This paper appeared in: Integration, the VLSI Journal 52 (2016), 355-366

An Exact Algorithm for Wirelength Optimal Placements in VLSI Design

Julia Funke[1], Stefan Hougardy[2], Jan Schneider[2]

[1]Lehrstuhl für Logistik, University of Bremen, Wilhelm-Herbst-Str. 5, 28359 Bremen, Germany
[2]Research Institute for Discrete Mathematics, University of Bonn, Lennéstr. 2, 53113 Bonn, Germany

June 17, 2015

Abstract

We present a new algorithm designed to solve floorplanning problems optimally. More precisely, the algorithm finds solutions to rectangle packing problems which globally minimize wirelength and avoid given sets of blocked regions. We present the first optimal floorplans for 3 of the 5 intensely studied MCNC block packing instances and a significantly larger industrial instance with 27 rectangles and thousands of nets. Moreover, we show how to use the algorithm to place larger instances that cannot be solved optimally in reasonable runtime.

keywords: Floorplanning; Placement; Rectangle Packing; Wirelength Minimization; Exact Algorithm

1 Introduction

In this paper we consider the *floorplanning* (or *facility layout / placement*) problem [1, 31], which asks for optimal positions for a given set of rectangles (the circuits) within a facility (the chip area), such that the rectangles do not overlap. The objective is to minimize distances, in our case the weighted half-perimeter wirelength, between some specified subsets of the rectangles. In the VLSI context the wirelength is a measure for the amount of metal needed to connect the circuits in a way that ensures the functionality of the chip. In some floorplanning approaches, the problem instances consist of so-called *soft* rectangles, i.e., rectangles varying in their height and width. Then, finding optimal values for the rectangles' widths and heights is also a part of the problem. However, in the following we only take *hard* rectangles into account, i.e., rectangles having a fixed outline.

Even without the optimization aspect the problem of packing given rectangles in a rectangular area is NP-complete, and it stays NP-complete under very strong restrictions like requiring the rectangular region to be larger by any factor than the area covered by the circuits themselves [30]. Thus there is no hope for a polynomial-time algorithm that solves the floorplanning problem optimally.

As a consequence, approaches looking for exact solutions usually consider the packing problem: What is the smallest rectangle into which a given set of smaller rectangles can be packed? On the

other hand, floorplanning algorithms that consider a target function like wirelength are usually heuristics without provable guarantees on the quality of their solutions.
In this paper, both aspects are combined: We extend the exact packing algorithm by Moffitt and Pollack [32] so that it optimizes wirelength, resulting in an algorithm that finds provably optimal solutions to wirelength minimization problems. Moreover, we present a framework that uses this method as a subroutine to solve larger instances for which no optimal solution can be found in reasonable runtime.
The remaining paper is organized as follows: After revisiting previous work in Section 2, Section 3 provides some basic definitions and notation used throughout this paper. In Section 4, we review the CONTAINMENT algorithm from [32], a method to pack a set of rectangles and which will serve as the basis for our extensions discussed in Section 5. These extensions enable the algorithm to solve an optimization problem, namely the wirelength minimization problem. Finally, Section 6 presents computational results on widespread MCNC benchmarks, additional synthetic benchmarks and real-world instances provided by IBM, and Section 7 summarizes our contributions.

2 Related Work

The problem of arranging multiple rectangles disjointly into a larger region, usually another rectangle, is known as rectangle packing. As a fundamental geometric problem, it has been studied for many decades [33]. Algorithmic advances were made in the context of VLSI design, where arrangements of rectangles are represented with certain structures, for example sequence pairs [34], bounded sliceline grids [35], Q-sequences [50] or B*-trees [41], that can be improved by heuristics like simulated annealing. To compute wirelength optimal placements a floorplan representation is needed that allows to represent placements of floorplans in general position. Good surveys on these floorplan representations can be found in [48] and [3].
A series of papers by Korf et al., e.g. [21, 22, 18], present optimal algorithms that find minimum-area rectangles into which a given set of rectangles can be packed. They present solutions up to $n = 32$ for a benchmark problem where n squares of size 1 to n have to be packed into a smallest possible rectangle. In some cases, for example when most or all of the available area must be covered by rectangles, even larger instances can be solved [16]. Moffit and Pollack [32] introduced a branch and bound method for the rectangle packing problem that we extend in the following chapters to compute wirelength optimal rectangle packings.
Most papers concerned with target functions like wirelength do not present algorithms that guarantee to find an optimal solution. The approaches used often rely on heuristics similar to the methods that are also applied for area minimization. For example, Tang and Sebastian [43] use a genetic algorithm to optimize over O-trees, Fernando and Katkoori [11] use sequence pairs, and Lin et al. [28] use B*-trees. Yan and Chu [47] use generalized slicing trees in combination with local operations to optimize the wirelength. Sutanthavibul, Shragowitz, and Rosen [42] present an approach that allows to bound the length of each net individually. Onodera, Taniguchi, and Tamaru [36] present an algorithm for computing netlength optimal rectangle placements. Their branch-and-bound algorithm is based on the enumeration of all $4^{\binom{n}{2}}$ possible relations that can exist between n rectangles. The largest instance that they present in their paper with an optimal

solution had 6 rectangles.
A more extensive survey on floorplanning literature is presented by Bortfeldt [2].

3 Preliminaries

We start by formalizing the notions used throughout this paper. Unless otherwise noted the problem instance always consists of n axis-parallel rectangles $\mathcal{R} = \{r_1, \dots, r_n\}$ having integer widths w_i and heights h_i, for $1 \leq i \leq n$, and two integers W and H representing the size of the rectangular box in which $\mathcal{R}$ must be packed. We always assume that $w_i \leq W$ and $h_i \leq H$ for $1 \leq i \leq n$ and that rotation of the rectangles is not allowed.
We call a rectangle r_i *placed* if there is a location (x_i, y_i) assigned to it. In this case, the rectangle is identified with the set $[x_i, x_i + w_i] \times [y_i, y_i + h_i]$ such that (x_i, y_i) can be seen as the lower left corner of r_i. We also write $x(r_i)$ instead of x_i and $y(r_i)$ instead of y_i. An assignment of locations to a subset of $\mathcal{R}$ is a *placement* or, if locations are assigned to all n rectangles, a *complete placement*. A placement is *legal* if both of the following conditions hold:

1. *Containment constraints*: All placed rectangles lie within the outer box, i.e. $0 \leq x_i \leq W - w_i$ and $0 \leq y_i \leq H - h_i$ hold for every index i of a placed rectangle.

2. *Non-overlap constraints*: The placed rectangles are pairwise interior-disjoint. More formally, for all distinct indices i and j of placed rectangles at least one of the following inequalities holds: $x_i + w_i \leq x_j$, $x_j + w_j \leq x_i$, $y_i + h_i \leq y_j$, or $y_j + h_j \leq y_i$.

To simplify the notation, we will replace the constraints of the second condition by the symbols ${}_iH_j$ (r_i is *to the left of* r_j), ${}_jH_i$ (r_i is *to the right of* r_j), V_i^j (r_i is *below* r_j), and V_j^i (r_i is *above* r_j).
Since the algorithms presented in the following chapters do not directly look for placements but rather describe the relative positions of placed rectangles, we introduce the notion of relative placements. Let ρ be a function that maps pairs of distinct rectangles to relative positions of these pairs, in other words $\rho(r_i, r_j) \in \{{}_iH_j,\ {}_jH_i,\ V_i^j,\ V_j^i\}$, then we call ρ a *relative placement*, or, if all pairs of distinct rectangles are in its domain, a *complete relative placement*. We say a placement *satisfies* ρ if it satisfies all inequalities in the image of ρ, and a relative placement is called *consistent* if there exists a complete placement that satisfies ρ. In the former sections we start to compute a solution to the *containment problem*, i.e. the problem of deciding for a set $\mathcal{R}$ of rectangles and numbers W and H if there exists a legal complete placement.
As the aim of this paper is not only to produce placements of rectangles, but also to minimize wirelength, we move on to definitions that are used in the field of VLSI design.

Definition 1. A *netlist* is an instance $(\mathcal{R}, W, H)$ of the containment problem together with a set P (the *pins*), an assignment $\gamma : P \to \mathcal{R} \cup \{\Box\}$, pin offsets $x_{\text{offs}} : P \to \mathbb{Z}$ and $y_{\text{offs}} : P \to \mathbb{Z}$, a partition of the pins $\mathcal{N} = \{N_1, \dots, N_k\}$ (the *nets*), i.e. $P = N_1 \dot\cup \dots \dot\cup N_k$, and *net weights* $w : \mathcal{N} \to \mathbb{R}_+$.

Given a netlist and a complete placement of $\mathcal{R}$, the *location* of a pin $p \in P$ is given by $x(p) := x(\gamma(p)) + x_{\text{offs}}(p)$ if $\gamma(p) \neq \Box$ and $x(p) := x_{\text{offs}}(p)$ otherwise. Vertical coordinates are defined analogously. In VLSI design, the rectangles correspond to circuits that have to be arranged on the rectangular chip area. Pins are the positions where the circuits have to be connected by wires, and nets are sets of pins that have to be connected electrically. The goal is to minimize the expected length of these connections, often modeled by the size of the smallest box containing all pins of a net.

Definition 2. Let $(\vec{x}, \vec{y})$ be a complete placement of a netlist. Then the *weighted half-perimeter wirelength* is defined as

$$\mathrm{HPWL}(\vec{x}, \vec{y}) := \sum_{N \in \mathcal{N}} w(N) \cdot \Bigg(\max_{p \in N} x(p) - \min_{p \in N} x(p) + \max_{p \in N} y(p) - \min_{p \in N} y(p) \Bigg) .$$

Here $\vec{x}$ and $\vec{y}$ denote vectors containing the coordinates of the placed rectangles. Using this notion of wirelength, the *(fixed-outline) floorplanning* problem, also called *placement* problem, is the following: Given a netlist and a legal placement of $\mathcal{R}$, find a legal complete placement $(\vec{x}, \vec{y})$ extending it that minimizes $\mathrm{HPWL}(\vec{x}, \vec{y})$ or decide that no such placement exists. The rectangles that are already placed in the input are called *blockages*.
If all rectangle sizes w_i and h_i and all pin offsets x_{offs} and y_{offs} are integers, then we call the instance of the floorplanning problem *nearly integral*. When we speak about the floorplanning problem we only consider nearly integral instances as an input within this paper.
As many of our results are obtained on graphs, some notations of graphs conclude this section. When we talk about a graph $G = (V, E)$, then we always assume that G is directed. Its node set is denoted by V and its edge set by E. We also write $V(G)$ to indicate its nodes and $E(G)$ to indicate its edges instead.

Definition 3. A directed graph $G = (V, E)$ together with capacities u and costs c on its edges, i.e., $u, c : E \to \mathbb{R}$ and balances $b : V \to \mathbb{R}$ on its nodes is called **network**. A **b-flow** is a function $f : E \to \mathbb{R}_+$, such that $0 \leq f(e) \leq u(e)$ holds for all $e \in E$ and the flow conservation $\sum_{(v,w)\in E} f(v, w) - \sum_{(w,z)\in E} f(w, z) = b(w)$ is satisfied for all $w \in V$.

If a network (G, u, b, c) is given, the *Minimum Cost Flow Problem* asks for a b-flow of minimum costs where the cost of a flow f is defined as $c(f) := \sum_{e\in E} f(e) \cdot c(e)$.

4 Solving the Containment Problem

In this section, we consider the packing aspect of floorplanning. We present an algorithm that extends the core routine of BlueBlocker from Moffitt and Pollack [32] and solves the containment problem. Similar to a method of Taniguchi et al. [36] the Containment algorithm iteratively generates a relative placement ρ, starting with the trivial mapping that does not assign any relative positions. In a branch and bound approach, subsequently assigning relations to pairs without a relation in such a way that ρ remains consistent.
Algorithm 1 shows the recursive formulation of the algorithm. Here $\rho[\circ]$ denotes the relative placement which is obtained from ρ by mapping (r_i, r_j) to $\circ$. When extending ρ by, say, mapping

```
Input  : A finite rectangle set $\mathcal{R}$, positive numbers $W$ and $H$, and a consistent relative
         placement $\rho$.
Output: SUCCESS, if there exists a legal complete placement of $\mathcal{R}$ that satisfies the
         assignments of $\rho$, FAILURE otherwise.
1  if $\rho$ is complete then
2      Return SUCCESS;
3  end
4  Choose pair $(r_i, r_j)$ with $i \neq j$ which is not in the domain of $\rho$;
5  for $\circ \in \{{}_iH_j,\ {}_jH_i,\ V_i^j,\ V_j^i\}$ do
6      if $\rho[\circ]$ is consistent then
7          if Containment($\mathcal{R}, W, H, \rho[\circ]$) then
8              return SUCCESS;
9          end
10     end
11 end
12 return FAILURE;
```

Algorithm 1: Containment algorithm

the pair (r_i, r_j) to ${}_iH_j$, it is always possible to implicitly map (r_j, r_i) to ${}_jH_i$, too. This is a simple consequence from the symmetry of the inequalities.
As soon as this approach leads to a complete relative placement which is consistent, the algorithm stops. Observe that in this case, by definition, a legal placement exists. If on the other hand the algorithm fails, no legal placement exists.

4.1 The Consistency Check

In every step the Containment algorithm needs to check whether the extended relative placement is still consistent. In order to do so, two so-called *distance graphs* G_{hor} and G_{ver}—one for each direction—are used. G_{hor} contains n vertices $v_1, \dots, v_n$ and a directed arc (v_j, v_i) of weight $-w_i$ if $\rho(r_i, r_j) = {}_iH_j$. We also include nodes v_0 and v_{n+1} representing the lower left and upper right corners of the placement box. Then G_{hor} contains arcs (v_i, v_0) of weight 0 and arcs (v_{n+1}, v_i) of weight $-w_i$ for $1 \leq i \leq n$. It also contains a special arc $e_\square = (v_0, v_{n+1})$ of weight W. Analogously, G_{ver} is defined on the same vertex set and has an arc (v_j, v_i) of weight $-h_i$ if $\rho(r_i, r_j) = V_i^j$, arcs (v_i, v_0) of weight 0 and (v_{n+1}, v_i) of weight $-h_i$ for $1 \leq i \leq n$, and an arc $e_\square = (v_0, v_{n+1})$ of weight H.
Shostak [40], Liao and Wong [26] and Leiserson and Saxe [25] showed that these graphs can be used to determine the consistency of a placement.

Lemma 4. *A relative placement ρ is consistent if and only if G_{hor} and G_{ver} have no negative cycles.*

Using this lemma we can check in polynomial time whether a given relative placement is consistent.

Lemma 5. *Given distance graphs G_{hor} and G_{ver}, it can be checked in time $O(n^2)$ whether the associated relative placement is consistent.*

Proof. First we can check in linear time whether $G_{hor} - e_\square$ and $G_{ver} - e_\square$ are acyclic. If one of them contains a cycle, Lemma 4 implies that the relative placement is not consistent. Otherwise, we have to check if the distance from v_{n+1} to v_0 is less than $-W$ respective $-H$. This can be done in linear time in an acyclic graph. The lemma follows from the observation that G_{hor} and G_{ver} have $O(n^2)$ edges. $\square$

4.2 The Bounding Strategies

In general, Algorithm 1 solves the containment problem in $\mathcal{O}(4^{\binom{n}{2}} \cdot n^2)$ steps. Jerrum [19] showed how to encode all complete relative placements that are consistent for $W = H = \infty$ with two permutations of the set $\{1, 2, \ldots, n\}$, therefore bounding their number by $n!^2$. His attempt leads to an algorithm that can solve the containment problem in $O(n!^2 \cdot n^2) \approx O((\frac{n}{e})^{2n} \cdot n^2)$, which seems to be much faster than the technique we use.
However, in this section we will mention and extend some of the powerful bounding strategies from [32] that enable the Containment algorithm to outperform Jerrum's method in practice.

4.2.1 Transitive closure

Dechter et al. showed in [10] that for given all-pairs shortest path matrices of the distance graphs G_{hor} and G_{ver} one can easily extract a complete placement satisfying ρ. But those matrices are also useful for another reason: When extending the relative placement within the Containment algorithm, more relations might be implied directly. For example, if r_i is to the left of r_j and we add the relation that r_j is to the left of r_k, then we also know that r_i has to be to the left of r_k. When maintaining all-pairs shortest path matrices, these implied relations are available immediately:

Lemma 6. *The implied relations can be directly deduced from the all-pairs shortest path matrices:*

$$\begin{aligned} & dist_{G_{hor}}(v_j, v_i) \leq -w_i \;\Leftrightarrow \\ & \qquad {}_iH_j \textit{ holds in every placement satisfying } \rho \\ & dist_{G_{ver}}(v_j, v_i) \leq -h_i \;\Leftrightarrow \\ & \qquad V_i^j \textit{ holds in every placement satisfying } \rho \end{aligned}$$

Here we use $\mathrm{dist}_G(v, w)$ to denote the length of the shortest path from v to w in G, with $\mathrm{dist}_G(v, w) = \infty$ if no such path exists. Consequently, we can add those implied relations to ρ directly after computing the new all-pairs shortest path matrices of G_{hor} and G_{ver}, which can be obtained in runtime $\mathcal{O}(n^3)$ with the algorithm of Floyd and Warshall [12, 46, 17], leading to a much smaller depth of the branch and bound tree. For this reason we compute all-pairs shortest path matrices instead of using the algorithm suggested by Lemma 5. If we want to check consistency of a given relative placement we only have to look for negative entries on the diagonal of those matrices.

4.2.2 Semantic Branching

An important technique to speed up Algorithm 1 is called semantic branching. Assume that the algorithm chose (r_i, r_j) as the current rectangle pair and started by extending ρ with ${}_iH_j$, i.e. $x_i + w_i \leq x_j$. Further assume that the algorithm fails to find a legal complete placement using this assignment. Then we know that in all solutions that might be found in the current branch of the branch and bound tree, the rectangle r_i is *not* to the left of r_j, or $x_i + w_i > x_j$. The following lemma enables us to slightly strengthen this constraint.

Lemma 7. *If all widths and heights in the instance are integral and a complete legal placement exists, there also exists a complete legal placement where all x_i and y_i are integral.*

Proof. Starting with a complete legal placement, the process of iteratively moving rectangles to the lower left as far as possible will end in a placement with the required property. □

Since we assume all numbers in the instance to be integral, we can restrict the search to placements where all rectangles have integer coordinates. Thus the condition "r_i is not to the left of r_j" can be written as $x_i + w_i \geq x_j + 1$, which translates to an edge (v_i, v_j) of weight $w_i - 1$ in G_{hor}. By introducing such edges, the algorithm always stores the negation of an inequality when failing to find a legal complete placement satisfying this constraint.
In the following, we call distance graphs which have been extended by arcs representing such negations *extended distance graphs* and denote them by $\widetilde{G}_{hor}$ and $\widetilde{G}_{ver}$. Observe that Lemma 4 still holds for extended distance graphs and that by computing all-pairs shortest path matrices one can verify the consistency of their underlying relative placements. One should also note that because the weights of the arcs of the negated relations are positive, we cannot use Lemma 5 anymore.
When the negation of a constraint is added to the distance graphs, other constraints might be implied in the same way as before. Fortunately, those can also be deduced from the all-pairs shortest path matrices immediately.

Lemma 8. *Let $\widetilde{G}_{hor}$ and $\widetilde{G}_{ver}$ be the extended distance graphs, then:*

$$\begin{aligned} &dist_{\widetilde{G}_{hor}}(v_j, v_i) < w_i \;\Leftrightarrow \\ &\qquad {}_iH_j \textit{ does not hold in any placement satisfying } \rho \\ &dist_{\widetilde{G}_{ver}}(v_j, v_i) < h_i \;\Leftrightarrow \\ &\qquad V_i^j \textit{ does not hold in any placement satisfying } \rho \end{aligned}$$

A third way to determine implied relations uses the outer placement box. For example, if ${}_iH_j$ would require that a set of rectangles have to lie next to each other, but the sum of the widths of these rectangles is larger than W, then the negation of ${}_iH_j$ can be assumed in the current branch. This can also be checked in constant time for each rectangle pair when all-pairs shortest path matrices are available.

Lemma 9. *Let $\widetilde{G}_{hor}$ and $\widetilde{G}_{ver}$ be extended distance graphs, then:*

$$dist_{\widetilde{G}_{hor}}(v_{n+1}, v_j) - w_i + dist_{\widetilde{G}_{hor}}(v_i, v_0) < -W \Leftrightarrow$$
$${}_iH_j \textit{ does not hold in any placement satisfying } \rho$$
$$dist_{\widetilde{G}_{ver}}(v_{n+1}, v_j) - h_i + dist_{\widetilde{G}_{ver}}(v_i, v_0) < -H \Leftrightarrow$$
$$V_i^j \textit{ does not hold in any placement satisfying } \rho$$

4.2.3 Sorting the Rectangle Pairs

The branching rule, i.e. the order in which the rectangle pairs in step 4 in Algorithm 1 are chosen, can have a big impact on the algorithm's runtime in practice. If one assigns relations between huge rectangles first, arrangements between those and others can become impossible. Thus Moffitt and Pollack [32] propose an ordering of the rectangles that considers their size. As the areas of both rectangles should be relevant for the branching rule, the algorithm processes the pairs in descending order according to the maxmin function of their volumes

$$\max_{i \neq j} \min\{w_i h_i, w_j h_j\}.$$

This function determines the base ordering of the rectangle pairs. However, if at any point in the algorithm there exists a rectangle pair $(r_i, r_j) \in \mathcal{R}^2$ that already has three negated relations assigned to it, the next branching step adds the unique missing relation to $\rho(r_i, r_j)$.

5 Finding Optimal Floorplans

In the previous chapter we described a powerful tool to enumerate all legal complete placements, mainly due to Moffitt and Pollack [32]. Now we want to apply this tool to optimally solve the floorplanning problem, a task originating in VLSI design which adds an optimization component to the packing aspect discussed so far.

5.1 Basic Algorithm

When we talk about the floorplanning problem, we only consider nearly integral instances. During the course of this section, it will be pointed out why we take this restriction into account. We use the notation $\mathcal{B} \subset \mathcal{R}$ for the blockages, i.e. the rectangles that have already fixed positions in the input.
Algorithm 2 outlines the structure of our method, called SPARK, to solve the floorplanning problem. A preliminary version of the method is described in [13]. Besides a nearly integral placement instance and a relative placement, it expects a legal placement f and a value $s \in \mathbb{R} \cup \{\infty\}$ as input parameters. When SPARK is called for the first time, f should contain a legal complete placement of $\mathcal{B}$ and s should be set to ∞. The return values constitute of these two parameters: While f is extended to obtain a legal complete placement of $\mathcal{R}$ with minimum wirelength, without changing the locations of $\mathcal{B}$, and s should be equal to its weighted half-perimeter wirelength. If no legal complete placement exists, f and s will not be modified.

```
Input  : A nearly integral floorplanning problem instance $\mathcal{I}$, a consistent relative
         placement $\rho$ of $\mathcal{R} \cup \mathcal{B}$, a legal placement $f$ of $\mathcal{R} \cup \mathcal{B}$ and a value $s \in \mathbb{R} \cup \{\infty\}$.
Output : A pair $(f^*, s^*)$ with $f^*$ is a legal complete placement of $\mathcal{R} \cup \mathcal{B}$ and $s^*$ is the
         minimal weighted half-perimeter wirelength of $\mathcal{I}$ induced by $f^*$, if such a
         placement of $\mathcal{I}$ exists and $s = \infty$ otherwise.
1  if $\rho$ is complete then
2      if $HPWL(\rho) < s$ then
3          Update $(f, s)$;
4      end
5      return $(f, s)$;
6  end
7  Choose pair $(r_i, r_j)$ with $i \neq j$ which is not in the domain of $\rho$;
8  for $\circ \in \{{}_iH_j,\ {}_jH_i,\ V_i^j,\ V_j^i\}$ do
9      if $\rho[\circ]$ is consistent then
10         if $HPWL(\rho) < s$ then
11             Set $(f, s)$ to SPARK$(\mathcal{I}, \rho, f, s)$;
12         end
13     end
14 end
15 return $(f, s)$;
```

Algorithm 2: SPARK Algorithm

Compared to Algorithm 1, SPARK is changed regarding two main aspects: On the one hand, it does not abort in line 11, when the first solution is found – instead the whole branch and bound tree is traversed in order to find a global optimum. On the other hand, a new bounding rule is integrated, measuring the wirelength of the current relative placement ρ. Within SPARK, HPWL(ρ) denotes the minimal weighted half-perimeter wirelength of the netlist satisfying the relations of ρ. In other words, whenever a consistent relative placement is computed, s and f are updated in line 3 to HPWL(ρ) and a placement obtaining HPWL(ρ) that satisfies the relations of ρ, if HPWL(ρ) is better than the best known value for the wirelength. In line 10 those branches are bounded, the current relative placement includes a larger wirelength than the best known wirelength.
Only small changes are needed to modify SPARK such that it is able to prove the optimality of a given complete legal placement. A further deduction is that there might be large improvements in terms of the practical runtime of SPARK if s is set to the wirelength of a known legal complete placement, e.g. a placement obtained by heuristics. The subsequent content of this section explains how to extend Algorithm 1 that it is able to deal with blockages and a netlist.

5.2 Handling of Blockages

From now on, we distinguish between two sets of rectangles: placed rectangles $\mathcal{B}$ (also called blockages) and non-placed rectangles $\mathcal{R}$. Due to Section 3, a blockage $b_i \in \mathcal{B}$ has assigned a location (x_i, y_i), while a rectangle in $\mathcal{R}$ has not and thus there is a need of optimization: The aim of this section is to compute a complete legal placement of $\mathcal{R}$ within the outer box, additionally satisfying the non-overlap constraints of $\mathcal{R} \cup \mathcal{B}$. In other words, we are only interested in the region that is not covered by blockages, not in the blockages themselves. We can assume that blockages are pairwise interior-disjoint because overlaps could not be resolved by the algorithm.

Lemma 10. *Let ρ be a relative placement of $\mathcal{R} \cup \mathcal{B}$, we can check whether ρ is consistent in $O(n^2)$ time, where still $n = |\mathcal{R}|$ holds.*

Proof. We consider the weighted distance graphs (G_{hor}, w_{hor}) and (G_{ver}, w_{ver}) for the instance $(\mathcal{R}, W, H)$. In the following, we only define new weight functions w'_{hor}/w'_{ver} on the arcs of G_{hor}/G_{ver}, while the set of nodes and arcs stay the same. By doing that, we obtain weighted distance graphs $(G_{hor}, w'_{hor})/(G_{ver}, w'_{ver})$, in which the handling of blockages is included. The remaining proof is restricted to G_{hor} only, but it can be applied to G_{ver} analogously. We define weights according to

$$\forall w'_{hor}((i,j)) := \begin{cases} -R_i, & j = 0, i \neq n+1 \\ L_j - W, & j \neq 0, i = n+1, \\ w_{hor}(i,j), & \text{otherwise.} \end{cases} \tag{1}$$

for all $(i,j) \in E(G_{hor})$, whereby

$$\begin{aligned} R_i &:= \max\{x_j + w_j \mid b_j \in \mathcal{B}, \rho((r_i, b_j)) = {}_jH_i\}, \\ L_j &:= \min\{x_i \mid b_i \in \mathcal{B}, \rho((r_j, b_i)) = {}_jH_i\} - w_j, \end{aligned}$$

with $\min \emptyset := W$ and $\max \emptyset := 0$.
These weights are achieved, by eliminating redundancies of constraints referring to one of the two borders first and building arcs following Equation (1) afterwards. The constraints related to the left border are the following:

$$\begin{aligned} &\forall r_i \in \mathcal{R} : x_0 \leq x_i \\ &\forall r_i \in \mathcal{R}, \forall b_j \in \mathcal{B} : \rho((r_i, b_j)) = {}_jH_i : x_0 + (x_j + w_j) \leq x_i \end{aligned}$$

While the former set of constraints refer to the containment constraints, the latter refers to the non-overlap constraints of rectangles with blockages. These constraints are set in relation to the left border of the outer box represented by x_0. Obviously it holds, that a rectangle that is placed to the right of the right most blockage is also placed to the right of the other blockages and additionally, it is placed to the right of the left corner of the outer box. The second implication follows as we assume that all blockages are placed legally within the outer box.
The constraints related to the right border of the outer box are the following (again, by eliminating redundancies w' is obtained):

$$\begin{aligned} &\forall r_i \in \mathcal{R} : x_i + w_i \leq x_{n+1} \\ &\forall b_i \in \mathcal{B}, \forall r_j \in \mathcal{R} : \rho((r_j, b_i)) = {}_jH_i : \\ &\qquad x_j + w_j + (W - x_i) \leq x_{n+1} \end{aligned}$$

As the constants of the left-hand side of all constraints have non-negative values and the number of the arcs of the distance graphs does not change, the proof for the runtime follows by the proof of Lemma 5. □

To achieve good practical runtimes, it is important to enable SPARK to use the bounding strategies from [32]. First, we will show that it is possible to obtain *transitive relations* regarding relations between rectangles and blockages. As we compute shortest paths originating at v_{n+1} within the proof of Lemma 10, we obtain a placement, in which the rectangles are placed as right/high as possible satisfying all constraints. The current placement of a rectangle $r_i \in \mathcal{R}$ can be deduced by considering the distance of v_{n+1} to v_i:

$$x_i = W + \mathrm{dist}_{G_{hor}}(v_{n+1}, v_i), \quad y_i = H + \mathrm{dist}_{G_{ver}}(v_{n+1}, v_i)$$

Thus, we can draw conclusions regarding the transitive closure, if a rectangle $r_i \in \mathcal{R}$ is placed to the left/below of a blockage $b_j \in \mathcal{B}$ within the current placement that is oriented towards the right/top:

$$\begin{aligned}
&W + \mathrm{dist}_{G_{hor}}(v_{n+1}, v_i) \le x_j \;\Leftrightarrow\\
&\qquad {}_iH_j \text{ holds in every placement satisfying } \rho\\
&H + \mathrm{dist}_{G_{ver}}(v_{n+1}, v_i) \le y_j \;\Leftrightarrow\\
&\qquad V_i^j \text{ holds in every placement satisfying } \rho\\
&W + \mathrm{dist}_{\widetilde{G}_{hor}}(v_{n+1}, v_i) < x_j + w_j \Leftrightarrow\\
&\qquad {}_jH_i \text{ does not hold in any placement satisfying } \rho\\
&H + \mathrm{dist}_{\widetilde{G}_{ver}}(v_{n+1}, v_i) < y_j + h_j \Leftrightarrow\\
&\qquad V_j^i \text{ does not hold in any placement satisfying } \rho
\end{aligned}$$

The missing relations are obtained by reversing the arcs of the distance graphs. Meaning we want to create a placement of the rectangles, in which they are placed as left/below as possible satisfying all constraints by computing the shortest paths originating at v_0.

Definition 11. Let $G = (V, E)$ be a digraph. Then $G^* := (V, E^*)$, with $E^* := \{(u, v) \mid (v, u) \in E\}$ is called its **reverse**. We define weights $w^* : E^* \to \mathbb{R}$ by $w^*(u, v) := c(v, u)$.

As G^*_{hor} and G^*_{ver} contain all relations of ρ in form of arcs, a placement satisfying ρ is obtained when computing shortest paths in the reverse order than within G_{hor} and G_{ver}. Then, the current placement of a rectangle $r_i \in \mathcal{R}$ can be deduced by considering the distance of v_0 to v_i:

$$x_i = |\mathrm{dist}_{G^*_{hor}}(v_0, v_i)|, \quad y_i = |\mathrm{dist}_{G^*_{ver}}(v_0, v_i)|$$

The missing relations can be extracted from the current placement as follows:

$$\begin{aligned}
&x_j + w_j \leq |\mathrm{dist}_{G^*_{hor}}(v_0, v_i)| \;\Leftrightarrow \\
&\qquad {}_jH_i \text{ holds in every placement satisfying } \rho \\
&y_j + h_j \leq |\mathrm{dist}_{G^*_{ver}}(v_0, v_i)| \;\Leftrightarrow \\
&\qquad V^i_j \text{ holds in every placement satisfying } \rho \\
&x_j < |\mathrm{dist}_{\widetilde{G}^*_{hor}}(v_0, v_i)| \;\Leftrightarrow \\
&\qquad {}_iH_j \text{ does not hold in any placement satisfying } \rho \\
&y_j < |\mathrm{dist}_{\widetilde{G}^*_{ver}}(v_0, v_i)| \;\Leftrightarrow \\
&\qquad V^j_i \text{ does not hold in any placement satisfying } \rho
\end{aligned}$$

As *semantic branching* only means to include the negation of a constraint, that will not hold for any placement within the current branch, to the set of constraints, it is no problem to apply this technique to a setting containing blockages.
Thus, the last thing that is missing concerning the handling of blockages within Spark is the *sequence* of rectangle pairs, in which they are processed in line 8 of Spark. We keep the order of the non-placed rectangle pairs as it is introduced in Section 4.2.3. But, whenever a pair $(r_i, r_j) \in \mathcal{R}^2$ is chosen, Spark continues by processing all blockages and the rectangle of the pair that has the larger volume. Afterwards, (r_i, r_j) is chosen to get assigned relations. And, at the end the smaller rectangle gets assigned relations to all blockages. Then, it is proceeded like in Section 4.2.3 with choosing the next rectangle pair non-overlap constraints should be applied to. The rectangle pair is treated as mentioned before by also including relations with blockages to it.

5.3 Minimizing Wirelength

Vygen [45] demonstrates how to extract from a given consistent relative placement ρ a placement having minimum wirelength that satisfies all relations indicated by ρ. To achieve this, he formulates the problem as a linear program. In the following, we use his results to implement HPWL(ρ) that is called in lines 3 and 10 of Spark.
We are given an instance $\mathcal{I}$ of the floorplanning problem together with a legal relative placement ρ, then two decision variables x_i and y_i are introduced for each rectangle $r_i \in \mathcal{R}$. These decision variables correspond to the lower left corner of r_i. Additionally, each net $N \in \mathcal{N}$ induces four decision variables x^+_N, x^-_N, y^+_N and y^-_N representing the smallest axis-parallel rectangle $[x^-_N, x^+_N] \times [y^-_N, y^+_N]$ that covers all pins of N. The following formulation arises:

$$\min \sum_{N \in \mathcal{N}} w(N) \cdot (x^+_N - x^-_N + y^+_N - y^-_N) \tag{2}$$

$$
\begin{aligned}
&\forall N \in \mathcal{N}, P \in N, \gamma(P) \neq \square : \\
&\qquad x_N^- \leq x_{\gamma(P)} + x(P), \qquad x_{\gamma(P)} + x(P) \leq x_N^+, \\
&\qquad y_N^- \leq y_{\gamma(P)} + y(P), \qquad y_{\gamma(P)} + y(P) \leq y_N^+, \\
&\forall N \in \mathcal{N}, P \in N, \gamma(P) = \square : \\
&\qquad x_N^- \leq x(P), \qquad x(P) \leq x_N^+, \\
&\qquad y_N^- \leq y(P), \qquad y(P) \leq y_N^+, \\
&\forall r_i \in \mathcal{R} : \\
&\qquad 0 \leq x_i, \qquad x_i + w_i \leq W, \\
&\qquad 0 \leq y_i, \qquad y_i + h_i \leq H. \\
&\forall r_i, r_j \in \mathcal{R} \cup \mathcal{B}, r_i \neq r_j : \\
&\qquad x_i + w_i \leq x_j \qquad \text{if } \rho(r_i, r_j) = {}_iH_j, \\
&\qquad x_j + w_j \leq x_i \qquad \text{if } \rho(r_i, r_j) = {}_jH_i, \\
&\qquad y_i + h_i \leq y_j \qquad \text{if } \rho(r_i, r_j) = V_i^j, \\
&\qquad y_j + h_j \leq y_i \qquad \text{if } \rho(r_i, r_j) = V_j^i).
\end{aligned}
$$

While the objective is to minimize the weighted wirelength, the former two sets of constraints model the bounding boxes of the pins of the nets and the last two sets of constraints are restricted to observe the containment, as well as non-overlap constraints.
By introducing an auxiliary variable v_0, whose value should be equal to zero, all constraints can be written in the form $v_j - v_i \geq a_{ij}$, where a_{ij} is a constant and v_i, v_j are variables. The dual of this linear program is of the form

$$\max \sum_{i,j} a_{ij} f_{ij} \tag{3}$$

$$\forall j : \sum_i f_{ij} - \sum_k f_{jk} = \begin{cases} w(N), & v_j \in \{x^-(N), y^-(N)\} \\ -w(N), & v_j \in \{x^+(N), y^+(N)\} \\ 0, & \text{otherwise.} \end{cases}$$

As this linear program models a *Maximum Cost Flow Problem* in a digraph [45], it can be solved in polynomial time. Anyway, due to practical reasons, we chose the Network Simplex Algorithm to obtain solutions to it. In Section 5.5.2 it is discussed how to time saving do that. As in literature the term Minimum Cost Flow Problem is used more frequently than Maximum Cost Flow Problem, in the following we also say that a Minimum Cost Flow Problem has to be solved to compute a wirelength optimal placement out of the relations of a relative placement. Such an instance can be obtained by negating the weights of the instance.

5.4 Optimality

The results of Section 5.3 directly imply an important lemma.

Lemma 12. *Every feasible, nearly integral instance of the floorplanning problem has an optimum solution where all coordinates are integral.*

Proof. As the dual of the linear program that minimizes wirelength for a given relative placement is a Minimum Cost Flow Problem, it exists an integral optimum solution, if the input is integral [9]. □

This lemma also is reason to the fact, why we restrict the instances of Spark to be nearly integral, now we can proof Spark's optimality:

Corollary 13. Spark *computes an optimal solution of the floorplanning problem.*

Proof. The former lemma proves that we can restrict the solution space of Spark to only include integral placements. Thus, it is shown that the constraints that are included in Section 4.2.2 for branching are really the negations of the non-overlap constraints. As these constraints are again of the form $v_j - v_i \geq a_{ij}$, with a_{ij} is a constant and v_i, v_j are decision variables, the dual of the linear program definition stays a Minimum Cost Flow Problem. □

5.5 Runtime

While Spark is not a polynomial-time algorithm, it can be shown that it stops after a finite number of steps, whereby each step can be implemented to run in polynomial time.

Theorem 14. Spark *terminates after* $4^{\binom{n}{2}} \cdot 4^{|\mathcal{B}|n}$ *steps and a single step can be implemented with runtime* $O(p \log m(m + p \log p))$*, where* $p = O(|\mathcal{N}| + n)$ *and* $m = O(n^2 + |\mathcal{P}|)$.

Proof. The branch and bound approach branches at most $4^{\binom{n}{2}} \cdot 4^{|\mathcal{B}|n}$ times. A single step of the algorithm refers to one of the three operations:

- The consistency check that can be implemented with the algorithm of Floyd and Warshall [12, 46, 17] to run in $\mathcal{O}(n^3)$,
- The computation of the transitive closure, which again is implemented with a runtime of $\mathcal{O}(n^3)$ using the algorithm of Floyd and Warshall,
- The determination of the wirelength optimal placement satisfying the current relative placement, i.e., solving a Minimum Cost Flow Problem. This can be done with the algorithm of Orlin [37] in a runtime of $O(p \log m(m + p \log p))$.

□

The rest of this section is dedicated to techniques that will speed up the runtime of single steps.

5.5.1 Simplification of the Netlist

By Theorem 14 the runtime of Spark depends on both the number of pins and the number of nets. Decreasing these numbers will lead to a reduced runtime of the algorithm in practice. When Spark is called within the framework of our placement tool, usually small sections of the whole chip area are processed. Section 6.4 provides an overview over such a flow. As a result, many pins of the netlist are located outside of these regions and therefore contribute the same amount to the wirelength no matter how the placement within this section changes and thus do not have an impact on the wirelength optimal solution. This motivates the simplification of the netlist as it is done in the following. First of all, two definitions are needed.

Definition 15. For a net $N \in \mathcal{N}$ we define its *core* $\mathrm{IO}(N)$ to be the bounding box of the IO pins of N, and $R(N) := \mathrm{IO}(N) \cap C$, where $C := [0, W] \times [0, H]$ denotes the section of the chip area a placement should be computed for.

Definition 16. Given a netlist $\mathcal{N}$, we define the *reduced netlist* $\mathcal{N}$' as the netlist obtained from $\mathcal{N}$ by iterating each of the following three steps until no step can be applied anymore:

- If a net $N \in \mathcal{N}$ consists of more than two IO pins, then replace its IO pins by two pins in the lower left and upper right corners of $R(N)$.
- If there exists a net $N \in \mathcal{N}$ with $R(N) = C$, then remove N from the netlist.
- If there exist two nets $N_1, N_2 \in \mathcal{N}$ and a bijection $\sigma : N_1 \to N_2$ with $\gamma(p) = \gamma(\sigma(p))$, $x(p) = x(\sigma(p))$, $y(p) = y(\sigma(p))$, and $R(N_1) = R(N_2)$, then remove N_2 from the netlist and set the weight from N_1 to $c(N_1) + c(N_2)$.

Obviously it holds:

Lemma 17. *Let $\mathcal{I}$ and $\mathcal{I}'$ be instances of the floorplanning problem such that $\mathcal{I}'$ results from $\mathcal{I}$ by replacing the netlist by the reduced netlist. Then $\mathcal{I}$ and $\mathcal{I}'$ have got the same sets of optimum solutions.*

As replacing the netlist of an instance of the floorplanning problem by its reduced netlist does not influence the optimum solution of Spark, the steps of Definition 16 are implemented in a preprocessing step that is executed before calling Spark.

5.5.2 Integration of the Network Simplex Algorithm

This section focuses on the function $\mathrm{HPWL}(\rho)$ that is called in lines 3 and 10 of Spark. In this step, a Minimum Cost Flow Problem has to be solved. Although the algorithm of Orlin [37] achieves the best known theoretical runtime, we prefer for practical reasons the Network Simplex Algorithm of Cunningham and Dantzig [7], [8] in our implementation. Within Spark, the Network Simplex Algorithm should be incrementally invoked, such that the special structure of our instances resulting from the underlying branch and bound method is combined with the properties of the Network Simplex Algorithm. Experimental results have proved that the approach to incrementally invoke the Network Simplex Algorithm compared to a normal call of it improves the runtime of Spark on instances consisting of many rectangles about a factor of 16 and more. We first repeat some important definitions following the notations of [23] that are needed afterwards to show how the output of the Network Simplex Algorithm can be reused as an input for the next time it is invoked within Spark. Throughout this section, we assume that each graph is connected. This property is included in the following definitions.

Definition 18. Let (G, u, b, a) be an instance of the Minimum Cost Flow Problem. A b-flow f in (G, u, b, a) is called *spanning tree solution*, if

$$H := (V(G), \{e \in E(G) : 0 < f(e) < u(e)\})$$

does not contain any cycle. A triple (r, T, f) is called *feasible spanning tree solution*, if:

- $r \in V(G)$,
- $T \subseteq E(G)$ such that $(V(G), T)$ is a spanning tree,
- f is a b-flow.

An edge $e = (v, w) \in E(T)$ is said to be a *downward* edge, if v belongs to the undirected r-w-path in T. Otherwise we say e is an *upward* edge. A feasible spanning tree solution is called *strongly feasible*, if:

- $0 < f(e)$ holds for downward edges,
- $f(e) < u(e)$ holds for upward edges.

The recursive defined function $\pi : V(G) \to \mathbb{R}$

$$\pi(w) := \begin{cases} 0, & \text{for } w = r, \\ \pi(v) + c((v, w)), & \text{for } (v, w) \in E(T), \\ \pi(v) - c((w, v)), & \text{for } (w, v) \in E(T). \end{cases}$$

is called *potential associated with the spanning tree structure* (r, T, f).

Besides an instance of a Minimum Cost Flow Problem, the Network Simplex Algorithm expects a strongly feasible spanning tree structure as an input. Then, its output comprises of a strongly feasible spanning tree structure (r, T, f), with f is an optimum b-flow. In the following, it is shown how to incrementally invoke the Network Simplex Algorithm for a network (G, u, b, a) that is obtained by considering the dual of the linear program formulation of the floorplanning problem given in Section 5.3.
Thus, we first recall the structure of the network (G, u, b, a). It contains a node $\bar{v}_0$ associated to the auxiliary variable v_0 of the Linear Program 3, and four nodes x_N^-, x_N^+, y_N^- and y_N^+ associated to the left, right, lower and upper border of a net $N \in \mathcal{N}$, each rectangle $r_i \in \mathcal{R}$ induces two nodes x_{r_i}, y_{r_i} referring to its lower left corner. While the capacity is unrestricted, i.e., $u \equiv \infty$, the balance value of a node $v \in V$ is set according to:

$$b(v) := \begin{cases} w(N), & v \in \{x^-(N), y^-(N)\} \\ -w(N), & v \in \{x^+(N), y^+(N)\} \\ 0, & \text{otherwise.} \end{cases}$$

Let us come back to the input the Network Simplex Algorithm expects, i.e., how to obtain a strongly feasible spanning tree structure for a network (G, u, b, a). It is well-known, that every graph G can easily be modified such that an initial spanning tree structure is obtained by inserting $|V(G)|$ auxiliary edges and one auxiliary node. The following Lemma demonstrates how such a spanning tree structure can be obtained for our specific network without inserting nodes or edges.

Lemma 19. *Let $\mathcal{I}$ be an instance of the floorplanning problem and let (G, u, b, a) denote the appropriate Minimum Cost Flow Problem instance, i.e., the dual of the floorplanning problem formulation. A strongly feasible spanning tree structure for G can be computed with a runtime of $O(n + |\mathcal{P}|)$.*

Proof. The problem decomposes for horizontal and vertical dimensions. This proof is restricted to the horizontal dimension, but it can be applied analogously to the vertical dimension. Let (G_{hor}, u, b, a) be the network containing only horizontal relations. Then, the set $T \subseteq E(G_{hor})$ is defined as follows:

$$\begin{aligned} T &:= T_1 \cup T_2 \text{ with} \\ T_1 &:= \{ (x_{r_i}, \bar{v}_0) \mid r_i \in \mathcal{R} \} \\ T_2 &:= \{ (x_{\gamma(P)}, x_N^-), (x_N^+, x_{\gamma(P)}) \mid \\ &\qquad N \in \mathcal{N} : \exists P \in N : \gamma(P) \in \mathcal{R} \} \end{aligned}$$

Furthermore we define a function $f : E(G_{hor}) \to \mathbb{R}^+$ according to:

$$f((v,w)) := \begin{cases} w(N), & \text{if } w = x_N^- \text{ or } v = x_N^+ \text{ for } N \in \mathcal{N} \\ 0, & \text{otherwise.} \end{cases}$$

Claim 20. $(\bar{v}_0, T, f)$ *is a strongly feasible spanning tree structure.*

Proof. First we show that T is a spanning tree. This statement follows as $(V(G), T)$ is connected and $|T| = n + 2|\mathcal{N}| = |V(G)| - 1$.
The function f is chosen in such a way that it is a b-flow. Thus, it has to be shown that

1. $0 < f(e)$ holds for downward edges,
2. $f(e) < u(e)$ holds for upward edges.

Because $u \equiv \infty$ and $f(e) > 0$ for $e \in T_2$, f satisfies this properties on T_2. As per definition T_1 is chosen such that it only contains upward edges. The claim follows as $f\big|_{T_1} \equiv 0$. (Claim)$\square$

The set T can be computed in a runtime of $O(n + |\mathcal{P}|)$ and f can be computed in a runtime of $O(n + |\mathcal{N}|)$. (Lemma)$\square$

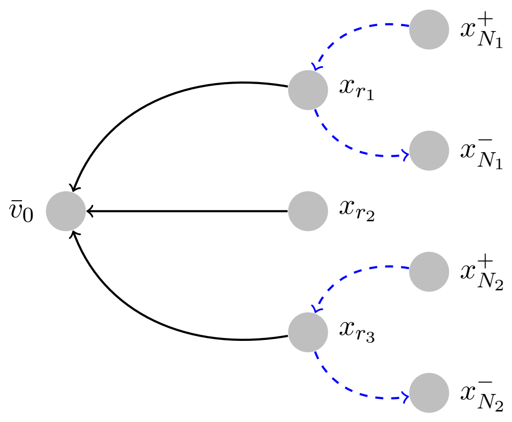

Figure 1: The strongly feasible spanning tree structure that is constructed in Lemma 19.

Figure 1 shows an initial tree solution due to Lemma 19 for an instance containing three rectangles r_1, r_2, r_3 and two nets N_1, N_2, whereby N_1 contains a pin located on r_1 and N_2 a pin located on r_3. While f on the consistent lines equals zero, the dashed lines are related to arcs having a positive, non-zero flow.
Due to Lemma 19 an initial spanning tree solution is obtained for the first time the Network Simplex Algorithm is called. Afterwards, we can incrementally invoke the Network Simplex Algorithm within Spark by using the output of a former step as an input for the next branching step. As every branching step only adds more constraints to the instance, what creates more edges and thus does not destroy the property of the former output to be a strongly feasible spanning tree structure. Only if it is backtraced within Spark, edges are deleted through removing constraints. Thus, only in case of a backtrack, in which an edge is deleted that is part of the former spanning tree solution, it has to be thrown away and a new initial spanning tree structure has to be computed (as it is shown by Lemma 19) to call the Network Simplex Algorithm in the next step.
A placement for $\mathcal{I}$ can be extracted out of the solution of the Network Simplex Algorithm as follows:

Remark 21. *Let $(\bar{v}_0, T, f)$ be the output of a call of the* Network Simplex Algorithm *and let π denote the potential associated with the spanning tree structure. If all rectangles $r_i \in \mathcal{R}$ are assigned to locations $(\pi(x_{r_i}), \pi(y_{r_i}))$, this leads to a wirelength optimal placement.*

A further simple conclusion can be drawn regarding instances all pins have the same offsets:

Corollary 22. *Let $(\bar{v}_0, T, f)$ be the strongly feasible spanning tree structure as it is constructed in Lemma 19. f is optimal, if all pin offsets are equal.*

Proof. The solution of Lemma 19 places the rectangles on the lower left corner of C. The wirelength of this solution equals zero and therefore is minimal. We get the statement by the duality theorem of [44, 15]. □

6 Experimental Results

6.1 MCNC Benchmarks

The most common benchmark instances for floorplanning problems are the MCNC block packing instances [24]. The set contains five block packing instances, the details of which are given in Table 1.
Three of these instances are sufficiently small so that our algorithm computes provably optimal solutions in reasonable time. Figure 2 shows wirelength optimal packings of the three smallest instances apte, xerox, and hp, optimal wirelengths and runtimes are given in Table 2. The table also includes the results of a second version of the same instances for which rescaling of the die sizes is allowed. In this version, one looks for the scaled die, which includes scaled positions of the IO pads, for which the minimum wirelength is as small as possible. We were able to find optimal solutions to this problem as well, details are given in [14]. Rotation or flipping of blocks is not allowed in these benchmarks.

This paper appeared in: Integration, the VLSI Journal 52 (2016), 355-366

instance	number of				die area	block area	whitespace
	blocks	IO-pads	pins	nets			
apte	9	73	214	97	10,500 × 10,500	46,561,628	57.77%
xerox	10	2	696	203	5,831 × 6,412	19,350,296	48.25%
hp	11	45	264	83	4,928 × 4,200	8,830,584	57.34%
ami33	33	42	480	123	2,058 × 1,463	1,156,449	61.59%
ami49	49	22	931	408	7,672 × 7,840	35,445,424	41.07%

Table 1: Characteristics of the MCNC block instances. Areas are given in μm^2.

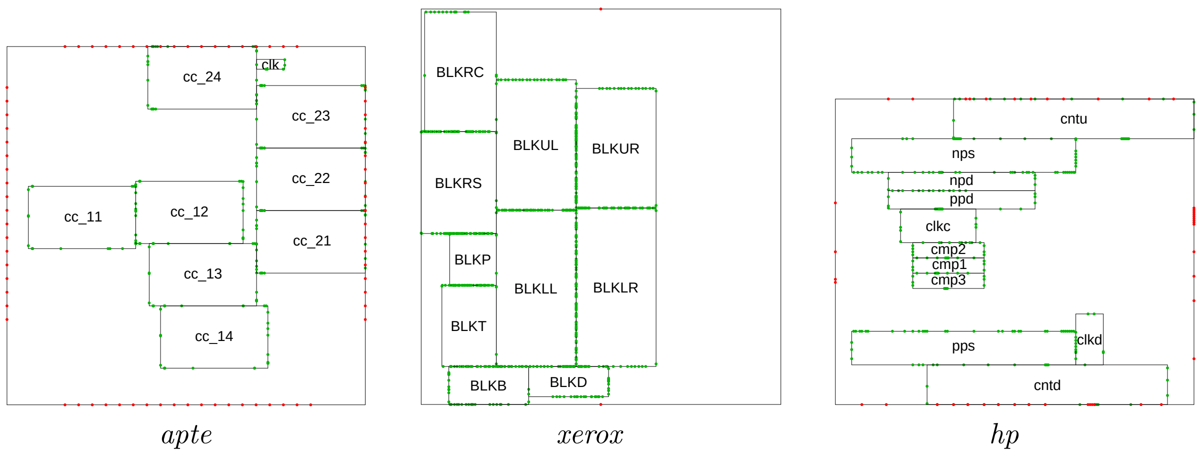

Figure 2: Wirelength optimal block packings of *apte*, *xerox*, and *hp* with placement area, pin locations (green dots), and IO-pad locations (red dots) as defined in the original `yal` files. Rotation or flipping of blocks is not allowed.

Our experiments show that often wirelengths are reported in the literature which are either much larger or even smaller than the optimum. Reasons for the inconsistencies in these reports include contradictory numbers in the original data as well as problems with the conversion of the original data into more recent file formats. An extensive analysis of these benchmarks is given in [14].

6.2 Synthetic Benchmarks

While we are not aware of any synthetic benchmarks used in the literature to test exact floorplanning tools, we generated a class of instances based on the square benchmarks regularly used to evaluate rectangle packing methods. The instance SQn consists of the n squares with edge lengths $1, \ldots, n$ and $n-1$ nets connecting the lower left corners of the squares with edge lengths w and $w+1$ for $1 \leq w < n$. The regions available for the placement are the minimum-area boxes reported in [32]. The runtimes needed to find optimal solutions for these instances are stated in Table 3.

Instance	fixed outline			rescaling allowed		
	original size	wirelength	runtime	optimal size	wirelength	runtime
apte	10,500 × 10,500	513,061	13 s	6,372 × 7,608	404,510	2 s
xerox	5,831 × 6,412	370,993	48 s	3,808 × 6,139	370,930	2 s
hp	4,928 × 4,200	153,328	102 s	4,263 × 3,108	143,302	100 s

Table 2: Optimal wirelengths for *apte*, *xerox*, and *hp* for the original die size and when rescaling of the die is allowed. Runtimes were measured on an Intel X5680 CPU at 3.33 GHz and refer to the instances with the given die sizes.

Instance	*SQ11*	*SQ12*	*SQ13*	*SQ14*	*SQ15*	*SQ16*	*SQ17*	*SQ18*	*SQ19*	*SQ20*
Box size	19 × 27	23 × 29	22 × 38	23 × 45	23 × 55	27 × 56	39 × 46	31 × 69	47 × 53	34 × 85
Runtime	0 s	0 s	0 s	1 s	1 s	25 s	15 s	521 s	1,392 s	7,669 s

Table 3: Runtimes to find optimal floorplans using Spark for the synthetic *SQn* benchmarks. Runtimes were measured on an Intel X5680 CPU at 3.33 GHz and refer to the instances with the given die sizes.

6.3 Industrial Testcase

The largest real-life instance that has been solved with our algorithm is depicted in Figure 3. It is an actual VLSI instance provided by our industry partners at IBM and contains 27 rectangles, 6 of which are fixed (dark gray), with 8837 pins and 3661 nets. However, to find an optimal solution additional information was used on the instance's structure. In particular, the bin-like structure of the blockages were exploited by enumerating assignments of movable blocks to the five regions between the blockages. The depicted floorplan provably optimizes the half-perimeter wirelength.

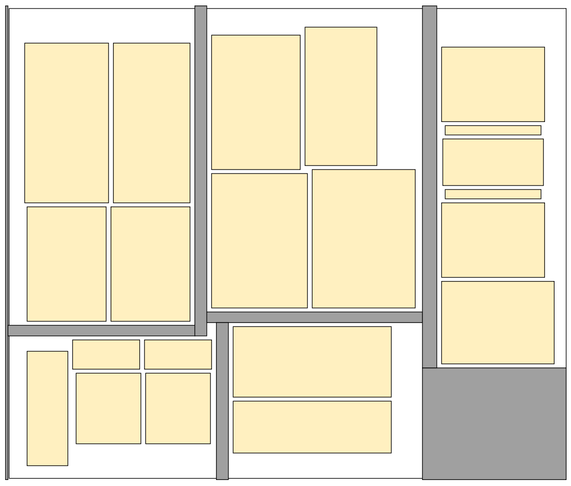

Figure 3: Optimal floorplan for an IBM core macro with 27 circuits, including 6 blockages.

Paper	*apte*	*xerox*	*hp*	*ami33*	*ami49*	Remarks
this	513,061	370,993	153,328	58,627	640,509	Packings for *apte*, *xerox*, and *hp* are optimal
[29]	614,602	404,278	253,366	96,205	1,070,010	–
[38]	545,136	755,410	155,463	63,125	871,128	Packings for *hp*, *ami33*, and *ami49* have overlaps

Table 4: Wirelengths for MCNC benchmark instances.

6.4 Larger Floorplans

As SPARK is designed to find optimal solutions, it is unable to find solutions for large instances in reasonable runtime. While it is possible to specify a runtime limit and return the best solution that is encountered until the limit is reached, not much can be said about the quality of the output in this case. For large instances—and thus for most of the applications occurring in practical VLSI design—we therefore propose a flow, called BONNMACRO, that uses the presented algorithm as a subroutine to place local groups of circuits optimally.

The first step of BONNMACRO is to generate a wirelength-optimal placement that ignores the non-overlap constraints, for example by solving a linear program. After this step, overlaps will occur that have to be resolved. To do so, a routine `Place(`R`)` is called for every $R \in \mathcal{R}$ and set $\mathcal{F}$ of *fixed* circuits is maintained that is initially empty. `Place` chooses a group of circuits including R and some additional circuits in $\mathcal{R}$ in its proximity. This group is then placed using SPARK presented above, but in addition to the blockages specified in the input, all fixed circuits are also considered as blockages. Furthermore, the region available for the placement is restricted to a local neighborhood of R so that the number of blockages is restricted. Then, R is added to the set of fixed circuits. In cases where SPARK fails, a fallback mechanism moves single rectangles to the closest available position.

After `Place` has been called for every $R \in \mathcal{R}$, the circuits have been legalized in the sense that no pair of circuits overlaps with their interiors (unless the algorithm failed in some step). Afterwards, a post-optimization phase chooses local groups of circuits and finds optimal solutions with SPARK, treating all non-chosen circuits as blockages. This procedure does not introduce overlaps and might improve the wirelength. An example of a unit processed by BONNMACRO is shown in Figure 4.

While in theory no statements can be made about the quality of the algorithm's output, and it may even fail in cases where feasible solutions exist, it performs very well in practice. On real-life instances provided by our industry partner IBM, the method almost always finds solutions which are very competitive. The tool is in regular use during the design of ASICs and server chips at IBM. Details of an earlier version of BONNMACRO are provided in [39].

6.5 ami33 and ami49

The two MCNC benchmarks that are too large to be solved optimally with our algorithm are *ami33* and *ami49*. To test the applicability of SPARK in such cases, we processed them with the tool described in the previous section. Even though the MCNC instances are frequently used by the rectangle packing and floorplanning community, we only found two papers, [29] and [38], that contained comparable wirelengths—and one of them included solutions with overlaps. The

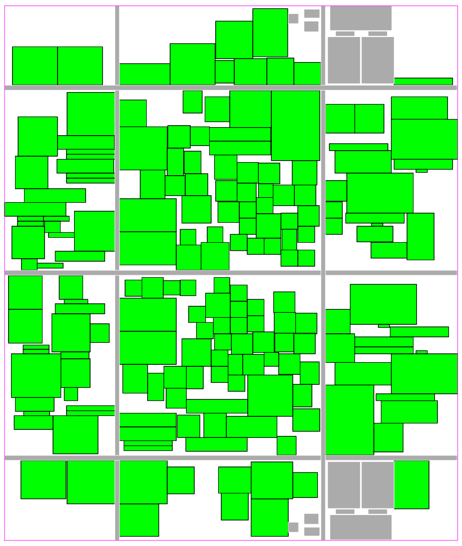

Figure 4: Floorplan for a real-life IBM core macro with 210 circuits, including 21 blockages.

BonnMacro placements are depicted in Figure 5, wirelength comparisons are given in Table 4. The runtime required to generate these placements was 13 seconds for *ami33* and 73 seconds for *ami49* on an Intel E5-2667 CPU at 3.30 GHz.
The small number of comparisons has several reasons. Many works present packing algorithms and do not discuss wirelength minimization. Moreover, works that do address wirelength most often do not use the original die area but scale or completely change it in order to reduce whitespace [4, 5, 6, 11, 20, 28, 43, 49]. In these cases it is usually not clear how the locations of the original IO pads are adjusted or if they are included at all. Among the few papers that do use the fixed outline from the original MCNC files, one allowed rotation of the blocks and is thus incomparable, too [27]. In some cases, placement objects are considered soft, meaning that their aspect ratio may be altered from the original shape, and, lastly, some authors only consider slicing floorplans—a restriction that we do not impose.

7 Conclusion

In this paper, we present a powerful algorithm to compute wirelength-optimal floorplans. This approach differs from most available literature in the sense that existing approaches either try to solve the rectangle packing problem optimally, leaving out the wirelength optimization aspect, or use heuristics to solve large instances without finding optimal solutions (or giving bounds for the quality of their solutions at all). The largest reported result for a provably optimal solution of the floorplanning problem that we are aware of has only 6 rectangles [36].
Our algorithm, however, is able to solve synthetic floorplanning instances with up to 20 rectangles in roughly 2 hours and, with some instance-specific tuning, a real-world chip with 27 rectangles, 6 of which are blockages, and thousands of nets. Moreover, we were able to generate the first provably netlength optimal floorplans for 3 of the 5 MCNC block packing instances that are

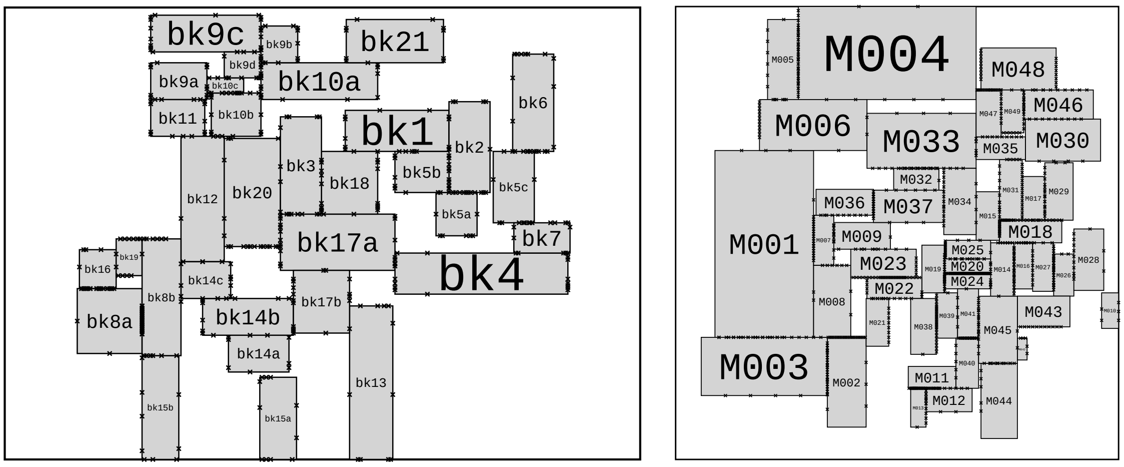

Figure 5: Block packings of *ami33* and *ami49*. Rotation or flipping of blocks is not allowed.

frequently discussed in the related literature.
Finally, we propose a framework in which our optimal algorithm can be used to find solutions to instances which are too large to be solved optimally. This framework is in constant use at our industry partner IBM and generates good solutions in practice. It is able to generate placements for the 2 large MCNC instances that have shorter wirelengths than any previous comparable result.

8 Acknowledgements

The authors wish to express their gratitude towards the anonymous reviewers for their valuable comments and suggestions and further thank IBM Corporation, the industry partner of the Research Institute for Discrete Mathematics, and in particular Peter Verwegen for the opportunity to integrate the algorithm in an industrial tool and an overall successful and constructive collaboration.

9 Bibliography

References

[1] M.F. Anjos and A. Vannelli. An attractor-repeller approach to floorplanning. *Mathematical Methods of Operations Research*, 56(1):3–27, 2002.

[2] A. Bortfeldt. A reduction approach for solving the rectangle packing area minimization problem. *European Journal of Operational Research*, 224(3):486–496, 2013.

[3] Tung-Chieh C. and Yao-Wen C. Handbook of algorithms for physical design automation. In Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, editors, *Packing Floorplan Representations*, pages 203–238. CRC Press, 2009.

[4] H.H. Chan and I.L. Markov. Practical slicing and non-slicing block-packing without simulated annealing. In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 282–287, New York, 2004. ACM.

[5] G. Chen, W. Guo, and Y. Chen. A pso-based intelligent decision algorithm for vlsi floorplanning. *Soft Computing*, 14(12):1329–1337, 2010.

[6] X. Chen, J. Hu, and N. Xu. Regularity-constrained floorplanning for multi-core processors. In *Proceedings of the 2011 International Symposium on Physical Design*, ISPD '11, pages 99–106, New York, 2011. ACM.

[7] W.H. Cunningham. A network simplex method. *Mathematical Programming*, 11:105–116, 1976.

[8] G.B. Dantzig. Application of the simplex method to a transportation problem. *Activity Analysis of Production and Allocation*, pages 359–373, 1951.

[9] G.B. Dantzig and D.R. Fulkerson. On the max flow min cut theorem of networks. In *Linear Inequalities and Related Systems*, pages 215–221. Princeton University Press, 1956.

[10] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[11] P. Fernando and S. Katkoori. An elitist non-dominated sorting based genetic algorithm for simultaneous area and wirelength minimization in vlsi floorplanning. In *21st International Conference on VLSI Design*, pages 337–342, 2008.

[12] R.W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[13] J. Funke. Netzlängenoptimale Platzierung von VLSI-Chips. Diploma thesis, University of Bonn, 2011.

[14] J. Funke, S. Hougardy, and J. Schneider. Wirelength optimal rectangle packings. In *Proceedings of the fourth International Workshop on Bin Packing and Placement Constraints*, Nantes, France, 2012.

[15] D. Gale, H.W. Kuhn, and A.W. Tucker. Linear programming and the theory of games. In *Activity Analysis of Production and Allocation*, pages 317–329, New York, 1951. Wiley.

[16] S. Hougardy. A scale invariant algorithm for packing rectangles perfectly. In *Proceedings of the fourth International Workshop on Bin Packing and Placement Constraints*, Nantes, France, 2012.

[17] Stefan Hougardy. The floyd–warshall algorithm on graphs with negative cycles. *Information Processing Letters*, 110(8):279–281, 2010.

[18] E. Huang and R.E. Korf. Optimal rectangle packing: An absolute placement approach. *CoRR*, abs/1402.0557, 2014.

[19] M. Jerrum. Complementary partial orders and rectangle packing. Technical Report CSR-190-85, University of Edinburgh, Department of Computer Science, 1985.

[20] Y. Kimura and K. Ida. Improved genetic algorithm for vlsi floorplan design with non-slicing structure. *Computers & Industrial Engineering*, 50(4):528–540, 2006.

[21] R.E. Korf. Optimal rectangle packing: Initial results. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, 2003.

[22] R.E. Korf, M.D. Moffitt, and M.E. Pollack. Optimal rectangle packing. *Annals of Operations Research*, 179(1):261–295, 2010.

[23] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms (Algorithms and Combinatorics)*. Springer, Berlin, 5th edition, 2012.

[24] K. Koźmiński. Benchmarks for layout synthesis — evolution and current status. In *28th ACM/IEEE Design Automation Conference*, pages 265–270, 1991.

[25] C.E. Leiserson and J.B. Saxe. A mixed-integer linear programming problem which is efficiently solvable. *Journal of Algorithms*, 9(1):114–128, 1988.

[26] Y.-Z. Liao and C.-K. Wong. An algorithm to compact a vlsi symbolic layout with mixed constraints. In *Proceedings of the 20th Design Automation Conference*, pages 107–112. IEEE Press, 1983.

[27] J.-M. Lin and Y.-W. Chang. Tcg: A transitive closure graph-based representation for general floorplans. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 13(2):288–292, 2005.

[28] P.-H. Lin, Y.-W. Chang, and S.-C. Lin. Analog placement based on symmetry-island formulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(6):791–804, 2009.

[29] W. Liu and A. Nannarelli. Net balanced floorplanning based on elastic energy model. In *NORCHIP*, pages 258–263, 2008.

[30] J. Maßberg and J. Schneider. Rectangle packing with additional restrictions. *Theoretical Computer Science*, 412(50):6948–6958, November 2011.

[31] R.D. Meller and K.-Y. Gau. The facility layout problem: recent and emerging trends and perspectives. *Journal of manufacturing systems*, 15(5):351–366, 1996.

[32] M.D. Moffitt and E. Pollack. Optimal rectangle packing: A meta-csp approach. In *ICAPS*, pages 93–102, 2006.

[33] J.W. Moon and L. Moser. Some packing and covering theorems. In *Colloquium Mathematicae*, volume 17, pages 103–110. Institute of Mathematics Polish Academy of Sciences, 1967.

[34] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. Rectangle-packing-based module placement. In *International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 472–479. IEEE/ACM, 1995.

[35] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module placement on bsg-structure and ic layout applications. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 484–491, Washington, D.C., 1996. IEEE Computer Society.

[36] H. Onodera, Y. Taniguchi, and K. Tamaru. Branch-and-bound placement for building block layout. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, DAC '91, pages 433–439, New York, 1991. ACM.

[37] J. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *STOC '88: Proceedings of the 20th annual ACM symposium on Theory of computing*, pages 377–387, New York, 1988. ACM.

[38] M. Samaranayake, H. Ji, and J. Ainscough. Development of a force directed module placement tool. In *Research in Microelectronics and Electronics*, pages 152–155, 2009.

[39] J. Schneider. Macro placement in vlsi design. Diploma thesis, University of Bonn, 2009.

[40] R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28:769–779, 1981.

[41] T.-Y. Sun, S.-T. Hsieh, H.-M. Wang, and C.-W. Lin. Floorplanning based on particle swarm optimization. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, ISVLSI '06, page 7, Washington, D.C., USA, 2006. IEEE Computer Society.

[42] S. Sutanthavibul, E. Shragowitz, and J.B. Rosen. An analytical approach to floorplan design and optimization. *IEEE Transactions on Computer-Aided Design*, 10:761–769, 1991.

[43] M. Tang and A. Sebastian. A genetic algorithm for vlsi floorplanning using o-tree representation. In *Applications of Evolutionary Computing*, volume 3449 of *Lecture Notes in Computer Science*, pages 215–224. Springer Berlin Heidelberg, 2005.

[44] J. von Neumann. Discussion of a maximum problem. Working Paper. Published in: John von Neumann, Collected Works; Vol VI (A.H. Taub, ed.), Pergamon Press, Oxford 1963, 1947.

[45] J. Vygen. *Platzierung im VLSI-Design und ein zweidimensionales Zerlegungsproblem.* Dissertation, University of Bonn, 1996.

[46] T. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962.

[47] Jackey Z Yan and Chris Chu. Defer: deferred decision making enabled fixed-outline floorplanning algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):367–381, 2010.

[48] E.F.Y. Young. Floorplan representations. In Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, editors, *Handbook of Algorithms for Physical Design Automation*, pages 185–202. CRC Press, 2009.

[49] S. Zhou, S. Dong, C.-K. Cheng, and J. Gu. Ecbl: An extended corner block list with solution space including optimum placement. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01, pages 150–155, New York, 2001. ACM.

[50] C. Zhuang, K. Sakanushi, L. Jin, and Y. Kajitani. An enhanced q-sequence augmented with empty-room-insertion and parenthesis trees. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 61–68, 2002.