1992

Minimizing Total Wire Length During 1-Dimensional Compaction

Susanne E. Hambrusch
Purdue University, seh@cs.purdue.edu

Hung-Yi Tu

Report Number:
92-033

Hambrusch, Susanne E. and Tu, Hung-Yi, "Minimizing Total Wire Length During 1-Dimensional Compaction" (1992). *Department of Computer Science Technical Reports*. Paper 955.
<https://docs.lib.purdue.edu/cstech/955>

MINIMIZING TOTAL WIRE LENGTH DURING
1-DIMENSIONAL COMPACTION

Susanne E. Hambruch
Hung-Yi Tu

CSD-TR-92-033
May 1992

Minimizing Total Wire Length During 1-Dimensional Compaction

Susanne E. Hambrusch *

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Hung-Yi Tu †

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

May 28, 1992

Abstract

Minimizing the total wire length is an important objective in VLSI layout design. In this paper we consider the problem of minimizing the total wire length during 1-dimensional (1-D) compaction. Assume we are given a layout containing n_h horizontal wires, n_v vertical wires, and rectilinear polygonal layout components composed of n_r vertical edges. We present an $O(n_h \cdot n \log n)$ time algorithm for generating, from the constraint graph corresponding to the initial layout, a layout of minimum total wire length, $n \leq n_v + n_r$. Our algorithm generates, among all the layouts having minimum total wire length, one of minimum layout width, assuming that compaction is done along the horizontal direction. We also consider a number of other compaction problems in which a relationship between the layout width and the total wire length is specified. For example, we present an $O(n_h \cdot n \log n)$ time algorithm to determine a layout minimizing the objective function $\alpha \cdot w + \beta \cdot l$, where w is the layout width and l is the total wire length, $\alpha, \beta > 0$; i.e., we consider optimizing a tradeoff function between the layout width and the total wire length.

Keywords: Analysis of algorithms, constraint graphs, layout width, 1-dimensional compaction, wire length, VLSI layouts.

*Research supported in part by ONR under contracts N00014-84-K-0502 and N00014-86-K-0689.

†Research supported in part by ONR under contract N00014-84-K-0502.

1 Introduction

The process of converting a symbolic layout into an actual layout satisfying the design rules and minimizing a set of objective functions is known as compaction [3, 5, 7, 9]. Minimizing the area is generally considered the most crucial objective function, with minimizing the wire length being the next relevant one. In this paper we present new and efficient 1-dimensional compaction algorithms that minimize the area as well as the total wire length. W.l.o.g. we assume that compaction is done in the horizontal direction. Hence, minimizing the area of the layout is equivalent to minimizing the layout width, and minimizing the total wire length is equivalent to minimizing the total length of the horizontal wires. We present an algorithm that generates, from a given constraint graph, a layout of minimum total wire length having minimum width among all layouts having this minimum total wire length. We also consider a number of other relationships between the layout width and the total wire length and present efficient algorithms for the following problems:

- Given a width w , find a layout of minimum total wire length among all layouts having width at most w .
- Given a length l , find a layout of minimum width among all layouts having total wire length at most l .
- Given a tradeoff function $\alpha \cdot w + \beta \cdot l$ between the width w and the total wire length l , and constants $\alpha, \beta > 0$, find a layout minimizing $\alpha \cdot w + \beta \cdot l$. Conceptually, decreasing the width by β units is equal to decreasing the total wire length by α units.

Like many layout problems, the objective of compaction can be phrased in terms of well-studied combinatorial problems. In [4, 5, 8] the problem of minimizing the total wire length during 1-D compaction is formulated in terms of a minimum cost network flow problem. Using these formulations and the fastest known algorithms for network flow [2], results in an $O(n_h \cdot n \log n)$ time algorithm for minimizing the total wire length, where n is the number of vertices in the constraint graph and n_h is the number of horizontal wires in the layout. Compaction algorithms that can solve the system of inequalities arising from the compaction constraints by solving shortest path computations are often referred to as graph-based compaction algorithms [5]. In [11, 13], graph-based compaction algorithms minimizing the total wire length are described, but

the algorithms have an exponential worst-case running time. Graph-based compaction algorithms provide insight into the compaction process that is different from the insight gained from a network flow formulation. In this paper we present an $O(n_h \cdot n \log n)$ time graph-based algorithm that generates a layout of minimum total wire length having minimum width among all the layouts achieving the minimum total wire length. Our algorithm thus represents an alternative approach to the network flow formulation with same asymptotic running time. Our algorithm requires only elementary data structures and graph-theoretic concepts to achieve this running time. At the same time, it generalizes in a natural and intuitive way to the other compaction problems defined above.

Our algorithms assume that the layout elements in the layout area are either layout components of rectilinear polygonal shape or vertical and horizontal wires. During compaction, layout elements are allowed to slide horizontally as long as no two layout elements overlap and the relative order of the layout elements is preserved. A *configuration* of a layout assigns every layout element an x -position in the layout area. If not stated otherwise, we assume that a configuration is feasible (i.e., the x -positions associated with the layout elements result in a layout satisfying the constraints). During compaction we allow horizontal wires to change their length (i.e., they are considered to be flexible objects), but we do not consider the automatic insertion of jogs into vertical wires.

Assume we are given a configuration $\mathcal{C}$ consisting of n_h horizontal wires, n_v vertical wires, and rectilinear polygonal layout components composed of n_r vertical edges. Our wire-length minimizing algorithm generates a layout having minimum total wire length and whose width is minimized among all layouts having this total wire length. Our algorithm generates this layout in $O(n_h \cdot n \log n)$ time with an $O(n_h + (n_v + n_r) \log(n_v + n_r))$ preprocessing time for generating the constraint graph, where n is the number of vertices in the constraint graph. The main idea underlying our algorithm is the following. We identify groups of layout elements whose simultaneous movement to the right reduces the total wire length. We then move such groups of layout elements to the right as much as possible without changing the positions of the other layout elements and the topology of the layout. We iteratively perform these steps until the total wire length can no longer be reduced by moving layout elements to the right. This approach is conceptually similar to the ones used in [11, 13]. Our contribution lies in developing an efficient method for identifying layout elements whose movement to the right reduces the total wire length and which guarantees that this process terminates in at most n_h iterations.

The paper is organized as follows. In Section 2 we give the relevant definitions. In Section 3 we address the correctness of the approach used by our algorithms. Sections 4 and 5 present algorithms for partitioning problems on directed trees which are a crucial component of our wire-length minimizing algorithm. In Section 6 we describe the algorithm generating a configuration of minimum total wire length, prove its correctness and claimed running-time. Section 7 discusses how to generalize the approach used in our wire-length minimizing algorithm to solve other width/wire-length optimization problems. In Section 8 we discuss possible extensions and modifications in the underlying layout model.

2 Preliminaries and definitions

Let $\mathcal{C}$ be the initial configuration containing n_v vertical wires, n_h horizontal wires, and rectilinear polygonal layout components composed of n_r vertical edges. A vertical (resp. horizontal) wire is defined as a vertical (resp. horizontal) line segment which at each endpoint is incident to either a layout component or a horizontal (resp. vertical) wire and no endpoint of another wire is incident to any other position of this vertical (resp. horizontal) wire. Figure 1(a) also shows a partitioning of the wires into vertical and horizontal wires. The information about configuration $\mathcal{C}$ is represented by a constraint graph which contains the visibility information and the distances between layout elements in the horizontal direction. As will be discussed further in Section 8, our approach does not depend on the assumption that visibility and distances are the constraints.

The vertices of the constraint graph correspond to the *cells* of the layout which are determined as follows. Partition the layout components and wires into maximal sets, so that each set represents one rigid object that can only move as one entity. One such rigid object is called a *cell* of the layout. Since our algorithms treat the horizontal wires as flexible objects, cells contain only layout components and vertical wires. We introduce two fictitious rectangular cells having a height equal to the height of the layout. One of these cells, U_l , is positioned immediately to the left of the leftmost cell in the layout and the other one, U_r , is positioned immediately to the right of the rightmost cell. The distance between U_l and U_r in $\mathcal{C}$ corresponds to the width of the initial layout. The existence of these two cells will allow us to minimize and control the width of the generated configuration while minimizing the total wire length. Figure 1(b) shows the cells induced by the configuration shown in Figure 1(a). The dashed lines represent cells U_l and U_r .

(a)

(b)

(c)

Figure 1: (a) A layout. (b) The cells of the layout. (c) The constraint graph of the layout of (a).

Figure 2: Three wires forming a “step”.

Let n be the number of cells in configuration $\mathcal{C}$ to which we added U_l and U_r , $n \leq n_v + n_r + 2$. Let $G_{\mathcal{C}} = (U_{\mathcal{C}}, A_{\mathcal{C}})$ be the constraint graph corresponding to configuration $\mathcal{C}$ with $|U_{\mathcal{C}}| = n$. Throughout, vertex u_i represents cell U_i . Cell U_j is *visible* from cell U_i if and only if one can draw a horizontal line H connecting U_i and U_j such that H starts at position x_1 and ends at position x_2 , $x_1 \leq x_2$, H connects U_i and U_j , and no position between x_1 and x_2 on line H is occupied by a cell. Every horizontal line H has a length associated with it. In general, its length is $x_2 - x_1$. However, when position x_1 on cell U_i belongs to a vertical wire and position x_2 on cell U_j also belongs to a vertical wire such that the two vertical wires are connected by a horizontal wire forming a “step” (as shown in Figure 2), then the length of horizontal line H is the distance between the central lines of these two vertical wires. In a unit-grid model with unit-width wires, this length is $\overline{x_2 - x_1} + 1$. The reason for this definition of the distance in this particular situation is to allow the two vertical wires to collapse into one vertical wire should the length of the horizontal wire go to zero. An arc from vertex u_i to u_j in $G_{\mathcal{C}}$ models the fact that cell U_j is visible from cell U_i . Every edge (u_i, u_j) of the constraint graph has a nonnegative weight $w(u_i, u_j)$ assigned to it. The weight is the minimum over all distances associated with horizontal lines connecting cells U_i and U_j .

With each vertex u_i of the constraint graph we associate a value $pot(u_i)$, called its *potential*. For vertices u_r and u_l , we set $pot(u_r) = -\frac{1}{2}$ and $pot(u_l) = +\frac{1}{2}$. For any other vertex u_i , $pot(u_i)$ is equal to the number of the left endpoints minus the the number of right endpoint of the horizontal wires incident to cell U_i . In other words, a left endpoint of a horizontal wire incident to a cell contributes $+1$, a right endpoint contributes -1 , and the potential is the sum of the contributions. Figure 1(c) shows the constraint graph corresponding to the configuration shown in Figure 1(a). The entry above or below each vertex name represents the potential of the vertex.

Given a configuration, its constraint graph $G_C = (U_C, A_C)$ can be built in $O(n_h + (n_v + n_r) \log(n_v + n_r))$ time by using standard balanced tree operations [10, 12]. Vertex u_r (resp. u_l) is the sink (resp. source) in G_C . Furthermore, $|A_C| = O(n)$ and, if the constraint graph contains the arcs induced by visibility as described above, G_C is a planar graph that can contain cycles.

We next define a number of graph-theoretic terms that will be used throughout the paper. Let $G = (V, A)$ be a directed graph and $G' = (V, E)$ be its undirected version. We say G is *connected* if G' is connected (in the undirected sense). If G' is a tree, then we call G a *directed tree*. A *directed spanning forest* F of G is a subgraph of G consisting of mutually vertex-disjoint directed trees so that every vertex of G is in one directed tree. A vertex v_i is *reachable* from vertex v_j in G if there exists a directed path from v_j to v_i in G . We say that every vertex is reachable from itself. A non-empty subgraph $G_r = (V_r, A_r)$ (not necessarily connected) of graph G is called *right-splitable* from G if every vertex in V reachable from a vertex in V_r is also in V_r . A non-empty subgraph $G_l = (V_l, A_l)$ of graph G is called *left-splitable* from G if the subgraph of G induced by vertex set $V \setminus V_l$ is right-splitable from G .

Let (u_i, u_j) be an arc in the constraint graph G_C . If $w(u_i, u_j) = 0$, we say arc (u_i, u_j) is *tight*. Let $G_t = (V_t, A_t)$ be a connected subgraph of G_C containing only tight arcs. G_t is a *tight component* of G_C if G_t is a maximal connected subgraph containing only tight arcs. A vertex incident to no tight arc is considered to form a tight component by itself. The *potential* of G_t is defined by the summation of the potentials of all vertices in V_t ; i.e., $pot(G_t) = \sum_{v \in V_t} pot(v)$. G_t is *positive* (resp. *negative*) if the potential of G_t is positive (resp. negative). Figure 3 shows the tight component of the constraint graph in Figure 1(c). The subgraph induced by the vertices u_1, u_2, u_3 , and u_4 is a positive right-splitable subgraph of potential 3.

3 Correctness of overall approach

In this section we prove the correctness of a wire-length minimizing algorithm based on the idea of identifying positive right-splitable subgraphs in the tight components. Our wire-length minimizing algorithm uses this idea, but in order to achieve the claimed time bound, the approach is somewhat modified, as is discussed at the end of this section.

Consider a configuration $\mathcal{C}$ and its constraint graph G_C . Let G_t be a tight component of G_C , and let $G_{t,r}$ be a right-splitable subgraph of G_t . If $G_{t,r}$ is positive, the cells corresponding to the

Figure 3: The tight component of the constraint graph in Figure 1(c).

vertices in $G_{t,r}$ are incident to more left endpoints than right endpoints of horizontal wires. Hence, if we simultaneously move the cells corresponding to the vertices in $G_{t,r}$ a positive distance to the right, without changing the positions of the other cells and the relative order of the cells, the total wire length in C is reduced. Since $G_{t,r}$ is a positive right-splitable subgraph of a tight component, the cells corresponding to the vertices in $G_{t,r}$ can move at least one position to the right (without affecting the positions of the other cells). For example, the vertices corresponding to cells U_3 and U_4 form a positive right-splitable subgraph in the tight components of the constraint graph shown in Figure 1(c). Moving cells U_3 and U_4 one position to the right reduces the total wire length by 3. We summarize the above discussion in the following lemma.

Lemma 3.1 *Let G_t be a tight component of a constraint graph G_C and $G_{t,r}$ be a positive right-splitable subgraph of G_t . Then, moving the cells corresponding to the vertices in $G_{t,r}$ one position to the right results in a configuration of shorter total wire length.*

Clearly, if C^* is a configuration of minimum total wire length, then no tight component of its constraint graph contains a positive right-splitable subgraph. In order to use the existence of positive right-splitable subgraphs as a termination criterion for an algorithm, we need to show that, if a configuration is not a configuration of minimum total wire length, then at least one of its tight components contains a positive right-splitable subgraph. This is shown in the next lemma.

Lemma 3.2 *If configuration C is not a configuration of minimum total wire length, then there exists at least one tight component in the constraint graph G_C containing a positive right-splitable subgraph.*

Proof: Let $x_C(U_i)$ denote the x -position of the leftmost vertical edge of cell U_i in any configuration C . We first show that there exists a configuration C^* of minimum total wire length that contains a cell U_i such that $x_C(U_i) = x_{C^*}(U_i)$ and $x_C(U_j) \leq x_{C^*}(U_j)$ for all $j \neq i$. Let C^* be a configuration of minimum total wire length. Align configurations C and C^* so that cell U_i is at position 0 in both configurations; i.e., $x_C(U_i) = x_{C^*}(U_i)$. Let $dis(j) = x_{C^*}(U_j) - x_C(U_j)$ for every cell U_j . If all $dis(j)$'s are nonnegative, the claim holds with $i = l$. Hence, assume that some $dis(j)$'s are negative. Among the dis -values, let $dis(i)$ be the smallest one. Consider the configuration obtained by shifting all cells in C^* $-dis(i)$ positions to the right. It is easy to see that the new configuration, call it C' , is also one of minimum total wire length. Moreover, $x_C(U_i) = x_{C'}(U_i)$ and $x_C(U_j) \leq x_{C'}(U_j)$ for all $j \neq i$. Hence, the claim follows.

Let C^* be a configuration of minimum total wire length satisfying the above claim. By using the positions of the cells in C^* we show the existence of a positive right-splitable subgraph in configuration C . We define a sequence of configurations $C_0, C_1, C_2, \dots, C_k$ as follows. $C_0 = C$. Let $S_0 = \{U_j | x_C(U_j) < x_{C^*}(U_j)\}$; i.e., set S_0 contains all the cells in C_0 that need to move to the right in order to get to the position assigned in configuration C^* . Push all cells in S_0 simultaneously to the right until at least one cell reaches its position in C^* . The resulting configuration is C_1 . In general, for a configuration C_l , define $S_l = \{U_j | x_{C_l}(U_j) < x_{C^*}(U_j)\}$. When $S_l \neq \emptyset$, C_{l+1} is the configuration obtained by moving the cells in S_l simultaneously to the right until one cell reaches its position in C^* . Obviously, $C_k = C^*$.

Let V_l be the set of vertices in the constraint graph of G_C corresponding to the cells in S_l , $0 \leq l \leq k$. Since C is not a configuration having minimum total wire length, there must exist an l such that the total wire length is decreased when going from C_l to C_{l+1} . The subgraph induced by V_l has positive potential and, according to the construction of S_l , it is a right-splitable subgraph of G_C . Thus G_C contains a positive right-splitable subgraph. $\square$

The above two lemmas imply that an algorithm based on the idea of iteratively finding positive right-splitable subgraphs and generating a new configuration based on the movement to the right induced by these subgraphs, will eventually generate a configuration of minimum total wire length. Our wire-length minimizing algorithm does not exactly proceed this way. The reason is that we do not know of an efficient algorithm for determining a positive right-splitable subgraph in a tight

component. In our algorithm we represent every tight component G_t by a spanning forest F_t and find positive right-splitable subtrees in every tree of the spanning forest F_t . We show that positive right-splitable subtrees of maximum potential can be found in linear time for a given tree. Obviously, a right-splitable subtree of a tree in F_t does not necessarily correspond to a right-splitable subgraph of the tight component G_t . However, if G_t contains a positive right-splitable subgraph $G_{t,r}$ of potential $pot(G_{t,r})$, then there exists a set of right-splitable subtrees of total potential at least $pot(G_{t,r})$ in any spanning forest of G_t . We show how to apply, over a number of iterations, arc additions and arc deletions to the spanning forests representing the tight components so that we identify spanning forests whose right-splitable subtrees do indeed represent a right-splitable subgraph for G_t . We show that after at most n_h iterations, where n_h is the number of horizontal wires, with each iteration identifying positive right splitable subtrees, we generate a configuration of minimum total wire length. The next two sections describe algorithms that are applied to every tree in the spanning forests representing the tight components and which are a major component of our wire-length minimizing algorithm.

4 Finding positive right-splitable subtrees in a directed tree

Let $T = (V, A)$ be a directed tree in which every vertex has a potential associated with it. In this section we describe a linear time algorithm to determine a set of subtrees of T so that

- (i) every subtree in this set is a positive right-splitable subtree of T ,
- (ii) the subtrees are pairwise vertex-disjoint, and
- (iii) the potential of the set is a maximum over all sets of subtrees satisfying conditions (i) and (ii).

This algorithm is refined and extended in Section 5 to handle right-splitable subtrees of zero potential in a way crucial for our wire-length minimization algorithm.

The positive right-splitable subtrees of T are recorded by assigning to every vertex of T the status “S” or “M”. Vertices with status “M” correspond to the vertices in the positive right-splitable subtrees. As already mentioned in the previous section, positive right-splitable subgraphs correspond to sets of cells in the layout whose movement to the right reduces the total wire length. The algorithm finding right-splitable subtrees is used to determine right-splitable subgraphs in

tight components and the status “M” assigned to vertices stands for this possible future movement. Analogously, “S” stands for cells that stay at their current positions.

The positive right-splitable subtrees of T are determined in two phases. The first phase computes for every vertex u and a tree T_u containing vertex u two quantities, $pot_m(T_u, u)$ and $pot_s(T_u, u)$. The quantity $pot_m(T_u, u)$ (resp. $pot_s(T_u, u)$) is the maximum potential over all sets of (vertex-disjoint) right-splitable subtrees of T_u including vertex u (resp. not including u). Observe that $pot_s(T_u, u) \geq 0$, but that $pot_m(T_u, u)$ can be negative.

Let POT-PRS(T_u, u) be the algorithm computing entries $pot_m(T_u, u)$ and $pot_s(T_u, u)$ (“PRS” stands for positive right-splitable). Algorithm POT-PRS is recursive and which trees are associated with each vertex during the calls is determined during the recursion. Let v be an arbitrarily chosen vertex in tree T . The initial call to POT-PRS is made with tree T and vertex v as the two arguments. Then, $\max\{pot_m(T, v), pot_s(T, v)\}$ is the value of the maximal potential over all sets of (vertex-disjoint) right-splitable subtrees of T . The second phase of the algorithm uses the computed quantities to determine the status of every vertex so that the potential of all vertices having status “M” is $\max\{pot_m(T, v), pot_s(T, v)\}$.

The base case for the algorithm computing the $pot_m(\cdot)$ and $pot_s(\cdot)$ entries represents the situation when T_u consists of a single vertex, namely vertex u . In this case the only right-splitable subtree of T_u is T_u itself and thus we set $pot_m(T_u, u) = pot(u)$ and $pot_s(T_u, u) = 0$.

Assume now that T_u contains at least two vertices. Assume u has k arcs coming from vertices $u_1, u_2, \dots, u_k$ and l arcs going to vertices $u_{k+1}, u_{k+2}, \dots, u_{k+l}$, $k + l \geq 1$. Let T_{u_i} be the directed subtree containing vertex u_i after the arc (u, u_i) (or (u_i, u)) is deleted from T_u . The value of $pot_s(T_u, u)$ is determined by

$$\sum_{i=1}^k pot_s(T_{u_i}, u_i) + \sum_{j=k+1}^{k+l} \max\{pot_s(T_{u_j}, u_j), pot_m(T_{u_j}, u_j)\}.$$

The reasoning is the following. Since the vertex u cannot be included into any right-splitable subtree of T_u , no vertex u_i incident to an arc coming into vertex u can be in a right-splitable subtree of T_u . Hence, the contribution to $pot_s(T_u, u)$ made by subtree T_{u_i} is $pot_s(T_{u_i}, u_i)$. Every vertex u_j incident to an arc going out from vertex u is free to be included into a right-splitable subtree of T_u . The contribution made to $pot_s(T_u, u)$ by subtree T_{u_j} is thus $\max\{pot_s(T_{u_j}, u_j), pot_m(T_{u_j}, u_j)\}$.

Algorithm POT-PRS:

Input: A directed tree $T_u = (V_u, A_u)$ and a vertex $u \in V_u$.

Output: $pot_m(T_u, u)$ and $pot_s(T_u, u)$.

1. If $V_u = \{u\}$ then $pot_m(T_u, u) \leftarrow pot(u)$, $pot_s(T_u, u) \leftarrow 0$, return;
2. For each vertex u_i , where $(u_i, u) \in A_u$ do call POT-PRS(T_{u_i}, u_i);
3. For each vertex u_j , where $(u, u_j) \in A_u$ do call POT-PRS(T_{u_j}, u_j);
4. $pot_s(T_u, u) \leftarrow \sum_{(u_i, u) \in A_u} pot_s(T_{u_i}, u_i)$
 $\quad + \sum_{(u, u_j) \in A_u} \max \{pot_s(T_{u_j}, u_j), pot_m(T_{u_j}, u_j)\};$
5. $pot_m(T_u, u) \leftarrow \sum_{(u, u_j) \in A_u} pot_m(T_{u_j}, u_j)$
 $\quad + \sum_{(u_i, u) \in A_u} \max \{pot_s(T_{u_i}, u_i), pot_m(T_{u_i}, u_i)\} + pot(u);$
6. return.

Figure 4: Algorithm POT-PRS

The value of $pot_m(T_u, u)$ is determined by

$$\sum_{j=k+1}^{k+l} pot_m(T_{u_j}, u_j) + \sum_{i=1}^k \max \{pot_s(T_{u_i}, u_i), pot_m(T_{u_i}, u_i)\} + pot(u).$$

Since vertex u is required to be in a right-splitable subtree of T_u , every vertex u_j incident to an arc outgoing from u is also included in the right-splitable subtree. Hence, the contribution to $pot_m(T_u, u)$ coming from subtree T_{u_j} is $pot_m(T_{u_j}, u_j)$. Every vertex u_i incident to an arc coming into u is free to be included into a right-splitable subtree of T_u and its contribution to $pot_m(T_u, u)$ is $\max \{pot_s(T_{u_i}, u_i), pot_m(T_{u_i}, u_i)\}$. The potential of vertex u is also added to $pot_m(T_u, u)$.

Figure 4 gives a detailed description of algorithm POT-PRS.

From the above discussion and the description given in Figure 4 it is clear that the entries $pot_m(T, v)$ (resp. $pot_s(T, v)$) represent the maximum potential achievable by right-splitable subtrees of T that include (resp. do not include) vertex v . The running time of algorithm POT-PRS(T, v) is $O(|V|)$, where V is the vertex set of T .

After algorithm POT-PRS computed the $pot_s(\cdot)$ and $pot_m(\cdot)$ entries for every vertex in T , the second phase of the algorithm identifies the positive right-splitable subtrees yielding maximum potential by assigning the appropriate status to every vertex. Let algorithm SM-STATUS be the algorithm performing the assignment. Let u be a vertex and T_u be the subtree of T containing vertex u so that the entries $pot_m(T_u, u)$ and $pot_s(T_u, u)$ were computed by algorithm POT-PRS. The initial call to algorithm SM-STATUS is made with tree T and vertex v as the arguments, where v is the same vertex used in the initial call of algorithm POT-PRS.

At the begin of algorithm $\text{ST-STATUS}(T_u, u)$, the status of vertex u may or may not already have been determined by an earlier recursive call. If the status of vertex u has not yet been determined, it is determined as follows. If $\text{pot}_s(T_u, u) < \text{pot}_m(T_u, u)$, u is a vertex in the collection of subtrees achieving a potential of $\max\{\text{pot}_m(T_u, u), \text{pot}_s(T_u, u)\}$ and u is thus assigned the status “M”. Otherwise, u is assigned status “S”. Observe that we do assign u the status “S” if $\text{pot}_s(T_u, u) = \text{pot}_m(T_u, u)$; i.e., we decide for status “S” in case of a tie.

Assume again that vertex u has k arcs coming from vertices $u_1, u_2, \dots, u_k$, and l arcs going to vertices $u_{k+1}, u_{k+2}, \dots, u_{k+l}$. Let T_{u_i} be the directed subtree containing vertex u_i after the arc (u, u_i) (or (u_i, u)) is deleted from T_u .

Consider a vertex u_i incident an arc coming into vertex u , $1 \leq i \leq k$. If the status of u is “S”, then the status of u_i must also be “S”. If the status of u is “M”, u ’s status does not impose a restriction on the status of u_i and u_i ’s status is not yet decided. We invoke $\text{SM-STATUS}(T_{u_i}, u_i)$ to determine the status of u_i , if necessary, and to handle the remaining vertices in tree T_{u_i} . Let u_j be a vertex incident an arc going out from u , $k+1 \leq j \leq k+l$. If the status of u is “M”, then u_j must belong to the same right-splitable subtree and its status is set to “M”. If the status of u is “S”, the status of u_j is determined at the next level of the recursion. We invoke $\text{SM-STATUS}(T_{u_j}, u_j)$ to handle the remaining vertices in tree T_{u_j} . See Figure 5 for a detailed description of algorithm SM-STATUS .

After algorithm SM-STATUS has determined the status of all vertices in the directed spanning tree T , the vertices are partitioned into two sets, the vertices with status “M” and the vertices with status “S”. Every one of these two sets consists of a collection of subtrees of T . Let F_m be the forest representing the directed subtrees of T formed by the vertices with status “M”, and let F_s be the forest representing the directed subtrees of T formed by the vertices with status “S”. Observe that $\text{pot}(F_m) = \max\{\text{pot}_m(T, v), \text{pot}_s(T, v)\}$, where v is the vertex of T used in the first call to POT-PRS (and SM-STATUS). A subtree in F_m is called a *moving tree* and a subtree in F_s is called a *staying tree*, respectively. The following lemma is used in Section 6 when we prove the correctness of our algorithm.

Lemma 4.1 *Let F_m be the forest representing the set of right-splitable subtrees of T with maximum potential. Let T_m be a moving tree in forest F_m . Then, T_m is a positive right-splitable subtree of T and every left-splitable subtree of T_m has positive potential.*

Algorithm SM-STATUS:

Input: A directed tree $T_u = (V_u, A_u)$ and a vertex $u \in V_u$.

Output: Status of every vertex in T_u .

1. If $status(u)$ has not been determined then
begin
2. If $pot_m(T_u, u) > pot_s(T_u, u)$
then $status(u) \leftarrow "M"$
else $status(u) \leftarrow "S"$;
end;
3. For each vertex u_i , where $(u_i, u) \in A_u$ do
begin
4. If $status(u) = "S"$ then $status(u_i) \leftarrow "S"$;
5. call SM-STATUS(T_{u_i}, u_i);
end;
6. For each vertex u_j , where $(u, u_j) \in A_u$ do
begin
7. If $status(u) = "M"$ then $status(u_j) \leftarrow "M"$;
8. call SM-STATUS(T_{u_j}, u_j);
end;
9. return.

Figure 5: Algorithm SM-STATUS

Proof: By the rules of how the status assignments are made in algorithm SM-STATUS, it follows that T_m is right-splitable from T . If the potential of T_m would be negative, then deleting T_m from F_m would increase the potential of F_m , which contradicts our definition of F_m . Hence, the potential of T_m is not negative. Similarly, if T_m would contain a negative left-splitable subtree, then by deleting this left-splitable subtree from T_m , we would increase the potential of T_m and thus of F_m . Hence, any left-splitable subtree of T_m has a non-negative potential.

Assume now that $pot(T_m) = 0$. Let u be the first vertex (in time) in T_m whose status is determined in algorithm SM-STATUS. Let T_u be the subtree of T containing vertex u and for which the entries $pot_s(T_u, u)$ and $pot_m(T_u, u)$ were evaluated. From the choice of u it follows that T_m is a right-splitable subtree of T_u . Furthermore, for every arc (x, y) in T_u with $x \notin T_m$ and $y \in T_m$, we have $status(x) = "S"$. Since $pot(T_m) = 0$, changing the status of every vertex in T_m from "M" to "S" does not change the potential of the moving trees determined for T_u . Hence, $pot_s(T_u, u) = pot_m(T_u, u)$. However, u received status "M" which implies $pot_s(T_u, u) < pot_m(T_u, u)$, resulting in a contradiction for assuming $pot(T_m) = 0$. The argument for showing that T_m cannot contain a left-splitable subtree with potential zero is similar and is omitted. $\square$

5 Finding the zero trees in a staying tree

The last section described a linear time algorithm to partition a given directed tree T into two forests F_m and F_s so that the trees in F_m are positive right-splitable subtrees of T with maximum potential. In this section we present an algorithm that further refines the trees in F_s by determining for every tree T_s in F_s subtrees having a potential of zero. We call these subtrees the zero trees. The identification of zero trees is crucial for achieving the claimed running time of our wire-length minimizing algorithm. During the compaction process zero trees represent groups of cells whose movement to the right will not change the total wire length. We start by giving the precise definition of the zero trees. A non-empty subtree T_z of T_s is a *zero tree* of T_s if the following two conditions hold:

- (i) $pot(T_z) = 0$ and T_z contains no proper right-splitable subtree having potential zero.
- (ii) Let (u, v) be an arc of T_s with u in T_z and v not in T_z . Then, v belongs to another zero subtree of T_s .

Figure 6 shows a staying tree and its three zero trees. Observe that the zero trees of a given staying tree are unique. We record the zero trees by assigning the vertices belonging to a zero tree status “Z”. In addition, the zero trees are labeled; i.e., for every vertex u with status “Z” $label(u)$ corresponds to the index of the zero tree u belongs to. Let v_s be an arbitrarily chosen vertex in T_s . The first step in identifying the zero trees in T_s is a call to algorithm POT-PRS(T_s, v_s). Since T_s is a staying tree, we know that the maximum potential of a right-splitable subtree of T_s is zero; i.e., $pot_m(T_s, v_s) \leq pot_s(T_s, v_s) = 0$. However, the entries $pot_m(\cdot)$ and $pot_s(\cdot)$ computed during this call to algorithm POT-PRS are needed during the detection of the zero trees.

Let algorithm Z-STATUS be the algorithm detecting and labeling the zero trees. Algorithm Z-STATUS has two arguments, a tree $T_{s,u}$ and a vertex u of $T_{s,u}$, with $T_{s,u}$ being a subtree of staying tree T_s . As done in the algorithms described in the previous section, the trees are determined during the execution of the algorithm. Observe that $T_{s,u}$ is not necessarily a right-splitable subtree of T_s . At the time Z-STATUS($T_{s,u}, u$) is invoked, the status of vertex u may or may not already have been updated by an earlier recursive call. By updating we mean that it was determined that the status of u remains “S” or that u does belong to a zero subtree (and thus has now status “Z”). In Figure 7 we give a detailed description of algorithm Z-STATUS to which we refer in the following

Figure 6: A staying tree and its zero trees.

discussion. Let L be the global index used for numbering the zero subtrees, with $L=1$ initially.

Assume the status of vertex u has not been updated. The invariant condition that holds for a vertex not yet updated at the begin of $\text{Z-STATUS}(T_{s,u}, u)$ is $\text{pot}_m(T_{s,u}, u) \leq \text{pot}_s(T_{s,u}, u) = 0$. This condition is obviously true when Z-STATUS is invoked with staying tree T_s and vertex v_s . Furthermore, it is true whenever $T_{s,u}$ is a right-splitable subtree of T_s (otherwise the assumption that T_s is a staying tree would be violated). When $T_{s,u}$ is not a right-splitable subtree of T_s , the invariant may not hold. Actions taken by previous calls will guarantee that vertex u has already been updated whenever $\text{pot}_m(T_{s,u}, u) > \text{pot}_s(T_{s,u}, u)$. The updating is done as follows. If $\text{pot}_m(T_{s,u}, u) < \text{pot}_s(T_{s,u}, u) = 0$, there exists no right-splitable subtree of $T_{s,u}$ containing vertex u and having potential zero. Hence, the status of u remains "S". Otherwise (i.e., $\text{pot}_m(T_{s,u}, u) = \text{pot}_s(T_{s,u}, u) = 0$), there exists a zero tree containing u and we make the necessary updates (see lines 4-5 in Figure 7).

Let $u_1, u_2, \dots, u_k$ be the vertices incident to the incoming arcs and $u_{k+1}, u_{k+2}, \dots, u_{k+l}$ be the vertices incident to the outgoing arcs of vertex u in tree $T_{s,u}$, respectively. Let T_{u_i} denotes the tree containing vertex u_i after the arc (u, u_i) (or (u_i, u)) has been deleted from $T_{s,u}$. Consider a vertex u_j incident to an outgoing arc of u , $k+1 \leq j \leq k+l$. Tree T_{u_j} is a right-splitable subtree of $T_{s,u}$ and we thus have $\text{pot}_m(T_{u_j}, u_j) \leq \text{pot}_s(T_{u_j}, u_j) = 0$. The decision of what happens to vertex u_j depends on the already updated status of vertex u . Assume first that the status of u is "Z". In order to satisfy condition (ii) of the definition of a zero tree, vertex u_j must also belong to zero tree. The value of $\text{pot}_m(T_{u_j}, u_j)$ determines whether u_j belongs in the same zero tree as u or in a newly detected zero tree. If $\text{pot}_m(T_{u_j}, u_j) = 0$, then tree T_{u_j} contains a right-splitable subtree of zero potential containing vertex u_j . In order to guarantee that no zero tree contains another zero

Algorithm Z-STATUS:

Input: A staying tree $T_{s,u} = (V_{s,u}, A_{s,u})$ and a vertex $u \in V_{s,u}$.

Output: A labeling of the zero trees in $T_{s,u}$.

1. If $status(u)$ has not been updated then
begin
2. If $pot_m(T_{s,u}, u) < pot_s(T_{s,u}, u)$ then $status(u) \leftarrow "S"$;
3. If $pot_m(T_{s,u}, u) = pot_s(T_{s,u}, u)$ then
begin
4. $status(u) \leftarrow "Z"$;
5. $label(u) \leftarrow L; L \leftarrow L + 1$;
6. end;
7. end;
8. For each vertex u_j , where $(u, u_j) \in A_{s,u}$ do
begin
9. If $status(u) = "Z"$ then
begin
10. $status(u_j) \leftarrow "Z"$;
11. If $pot_m(T_{s,u}, u) = 0$ then $label(u_j) \leftarrow L, L \leftarrow L + 1$;
12. If $pot_m(T_{s,u}, u) < 0$ then $label(u_j) \leftarrow label(u)$;
13. end;
14. call Z-STATUS(T_{u_j}, u_j);
15. end;
16. For each vertex u_i , where $(u_i, u) \in A_{s,u}$ do
begin
17. If $status(u) = "S"$ then $status(u_i) \leftarrow "S"$;
18. If $status(u) = "Z"$ and $pot_m(T_{s,u_i}, u_i) > 0$
then $status(u_i) \leftarrow "Z", label(u_i) \leftarrow label(u)$;
19. call Z-STATUS(T_{u_i}, u_i);
20. end;
21. return.

Figure 7: Algorithm Z-STATUS

tree, this tree and thus vertex u_j must belong to a zero tree different from the one vertex u belongs to. Line 9 performs the necessary updates. If $\text{pot}_m(T_u, u_j) < 0$, then vertex u_j belongs to the same zero tree as vertex u and line 10 is executed. When the status of u is “S”, then the status of vertex u_j is not updated at this point, but at the next level of recursion. The last action to be taken for vertex u_j is to invoke a call to $\text{Z-STATUS}(T_{u_j}, u_j)$.

Consider now a vertex u_i so that the arc (u_i, u) is an incoming arc of vertex u , $1 \leq i \leq k$. If the status of u is “S”, then the status of u_i must also be “S” (otherwise condition (ii) of the definition of a zero tree is violated). Assume now that the status of u is “Z”. Since T_{u_i} is not a right-splitable tree, $\text{pot}_m(T_{u_i}, u_i) > 0$ is possible. If $\text{pot}_m(T_{u_i}, u_i) > 0$, then vertices u_i and u must belong to the same zero subtree. The reasoning is as follows: we know u belongs to a zero tree and if u_i would not belong to this zero tree, then we would have found a right-splitable subtree of positive potential (formed by the vertices in the zero tree containing u and the vertices of T_{u_i} giving a potential of $\text{pot}_m(T_{u_i}, u_i)$). This is not possible. Line 15 is executed in this case. If $\text{pot}_m(T_{u_i}, u_i) \leq 0$, vertex u_i is not updated. Observe that these actions guarantee that the invariant condition for updating a vertex are satisfied; namely, that a vertex u_i with $\text{pot}_m(T_{u_i}, u_i) > 0$ has been updated when u_i is an argument for algorithm Z-STATUS . Similar to the outgoing arcs, the last action to be taken for vertex u_i is to invoke a call to $\text{Z-STATUS}(T_{u_i}, u_i)$, as done in line 16.

It follows from the structure of Z-STATUS that the running time of $\text{Z-STATUS}(T_s, v_s)$ is linear in the number of vertices of tree T_s . After algorithm Z-STATUS has updated the status of every vertex in tree T_s , we have partitioned the vertices of T_s into those belonging to zero trees and those whose status remained “S”. From now on when we refer to a staying tree we mean a tree containing vertices with status “S” after algorithm Z-STATUS has identified the zero trees. We refer to the staying trees present before the zero trees were identified as the staying trees generated by algorithm SM-STATUS . The following two lemmas state properties of zero trees and staying trees that are relevant to the correctness of our wire-length minimizing algorithm.

Lemma 5.1 *Let T_s be a staying tree generated by algorithm SM-STATUS and let T_z be a zero tree of T_s identified by algorithm Z-STATUS . Then, any proper, right-splitable subtree of T_z has negative potential.*

Proof: By the definition of a zero tree, T_z contains no proper right-splitable subtree having potential zero. Assume T_z contains a right-splitable subtree $T_{z,r}$ having positive potential. Then,

$T_{z,r}$, together with the zero trees of T_s containing vertices that can be reached by vertices in $T_{z,r}$, represents a positive right-splitable subtree in T_s . However, T_s is a staying tree generated by algorithm SM-STATUS and contains no positive right-splitable subtree. The lemma follows. $\square$

Lemma 5.2 *Let T'_s be a staying tree produced by algorithm Z-STATUS. Then, any right-splitable subtree of T'_s has a negative potential.*

Proof: Assume that T_s is the staying tree containing T'_s before invoking algorithm Z-STATUS. Assume first that T'_s contains a right-splitable subtree $T'_{s,r}$ having positive potential. Then, $T'_{s,r}$ and all the zero trees of T_s containing vertices that can be reached by vertices of $T'_{s,r}$ form a positive right-splitable subtree of T_s . However, T_s cannot contain such a subtree. Consider now the situation when T'_s contains a right-splitable subtree of zero potential. Such a subtree would represent another zero tree for T_s and its existence violates our assumption about T'_s . Hence, the lemma follows. $\square$

6 Minimizing the total wire length

We now return to the problem of generating a configuration of minimum total wire length. Recall that relevant terminology and definitions were given in Section 2 and the overall approach was discussed in Section 3. We start by describing the algorithm, then prove its correctness and show its $O(n_h \cdot n \log n)$ running time.

6.1 The algorithm

Our wire-length minimizing algorithm performs of a number of iterations, with the information generated by each iteration corresponding to a configuration. Let $\mathcal{C}$ be such a configuration at the beginning of an iteration. At the start of an iteration we have available the constraint graph $G_{\mathcal{C}}$ whose arc weights are set with respect to the distances between cells in configuration $\mathcal{C}$, and the tight components of the constraint graph. Recall that every vertex of $G_{\mathcal{C}}$ represents a cell in $\mathcal{C}$ and that every cell belongs to exactly one tight component. For the first iteration of the algorithm, this information can be generated in $O(n_h + (n_v + n_r) \log(n_v + n_r))$ time from the initial configuration, as stated in Section 2. In addition, every tight component G_t is represented by a directed spanning forest F_t . The first iteration chooses an arbitrary directed spanning tree of G_t as F_t . Given these

graphs and spanning forests at the begin of an iteration, we start the iteration by partitioning every directed tree in a spanning forest into moving, zero, and staying trees using the algorithms described in the previous two sections.

Lemma 3.1 states that a positive right-splitable subgraph of a tight component G_t corresponds to a set of cells that can move at least one position to the right and that this movement reduces the total wire length. The positive right-splitable subtrees determined for the trees in spanning forest F_t do not necessarily represent a positive right-splitable subgraph of G_t . Thus, they may not correspond to cells that can move one unit to the right. The remainder of an iteration determines which positive right-splitable subtrees do correspond to positive right-splitable subgraphs of tight components, how much cells corresponding to vertices in moving and zero trees move to the right, and how to update the forests for the next iteration.

The process of how much the cells corresponding to vertices of status “M” or “Z” move to the right can be visualized through plowing. Imagine a plow in the form of a vertical straight line positioned to the left of the leftmost cell. The plow moves to the right pushing along some cells and passing over others, according to the following rules. When the plow touches a cell corresponding to a vertex of a moving or zero tree T , the plow pushes all cells in tree T along. On the other hand, if the plow touches a cell of status “S”, it simply moves across this cell. Once the plow has touched cells and is pushing them along, a cell being pushed can touch a cell not yet being pushed. The rules are now somewhat different. If a cell being pushed and belonging to a tree T' touches a cell in a moving or zero tree T , all the cells corresponding to vertices in T are being pushed along. On the other hand, if a cell of tree T' touches a cell of status “S”, then all the cells corresponding to vertices in T' stop being pushed by the plow. In terms of the status, we can interpret this as assigning to the vertices in T' the status “S”. The setup will be such that eventually every cell will have status “S” and the plow will have moved across all cells. At this point the plowing process stops and the new positions of the cells represent a new configuration.

The plowing described above is accomplished by performing a single-source shortest path computation. The shortest path computation is done on a directed, weighted graph $G = (V, A)$ obtained from the moving, zero, and staying trees and the constraint graph G_C as follows. Every vertex in G corresponds to either a moving, zero or staying tree. Let v_i and v_j be two vertices in G representing trees T_{v_i} and T_{v_j} , respectively. There is an arc from v_i to v_j in G , if the constraint graph G_C

contains an arc from a vertex in T_{v_i} to a vertex in T_{v_j} . The weight of the arc (v_i, v_j) in G is set to the minimum weight among all the weights of arcs in G_C that induce the arc (v_i, v_j) . In order to be able to use a single-source algorithm, we add a special vertex v_0 . For every vertex v_i representing a staying tree, we include an arc from v_i to v_0 with weight $w(v_i, v_0) = 0$.

In Section 6.2 we show that for every vertex of G representing a moving or a zero tree there exists a path to a vertex representing a staying tree. After having created graph G , we determine for every vertex v_i representing a moving or zero tree the shortest path from v_i to a vertex representing a staying tree. Since for every vertex v_j with $j \neq 0$ representing a staying tree we introduced an arc (v_j, v_0) with weight zero, we can view the path computation as a single-source shortest path problem (we simply compute the shortest path from v_0 to every vertex in the reversed graph). Let T be the tree containing the shortest path information we are interested in (i.e., T is the union of the shortest paths from all vertices representing a moving or zero tree to v_0). The length of the shortest path from vertex v_i to v_0 represents the distance the cells corresponding to the vertices in T_{v_i} move to the right. This movement results in a new configuration and we update the weights of the constraint graph G_C to correspond to this new configuration. The final action taken in an iteration is the generation of the new directed spanning forests representing the tight components of G_C and this is described next.

— The new spanning forests are formed by including the arcs in the moving, zero, and staying trees. These arcs were tight at the beginning of the iteration and, since the cells corresponding to vertices in these trees moved the same amount or not at all, these arcs are still tight in the generated constraint graph. In addition, we include the following arcs. For any arc (v_i, v_j) with $v_j \neq v_0$ belonging to the shortest path tree T there exists at least one corresponding arc (having the same weight) in the constraint graph G_C . For any such arc (v_i, v_j) choose one corresponding arc in the constraint graph G_C and call it a *critical arc*. In the just generated constraint graph the critical arcs are tight, i.e., their weight is equal to zero. All critical arcs are included into the new spanning forests. This completes the description of one iteration.

The running time of one iteration is $O(n \log n)$ and is established as follows. Given the spanning forests of the tight components of the constraint graph, the moving, zero, and staying subtrees are determined in $O(n)$ time. Using these subtrees and the arcs of the constraint graph, graph G is set up in $O(n)$ time. Graph G is not a planar graph (introducing the arcs from v_j to v_0 may destroy

planarity) and it has $O(n)$ arcs. Using Dijkstra's algorithm on G [1], we determine the shortest path tree T in $O(n \log n)$ time. The final updating (i.e., updating the weights of arcs in the constraint graph and forming the new directed spanning forests) can be completed in $O(n)$ time.

To complete the description of the algorithm, we need to state its stopping criterion. The first action taken in an iteration is the identification of moving, staying, and zero subtrees. The algorithm terminates (without constructing graph G and performing a shortest path computation) when no moving tree is identified. Observe that if the spanning forests contain no moving subtrees, they only contain zero trees. This follows since the potential of the constraint graph is zero and the existence of a staying tree implies the existence of a moving tree. Furthermore, when no moving tree is found, no tight component of the constraint graph contains a positive right-splitable subgraph.

6.2 Correctness

In this section we prove that the algorithm described in Section 6.1 generates a configuration of minimum total wire length that also minimizes the width over all configurations having minimum total wire length.

We start by showing that in graph G there exists, for every vertex v_i representing a zero or a moving tree, a path to a vertex representing a staying tree. Let T_{v_i} be the tree represented by vertex v_i in G . Consider first the case when T_{v_i} contains a vertex corresponding to a cell that is incident to a left endpoint of a horizontal wire whose right endpoint is not incident to a cell corresponding to a vertex in T_{v_i} . If the right endpoint of this horizontal wire connected to a cell in a staying tree, the claim follows. Assume the right endpoint is incident to another moving tree or a zero tree. Let T_{v_j} be this tree. Tree T_{v_j} must contain at least one cell incident to the left endpoint of a horizontal wire whose right endpoint is not incident to a cell in T_{v_j} . Continuing this argument, we traverse in G a sequence of vertices corresponding to either zero or moving trees. Eventually, we must reach a vertex whose corresponding tree contains cells incident to more right endpoints than left endpoints. Such a vertex corresponds to a staying tree.

Assume now that tree T_{v_i} is a zero tree in which for every cell incident to a left endpoint of a horizontal wire the right endpoint of this wire is incident to a cell also in T_{v_i} . In order to show that there exists a path from v_i to a vertex representing a staying tree, we make use of the two fictitious cells U_l and U_r , which are represented by vertices u_l and u_r , respectively, in the spanning

forest and the constraint graph. Let v_r be the vertex in G that corresponds to the tree containing vertex u_r . Let T_{v_r} be this tree. Cell U_r corresponds to a vertical line to the right of the rightmost cell in the layout and hence there exists in G a path from v_i to v_r . If T_{v_r} is a moving tree, then, according to the above discussion, there exists a path from v_r to some vertex representing a staying tree. Combining the path from v_i to v_r with this path, gives a path from v_i to a vertex representing a staying tree. Assume now that T_{v_r} is a zero tree. Since $pot(u_r) = -\frac{1}{2}$ and $pot(T_{v_r}) = 0$, vertex u_l , which has potential $+\frac{1}{2}$, is also in T_{v_r} . Cell U_l corresponds to a vertical line to the left of the leftmost cell in the layout and hence there exists in G a path from v_r to every vertex of G . Graph G is only constructed when there exists at least one moving tree. Let v_k be a vertex representing a moving tree. Then, there exists a path from v_r to v_k in G . Combining the path from v_i to v_r with the path from v_r to v_k , and using the result that there exists a path from every moving tree to a staying tree, gives the existence of a path from v_i to a vertex representing a staying tree.

We now turn to showing that the configuration generated by our algorithm has minimum total wire length. Let G_{C_f} be the constraint graph generated during the final iteration and let C_f be the corresponding configuration. If the wire length of C_f could be reduced further, then according to Lemma 3.2 we could locate in G_{C_f} a tight component containing a positive right-splitable subgraph. The last iteration failed to find a positive right-splitable subtree in the spanning forests representing tight components and thus no positive right-splitable subgraph can exist in the tight components of G_{C_f} . Hence, any further movement of the cells to the right either increases the total wire length or leaves it unchanged.

Now consider the width of configuration C_f . Assume that C_f is not a configuration of minimum width over all configurations having minimum total wire length. Then, we can find a proper subgraph G' of G_{C_f} such that (i) G' contains cell U_l , but does not contain cell U_r and (ii) pushing all cells in G' one unit to the right without changing the positions of the other cells results in another configuration and does not change the total wire length. Consider the potential of such a graph G' . It has a potential of $+\frac{1}{2}$, since $pot(u_l) = +\frac{1}{2}$ and the potential contributed by all other cells sums up to 0. This implies that G_{C_f} contains a positive right-splitable subgraph. However, the last iteration failed to find a positive right-splitable subtree, which contradicts the existence of G' . Hence, C_f is a configuration having minimum width over all configurations with minimum total wire length.

6.3 Time complexity

In this section we show that the number of iterations executed by our algorithm is at most n_h , the number of horizontal wires in the layout.

The algorithm determines positive right-splitable subtrees in the directed forests representing the tight components. As already stated, these subtrees may not correspond to right-splitable subgraphs. Let v_i be a vertex in G representing moving or zero tree T_{v_i} . If the length of shortest path from v_i to v_0 is zero, then the cells corresponding to the vertices in T_{v_i} do not move to the right and T_{v_i} does not represent a right-splitable subgraph of the corresponding tight component. Hence, it is possible that no cell moves to the right after the shortest path computation on G is completed and the configuration generated is identical to the configuration at the beginning of the iteration. When T_{v_i} does not represent a right-splitable subgraph of tight component G_i , then we change the spanning forest by including critical arcs and deleting all other arcs between moving, zero, and staying subtrees. Obviously, the resulting spanning forest may again not contain a positive right-splitable subtree corresponding to a right-splitable subgraph. We do not explicitly address the issue of how many iterations are required to guarantee that a right-splitable subgraph is found. We point out that there exist examples for which this takes $O(n_h)$ iteration until a right-splitable subgraph is generated. The argument we use to show that after at most n_h iterations no moving tree is found relies on another property of the generated configurations.

Assume that in the $(i - 1)$ -st iteration of the algorithm at least one moving tree was identified and we thus execute an i -th iteration, $i \geq 2$. Let M_{i-1} (resp. M_i) be the set of the vertices of the constraint graph that belong to moving trees (i.e., have status "M") in the $(i - 1)$ -st (resp. i -th) iteration. We then have $0 < \text{pot}(M_{i-1}) \leq n_h$. The potential of M_i is called the *moving potential* of the i -th iteration. In Lemma 6.1 we show that the moving potential is strictly decreasing from one iteration to the other. A straightforward consequence of this lemma is that the moving potential is zero after at most n_h iterations.

Lemma 6.1 *If $M_{i-1} \neq \emptyset$ and $i \geq 2$, then $\text{pot}(M_{i-1}) > \text{pot}(M_i)$.*

Proof: We first show that a change in the status of a vertex cannot increase the moving potential; i.e., $\text{pot}(M_{i-1}) \geq \text{pot}(M_i)$. Since the moving potential only includes the potential of vertices in moving trees, we do not need to consider a change of status from "S" to "Z" or from "Z" to "S".

Consider a vertex u that has either status “S” or “Z” in the $(i - 1)$ -st and status “M” in the i -th iteration. Assume in the $(i - 1)$ -st iteration u is in the staying tree T_s (resp. zero tree T_z). Since u has status “M” in the i -th iteration, we can locate in T_s (resp. T_z) a right-splitable subtree $T_{s,r}$ (resp. $T_{z,r}$) containing vertex u so that all the vertices in this right-splitable subtree have status “M” in the i -th iteration. Observe that $T_{s,r}$ (resp. $T_{z,r}$) could be identical to T_s (resp. T_z). Lemma 5.2 states that any right-splitable subtree of a staying tree has a negative potential. Lemma 5.1 states that any proper right-splitable subtree of a zero tree has a negative potential. Hence, $pot(T_{s,r}) < 0$ and $pot(T_{z,r}) \leq 0$. A vertex changing its status from “S” to “M” is thus associated with a set of vertices in a staying tree all of which change their status to “M” and whose contribution to $pot(M_i)$ is negative. On the other hand, a vertex changing its status from “Z” to “M” is associated with a set of vertices in a zero tree all of which change their status to “M” and whose contribution to $pot(M_i)$ is not positive.

The second change in status that needs to be considered is that of a vertex u having status “M” in the $(i - 1)$ -st iteration and having either status “S” or “Z” in the i -th iteration. Let T_m be the moving tree containing vertex u in the $(i - 1)$ -st iteration. Since u does not have status “M” in the i -th iteration, T_m contains a left-splitable subtree $T_{m,l}$ such that u is a vertex in $T_{m,l}$ and no vertex in $T_{m,l}$ has status “M” in the i -th iteration. Lemma 4.1 states that any left-splitable subtree of a moving tree has positive potential. This implies that $pot(T_{m,l}) > 0$. Hence, the change of status in vertex u is associated with a set of vertices all of which have no longer status “M” and whose removal from M_{i-1} reduces the moving potential by $pot(T_{m,l})$.

In order to complete the proof we need to show that in every iteration there exists at least one vertex that induces a decrease in the moving potential. As shown above, among the changes in the status effecting the moving potential, only a change from status “Z” to “M” may not cause a decrease. Since the $(i - 1)$ -st iteration is not the last one, at least one moving tree is identified. Let T_m be one such moving tree which is represented by vertex v_m in graph G , the graph used for the shortest path computation during the $(i - 1)$ -st iteration. In the shortest path computation on G we determine a shortest path from v_m to a vertex representing a staying tree. Let v_s and T_s be this vertex and its corresponding staying tree, respectively, and let $P = \langle v_1, v_2, \dots, v_{l-1}, v_l \rangle$ be the shortest path from $v_m = v_1$ to $v_s = v_l$, $l \geq 2$. W.l.o.g., assume that, when $l \geq 3$, vertices $v_2 \dots v_{l-1}$ represent zero trees $T_2, \dots, T_{l-1}$. Let T' be the spanning tree generated at the end of the

Figure 8: A spanning tree generated at the end of $(i-1)$ -st iteration.

$(i-1)$ -st iteration and which contains $T_1, T_2, \dots, T_{l-1}, T_l$ as subtrees and the arcs of the constraint graph inducing the arcs on path P (i.e., the critical arcs).

In every tree T_j , $2 \leq j \leq l-1$, we identify two vertices $u_{j,in}$ and $u_{j,out}$, so that arcs $(u_{j-1,out}, u_{j,in})$ and $(u_{j,out}, u_{j+1,in})$ are critical arcs. For T_m and T_s we identify one vertex each, namely $u_{1,out}$ and $u_{l,in}$. Observe that there does not have to exist a path between $u_{j,in}$ and $u_{j,out}$ in T_j . See Figure 8 for a possible situation. In the $(i-1)$ -st iteration $u_{1,out}$ has status “M” and $u_{l,in}$ has status “S”. If at least one of these two vertices changes its status in the i -th iteration, we have $pot(M_{i-1}) > pot(M_i)$ and the lemma follows. Assume now that neither vertex changes its status. Observe that when $l = 2$ (i.e., there exists no zero tree between T_m and T_s), the contradiction follows immediately. Assume thus $l \geq 3$. Vertex $u_{2,in}$ and all vertices in tree T_2 reachable from $u_{2,in}$ must have status “M” in the i -th iteration. Vertex $u_{l,in}$ maintaining status “S”, implies that vertex $u_{l-1,out}$ must have status “S” in the i -th iteration. Hence there exists a zero tree T_k such that $u_{k,in}$ has status “M” and $u_{k,out}$ has status “S” or “Z”. In order for this to happen, $u_{k,out}$ is not reachable from $u_{k,in}$. This implies that the vertices of a proper right-splitable subtree of T_k that contains vertex $u_{k,in}$, but not $u_{k,out}$, are assigned status “M”. Since any proper right-splitable subtree of a zero tree has negative potential, zero tree T_k causes a decrease in the moving potential and the lemma now follows. $\square$

We are now able to describe why it is crucial to partition the spanning trees into moving, zero, and staying trees and why two types of trees would not be enough. In order to reduce the wire length, it is not necessary that zero trees move to the right whenever possible. However, in order to guarantee that in at most n_h iterations a configuration of minimum total wire length is

generated, zero trees need to move to the right whenever possible. If we do not allow zero trees to move along with moving trees and thus view them as staying trees, we could only show that $\text{pot}(M_{i-1}) \geq \text{pot}(M_i)$. Moving trees would be stopped by zero trees that are not moving to the right and the moving potential could remain unchanged over a number of iterations. It is not hard to construct examples for which there is no change in the moving potential for $O(n_h)$ iterations. Partitioning the trees into moving, zero, and staying trees reduces the moving potential formed by all the vertices in the moving trees by at least one in each iteration. Hence, after at most n_h iterations no more moving trees are identified and the algorithm terminates.

We can now state the following result which summarizes the discussion of this section.

Theorem 6.1 *Given a configuration and its constraint graph, a configuration minimizing the total wire length can be generated in $O(n_h \cdot n \log n)$ time, where n_h is the number of horizontal wires and n is the number of cells in layout. Furthermore, this configuration also has the minimum width among all configurations minimizing the total wire length.*

7 Tradeoffs between the wire length and the layout width

In the previous sections we developed an algorithm that generates a configuration of minimum total wire length which has minimum layout width among all configurations achieving this minimum total wire length. In this section we consider additional relationships between the layout width and the total wire length and generalize the concepts developed to other such compaction problems. For any configuration C , let $w(C)$ be its width and $l(C)$ be its total wire length.

Let C be a given configuration. In the *min- l -problem* we are given, in addition to C , a width w and we are to determine a configuration C^* with $w(C^*) \leq w$ that has minimum total wire length among all configurations having a width of at most w . In the *min- w -problem* we are given a total wire length l and are to determine a configuration C^* with $l(C^*) \leq l$ that has minimum width among all configurations having total wire length of at most l . We show in Sections 7.1 and 7.2 that, given the constraint graph corresponding to the initial configuration C , both problems can be solved in $O(n_h \cdot n \log n)$ time. In Section 7.3 we consider the compaction process when a tradeoff between the layout width and the total wire length is specified. More precisely, given a configuration C and two constants α and β , $\alpha, \beta > 0$, we show how to determine, in $O(n_h \cdot n \log n)$ time, a configuration C^* such that $\alpha \cdot w(C^*) + \beta \cdot l(C^*) \leq \alpha \cdot w(C') + \beta \cdot l(C')$, for all other configurations C' .

7.1 Compacting with a given width w

In this section we show how to solve the min_L -problem by modifying the wire-length minimizing algorithm presented in Section 6. Assume the constraint graph G_C corresponding to the initial configuration C has been determined. We next generate a configuration C_{left} in which every cell of configuration C is positioned as far to the left as possible. Configuration C_{left} corresponds to a configuration of minimum width and can be generated in $O(n \log n)$ time [6]. If $w(C_{left}) > w$, there exists no solution to the min_L -problem. Otherwise, we add to configuration C_{left} a U-shaped cell that “encloses” all other cells. The two vertical sections of this new cell have unit width and a height of h , where h is the height of the layout, and the horizontal section has unit height and a length of w . The introduction of the U-shaped cell guarantees that no configuration having a width larger than w is generated. We then apply our wire-length minimizing algorithm to this configuration. Let $C_\square^*$ be the configuration generated by our algorithm minimizing the total wire length. Removing the U-shaped cell from configuration $C_\square^*$ results in a configuration C^* representing a solution to the min_L -problem.

Configuration C^* satisfies $w(C^*) \leq w$ and $l(C^*)$ is as small as possible without exceeding width w . However, the configuration generated does not necessarily correspond to a configuration of minimum width among all configurations having a total wire length of $l(C^*)$. The reason for this lies in the fact that our algorithm minimizing the total wire length does not necessarily minimize the width of sublayouts in the configuration it generates, it only minimizes the overall width. Removing the U-shaped cell leaves a sublayout to which this non-optimality could apply. However, in an additional $O(n \log n)$ time we can generate from C^* a configuration of total wire length $l(C^*)$ and having minimum width. The approach for doing this is described in next.

Let $G_{C_\square^*}$ and $F_\square^*$ be the constraint graph and the spanning forest representing the constraint graph at the termination of the wire-length minimizing algorithm. From both structures we remove the U-shaped cell and, in order to preserve width w , we move cell U_l (resp. U_r) one position to the right (resp. to the left). Let G_{C^*} and F^* be the new constraint graph and the spanning forest, respectively. Forest F^* contains a moving tree of potential $+\frac{1}{2}$ consisting of vertex u_l only and a staying tree of potential $-\frac{1}{2}$ consisting only of vertex u_r . F^* can contain other moving trees; namely moving trees whose movement to the right would decrease the total wire length, but increase in the width. Hence, moving any of these moving trees to the right would cause cell U_r to move to

the right as well. At the same time, forest F^* can contain zero trees whose movement to the right does not change the total wire length, but reduces the width (to be more precise, their movement would allow cell U_l to move to the right as well). We generate a configuration minimizing the width by completing another iteration of the wire-length minimizing algorithm with the following modification. We assign vertex u_r a potential of $pot(u_r) = -\frac{1}{2} - n_h$ and then identify moving, staying, and zero trees in forest F^* . This new potential guarantees that cell U_r remains at its position. If possible, zero trees and the moving tree containing vertex u_l move further to the right and thus reduce the initial width of w .

Summarizing the above discussion allows us to state the following theorem.

Theorem 7.1 *Given a configuration and its constraint graph, the min- l -problem can be solved in $O(n_h \cdot n \log n)$ time.*

7.2 Compacting with a given total wire length l

Consider now the min- w -problem in which we are given a configuration C_f and a bound l and are to determine a configuration C^* having total wire length at most l and minimizing the width among all those configurations. In this section we present an algorithm solving the min- w -problem in $O(n_h \cdot n \log n)$ time.

--- The first step of this algorithm generates the configuration minimizing the total wire length by using the algorithm presented in Section 6. Let C_f be the generated configuration, G_{C_f} be the corresponding constraint graph, and F_{C_f} be the spanning forest representing G_{C_f} . Similar to the min- l -problem, if $l(C_f) > l$, the min- w -problem has no solution. If $l(C_f) = l$, then $C_f = C^*$. In addition, if there exists a path from vertex u_l to vertex u_r in F_{C_f} , C_f represents a configuration of minimum width and $C_f = C^*$. When $l(C_f) < l$ and there exists no path from u_l to u_r in F_{C_f} , we generate the final configuration C^* from C_f by executing at most n_h iterations, with each iteration using $O(n \log n)$ time.

Assume now that $l(C_f) < l$ and that there exists no path from u_l to u_r in F_{C_f} . Configuration C^* is found by keeping the position of cell U_r fixed and identifying, over a sequence of iterations, cells whose movement to the right increases the total wire length to at most l , while reducing the layout width by as much as possible. Let C_i be the configuration and F_i be the tight spanning forest representing the constraint graph at the beginning of the i -th iteration, $i \geq 1$, with $C_1 = C_f$

and $F_1 = F_{C_1}$. The objective of the i -th iteration is to partition the cells into two sets, S_l and S_r , such that $U_l \in S_l$, $U_r \in S_r$, the potential of S_l is as large as possible, and moving the cells in S_l one position to the right reduces the width of the layout by one. Similar to the approach used for the wire-length minimizing algorithm, we do not identify the cells belonging to S_l using the constraint graph. Instead, we use forest F_i . Forest F_i does not contain all the arcs of weight 0 of the constraint graph and thus identifying set S_l on F_i does not necessarily correspond to a collection of cells that can move to the right. During the iterations the forest representing the constraint graph changes so that after at most n_h iterations no further movement of cells to the right is possible without exceeding a total wire length of l .

We next describe the i -th iteration in detail. Let T be the tree in F_i containing u_l . At the beginning of the i -th iteration, there exists no path from u_l to u_r . Vertex u_r may or may not be in tree T . The first step of the i -th iteration identifies in T a right-splitable subtree of maximum potential containing vertex u_l , but not vertex u_r . In order to guarantee that u_r does not get included into the right-splitable subtree, we set $pot(u_r) = -\frac{1}{2} - n_h$. Observe that after this change in potential for vertex u_r , any right-splitable subtree in forest F_i has negative potential. This holds in the first iteration since C_1 is a configuration of minimum total wire length. It holds in the later iterations since the total wire length can only be reduced when cell U_r is pushed to the right as well. In order to find the right-splitable subtree of maximum potential in T , we invoke algorithm POT-PRS(T, u_l). The entry $pot.m(T, u_l)$ generated by this algorithm represents the maximum potential of the right-splitable subtree of F_i containing vertex u_l but not u_r . We identify the vertices in this subtree by assigning to vertex u_l the status "M" and invoking algorithm SM-STATUS(T, u_l) for completing the status assignment. Let T_{m_l} be this tree. Even though tree T_{m_l} has negative potential, we refer to it as a moving tree (since its cells may move to the right). We apply algorithm Z-STATUS to every tree in F_i to identify the zero subtrees.

After having completed the status assignments described above, we partitioned forest F_i into tree T_{m_l} containing vertex u_l , a number of trees having potential zero, and a number of staying trees one of which contains vertex u_r . Let T_{s_r} be the staying tree containing vertex u_r . The cells corresponding to the vertices in staying tree can be viewed as forming set S_r defined earlier for describing the overall approach of the i -th iteration. The second step of the i -th iteration determines how much the cells corresponding to the vertices of tree T_{m_l} and the zero trees can

move to the right. This is done by using an approach similar to the one used in Section 6.1. We set up a graph G in which every vertex corresponds to a just identified tree and we use the constraint graph to determine the arcs between the vertices. We then determine the length of the shortest path from the vertex representing tree T_{m_i} and from every zero tree to a vertex representing a staying tree. The length of every such shortest path represents the maximum distance the cells corresponding to the vertices of T_{m_i} (or a zero tree) can move to the right. Assume the vertices in every zero tree have moved to the right according to the length of the shortest path. Their movement does not change the total wire length. Let d be the shortest path length determined for the vertex representing T_{m_i} . If $d > 0$, the cells corresponding to the vertices in T_{m_i} can move at least one unit to the right and doing so increases the total wire length by $-pot(T_{m_i} \setminus \{u_i\})$ (recall that $pot(T_{m_i} \setminus \{u_i\}) < 0$). If $l(C_i) - d \cdot pot(T_{m_i} \setminus \{u_i\}) \leq l$, we push the cells in T_{m_i} d positions to the right and update the constraint graph accordingly. If $l(C_i) - d \cdot pot(T_{m_i} \setminus \{u_i\}) > l$, we push the cells in T_{m_i} $\lfloor (l - l(C_i)) / (-pot(T_{m_i} \setminus \{u_i\})) \rfloor$ units to the right and terminate the algorithm. In this case any further decrease in the width would result in a total wire length exceeding l . If $d = 0$, the cells in T_{m_i} cannot be pushed to the right.

The last step of the iteration generates forest F_{i+1} . Forest F_{i+1} contains tree T_{m_i} , tree T_{s_r} , zero trees, the other staying trees, and the arcs in the shortest paths. Before proceeding with the $(i + 1)$ -st iteration we check whether there exists a path from u_l to u_r in a tree of F_{i+1} . If there is, the algorithm has found configuration C^* ; i.e., $C_{i+1} = C^*$.

Theorem 7.2 *Given a configuration and its constraint graph, the min_w-problem can be solved in $O(n_h \cdot n \log n)$ time.*

Proof: We first show that the algorithm described above generates a configuration of minimum width among all configurations having total wire length not more than l . Assume it does not. Then, in the generated configuration, we could push cell U_l , together with other cells, to the right without changing the position of cell U_r and without increasing the total wire length. In other words, there would exist a partition of the cells into S_l and S_r so that $U_l \in S_l$, $U_r \in S_r$ and the cells in S_l can be pushed at least one unit to the right without changing the positions of the cells in S_r . The termination rules used in the algorithm (the algorithm terminates when either there is a path from u_l to u_r in F_i or the total wire length is at least l) guarantees that we can not find such a partition in the final configuration.

It is clear that one iteration of the algorithm uses $O(n \log n)$ time. We sketch the argument similar to the one used in the proof of Lemma 6.1 showing that the algorithm terminates in at most n_h iterations. Define the notion of a moving potential, as done in Section 6.3, to be the potential of the vertices in T_{m_i} . The moving potential of the i -th iteration is $pot(T_{m_i})$. From the construction of forest F_{i+1} and the property that every right-splittable subtree of a staying tree detected during the i -th iteration has negative potential, it follows that $pot(M_i) > pot(M_{i+1})$. Since $pot(M_1) \leq -\frac{1}{2}$, the moving potential decreases by at least 1 in each iteration, and $pot(M_i) \geq -\frac{1}{2} - n_h$, the algorithm terminates after at most n_h iterations. $\square$

7.3 Minimizing $\alpha \cdot w + \beta \cdot l$

In this section we show how to determine, in $O(n_h \cdot n \log n)$ time, a configuration C^* that minimizes $\alpha \cdot w(C^*) + \beta \cdot l(C^*)$ when α and β are positive constants. Observe that the cases $\alpha = 0$ and $\beta = 0$ correspond to minimizing the total wire length and the width, respectively, and have already been discussed. Instead of minimizing $\alpha \cdot w(C^*) + \beta \cdot l(C^*)$, we minimize $\gamma \cdot w(C^*) + l(C^*)$, where $\gamma = \frac{\alpha}{\beta}$.

Depending on the relationship between $\gamma = \frac{\alpha}{\beta}$ and n_h , the algorithm will proceed differently. Assume first that $\gamma \leq n_h$. In this case we generate the final configuration C^* by assigning to the vertices u_l and u_r of the constraint graph a potential depending on γ and then use the algorithm described in Section 6. Recall that in our wire-length minimizing algorithm we set $pot(u_l) = +\frac{1}{2}$ and $pot(u_r) = -\frac{1}{2}$. Now we set $pot(u_l) = +\gamma$ and $pot(u_r) = -\gamma$. Conceptually, this corresponds to introducing γ wires of unit width between cell U_l and cell U_r . For any configuration C , moving cell U_r (resp. U_l) one unit to the right without changing the positions of the other cells results in a configuration having cost $\gamma \cdot (w(C) + 1) + l(C) = \gamma \cdot w(C) + \gamma + l(C)$ (resp. $\gamma \cdot w(C) - \gamma + l(C)$). The total wire length of configuration C is $\gamma \cdot w(C) + l(C)$ and thus an algorithm minimizing the total wire length minimizes the tradeoff function. With respect to the achieved running time, we again reduce the potential associated with all moving cell by at least 1 in each iteration and thus perform at most $n_h + \gamma$ iterations. Hence, configuration C^* is generated in $O((\gamma + n_h) \cdot n \log n) = O(n_h \cdot n \log n)$ time.

For $\gamma = O(n_h)$, the approach described above results in an $O(n_h \cdot n \log n)$ time algorithm. For $\gamma = \Omega(n_h)$, this algorithm still generates a correct configuration, but the time bound is $O(\gamma \cdot n \log n)$. By using a property of configuration C^* and the algorithm solving the min_1-problem, we can achieve

an $O(n_h \cdot n \log n)$ running time. Let C_{left} be a configuration of minimum width. We start by showing that when $\gamma > n_h$, $w(C^*) = w(C_{left})$; i.e., configuration C^* has minimum width. Clearly, we have $w(C^*) \geq w(C_{left})$. Assume now that $w(C^*) > w(C_{left})$. Let G_{C^*} and $G_{C_{left}}$ be the constraint graphs of C^* and C_{left} , respectively. Since C_{left} is a configuration of minimum width, the length of the shortest path from u_l to u_r in $G_{C_{left}}$ is zero. Since $w(C^*) > w(C_{left})$, the length of the shortest path from u_l to u_r in G_{C^*} is positive. This implies that there exists a set of arcs whose removal partitions G_{C^*} into two subgraphs G_l and G_r such that u_l and u_r are in G_l and G_r , respectively, and the weight of every arc going from a vertex in G_l to a vertex in G_r is positive. Thus, we can generate a new configuration C' with $w(C') < w(C^*)$ by keeping the cells corresponding to the vertices in G_r fixed at their position in C^* and moving the cells corresponding to the vertices in G_l at least one unit to the right. This move decreases the layout width by one unit and increases the total wire length by at most n_h units. Therefore,

$$\begin{aligned} \gamma \cdot w(C') + l(C') &\leq \gamma \cdot (w(C^*) - 1) + (l(C^*) + n_h) \\ &\leq \gamma \cdot w(C^*) + l(C^*) + (n_h - \gamma) \\ &< \gamma \cdot w(C^*) + l(C^*) \end{aligned}$$

This contradicts the assumption that C^* is a configuration minimizing $\gamma \cdot w + l$. Hence, $w(C_{left}) = w(C^*)$.

The width of configuration C_{left} can be determined in $O(n \log n)$ time. Once the width of configuration C^* is known, we use the algorithm for the min_L-problem described in Section 7.1 to generate configuration C^* . We can thus state the following theorem.

Theorem 7.3 *Given a configuration and its constraint graph, a configuration C^* minimizing the tradeoff function $\alpha \cdot w + \beta \cdot l$ can be generated in $O(n_h \cdot n \log n)$ time.*

8 Extensions

In this section we show that the compaction algorithms presented in the previous sections are fairly robust towards changes in the underlying model. A number of assumptions can be modified without causing an increase in time complexity. The algorithms presented assumed the following model:

1. All wires have the same associated cost. Only the length of horizontal wires change during the compaction and thus the function minimized when minimizing the total wire length is

$\sum_{i=1}^{n_h} l(H_i)$, where $l(H_i)$ is the length of horizontal wire H_i .

2. The constraint graph is based on visibility between layout elements.
3. Two vertical wires incident to a common horizontal wire and forming a “step” have a constraint between them; i.e., a wire is partitioned into vertical and horizontal segments as shown in Figure 2.
4. No jogs are introduced into the vertical wires.
5. Compaction is done in a 1-layer environment; i.e., we do not allow any overlap between layout elements.

We next show how each one of these 5 assumptions can be relaxed and/or changed. Assume every horizontal wire H_i has an integer cost c_i associated with it. Minimizing the total wire length corresponds to minimizing $\sum_{i=1}^{n_h} c_i \cdot l(H_i)$, where $l(H_i)$ is the length of horizontal wire H_i . By viewing a horizontal wire of cost c_i as c_i horizontal wires of unit cost, our wire-length minimizing algorithms minimize this new cost function. However, the running time is now $O(C \cdot n \log n)$, where $C = \sum_{i=1}^{n_h} c_i$. Having the quantity C enter the running time is reasonable as long as the c_i 's are small integers, but it is undesirable for arbitrarily large integers. We point out that by recasting the problem in terms of the classical minimum cost network flow and using Orlin's [2] algorithm, the problem can be solved in $O(n \log n \cdot n \log n)$ time.

While we defined the constraint graph as a graph containing information about the visibility and distance between layout elements, no step of our algorithms makes use of this fact in a crucial way. Non-visibility constraints inducing arcs with non-negative weight can easily be added and the resulting constraint graph can be used by our algorithms. Observe that our algorithms do not check whether the configuration described by the initial constraint graph is indeed feasible, but assume that it is.

The constraint graph defined in Section 2 partitioned wires into vertical and horizontal wires as shown in Figure 2. This partitioning does not allow vertical wires on “opposite sides” of a horizontal wire to slide freely. For example, we may want to allow the situation shown in Figure 9(a), as well as the one shown in Figure 9(b). A simple change in the definition of the constraint graph allows us to treat vertical wires this way. We chose the more restrictive version of partitioning the wires

Figure 9: The length of the visible segment connecting two vertical wires.

since it lends itself to somewhat simpler constraint graph. For the alternative partitioning into vertical and horizontal wires we use the following constraint graph. As before, we partition layout components and vertical wires into cells. In addition, every horizontal wire H_i introduces two cells, one, $U_{i,1}$, corresponds to the left endpoint and the other one, $U_{i,2}$, corresponds to the right endpoint of wire H_i . These at most $2n_h + n_v + n_r$ cells form the vertices of the constraint graph. The arcs of the constraint graph include the arcs based on the visibility and the distance between layout components and vertical wires, as described in Section 2. In addition, we add the following arcs incident to cells corresponding to endpoints of horizontal wires.

For each horizontal wire H_i , we add the arc $(u_{i,1}, u_{i,2})$ whose weight is the length of horizontal wire H_i . Let U_j be a cell connected to horizontal wire H_i . If U_j is connected to H_i through a vertical wire, we add an arc from $u_{i,1}$ to u_j and an arc from u_j to $u_{i,2}$. The weight of arc $(u_{i,1}, u_j)$ (resp. $(u_j, u_{i,2})$) is the distance between the left (resp. right) endpoint of wire H_i and the vertical wire of U_j . In Figure 10 such a situation occurs between cell U_2 and horizontal wire H_1 . Assume that cell U_j is connected to the left (resp. right) endpoint of wire H_i through a layout component, as, for example, are cell U_1 and the left endpoint of horizontal wire H_1 in Figure 10. In this case we add the arc $(u_{i,1}, u_j)$ (resp. $(u_j, u_{i,2})$) having weight zero to ensure that cell U_j remains attached to the left (resp. right) endpoint of wire H_i . The potential of every vertex in the constraint graph is set as follows. For vertices u_r and u_l , we have $pot(u_r) = -\frac{1}{2}$ and $pot(u_l) = +\frac{1}{2}$. Every vertex $u_{i,1}$ has $pot(u_{i,1}) = +1$ and every vertex $u_{i,2}$ has $pot(u_{i,2}) = -1$. For every other vertex u_j , we set $pot(u_j) = 0$. Conceptually, when the cell corresponding to a left (resp. right) endpoint of a horizontal wire is pushed 1 unit to the right without changing the positions of the other cells, the total wire length decreases (resp. increases) by 1 unit. Given a configuration, this new constraint

Figure 10: A layout and its constraint graph.

graph can be built in $O((n_h + n_v + n_r) \log(n_h + n_v + n_r))$ time. We then use the constraint graph without any further changes in our algorithms and determine the configuration minimizing the total wire length in $O(n_h \cdot n \log n)$ time.

The algorithms presented can change the length of the horizontal wires, but they do not introduce jogs into vertical wires. The positions in the vertical wires where jogs might be useful can be bounded in terms of n_v and n_r . For the model used in the paper, every vertical edge of a layout component introduces at most $2n_v$ jogs and every vertical wire introduces at most $2n_v$ jogs. Hence, the total number of positions in vertical wires where a jog might be beneficial is $O(n_v(n_v + n_r))$. These positions can be determined before the compaction algorithm is invoked. Hence, by splitting the n_v vertical wires into $O(n_v(n_v + n_r))$ vertical wires and running our algorithms on the resulting configuration, jogs are introduced whenever they reduce the total wire length or the layout width. Of course, doing this does not guarantee any bound on the total number of jogs introduced. The overall running time is now $O(N_h \cdot N \log N)$, where $N \leq n_v(n_v + n_r) + n_h + 2$ and $N_h \leq n_v(n_v + n_r) + n_h$.

Our algorithms and their extensions/modifications described above perform compaction in a 1-layer model. Another commonly used model is the 3-layer model in which layout components, vertical wires, and horizontal wires are placed on a separate layer, respectively. Overlap between layout elements placed on different layers is now allowed. By deleting arcs of the constraint graph

that correspond to constraints no longer valid, our algorithms can perform compaction in a 3-layer model. We omit any further details since they tend to be rather model-specific and do not introduce new aspects to the problem.

References

- [1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] R. K. Ahuja, T. L. Magnant, and J. B. Orlin. Some recent advances in network flows. *SIAM Review*, 33(2):175–219, June 1991.
- [3] D. G. Boyer. Symbolic layout compaction review. In *Proceedings of 25th ACM/IEEE Design Automation Conference*, pages 383–389, 1988.
- [4] S. Gao, M. Kaufmann, and F. M. Maley. Advances in homotopic layout compaction. In *Proceedings of 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 273–282, 1989.
- [5] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.
- [6] Y. Z. Liao and C. K. Wong. An algorithm to compact a VLSI symbolic layout with mixed constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(2):62–69, April 1983.
- [7] F. M. Maley. *Single-Layer Wire Routing and Compaction*. The MIT Press, 1990.
- [8] D. Marple. A hierarchy preserving hierarchical compactor. In *Proceedings of 27th ACM/IEEE Design Automation Conference*, pages 375–381, 1990.
- [9] D. A. Mlynski and C. H. Sung. Layout compaction. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 199–235. Elsevier Science Publ., 1986.
- [10] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer Verlag, 1985.
- [11] W. L. Schiele. Improved compaction by minimized length of wires. In *Proceedings of 20th ACM/IEEE Design Automation Conference*, pages 121–121, 1983.
- [12] M. Schlag, F. Luccio, P. Maestrini, D. T. Lee, and C. K. Wong. A visibility problem in VLSI layout compaction. In F. P. Preparata, editor, *Advances in Computing Research: VLSI Theory*, pages 259–282, Greenwich, Connecticut, 1984. JAI Press.
- [13] B. X. Weis and D. A. Mlynski. A graphtheoretic approach to the relative placement problem. *IEEE Transactions on Circuits and Systems*, 35(3):286–293, 1988.