.41. *A function $f : U \to \mathbb{R}$ is exponentially convex iff*

$$f(x^t \cdot y^{t-1}) \leq tf(x) + (1-t)f(y)$$

for all $x, y \in U$ and $t \in [0, 1]$.

Proof. Write g for the function $u \mapsto f(e^u)$. Assuming the condition holds we have

$$\begin{aligned} g(tx + (1-t)y) &= f(e^{tx+(1-t)y}) \\ &= f((e^x)^t \cdot (e^y)^{(1-t)}) \\ &\leq tf(e^x) + (1-t)f(e^y) \\ &= tg(x) + (1-t)g(y) \end{aligned}$$

for all $x, y \in U$ and $t \in [0, 1]$ and so g is convex. The other direction of the proof is analogous. □

In words, what this means is that a function is exponentially convex iff the (weighted) *arithmetic* mean of $f(x), f(y)$ is always an overestimator of the function value at the (weighted) *geometric* mean of x, y.

Geometric Programming

Note that if $u, v \in U$ such that $f(u) \geq f(v)$, we have that

$$F(\log u) = \log f(u) \geq \log f(v) = F(\log v)$$

and so the local / global minima of f and F are in one-to-one correspondence.

A *Geometric Program* (GP) has the form

$$\begin{array}{ll} \min & f_0(x) \\ s.t. & f_i(x) \leq 1, \quad i = 1, \ldots, I \\ & g_j(x) = 1, \quad j = 1, \ldots, J \\ & x \in \mathbb{R}^n_{>0}, \end{array}$$

where the f_i are posynomials and the g_j are monomials.

From our previous discussion, it follows that we may transform any GP to the equivalent convex program:

$$\begin{array}{ll} \min & F_0(y) \\ s.t. & F_i(y) \leq 0, \quad i = 1, \ldots, I \\ & G_j(y) = 0, \quad j = 1, \ldots, J \\ & y \in \mathbb{R}^n_{>0}. \end{array}$$

We shall refer to the first a program as being in *posynomial form*, and to the second as being in *in geometric form*. For a more detailed overview, see for instance [24].

A major drawback of geometric programming is its inability to model even simple (linear) constraints with negative coefficients or equality constraints with multiple terms. For instance, the rather innocent looking program

$$\begin{array}{ll} \min & \dfrac{x_1}{x_2} \\ s.t. & x_1 + x_2 = 1, \\ & x \in \mathbb{R}^2_{>0}, \end{array}$$

is not geometric. The problem with this program is that the objective function is only *exponentially* convex, but not convex, whereas the equality constraint is affine, but not exponentially affine. The result is that the program is convex neither before nor after performing the variable change of section 2.4.

A program of the form:

$$\begin{aligned} \min \quad & f_0(x) \\ s.t. \quad & f_i(x) \leq 0, \quad i = 1, \ldots, I \\ & x \in \mathbb{R}^n_{>0}, \end{aligned}$$

where the f_i are now only restricted to being *signomials* (i.e. arbitrary linear combinations of monomials) is called a *generalized geometric* or *signomial* program. Signomials programs are not convexifiable in general and can have multiple local minima that are not global minima. By reordering terms, we may rewrite any program of this form to an equivalent program of the form

$$\begin{aligned} \min \quad & f_0^+(x) - f_0^-(x) \\ s.t. \quad & f_i^+(x) - f_i^-(x) \leq 0, \quad i = 1, \ldots, I \\ & x \in \mathbb{R}^n_{>0}, \end{aligned}$$

where all f_i^+, f_i^- are posynomials.

In practice, geometric programs can be solved rather efficiently. As they are convexifiable, general methods for convex programming may be used to solve them, although more specialized methods have also been suggested, making use for instance of the fact that the constraints of a geometric program are log-convex after the exponential variable transformation.

To find a locally optimal solution for signomial programs the negated posynomials f_i^- are usually approximated either by linear or monomial functions to obtain a geometric program whose solution provides a bound for the original signomial program. These approximations are then improved iteratively. For an overview of different algorithms for solving geometric and signomial programs and their practical performance, see [20].

2.5 Quadratic Programming

A program of the form

$$\begin{aligned} \min \quad & x^\top Q x + r^\top x + s \\ s.t. \quad & Ax = b, \\ & x \in \mathbb{R}^n, \end{aligned} \tag{2.5.1}$$

with $Q \in \mathbb{R}^{n\times n}$, $r \in \mathbb{R}^n$ and $s \in \mathbb{R}, A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m$ is called a *quadratic program*. If Q is positive semi-definite, 2.5.1 is a convex program.

Proposition 2.42. *Let $Q \in \mathbb{R}^{n\times n}$. The function $f : \mathbb{R}^n \to \mathbb{R}$ given by $f(x) = x^\top Qx$ is convex iff Q is positive semi-definite.*

Proof. As

$$\nabla^2 f(x) = 2Q,$$

this is a direct consequence of the second-order condition for convexity. □

As a result of the above proposition, quadratic programs can be solved using general methods for convex programming, although more specialized methods have also been suggested, see for instance [23].

2.6 Resource Sharing

The problem of finding an optimum distribution of a set of limited resources among several competing consumers occurs frequently in many applications. One way of modeling such a problem mathematically is as a so-called *min-max resource sharing* problem.

Min-Max Resource Sharing

Input: Finite sets $\mathcal{R}$ of *resources* and $\mathcal{C}$ of *customers*. A convex set $\mathcal{B}_c$ of feasible solutions for each customer $c \in \mathcal{C}$ (also called *block*), together with a nonnegative convex function $g_c : \mathcal{B}_c \to \mathbb{R}^{\mathcal{R}}_{\geq 0}$ indicating the consumption of each resource for each feasible solution.

Output: Feasible solutions $b = (b_c)_{c\in C}$, where $b_c \in \mathcal{B}_c$ for each customer $c \in \mathcal{C}$ such that the maximum resource consumption

$$\lambda(b) := \max_{r\in\mathcal{R}} \sum_{c\in\mathcal{C}} (g_c(b_c))_r$$

is minimized. We shall write λ^* for the value of such a minimum solution.

Generally, we assume that we are provided with *block solvers* $f_c : \mathbb{R}^{\mathcal{R}}_{\geq 0} \to \mathcal{B}_c$ for each customer c, which (approximately) solve the minimization problem $\min_{x \in \mathcal{B}_c}\{y^\top g_c(x)\}$ for any $y \in \mathbb{R}^{\mathcal{R}}_{\geq 0}$. That is, given a *weight* for each resource, they find a feasible solution for customer c that minimizes resource consumption with respect to these weights. The quality of the block solvers is given as a constant $\sigma \geq 1$ for which it is guaranteed that $y^\top f_c(y) \leq \sigma \inf_{x \in \mathcal{B}_c}\{y^\top g_c(x)\}$ for all $y \in \mathbb{R}^{\mathcal{R}}_{\geq 0}$. Block solvers are called *exact* if $\sigma = 1$, *strong* if σ can be chosen arbitrarily close to 1 and *weak* if $\sigma > 1$.

The min-max resource sharing model is quite general, having well-known problems such as maximum concurrent flow and fractional packing as special cases.

Example 2.4. In the *fractional packing problem* we are given a convex set $P \subseteq \mathbb{R}^n_{\geq 0}$ and a set of m linear inequalities $Ax \leq d$, with $A \geq 0$. The task is to find an $s \in P$ such that $As \leq d$ or determine no such s exists. We can model the problem as an instance of the min-max resource sharing problem by considering a single customer c with $\mathcal{B}_c := P$, and a resource for each $i \in [m]$. Writing a_i for the rows of A, we define the nonnegative convex function $g_c : P \to \mathbb{R}^m_{\geq 0}$ by

$$g_c(p)_i := \max\{a_i^\top p - d_i, 0\}, \qquad \text{for } i \in [m].$$

It can be seen easily that any feasible solution b_c with $\lambda(b_c) = 0$ satisfies $A^\top b_c \leq d$, and conversely that any $s \in P$ with $A^\top s \leq d$ is a feasible solution to the resource sharing instance with $\lambda(s) = 0$. A block solver f_c for c is only required to (approximately) minimize linear functions over P.

The Resource Sharing Algorithm

In this section we will present an FPTAS for the resource sharing problem proposed by Müller, Radke and Vygen in [7]. By FPTAS here we mean that for any $\omega > 0$ the algorithm should compute a solution $b = (b_c)_{c \in \mathcal{C}}$ for which $\lambda(b) \leq \sigma(1+\omega)\lambda^*$ in running time polynomial in ω^{-1}. That is, the quality of the approximation is allowed to depend on the quality of the provided block solvers.

Resource Sharing Algorithm

Input: An instance of the min-max resource sharing problem, together with a constant $\sigma \geq 1$ and block solvers $f_c : \mathbb{R}^{\mathcal{R}}_{\geq 0} \to \mathcal{B}_c$ with quality guarantee σ. Parameters $\epsilon > 0$ and $t \in \mathbb{N}$.

Output: A convex combination $\sum_{b \in \mathcal{B}_c} x_{c,b} b$ of feasible solutions for each customer c, and a weight vector $y \in \mathbb{R}^{\mathcal{R}}_{\geq 0}$.

Algorithm 1 Resource Sharing Algorithm

1: $\alpha_r \leftarrow 0$, $y_r \leftarrow 1$ for each $r \in \mathcal{R}$.
2: $X_c \leftarrow 0$, $x_{c,b} \leftarrow 0$ for each $c \in \mathcal{C}$ and $b \in \mathcal{B}_c$.
3: **for** $p = 1 \dots t$ **do**
4: **while** exists $c \in \mathcal{C}$ with $X_c < p$ **do**
5: Pick a $c \in C$ with $X_c < p$.
6: AllocateResources(c)
7: $x_{c,b} \leftarrow \frac{1}{t} x_{c,b}$ for each $c \in C$ and $b \in \mathcal{B}_c$.

8: **procedure** AllocateResources(c)
9: $b \leftarrow f_c(y)$.
10: $\xi \leftarrow \min\{p - X_c, 1/\max_{r \in \mathcal{R}}\{g_c(b)_r\}\}$.
11: $x_{c,b} \leftarrow x_{c,b} + \xi$, $X_c \leftarrow X_c + \xi$.
12: $\alpha \leftarrow \alpha + \xi g_c(b)$.
13: **for all** $r \in \mathcal{R}$ with $a_r \neq 0$. **do**
14: $y_r \leftarrow e^{\epsilon \alpha_r}$.

We will refer to the iterations of the for-loop in line 3 of the algorithm as *phases*.

Theorem 2.43 (Müller et al. [7])**.** *Using the resource sharing algorithm, a $\sigma(1+\omega)$ approximate solution to the resource sharing problem can be computed in $\mathcal{O}(\theta \log |\mathcal{R}|((|\mathcal{C}| + |\mathcal{R}|) \log\log |\mathcal{R}| + (|\mathcal{C}| + \mathcal{R}\sigma)\sigma\omega^{-2}))$ time, where θ is an upper bound on the running time of the provided block solvers.*

Modifications in practice

In practice, the resource sharing algorithm is often run for a fixed number of phases to limit runtime (in which case perfomance guarantees no longer hold). Additionally, it is possible to apply the algorithm to problems for which the set of feasible solutions $\mathcal{B}_c$ for some customer c is finite and nonconvex (for instance the set of all spanning trees on a graph). In this case, one simply defines a new convex set $\mathcal{B}'_c$ of feasible solutions for c consisting of all convex combinations of element of $\mathcal{B}_c$, and extends the resource consumption function g_c accordingly. In order to extract an original feasible solution from the convex combination $\sum_{b \in \mathcal{B}_c} x_b b$ returned by the algorithm, one draws an element from $\mathcal{B}_c$ at random with probabilities given by the x_b. Müller et al. refer to this procedure as *randomized rounding* and prove that the resource consumption of the rounded solution is not much worse than that of the original with high probability.

Chapter 3

Chip Design

The production of ever faster and smaller computer chips has undoubtedly had a large impact on modern life. The development of these chips is often referred to as *chip design* or *VLSI (Very Large Scale Integration)* design, and is an important application of mathematical optimization. Many well-known combinatorial problems arise quite naturally in this field, although often with minor modifications. This, combined with the fact that the scale of modern instances can be very large (several billions of transistors) neccessitates the use of specialized methods and algorithms to solve or approximate them. The introduction we will give here is most certainly not complete, and we will leave out many details that are beyond the scope of this thesis. For a more complete overview, see for instance [3] and [29].

A computer chip can be thought of as a set of several *cells* placed on a rectangular surface, connected by metal wires. These cells can be *gates*, implementing some simple boolean function such as AND or NOR, but also small (predesigned) chips themselves, implementing a more complicated function. For simplicity we will assume that each cell is a gate, having multiple input pins and a single output pin. The set of all gates on a chip is denoted as $\mathcal{G}$. The points on the cell at which wires should connect are refered to as *pins*, and they correspond to either one of the inputs or the output of the implemented logical function. Additionally, there are special *primary* pins, which connect directly to the chip's surface and are not associated with a cell.

Wires are always arranged as trees, connecting the output pin of a cell (or a primary input pin) to an input pin of several different cells (or primary output pins). One such tree together with the pins it connects is called a *net*, and the set of all nets is a partition of the set of all pins. The single output (or primary input) pin of a net is called the *source*, and the several input (or primary output) pins are called *sinks*. If the source belongs to a cell, this cell is called the *driver* of the net.

On a more abstract level, we might model a chip as a directed acyclic graph $G = (V, E)$ whose vertices represent boolean functions f_v and whose arcs (v, w) represent a connection between the output of f_v and an input of f_w. Importantly, this model does not contain a description of exactly *how* the functions or these connections should be realized. We will think of G as a description of the logical interconnect of a chip. A high level interpretation of chip design, then, is the task of finding a physical implementation of G (in the above sense) that is optimal with respect to some given set of criteria. In this work we will be particularly interested in the choice of implementation of the gates (or cells) on the chip. Such a choice is refered to as a *book*. The set of all available books is called a *library*.

3.1 Signals

A chip operates by propagating *signals* along its cells. Writing V_0 for the ground voltage and V_{dd} for the voltage of the chip's power supply, a signal here refers to the change of a pin's voltage from V_0 (representing the logical value 0) to V_{dd} (representing the logical value 1) or the other way around. Signals starting at V_0 and ending at V_{dd} are called *rising*, whereas those starting at V_{dd} and ending at V_0 are called *falling*. The voltage change at a pin is not instantaneous. Instead, the voltage changes in a non-linear fashion over a short period of time. The exact shape of the signal differs depending on the implementation. The total time it takes for the signal to change from $\frac{1}{10}V_{dd}$ to $\frac{9}{10}V_{dd}$ is called the *slew*. We will often ignore the effects of slew in this work, and explicitely mention when we do take them into account. Instead, we will consider the *arrival time* of a signal at a pin, which is the time at which the voltage reaches $0.5V_{dd}$.

3.2 Timing Propagation

For a given implemented chip, the *timing graph* is the directed acyclic graph $G = (V, E)$ having a vertex for each pin and arcs between connected pins. The vertex set G can be partioned into three sets $V = P_{in} \dot\cup P_{out} \dot\cup P_{gate}$ corresponding to the primary inputs, primary outputs and gate pins of the chip respectively.

In *timing analysis* we consider the time it takes for the signal to propagate from the primary inputs of a chip to its primary outputs. We assume that we are given some initial *arrival time* $\mathrm{at}(p)$ for each primary input $p \in P_{in}$, as well as a *required arrival time* $\mathrm{rat}(p)$ for each primary output $p \in P_{out}$. If we know the *delay* $\mathrm{delay}_{q,r}$ between a signal reaching the pins q and r for all $(q, r) \in E$, we may compute arrival times for all pins p by setting

$$\mathrm{at}(p) := \max_{(q,p)\in E} \mathrm{at}(q) + \mathrm{delay}_{q,p},$$

using the fact that G is a DAG. Similarly, we can compute required arrival times for all pins by setting

$$\mathrm{rat}(p) := \max_{(p,q)\in E} \mathrm{rat}(q) - \mathrm{delay}_{p,q}\,.$$

We say that a chip *meets timing requirements* if the arrival time $\mathrm{at}(p)$ is smaller than the required arrival time $\mathrm{rat}(p)$ for each pin (or, equivalently, for each primary output). The *slack* of a pin, defined as

$$\mathrm{slack}(p) := \mathrm{rat}(p) - \mathrm{at}(p),$$

represents the degree to which a specific pin meets timing. In particular, the chip meets timing if $\mathrm{slack}(p) \geq 0$ for all $p \in V$.

3.3 Elmore Delay

The question left open is how to actually compute $\mathrm{delay}_{p,q}$ for each arc (p, q) in the timing graph. The most commonly used model for this purpose is that of *Elmore delay* [9]. We assume that for each book b in our library we have an associated *size* s_b, base *capacity* γ_b and base *resistance* ρ_b. Additionally,

we assume a base capacity per length γ_{wire} and base resistance ρ_{wire} for all wire in the chip.

For an input pin p of a book b we define the *input capacitance* of p as $\gamma_b x_b$. For an output pin q of b, we define the *downstream capacitance* $\mathrm{downcap}(q)$ as the sum of all input capacitances of pins connected to q, as well as the capacitance of the wire connecting q to these pins.

The Elmore delay between an input pin p and an output pin q of a gate with book b is now given by

$$\mathrm{delay}_{p,q} := \frac{\rho_b}{s_b} \cdot \mathrm{downcap}(q).$$

If instead p is the output pin of a gate and q is an input pin of some other gate we need to consider the entire Steiner tree of wire segments connecting p to input pins of other gates. For each node w in this directed Steiner tree, we define the downstream capacitance $\mathrm{downcap}(w)$ as the sum of all input capacitances of pins downstream from w as well as the capacitance of all wire downstream from w. Now if P is the (unique) path connecting p and q in this tree, we have

$$\mathrm{delay}_{p,q} := \sum_{(v,w)\in E(P)} l_{v,w} \cdot \rho_{wire} \cdot \big(\frac{1}{2} \cdot \mathrm{cap}(v,w) + \mathrm{downcap}(w)\big), \qquad (3.3.1)$$

where $l_{v,w}$ is the length of the wire segment (v,w).

3.4 Power Analysis

Besides timing, the choice of a book b for a gate g also affects the *power consumption* of that gate. A gate consumes power mainly in the following two ways. Firstly, the gate continuously leaks a constant amount of current. Secondly, the gate consumes power each time the logical value of its transistors switch. The first effect is known as *static power consumption*, the second as *dynamic power consumption*. The dynamic power consumption depends linearly on the *switching frequency* χ_g of a gate g, which is independent of the chosen book.

For our purposes, the power consumption of a gate g is simply the sum of the static and dynamic consumptions, which we we will assume at all times

to be given for each book b and be linear in its size s_b (for fixed χ_g). As a result, the total power consumption of a chip is proportional to the sum of the sizes of the chosen book for each gate.

3.5 Gate Sizing

Assuming that the location of all gates and wires is fixed, an important task in chip design is to select the size s_g of each gate $g \in \mathcal{G}$ minimizing power consumption while still meeting timing requirements. This is known as the *gate sizing problem*.

Gate Sizing Problem	
Input:	A timing graph $G = (V, E)$ with gates $\mathcal{G}$. A set of potential sizes S_g for each $g \in \mathcal{G}$. Required arrival times for all primary output pins, and arrival times for all primary input pins.
Output:	A vector s of sizes $s_g \in S_g$ for each $g \in \mathcal{G}$ for which the chip meets timing requirements minimizing the total power consumption.

If S_g is an interval $[u_g, v_g]$ for each $g \in \mathcal{G}$ we shall call the resulting problem *continuous*. If instead S_g is some finite set for each $g \in \mathcal{G}$, we call the resulting problem *discrete*. It is shown for instance by Daboul [8] that the discrete gate sizing problem is NP-hard.

Under the Elmore model, the delay between two subsequent vertices p, q of the timing graph can always be written as

$$\text{delay}_{p,q}(s) = \sum_{m=1}^{M} c_m \prod_{g \in \mathcal{G}} s_g^{a_{g,m}},$$

where $c_m \in \mathbb{R}_{>0}$ and $a_{g,m} \in \mathbb{R}$ (in fact $a_{g,m} \in \{-1, 0, 1\}$). Note that this means that $\text{delay}_{p,q}(s)$ is a posynomial function, and therefore convexifiable. The total power consumption is equal to

$$P(s) = \sum_{g \in \mathcal{G}} p_g s_g,$$

for certain constants $p_g \in \mathbb{R}_{\geq 0}$. As is noted by Szegedy in [28] (and before by Fishburn and Dunlop in [14]), one way of solving the gate sizing problem therefore is to solve the program:

$$\begin{aligned} \min \quad & \sum_{g \in \mathcal{G}} p_g s_g, \\ s.t. \quad & a(p) + \text{delay}_{p,q}(s) \leq a(q) \quad \forall (p,q) \in E, \\ & s_g \in S_g \quad \forall g \in \mathcal{G}, \\ & a(p) = \text{rat}(p) \quad \forall p \in P_{out}, \\ & a(p) = \text{at}(p) \quad \forall p \in P_{in}. \end{aligned} \tag{3.5.1}$$

Here, the a variables can be viewed as the propagated (required) arrival times as in section 3.2. In the continuous case, 3.5.1 is a geometric program in posynomial form (see section 2.4). The currently fastest known method for solving this program is due to Chen Chu and Wong [5]. Their main idea is that the dual objective function $L(s, a, \lambda)$ can be separated as

$$L(s, a, \lambda) = L_1(s, \lambda) + L_2(a, \lambda),$$

with

$$L_1(s, \lambda) = \sum_{g \in \mathcal{G}} p_g s_g + \sum_{(p,q) \in E} \lambda_{p,q} \, \text{delay}_{p,q}(s).$$

The function L_1 can then be optimized efficiently for fixed λ by successively optimizing over s_g for a single gate g while fixing $s_{g'}$ for all other gates $g' \in \mathcal{G}$. They refer to this method as *local refinement*.

A second, much more recent method for solving the continuous gate sizing problem is due to Daboul, Hähnle, Held and Schorr [25] and makes use of the resource sharing algorithm (see section 2.6). They model the problem as an instance of resource sharing by considering

- A single resource for power usage.
- A resource for each directed path P from a primary input to a primary output in the timing graph, called *timing resources*.
- A single customer for all gates with feasible solution set $X = \prod_{g \in \mathcal{G}} S_g$.

They then assume a power budget $B \in \mathbb{R}_{\geq 0}$ and define the resource consumption g of a solution s as

$$g_{pow}(s) := \frac{\sum_{g \in \mathcal{G}} p_g s_g}{B}$$

for the power resource and

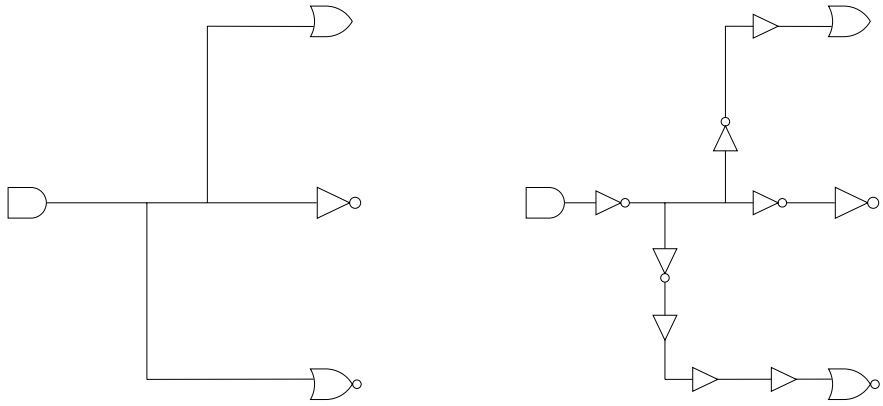
$$g_P(s) := \frac{\sum_{(p,q) \in E(P)} \text{delay}_{p,q}(s)}{\text{rat}(\hat{q}) - \text{at}(\hat{p})}$$

for the timing resource belonging to a path P starting in $\hat{p}$ and ending in $\hat{q}$. It should be noted that although the total number of timing resources may be exponential, it is still possible to compute the resource weights efficiently [12] using the conditional gradient method of Frank and Wolfe [10] as a basis for the block-solver. Clearly, a solution s meets the timing requirements and satisfies the power budget iff all resource consumptions are less than 1. It can be decided whether such a solution exists by running the resource sharing algorithm. In order to actually minimize the power usage one can perform binary search on B.

The method described above can easily be adapted into a heuristic for the discrete gate sizing problem by providing an alternative block-solver. Daboul, Hähnle, Held and Schorr suggest to use local refinement: They successively select a gate g and optimize the weighted resource consumption for $s_g \in S_g$ while fixing $s_{g'}$ for all other gates g'.

3.6 Buffering

From (3.3.1) it can be seen that the signal propagation delay through a wire segment scales quadratically in the length of that segment. As a result, it is highly undesirable to have long wires connecting gates on a chip. As it is not always possible to place connected gates within close proximity of each other, it is therefore useful to insert additional gates on long wire segments that do not change the circuits logic, but instead serve only to refresh the signal, preventing large delays due to quadratic scaling. These gates are called *repeaters* and they come in two basic types: *buffers* simply duplicate



(a) A net before repeaters are inserted. (b) A net after repeaters are inserted.

Figure 3.1: An example of a single net before and after repeater insertion.

the signal, while *inverters* invert it. When inserting inverters one should always make sure that every path between the output and input of two gates contains an even amount of them. Figure 3.1 shows an example of a net before and after inserting repeaters.

A somewhat informal definition of *buffering* is as follows:

Buffering

Input: A Steiner tree $G = (V, E)$ representing a single net, whose vertex set can be partioned as $V = \{source\} \dot\cup V_{sink} \dot\cup V_{rep}$, corresponding to the source gate of the net, the sink gates of the net and locations where a repeaters may be inserted, respectively, and whose edges correspond to wire segments. An arrival time $\mathrm{at}(source)$ for the source and required arrival time $\mathrm{rat}(s)$ for each sink $s \in V_{sink}$. A library $\mathcal{R} \cup \square$ of available repeaters ($\square$ indicates no repeater is to be inserted).

Output: A *buffering* of G, that is, a map $\beta : V_{rep} \to \mathcal{R} \cup \{\square\}$ indicating which (if any) repeaters should be inserted at each location maximizing the worst slack

$$\mathrm{slack}(\beta) := \min_{s \in V_{sink}} \mathrm{slack}_\beta(s),$$

where $\mathrm{slack}_\beta(s)$ is the slack at sink s after insertion of all buffers according to β with respect to the given (required) arrival times.

The buffering problem as stated above can be solved using a dynamic programming algorithm by Li and Shi [27] in time $\mathcal{O}(|V|^2 \cdot |\mathcal{R}|)$, improving on earlier results by Lillis [13] and Van Ginneken [11]. The basic idea of these algorithms is to define *candidate solutions* C consisting of a vertex $v(C) \in V$, a set $s(C) \subseteq V_{sink}$ and a partial buffering of the sub-tree of G rooted at $v(C)$ having leaves $s(C)$. To each such a candidate we can associate an input capacitance $\mathrm{cap}(C)$ and a worst slack $\mathrm{slack}(C)$. After an initial candidate is created at each sink, G is traversed in reverse topological order, creating new candidates by buffer insertion or the merging of two existing candidates. Pruning based on dominance checks keeps the total amount of created candidates low, by removing a candidate C if another candidate C' exists with $v(C) = v(C'), s(C) = s(C'), \mathrm{cap}(C) \geq \mathrm{cap}(C')$ and $\mathrm{slack}(C) \leq \mathrm{slack}(C')$.

If instead of maximizing worst slack we wish to minimize power consumption while meeting some bound on worst slack (for instance $\mathrm{slack}(\beta) \geq 0$),

the buffering problem is NP-hard [30]. However, if we allow the arrival time requirements to be violated by a factor of $(1+\epsilon)$, a $(1+\epsilon)$-approximate solution can be computed in time

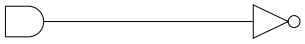
$$\mathcal{O}\left(\frac{|V_{sinks}|^2 \cdot |V|^2 \cdot |\mathcal{R}|^2}{\epsilon^3} + \frac{|V_{sinks}|^3 \cdot |\mathcal{R}|^2|}{\epsilon}\right)$$

using again a dynamic programming algorithm by Shi et al [30]. This algorithm was adapted by Romen [21] to approximately minimize a weighted sum of the power consumption and worst slack of β.

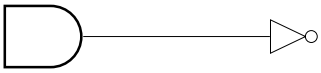
3.7 Gate Sizing and Buffering

The problems of buffering and sizing the gates of a chip are heavily codependent. Consider for example a net consisting of a source and a single sink connected by wire, with some initial sizing. If the size of the source is rather large, inserting a buffer along the connecting wire segment is unlikely to improve timing. If, on the other hand, the source is rather small, buffer insertion probably will lead to some improvement. The situation is illustrated in Figure 3.2. The issue is that traditionally, gate sizing algorithms do not account for the possibility of inserting additional buffers (and therefore tend to oversize the source gate in our example), while conversely buffering algorithms do not account for the possibility of resizing the source and sink gates. Rather, buffering is performed using some heuristically chosen intial gate sizes, after which gate sizing is run on the buffered chip. This order of operations is used as it allows inserted buffers to be sized.

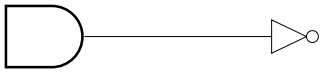
It seems that a single algorithm is needed that simultaneously sizes gates and buffers the nets. The biggest problem in finding such an algorithm is that gate sizing affects the chip *globally*, i.e. changing the size of a single gate impacts the timing of multiple nets. A simultaneous algorithm for gate sizing and buffering would therefore have to act on the entire chip at once. Since the the entire chip, as opposed to a single net, is not a tree, this means that the dynamic programming approaches mentioned in the previous section will not work. In fact, no efficient algorithms are known for buffering graphs that are not trees.



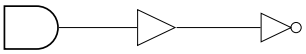
(a) The original net, sizing and buffering have yet to be applied.



(b) In order to meet timing, the first gate is greatly enlarged.



(c) As timing requirements are already met, no repeaters are inserted.



(d) An actual optimum solution, requiring both sizing and repeater insertion

Figure 3.2: Illustration of a situation where gate sizing and buffering are codependent and an optimum solution is not found by simply performing one after the other.

Much of the purpose of the rest of this work is to explore some strategies to unify buffering and gate sizing, both in theory (chapter 4) and in practice (chapter 5).

Chapter 4

Gate Sizing with Movement

The goal of this chapter is to explore an optimization problem we will call *gate sizing with movement*, which extends the gate sizing problem to also allow movement of buffers along wire segments. The idea is that this problem can be thought of as a unified instance of gate sizing and buffering, in which the (admittedly large) concession has been made that the total number of repeaters to be inserted on each wire segment has already been fixed.

As providing a precise model for gate sizing with movement on a general circuit is very involved, we will only consider the case in which the circuit is a chain of repeaters. In principle this is enough to deal with general circuits as well, as the delay between any two subsequent vertices of the timing graph can be modeled in terms of these repeater chains (as in the case of regular gate sizing). Even in this simple case the resulting problem is not straightforwardly convexifiable. In what follows, we will introduce repeater chains in detail, study some properties of their delay functions, and propose two methods for solving the gate sizing problem with movement on these special circuits.

4.1 Repeater Chains

A *repeater chain* consists of $n > 1$ repeaters $R_1, R_2, \ldots R_n$ with wire connections between repeaters R_i and R_{i+1} for all $i < n$. Each repeater has a base capacitance $\bar{\gamma}_i := \bar{\gamma}(R_i) > 0$ and resistance $\bar{\rho}_i := \bar{\rho}(\gamma_i) > 0$. For simplicity, we

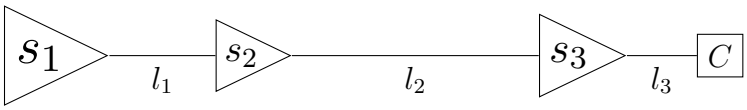


Figure 4.1: A repeater chain for $n = 3$. The locations of the first repeater and the capacitance C are fixed. The sizes s_1, s_2, s_3 and the distances l_1, l_2, l_3 are to be optimized.

will assume that base capacitance and resistance are equal for each repeater, i.e. there are constants $\bar{\gamma}_{rep}, \bar{\rho}_{rep} > 0$ such that $\bar{\gamma}_i = \bar{\gamma}_{rep}$ and $\bar{\rho}_i = \bar{\rho}_{rep}$ for all $i \in [n]$. The wire used has a capacity per length $\bar{\gamma}_{wire} \geq 0$ and resistance $\bar{\rho}_{wire} \geq 0$. The chain is required to drive a fixed capacity $C > 0$. Usually, the size of the first repeater will be fixed to some value $s_1 = \eta > 0$.

We will concern ourselves with optimization problems in which the size s_i of each repeater R_i, the length l_i of the wire segment between repeaters R_i and R_{i+1} and the length l_n between the final repeater and the driven capacity must be chosen to either minimize delay, or minimize power usage while satisfying some bound on total delay. As the location of the first repeater and the driven capacity are fixed, the sum of all wire segment lengths must be equal to some constant L. That is, $\sum_{i=1}^{n} l_i = L$. We will consider both cases in which s_i, l_i are chosen from continuous or discrete domains. Furthermore, we will consider the case where either s or l is fixed. The latter case yields an instance of the regular gate sizing problem. Figure 4.1 illustrates the situation for $n = 3$.

4.1.1 Properties of the Delay Function

It is useful to consider the delay function arising from the Elmore delay model for repeater chains in some detail. We write $\gamma_i := \bar{\gamma}_{rep} \cdot s_i$, $\rho_i := \bar{\rho}_{rep} \cdot s_i$ for the effective capacitances and resistances of each repeater, and $s_{n+1} := C/\gamma_{rep}$, $\gamma_{n+1} := C$ for simplicity of notation (note that s_{n+1} is never treated as a variable). Let $s = (s_1, s_2, \ldots s_n)$, $l = (l_1, l_2, \ldots l_n)$ be given. Then the total delay $d(s, l)$ along the chain for sizes s and lengths l is given by

$$d(s, l) = \sum_{i=1}^{n} d_i(s, l), \tag{4.1.1}$$

where

$$\begin{aligned} d_i(s,l) &= \frac{\gamma_{rep} \cdot s_{i+1} + \gamma_{wire} \cdot l_i}{\rho_{rep} \cdot s_i} + \rho_{wire} \cdot l_i(\frac{1}{2}\gamma_{wire} \cdot l_i + \gamma_{rep} \cdot s_{i+1}) \\ &= \frac{\gamma_{i+1} + \gamma_{wire} \cdot l_i}{\rho_i} + \rho_{wire} \cdot l_i(\frac{1}{2}\gamma_{wire} \cdot l_i + \gamma_{i+1}). \end{aligned} \tag{4.1.2}$$

In the special case that $\gamma_{wire} = \rho_{wire} = 0$, we get the delay functions

$$\hat{d}_i(s) := \frac{\gamma_{rep}}{\rho_{rep}} \cdot \frac{s_{i+1}}{s_i} = \frac{\gamma_{i+1}}{\rho_i} \tag{4.1.3}$$

and

$$\hat{d}(s) := \sum_{i=1}^{n} \hat{d}_i(s,l) = \frac{\gamma_{rep}}{\rho_{rep}} \cdot \sum_{i=1}^{n} \frac{s_{i+1}}{s_i}, \tag{4.1.4}$$

which no longer depend on the choice of l. Clearly, $d_i(s,l) \geq \hat{d}_i(s)$ for any choice of s, l and i. Previous work on repeater chains (e.g. by Boyd [26] and Daboul [8]) usually deals with this case.

For each $a \in [n]$, we shall write $d_{>a}(s,l) := \sum_{i=a+1}^{n} d_i(s,l)$ for the delay after passing repeater R_a.

Lemma 4.1. *For each $a \in [n]$, $d(s,l)$ depends only on $d_{>a}(s,l)$, $s_1, \ldots, s_{a+1}$ and $l_1, \ldots, l_a$.*

Proof. We have

$$d(s,l) = \sum_{i=1}^{a} d_i(s,l) + \sum_{i=a}^{n} d_i(s,l) = \sum_{i=1}^{a} d_i(s,l) + d_{>a}(s,l).$$

From equation (4.1.2) we see that $d_i(s,l)$ depends only on s_i, s_{i+1}, l_i and so we are done. □

Note that the exact same also holds for $\hat{d}_{>a}(s,l) := \sum_{i=a+1}^{n} \hat{d}_i(s,l)$.

4.1.2 Structural Results

One special property of $\hat{d}$ is that we may assume monotonicity of its input when minimizing.

Theorem 4.2 (Daboul [8])**.** *Let $s = (s_1, s_2, \ldots, s_n) \in \mathbb{R}^n_{>0}$ be given. Then there exists a map $\pi : [n] \to [n]$ such that*

$$\hat{d}(s) \geq \hat{d}(\pi(s))$$

and $s_{\pi(1)} \leq s_{\pi(2)} \leq \cdots \leq s_{\pi(n)}$ or $s_{\pi(1)} \geq s_{\pi(2)} \geq \cdots \geq s_{\pi(n)}$. Furthermore, π can be chosen such that $\pi(1) = 1$.

Daboul notes that the above theorem yields a polynomial algorithm to minimize $\hat{d}$ if the set of possible sizes for each repeater is the same and of constant size (as there are only polynomially many *monotone* solutions in this case). The goal now is to find similar structural results for d_l in the case that l is an equidistant spacing.

Daboul's proof of theorem 4.2 is based on the so-called *rearrangement inequalities*.

Theorem 4.3 (Rearrangement inequalities)**.** *Let $x_1 \leq x_2 \leq \ldots \leq x_n$ and $y_1 \leq y_2 \leq \ldots \leq y_n$ be arbitrary reals. Let $\pi : [n] \to [n]$ be a permutation. Then*

$$x_1 y_n + \ldots + x_n y_1 \leq x_1 y_{\pi(1)} + \ldots + x_n y_{\pi(n)} \leq x_1 y_1 + \ldots + x_n y_n.$$

Proposition 4.4. *Let $s_1, s_2, \ldots, s_n$ be given such that $s_1 \leq s_2 \leq \ldots \leq s_n$ or $s_1 \geq s_2 \geq \ldots \geq s_n$. Let $\pi : [n] \to [n]$ be a permutation such that $\pi(1) = 1$ and $\pi(n) = n$. Then*

$$\sum_{i=1}^{n-1} \frac{s_{i+1}}{s_i} \leq \sum_{i=1}^{n-1} \frac{s_{\pi(i+1)}}{s_{\pi(i)}}.$$

Proof. Assume first that $s_1 \leq s_2 \leq \ldots \leq s_n$. Define $x_i := s_{i+1}$ and $y_i := \frac{1}{s_{n-i}}$ for $1 \leq i < n$ so that $x_1 \leq x_2 \leq \ldots \leq x_{n-1}$ and $y_1 \leq y_2 \leq \ldots \leq y_{n-1}$. Let

$\pi : [n] \to [n]$ be a permutation with $\pi(1) = 1$ and $\pi(n) = n$. Then there is some permutation $\tau : [n-1] \to [n-1]$ such that

$$\sum_{i=1}^{n-1} \frac{s_{\pi(i+1)}}{s_{\pi(i)}} = \sum_{i=1}^{n-1} x_i y_{\tau(i)} \geq \sum_{i=1}^{n-1} x_i y_{n-i} = \sum_{i=1}^{n-1} \frac{s_{i+1}}{s_i}$$

by Theorem 4.3.

If instead $s_1 \geq s_2 \geq \ldots \geq s_n$ we define $x_i := s_{n-i+1}$ and $y_i = \frac{1}{s_i}$ for $1 \leq i < n$ so that again $x_1 \leq x_2 \leq \ldots \leq x_{n-1}$ and $y_1 \leq y_2 \leq \ldots \leq y_{n-1}$. The proof continues analogously to the previous case. □

Proposition 4.5. *Let $s_1, s_2, \ldots, s_n$ be given. Let $\pi : [n] \to [n]$ be a permutation such that $\pi(1) = 1$ and $\pi(n) = n$ and*

$$f(\pi) := \sum_{i=1}^{n-1} \frac{s_{\pi(i+1)}}{s_{\pi(i)}}$$

is minimized amongst all such permutations. Then we may assume that the sequence $s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}$ can be split up into at most three monotone subsequences.

Proof. Let $z_i = s_{\pi(i)}$ for all $i \in [n]$. Let $a, b \in [n]$ such that $z_a \geq z_i$ and $z_b \leq z_i$ for all $i \in [n]$, respectively. Assume that $a < b$, the proof for $b > a$ is analogous. Using Proposition 4.4 we may assume that $z_a, z_{a+1}, \ldots, z_b$ is a monotone sequence and so it remains to show that $z_1, z_2, \ldots, z_a$ and $z_b, z_{b+1}, \ldots, z_n$ can also be assumed to be monotone.

Suppose the former is not. If $z_1 \leq z_i \leq z_a$ for all $1 \leq i \leq a$ we may apply Proposition 4.4 directly. If not, then there is some $1 < j < a$ such that $z_{j-1} > z_j \geq z_{j+1}$. Let $a \leq k \leq b$ such that $z_k \geq z_j \geq z_{k+1}$. Consider the sequence $z' : z_1, z_2, \ldots, z_{j-1}, \hat{z_j}, z_{j+1}, \ldots, z_k, z_j, z_{k+1}, \ldots, z_n$. As

$$\frac{z_j}{z_{j-1}} + \frac{z_{j+1}}{z_j} - \frac{z_{j+1}}{z_{j-1}} \geq 1$$

and

$$\frac{z_j}{z_k} + \frac{z_{k+1}}{z_j} - \frac{z_{k+1}}{z_k} = \frac{z_j^2 + z_{k+1} \cdot (z_k - z_j)}{z_k \cdot z_j} \leq 1$$

we have

$$\sum_{i=1}^{n-1} \frac{z'_{i+1}}{z'_i} \leq \sum_{i=1}^{n-1} \frac{z_{i+1}}{z_i}.$$

It follows that we may assume $z_1 \leq z_i \leq z_a$ and so we may assume $z_1, z_2, \ldots, z_a$ is monotone, again using Proposition 4.4. By the same reasoning $z_b, z_{b+1}, \ldots, z_n$ may be assumed to be monotone.

□

Corollary 4.6. *Let $s = (s_1, s_2, \ldots, s_n) \in \mathbb{R}^n_{>0}$ be given. Then there exists a permutation $\pi : [n] \to [n]$ such that*

$$\hat{d}(s) \geq \hat{d}(\pi(s))$$

and $s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}$ can be split up into at most three monotone subsequences. Furthermore, π can be chosen such that $\pi(1) = 1$.

Proof. This follows directly from proposition 4.5 (setting $s_{n+1} = C$, the load capacitance). □

Theorem 4.7. *Let $s = (s_1, s_2, \ldots, s_n) \in \mathbb{R}^n_{>0}$ be given and let $l = (\lambda, \lambda, \ldots, \lambda)$ be an equidistant spacing. Then there exists a permutation $\pi : [n] \to [n]$ such that*

$$d(s, l) \geq d(\pi(s), l)$$

and $s_{\pi(1)}, s_{\pi(2)}, \ldots, s_{\pi(n)}$ can be split up into at most three monotone subsequences. Furthermore, π can be chosen such that $\pi(1) = 1$.

Proof. Using corollary 4.6 we may choose a permutation $\pi : [n] \to [n]$ such that $\hat{d}(s) \geq \hat{d}(\pi(s))$. For ease of notation we extend $\pi(n+1) := n+1$. We

have

$$\begin{aligned} d(s,l) &= \hat{d}(s) + \sum_{i=1}^{n} \big(\frac{\gamma_{wire}}{\rho_{rep}} \frac{l_i}{s_i} + \rho_{wire}\gamma_{rep} \cdot l_i \cdot s_{i+1} \big) \\ &= \hat{d}(s) + \sum_{i=1}^{n} \big(\frac{\gamma_{wire}}{\rho_{rep}} \frac{\lambda}{s_i} + \rho_{wire}\gamma_{rep} \cdot \lambda \cdot s_{i+1} \big) \\ &= \hat{d}(s) + \sum_{i=1}^{n} \big(\frac{\gamma_{wire}}{\rho_{rep}} \frac{\lambda}{s_{\pi(i)}} + \rho_{wire}\gamma_{rep} \cdot \lambda \cdot s_{\pi(i+1)} \big) \\ &= d(\pi(s), l) \end{aligned}$$

making use of the fact that $\pi(1) = 1$. □

Note that like in the case of $\hat{d}$, this theorem provides a polynomial algorithm for minimizing $d(s,l)$ in the case that l is equidistant (as the amount of solutions consisting of at most three monotone subsequences is also polynomial).

As the following example shows, Theorem 4.7 is in some sense the best possible result, even if we allow π to be any map (instead of just a permutation).

Example 4.1. Consider a chain of $n = 3$ repeaters, driving a load $C = 1$ with constants $\gamma_{rep} = \rho_{rep} = \gamma_{wire} = 1, \rho_{wire} = 10$, and fixed l given by $l_1 = l_2 = l_3 = 1$. Consider the solution $s = (5, 1, 5)$. The only alternative solution in the sense of Theorem 4.2 we have to consider is $s' = (5, 5, 5)$. Indeed,

$$\hat{d}(s) - \hat{d}(s') = \frac{s_2}{s_1} + \frac{s_3}{s_2} - \big(\frac{s'_2}{s_1} + \frac{s_3}{s'_2} \big) = \frac{1}{5} + 5 - (1 + 1) \geq 0.$$

However,

$$d_l(s') - d_l(s) \geq 10(s'_1 - s_1) + \frac{s_3 + 1}{s_2} - \frac{s'_3 + 1}{s'_2} = 40 + 6 - \frac{6}{5} \geq 0.$$

Unfortunately, the theorem we have just proven for equidistant l do not hold for general l, and no useful generalizations are known.

4.1.3 Global Minimum of the Delay Function

We first give a (rather weak) lower bound for the delay function (4.1.1) in terms of the fraction between the largest and the smallest occuring repeater size.

Lemma 4.8. *Let $x_1, x_2, \ldots, x_m \in \mathbb{R}_{>0}$, and set $\alpha = x_m/x_1$. Then*

$$\max_{1 \le i < m} \frac{x_{i+1}}{x_i} \ge \alpha^{1/(m-1)}.$$

Proof. Suppose not. Then

$$x_m = x_1 \cdot \prod_{1 \le i < m} \frac{x_{i+1}}{x_i} < x_1 \cdot (\alpha^{1/(m-1)})^{(m-1)} = x_1 \cdot \alpha = x_m,$$

a contradiction. □

Proposition 4.9. *For any s, l, let $\alpha := \max_{1 \le j < i \le n+1} \cdot \frac{s_i}{s_j}$. There is a constant $c > 0$ depending only on γ_{rep}, ρ_{rep} such that*

$$d(s, l) \ge c \cdot \min\{\alpha, \alpha^{1/n}\}.$$

Proof. Let $1 \le a < b \le n+1$ such that $\alpha = s_a/s_b$ and write $m = b - a$. By Lemma 4.8 there exists an $a \le i < b$ such that $s_{i+1}/s_i \ge \alpha^{1/(m-1)}$. So

$$d(s, l) \ge \hat{d}_i(s) = \frac{\gamma_{rep}}{\rho_{rep}} \cdot \frac{s_{i+1}}{s_i} \ge \frac{\gamma_{rep}}{\rho_{rep}} \cdot \alpha^{1/(m-1)}.$$

We conclude the proof by noting that $\alpha \le \alpha^{1/(m-1)}$ if $\alpha < 1$, and $\alpha^{1/n} \le \alpha^{1/(m-1)}$ otherwise. □

Using this lower bound, we can prove that the delay function d attains a global minimum for any fixed $l \in \mathbb{R}^n_{>0}$ and $s_1 = \eta > 0$.

Corollary 4.10. *For fixed $l \in \mathbb{R}^n_{>0}, s_1 > 0$, the function*

$$d_l(s_2, s_3, \ldots, s_n) := d(s_1, s_2, s_3, \ldots, s_n, l)$$

has a stationary point on $\mathbb{R}^{n-1}_{>0}$ which is a global minimum. Furthermore, all of its stationary points are global minima.

Proof. As d_l is a posynomial, it is convexifiable. Since its domain is open, we only have to show existence of a global minimum. We can lower bound $\alpha := \max_{1 \leq j < i \leq n+1} s_i/s_j$ by

$$\alpha \geq \max_{2 \leq i \leq n} s_i/s_1 \text{ and } \alpha \geq s_{n+1}/\min_{2 \leq i \leq n} s_i.$$

Writing $s = (s_2, s_3, \ldots, s_n)$, it follows from Proposition 4.9 that $\lim_{||s|| \to 0} d_l(s) = \lim_{||s|| \to \infty} d_l(s) = \infty$. So we may restrict d_l to some appropriate compact subset of $\mathbb{R}^{n-1}_{>0}$, on which it attains a global minimum as it is continuous. □

It is shown by Boyd [26] that the global minimum of $\hat{d}(s)$ can be computed analytically for fixed $s_1 = \eta$ by setting $\partial \hat{d}(s)/\partial s_i$ to zero for each i. This results in equations

$$\frac{1}{s_{i-1}} = c \cdot \frac{s_{i-1}}{s_i^2}$$

for all $i < n$ and

$$\frac{1}{s_n} = \frac{c}{s_{n+1}^2}$$

for some constant c. These form a well-known recursion with solution

$$s_i = c \cdot (s_{n+1}/\eta)^{\frac{i-1}{n+1}}.$$

Attempts to do the same thing for $d(s, l)$ unfortunately fail as the resulting recursion relations on the s_i and l_i do not have a known analytical solution.

4.1.4 The Discrete Spacing Problem

Before considering methods for solving the continuous gate sizing problem with movement on repeater chains, we introduce the discrete variant of optimizing $d(s, l)$ for fixed s and show that it can be solved in polynomial time.

Discrete Repeater Chain Spacing Problem

Input: A directed path G having n nodes $v_1, v_2, \ldots, v_n$ and arcs $e_i := (v_i, v_{i+1})$ for all $i \in [n-1]$ with lengths $l(e_i) > 0$. A base wire resistance ρ_{wire} and capacitance γ_{wire}. A number $k < n$ and repeaters $r_1, r_2, \ldots r_k$ with sizes $s_1, s_2, \ldots s_n$, base resistance $\rho_{rep} > 0$ and base capacitance $\gamma_{rep} > 0$. A load $C \geq 0$ to be driven be the chain.

Output: A strictly increasing map $p : [k] \to [n-1]$ with $p(1) = 1$ indicating a placement of the repeaters minimizing the total delay along the path:

$$d(p) := d(s, l(p)),$$

where $l(p)_i := \sum_{j=p(i)}^{p(i+1)-1} l(e_j)$, the length of the path between $v_{p(i)}$ and $v_{p(i+1)}$ for $i < k$ and $l(p)_k := \sum_{j=p(k)}^{n} l(e_j)$.

Consider an instance $\mathcal{I} = (G, l, k, \rho, \gamma)$ of the DRCSP. For each $a \in [k]$ and $b \in [n]$ with $b - a \geq n - k$, a new instance $\mathcal{I}_a^b$ of the DRCSP can be constructed from $\mathcal{I}$ by removing the nodes $v_1, v_2, \ldots v_{b-1}$ from G, only considering the repeaters $r_a, r_{a+1}, \ldots r_k$ and appropriately reindexing. If $p : [k] \to [n-1]$ is a feasible solution to $\mathcal{I}$, a feasible solution to $\mathcal{I}_a^{p(a)}$ is given by the function $p_a : [k - a + 1] \to [n - p(a) + 1]$ defined by $p_a(i) = p(i + a - 1)$.

Lemma 4.11. *Let $\mathcal{I}, p$ as above, and let $a \in [k]$. If p is an optimum solution to $\mathcal{I}$, then p_a is an optimum solution to $\mathcal{I}_a^{p(a)}$.*

Proof. This follows directly from Lemma 4.1. □

We define a *label* to be a triple $l = (\text{depth}, \text{node}, \text{pred})$, where $\text{depth} \in [k], \text{node} \in [n]$ and pred is either $\emptyset$ or another label l' with $\text{depth}(l') = \text{depth}(l) + 1$ and $\text{node}(l') \geq \text{node}(l)$. A label l corresponds to a feasible solution p of $\mathcal{I}_{\text{depth}(l)}^{\text{node}(l)}$ by setting $p(1) = \text{node}(l), p(2) = \text{node}(\text{pred}(l)), \ldots$ etc. We shall write $\text{delay}(l)$ for the delay of this feasible solution. Using Lemma 4.11 we may compute a label $l_{i,j}$ corresponding to an optimium solution to the instance $\mathcal{I}_j^i$ for all $0 \leq j \leq k-1$ and $i \in [n]$ using dynamic programming.

Theorem 4.12. *An optimum solution to the DRCSP can be found in in $\mathcal{O}(n^2k)$ time.*

Proof. Define the labels $l_{k-1,i} = (k-1, i, \emptyset)$ for $i = 1, \ldots, n$ and

$$l_{j,i} = \arg\min\{\text{delay}(l) : l \in \{(j, i, l_{j+1,m}) : n \geq m > i\}\}$$

for $i = 1, \ldots, n$ and $j = 0, \ldots, k-2$. Note that for for each $i \in [n]$ the label $l_{k-1,i}$ corresponds to an optimimum solution to $\mathcal{I}^i_{k-1}$, as this instance has only a single feasible solution. It follows from Lemma 4.11 that each label $l_{j,i}$ corresponds to an optimum solution of $\mathcal{I}^i_j$. As a result, $l_{0,1}$ corresponds to an optimum solution of $\mathcal{I}^1_0 = \mathcal{I}$. For the running time, note that at most n labels need to be compared to compute a new label $l_{j,i}$, and each comparison can be performed in constant time (by caching the delays of previous labels). So the total running time is $\mathcal{O}(n + nkn) = \mathcal{O}(n^2k)$. □

4.1.5 Solving the Continuous Problem

Our goal is to solve the program

$$\begin{aligned} \min \quad & d(s, l), \\ s.t. \quad & \sum_i l_i = L, \\ & s \in [u, v], \\ & l \geq 0, \end{aligned} \tag{4.1.5}$$

for given constants $L \in \mathbb{R}_{\geq 0}$ and $u, v \in \mathbb{R}^n_{\geq 0}$.

Clearly, 4.1.5 is not convex as stated. As it is *almost* of geometric form, it makes sense to attempt a variable substitution $s_i = e^{\sigma_i}$, $l_i = e^{\lambda_i}$ for σ_i, λ_i suitably bounded (as in the case of regular gate sizing). Since $d(s, l)$ is a posynomial, $d(e^\sigma, e^\lambda)$ would indeed be convex. The problem is that the constraint $\sum_i l_i = L$ is transformed into $\sum_i e^{\lambda_i} = L$, which is nonconvex (as $\sum_i e^{\lambda_i}$ is not affine).

Instead, one could try to only substitute $s_i = e^{\sigma_i}$, leaving the l_i as they are. Unfortunately, the resulting function $d(e^\sigma, l)$ is nonconvex in general. To show this is the case, we consider a simple example.

Example 4.2. Let $n = 3$. Choosing constants for resistance and capacitance carefully, the delay function $d = d(s, l)$ can be written as $d = d_1 + d_2 + d_3 + d_4$, where

$$d_1 = \frac{s_2 + l_1}{s_1}, \quad d_2 = l_1(l_1 + s_2), \quad d_3 = \frac{s_3 + l_2}{s_2}, \quad d_4 = l_2(l_2 + s_3).$$

Now it is enough to find $x^1 = (s^1, l^1)$, $x^2 = (s^2, l^2)$ such that

$$d(s_1^3, s_2^3, s_3^3, l_1^3, l_2^3) > \frac{1}{2}d(s_1^1, s_2^1, s_3^1, l_1^1, l_2^1) + \frac{1}{2}d(s_1^2, s_2^2, s_3^2, l_1^2, l_2^2)$$

where $s_i^3 = \sqrt{s_i^1 s_i^2}$ (the geometric mean of s_i^2 and s_i^2) and $l_i^3 = \frac{1}{2}(l_i^1 + l_i^2)$ (the arithmetic mean of l_i^1 and l_i^2). Consider the x^1, x^2, x^3 given in the table below:

	s_1	s_2	s_3	l_1	l_2
x^1	10	5	10	0.9	0.1
x^2	6	7	6	0.2	0.8
x^3	7.75	5.92	7.75	0.55	0.45

Table 4.1: Values of three feasible solutions x^1, x^2, x^3 to the delay minimization problem of example 4.2. The values for x^3 are derived from those of x^1 and x^2 by taking the geometric mean of the s_i and the arithmetic mean of the l_i. They have been rounded to two decimals for readability.

Computing the delays using these values yields:

$$d(x^1) = \frac{5 + 0.9}{10} + 0.9(0.9 + 5) + \frac{10 + 0.1}{5} + 0.1(0.1 + 10) = 8.93$$
$$d(x^2) = \frac{7 + 0.2}{6} + 0.2(0.2 + 7) + \frac{6 + 0.8}{7} + 0.8(0.8 + 6) = \frac{1584}{175} \simeq 9.05$$
$$d(x^3) \simeq \frac{5.92 + 0.55}{7.75} + 0.55(0.55 + 5.92) + \frac{7.75 + 0.45}{5.92} + 0.45(0.45 + 7.75) \simeq 9.47$$

As $d(x^3) \geq \max\{d(x^1), d(x^2)\}$ the delay function is not convex (in fact, not even quasiconvex).

As a result, we cannot hope to solve 4.1.5 using convex programming directly. In this section we will outline two potential methods for finding a

local solution and analyze their practical performance. The first method is straightfoward. As 4.1.5 can be easily brought into signomial form, a local solution may be found by applying an algorithm for signomial programming (see section 2.4).

For the second method we make use of the fact that $d(s, l)$ can be minimized efficiently after either fixing s or l. Indeed, after fixing $l \geq 0$, $d(s, l)$ is a posynomial function and so the program

$$\begin{aligned} \min \quad & d_l(s) \\ s.t. \quad & s_i \in [u, v] \quad i = 1, 2, \ldots, n \end{aligned} \tag{4.1.6}$$

is geometric and can be solved efficiently (see section 2.4). If we fix the sizes s instead, we can write

$$d_i(s, l) = a_i l_i + b l_i^2 + c_i$$

for all $i \in [n]$, where $a_i = \gamma_{wire}/(\rho_{prep} \cdot s_i) + \rho_{wire} \cdot \gamma_{rep} \cdot s_{i+1}$, $b = \frac{1}{2}\rho_{wire} \cdot \gamma_{wire}$ and c_i is some constant. Consider the quadratic program

$$\begin{aligned} \min \quad & a^\top l + l^\top B l \\ s.t. \quad & \sum_{i=1}^{n} l_i = 1 \\ & l_i \geq 0 \quad i = 1, 2, \ldots, n, \end{aligned} \tag{4.1.7}$$

where $a = (a_i) \geq 0$ and $B = \mathrm{diag}(b) \succeq 0$. As B is positive definite, this program is convex (see section 2.5). Now, we can find a local solution to 4.1.5 by applying ACS (see section 2.3) to the programs 4.1.6 and 4.1.7. As it is trivial to find an initial feasible solution to either 4.1.6 and 4.1.7, we can opt to start the ACS with either of them. As the constraints on the l variables are usually more strict than those on the s variables, we choose to always solve 4.1.6 first, choosing as an initial solution for l either an equidistant spacing or a vector chosen uniformly at random having $\sum l = L$.

Practical performance

We will now attempt to get a feeling for the (relative) practical performance of the two methods for finding a local solution to 4.1.5 described above. For

this purpose, we generate random instances of 4.1.5 having different values of $n, \gamma_{wire}, \rho_{wire}$, C and L. We will then use the python package GPkit [4] to solve the signomial program 4.1.5 and the geomteric program 4.1.6. GPkit uses CVXOPT [19] as its solver for convex optimization, which we will use directly to solve the quadratic program 4.1.7. It should be noted that while both the signomial method and the ACS find a local solution to 4.1.5, they need not neccessarily find the *same* local solution. However, no instance was found in which this actually occured.

In the first experiment we seek to compare the performance of the two methods directly. For this purpose we pick $\gamma_{wire}, \rho_{wire}$ uniformly at random from $[1, 5]$, C from $[1, 10]$ and set $L = 2n$. We find a solution to the generated instance using the signomial solver, and then run ACS until a solution whose value is at most 0.1% greater is found (using a random initial solution for l). Figure 4.2 shows the average runtime of both methods over 100 generated instances for several values of n.

As can be seen from Figure 4.2 the signomial solver is significantly slower than the ACS. An explanation for this might be that solving the quadratic program 4.1.7 is particularly inexpensive. In contrast to ACS, the signomial solver cannot make use of the special structure of 4.1.5 and so is required to solve more (expensive) geometric programs.

In our second experiment, we are interested in the amount of iterations needed in the ACS to reach a local minimum. For this purpose we again pick $\gamma_{wire}, \rho_{wire}$ uniformly at random from $[1, 5]$ and C from $[1, 10]$. We then run ACS until the solution value in a given iteration improves by less than 0.1% for different values of n and L, choosing either a random or equidistant initial solution for l. Figure 4.3 shows the average amount of iterations required over 100 generated instances.

We make three observations from Figure 4.3. Firstly, it appears that the required amount of iterations does not grow as a function of n. Secondly, it seems that using an equidistant spacing as initial values for l lowers the required amount of iterations compared to choosing the initial values at random. Lastly, fewer iterations are required for smaller values values of L. This last observation may be explained by the fact that the impact of the wire on the total delay is smaller in these cases.

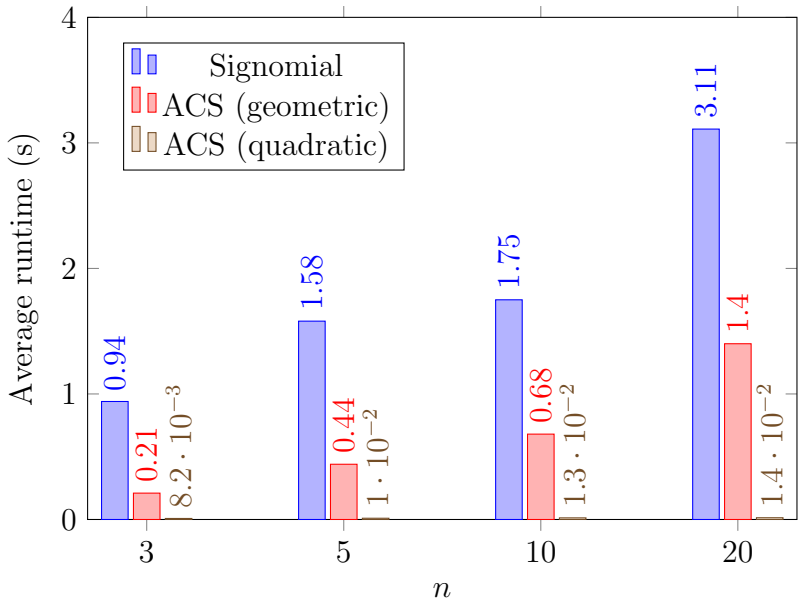


Figure 4.2: Average runtime for randomly generated instances of 4.1.5 for both the signomial solver and the ACS based algorithm for different values of n. Runtime for ACS is split into time spent solving 4.1.6 (geometric) and 4.1.7 (quadratic).

It is interesting to analyze the improvement made in each iteration of the ACS in some more detail. For this purpose Figure 4.4 shows the average percentual improvement in each iteration (with respect to the previous iteration) for runs in the previous experiment for $n = 50$, $L = n$ and equidistant initial l.

It should be noted that the rate of improvement diminishes rather quickly as more iterations are performed.

In conclusion, it appears that ACS significantly outperforms the signomial solver. Furthermore, the amount of iterations needed by the ACS does not seem to scale in n, and is rather stable under different choices of parameters. The runtime of ACS is therefore primarily dependent on the time required to solve a single instance of 4.1.6, which is a regular gate sizing instance. This is the case particularly when we bound the number of ACS iterations by some constant value. As most improvement appears to be made in the first few iterations anyways, this might be a feasible strategy for larger instances, where performing many iterations is not possible.

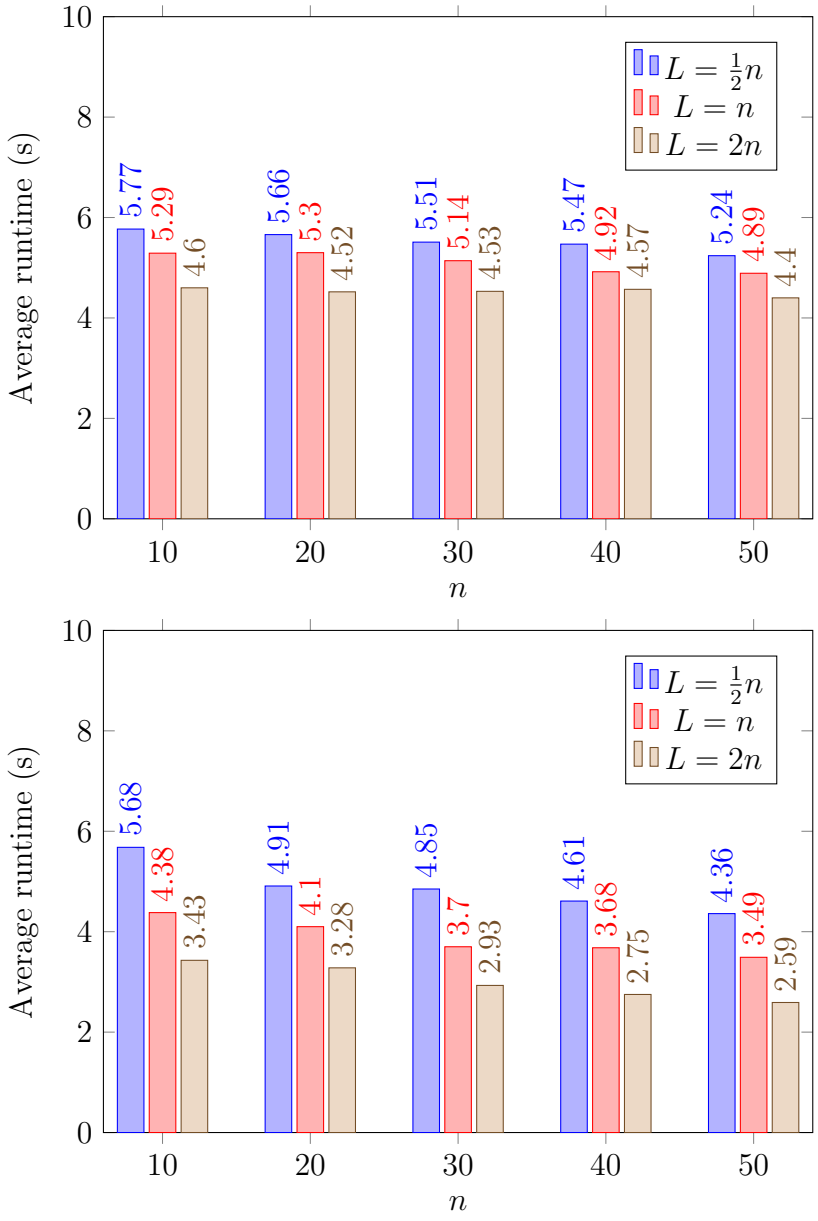


Figure 4.3: Average number of iterations until a local minimum is reached for randomly generated instances of 4.1.5 for different values of n and L. Results shown for both equidistant (below) and random (above) initial l.

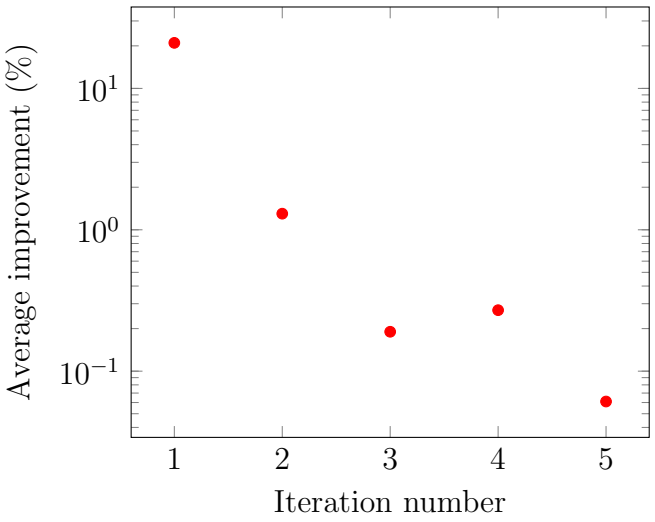


Figure 4.4: Average percentual improvement of the objective function per iteration during ACS, using equidistant inital l, for $n = 50$ and $L = n$.

Chapter 5

BonnRouteBuffer

In this chapter we discuss some additions made to BonnRouteBuffer, part of the BonnTools [3] software suite for automated chip design developed at the Institute for Discrete Mathematics in Bonn. We then run the altered version of BonnRouteBuffer on some circuits provided by IBM and discuss the results.

5.1 Buffering and Gate Sizing

Buffering and gate sizing are both performed within BonnRouteBuffer. In the current flow, buffering is performed first using heuristically chosen initial gate sizes which we consider to be part of the input. The buffering algorithm uses resource sharing to balance power consumption and total negative slack (among other criteria such as congestion etc. which we will not discuss here). Within each phase a dynamic programming algorithm by Rotter [22] finds a near optimum buffering with respect to the current resource weights, using ideas similar to those described in section 3.6. After a fixed number of phases a solution is selected using randomized rounding.

After buffering is finished, gate sizing is performed. Here the discrete version of the algorithm by Daboul et. al [25], which also uses resource sharing, is used (see section 3.5). As discussed earlier, even if we assume that these algorithms produce an optimum solution for buffering and gate sizing separately, the interdependency of these problems makes it so that the

combined solution need not be optimal at all.

5.1.1 Alternate Search

Inspired by the ACS algorithm for biconvex programs (section 2.3), one way of potentially improving the current BonnRouteBuffer flow is to simply alternate buffering and gate sizing several times. The first iteration would consist of the current flow. In the second iteration, the gate sizes produced in the first one are used for buffering. In the third iteration the sizes from the second iteration are used, and so forth. It should be noted that this approach will not neccessarily result in an optimum solution (allowing arbitrarily many iterations) even for simple circuits, as the following example shows.

Example 5.1. Consider a circuit consisting of two gates g_1, g_3 connected by a single segment of wire, on which a single buffer g_2 may (or may not) be inserted. Let s_1, s_2, s_3 be the sizes of g_1, g_2, g_3 respectively, and assume that $s_3 = 1$ is fixed. Assuming the right constants, the total delay d between g_1 and g_3 is given by

$$d = \begin{cases} \frac{s_3}{s_1} + c^2 = \frac{1}{s_1} + c^2 & \text{if the buffer is not inserted,} \\ \frac{s_2}{s_1} + \frac{s_3}{s_2} + \frac{1}{2}c^2 = \frac{s_2}{s_1} + \frac{1}{s_2} + \frac{1}{2}c^2 & \text{if the buffer is inserted,} \end{cases}$$

for some constant $c > 0$ and the total power consumption p is given by

$$p = \begin{cases} s_1 + 1 & \text{if the buffer is not inserted,} \\ s_1 + s_2 + 1 & \text{if the buffer is inserted.} \end{cases}$$

The constant terms c^2 and $\frac{1}{2}c^2 = (\frac{1}{2}c)^2 + (\frac{1}{2}c)^2$ in the delay function represent the quadratic effect of wire length on delay, which is reduced by inserting a buffer (see section 3.5). Now assume that $c = 2$ and we wish to minimize the power consumption while guaranteeing a delay of at most $\frac{13}{3}$. Assuming the initial value of s_1 is set as $s_1 = 3$, timing is met already, and so the buffering algorithm will not insert a buffer (as this will only increase power consumption). Similarly, the gate sizing algorithm will not change the size s_1 of the first gate, as any decrease would mean timing is no longer met, and any increase would only increase power consumption.

However, inserting a buffer and then setting $s_1 = s_2 = 1$ would decrease power consumption by 1, while simultaneously decreasing delay to $1+1+2 = 4 < \frac{13}{3} = \frac{1}{3} + 4$.

5.1.2 Driver Sizing

A second approach to improving the current flow is *driver sizing*. The idea is to update gate sizes within the phases of the buffering algorithm by making use of local refine (see section 3.5). Whenever a net is buffered we consider all candidate solutions that make it to the source (driver). For each of these candidates, we select the optimum driver size, considering power consumption and timing, both within the current net and in other nets (but keeping all other gate sizes fixed). Of course, the timing of other nets is dependent on their buffering, which may not have been computed yet for the current phase. To overcome this difficulty, we make use of the buffering found in the *previous* phase to compute timing in other nets. After the optimum driver size for each candidate has been selected, we recompute their resource usage (now using their optimum driver size). At the end of each phase, the size of each driver is updated to the optimum driver size for the best candidate of the corresponding net.

5.2 Practical Results

In this section we will present some practical results of the potential improvements to the BonnRouteBuffer flow described above. Unfortunately, results for driver sizing are not available at the time of writing, and so we will restrict ourselves to the results for alternate search.

We run BonnRouteBuffer on several chips from industry using two modes. In the fist one (called *full*) we run five iterations of buffering and gate sizing. In each iteration 25 phases of resource sharing are performed. In the second one (called *reduced*), we again run five iterations, now performing 25 phases of resource sharing in the first and final iteration, and only 5 in the other three. We measure worst slack (**ws**), sum of negative slacks (**fom**) and total power consumption (**pow**) after each iteration. The results can be found

in Table 5.1. It should be noted that the runtime of BonnRouteBuffer is approximately linear in the total amount of resource sharing phases run. As a result, runs in the full mode require roughly five times the normal runtime, whereas runs in the reduced require roughly three times normal runtime.

In addition to the full table of results, Figure 5.1 depicts the sum of negative slacks after each iterations both in full and reduced mode for all chips.

Conclusion

In general, it seems that alternate search succesfully reduces worst slack and sum of negative slacks (fom). Both in full and reduced mode their values are significantly reduced after the 5th iteration. This improvement does however come at the cost of increased power consumption, especially in reduced mode. It seems unclear whether or not enabling alternate search (in either mode) is practically benefical, especially considering the vastly increased runtime.

Looking at the fom development over time (see Figure 5.1) we notice two trends. Firstly, in the case of reduced mode, fom usually does not significantly decrease (or even increases) in iteration 2, 3 and 4, and then falls sharply in the final iteration. This is not surprising, as the quality of the buffering computed in the middle iterations should be much lower than the quality of the first and final buffering. The fact that the final result is still an improvement over the initial solution does seem to indicate that some useful gate sizing is happening between iterations.

Secondly, we note that in the case of full mode, most progress is made in the earlier iterations. In fact, there are some examples where the solution found after the final iteration is in fact worse than one of the solutions found before. A possible conclusion might be that 5 iterations is too much. Reducing the number of iterations will reduce runtime, seemingly without much loss in terms of fom reduction.

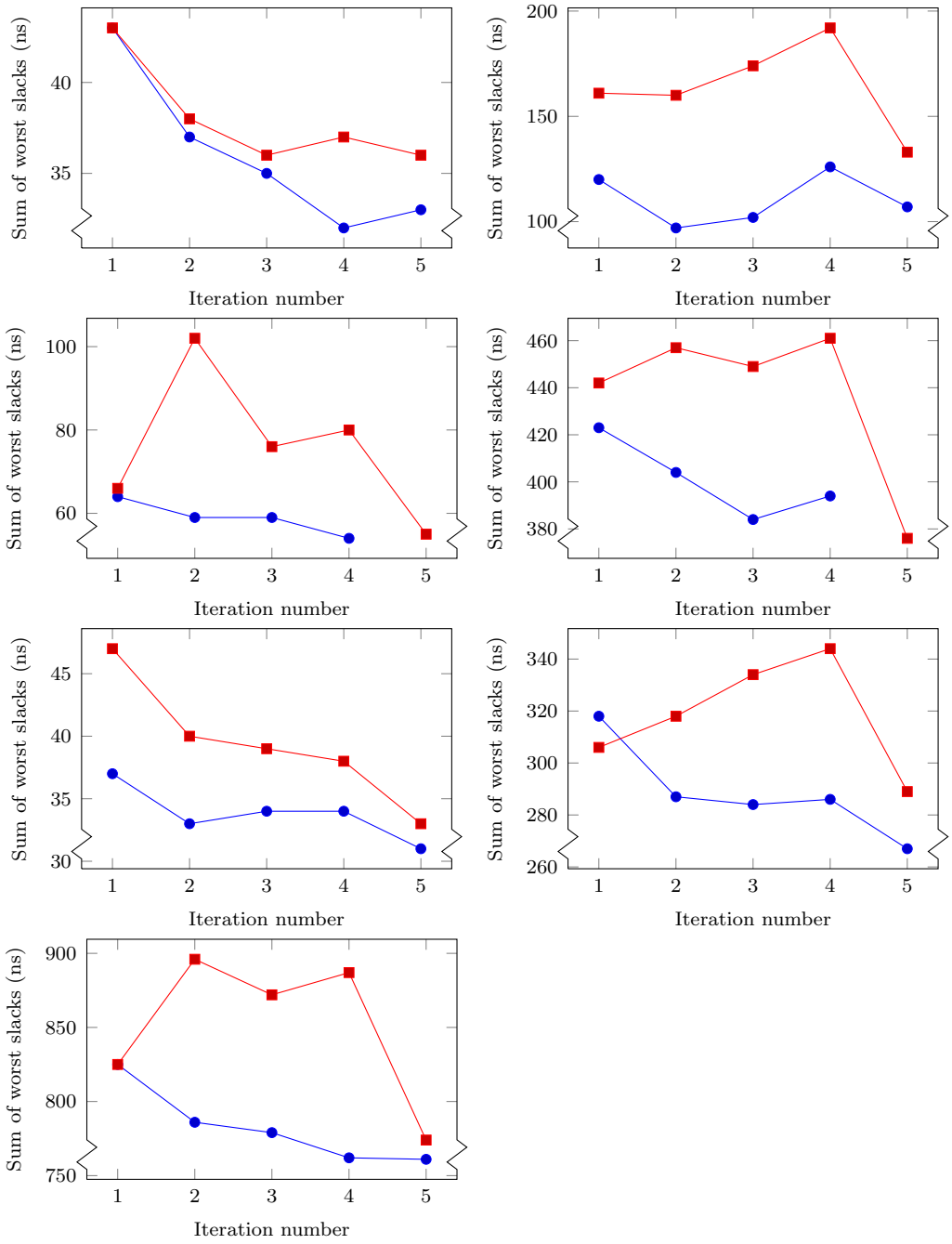


Figure 5.1: Sum of worst slacks after each iteration for both full (blue, dots) and reduced (red, squares) mode. The results for chips $C_1, C_2, \ldots C_7$ are ordered left to right, top to bottom.

Chip	Mode	Iteration 1			Iteration 2			Iteration 3			Iteration 4			Iteration 5			Comparison		
		ws (ps)	**fom** (ns)	**pow** (µW)	**ws** (ps)	**fom** (ns)	**pow** (µW)	**ws** (ps)	**fom** (ns)	**pow** (µW)	**ws** (ps)	**fom** (ns)	**pow** (µW)	**ws** (ps)	**fom** (ns)	**pow** (µW)	**ws** Δ %	**fom** Δ %	**pow** Δ %
C_1	**full**	120	43	151	100	37	152	100	35	153	100	32	153	92	33	160	-23.3	-23.4	6.2
C_2	**full**	184	120	222	178	97	225	172	102	228	258	126	228	208	107	232	13.0	-10.9	4.1
C_3	**full**	259	64	161	256	59	177	252	59	171	248	54	174	-	-	-	-4.2	-15.3	8.2
C_4	**full**	235	423	316	313	404	333	253	38	344	242	394	345	-	-	-	3.0	-7.0	9.1
C_5	**full**	94	37	80	79	33	83	76	34	84	82	34	84	78	31	84	-17.0	-14.6	4.4
C_6	**full**	175	318	269	163	287	269	173	284	276	167	285	273	148	267	276	-15.4	-16.0	2.5
C_7	**full**	608	825	258	591	786	254	597	779	256	588	762	256	586	761	259	-3.6	-7.7	0.2
C_1	**red.**	120	43	155	95	384	161	91	36	163	101	37	162	101	36	161	-15.8	-16.0	3.9
C_2	**red.**	244	161	225	303	160	247	280	174	254	295	192	250	209	133	235	-14.3	-17.8	4.3
C_3	**red.**	256	66	158	312	102	183	227	76	182	277	80	189	249	55	177	-2.7	22.1	19.5
C_4	**red.**	269	442	316	247	457	362	260	449	371	270	461	374	242	376	352	-10.0	4.2	18.2
C_5	**red.**	92	37	80	99	40	87	97	39	91	93	38	90	75	33	84	-18.5	-8.6	4.9
C_6	**red.**	161	306	271	215	318	276	212	334	278	223	344	275	165	289	275	2.5	-5.6	1.4
C_7	**red.**	607	825	260	605	896	261	588	872	262	618	887	266	598	774	258	-1.5	-6.2	-0.9

Table 5.1: Runtime tests for several chips provided by IBM. The final columns compare the situtation after iterations 5 and 1. For chips C_3, C_4 a timeout was reached before completion of the 5th iteration. The situation after iteration 4 was therefore used instead.

Chapter 6

Summary

In the second chapter of this thesis we reviewed some basic facts of convex optimization. We considered quadratic, geometric and signomial programming in particular. We then explored some notions of generalized convexity, including invexity, quasiconvexity, pseudoconvexity and biconvexity. For the latter we reviewed the alternate search algorithm, which produces a sequence of feasible solutions to a biconvex program converging to a local optimum.

In the third chapter, some basic notions from chip design were introduced. We described in some detail the problems of gate sizing and buffering, and presented some recent ideas for solving them. We also consider the interplay between these two problems and the difficulty in finding an algorithm solving both simultaneously.

In chapter four, we introduced the gate sizing problem with movement, extending the traditional gate sizing problem to also allow movement of repeaters. We studied this problem on repeater chains. We proved some new structural results for optimal solutions on repeater chains in the case that the repeaters are spaced equidistantly. We formulate the gate sizing problem with movement as a signomial problem that is also biconvex. Based on this, we suggested two methods for finding local optima of the program, using either alternate convex search or a general signomial solver. Performance of the ACS based method was substantially better on randomly generated test instances. Of particular interest was the fact that the amount of iterations required to reach a local optimum appears not to be dependent on the

amount of repeaters in the chain.

In the final chapter, we suggested two potential improvements to the BonnRouteBuffer flow, which is part of BonnTools. We tested an adaptation based on alternate search on chips provided by IBM. Although this adaptation succesfully reduces the sum of negative slacks on these chips, it does so at the cost of an increased power consumption and substantially increased runtime.

Bibliography

[1] B. Mond A. Ben-Israel. "What is invexity". In: *The ANZIAM journal* (1986).

[2] A.A. Ahmadi. *Lecture notes on convexity*. 2016. URL: `http://www.princeton.edu/~amirali/Public/Teaching/ORF523/S16/ORF523_S16_Lec7_gh.pdf`.

[3] D. Rautenbach B. Korte and J. Vygen. "BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip". In: *Proceedings of the IEEE, Vol. 95* (2016). URL: `http://www.or.uni-bonn.de/research/bonntools.pdf`.

[4] E. Burnell and W. Hoburg. *GPkit software for geometric programming*. `https://github.com/convexengineering/gpkit`. Version 0.7.0. 2018.

[5] Chris C. N. Chu Chung-Ping Chen and D. F. Wong. "Fast and Exact Simultaneous Gate and Wire Sizing by Lagrangian Relaxation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 7* (1999).

[6] B. D. Craven. "Duality for generalized convex fractional programs". In: *Generalized concavity in optimization and economics* (1981).

[7] K. Radke D. Müller and J. Vygen. "Faster Min-Max Resource Sharing in Theory and Practice". In: *Mathematical Programming Computation* (2011).

[8] S. Daboul. *Algorithms for the gate sizing and V_t assignment problem*. 2015.

[9] W. C. Elmore. *The transient response of damped linear networks with particular regard to wide-band amplifiers*. 1948.

[10] M. Frank and P. Wolfe. "An algorithm for quadratic programming". In: *Naval Research Logistics 3* (1956).

[11] L. P. P. P. Van Ginneken. "Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay". In: *Proceedings of the IEEE International Symposium on Circuits and Systems* (1990).

[12] N. Hähnle. "Time-cost Tradeoff and Steiner Tree Packing with Multiplicative Weights". In: *Technical report no. 1511115, Research Institute for Discrete Mathematics, University of Bonn* (2015).

[13] C. Cheng J. Lillis and T. Y. Lin. "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model". In: *IEEE Journal of Solid-State Circuits 31.3* (1996).

[14] A. E. Dunlop J. P. Fishburn. "TILOS: a posynomial programming approach to transistor sizing". In: *Proceedings of the IEEE International Conference on Computer-Aided Design* (1985).

[15] Kathrin Klamroth Jochen Gorski Frank Pfeuffer. "Biconvex Sets and Optimization with Biconvex Functions - A Survey and Extensions". In: *Mathematical Methods of Operations Research* (2007).

[16] M. Safonov G. Papavassilopoulos K. Goh L. Turan and J. Ly. "Biaffine matrix inequality properties and computational methods". In: *Proceedings of the American Control Conference Baltimore, Maryland* (1994).

[17] J. de Leeuw. "Block relaxation algorithms in statistics". In: *Information Systems and Data Analysis* (1994).

[18] B. Mond M. A. Anson. "Convex Transformable Programming Problems and Invexity". In: *J. Inj. Optim. Sci., 8* (1987).

[19] J. Dahl M. Andersen and L. Vandenberghe. *CVXOPT: A python package for convex optimization*. `http://cvxopt.org/`. 2018.

[20] M. J. Rijckaert and X. M. Martens. “A tutorial on geometric programming”. In: *Journal Of Optimization Theory And Applications: Vol. 26, No. 2* (1978).

[21] D. Romen. *Cost-Based Buffering for Multiple Resources*. 2015.

[22] D. Rotter. *Timing-Constrained Global Routing with Buffered Steiner Trees*. 2017.

[23] L. Vandenberghe S. Boyd. *Convex Optimization*. 2004.

[24] L. Vandenberghe A. Hassibi S. Boyd S. Kim. “A tutorial on geometric programming”. In: *Optimization and Engineering* (2005).

[25] S. Held S. Daboul N. Hähnle and U. Schorr. “Provably Fast and Near-Optimum Gate Sizing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[26] D. D. Patil M. A. Horowitz S. P. Boyd S. Kim. “Digital Circuit Optimization via Geometric Programming”. In: *Operations Research Vol 53, No. 6* (2005).

[27] W. Shi and Z. Li. “A Fast Algorithm for Optimal Buffer Insertion”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24.6* (2005).

[28] C. Szegedy. *Some Applications of the Weighted Combinatorial Laplacian*. 2005.

[29] J. Vygen. *Lecture notes on chip design*. 2016.

[30] Z. Li W. Shi and C. J. Alpert. “Complexity Analysis and Speedup Techniques for Optimal Buffer Insertion with Minimum Cost”. In: *Proceedings of the 2004 Asia and South Pacific Design Automation Conference. IEEE* (2004).

[31] R. Wendell and A. Hurter Jr. “Minimization of non-separable objective function subject to disjoint constraints”. In: *Operations Research 24 (4)* (1976).