1993

New Algorithms for Minimizing the Longest Wire Length During Circuit Compaction

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

Hung-Yi Tu

Report Number:
93-013

TR #93-013

Algorithmica (1997) 17: 426–448

Algorithmica

New Algorithms for Minimizing the Longest Wire Length During Circuit Compaction[1]

S. E. Hambrusch[2] and Hung-Yi Tu[3]

Abstract. Consider the problem of performing one-dimensional circuit compaction for a layout containing n_h horizontal wires and n layout cells. We present new and efficient constraint-graph-based algorithms for generating a compacted layout in which either the length of the longest wires or a user-specified tradeoff function between the layout width and the longest wire length is minimized. Both algorithms have an $O(n_h \cdot n \log n)$ running time. The concept employed by our algorithms is that of assigning speeds to the layout cells. Speeds are computed by performing path computations in subgraphs of the constraint graphs. A compacted layout is generated over a number of iterations, with each iteration first determining speeds and then moving the layout elements to the right according to the computed speeds. Each iteration produces a better layout and after at most $n \cdot n_h$ iterations the final layout is produced.

Key Words. Analysis of algorithms, Circuit layout, Compaction, Layout width, Longest wire length, Path computations.

1. Introduction. Circuit compaction is the process of converting a symbolic layout into an actual layout that satisfies the design rules and minimizes a set of objective functions [5], [9]. One-dimensional compaction allows layout elements to slide in one direction only and is often preferred over computationally intractable two-dimensional compaction. Consider performing one-dimensional compaction along the horizontal direction. Layout elements are now allowed to slide horizontally as long as no constraint is violated and the relative order of the layout elements is preserved. The length of a horizontal wire can change during this process. A layout generated by most conventional width-minimizing compaction algorithms contains unnecessarily long horizontal wires. Controlling the wire length is crucial in circuit design [4], [9], [10], [13]. In this paper we present new and efficient algorithms for minimizing the length of the longest horizontal wires during one-dimensional compaction and for minimizing a given tradeoff function between the length of the longest wires and the width of the layout.

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 a layout which minimizes the length of the longest wires and for which, among all layouts having the same minimum wire length,

[1] This research was supported in part by DARPA under Contract DABT63-92-C-0022. The views and conclusions contained in this paper are those of the authors and should not be interpreted as representing official policies, expressed or implied, of the U.S. government.

[2] Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA. seh@cs.purdue.edu.

[3] Department of Computer Science and Information Management, Providence University, Taichung, Taiwan, Republic of China.

Received December 10, 1994; revised October 15, 1995. Communicated by A. S. LaPaugh.

one of minimum width is generated, $n \leq 2n_h + n_v + n_r$. We also consider the problem of generating a layout minimizing a specified tradeoff between the longest wire length and the layout width. More precisely, given a tradeoff function $\alpha \cdot W + \beta \cdot L$ between the layout width W and the longest wire length L, and constants $\alpha, \beta > 0$, we present an $O(n_h \cdot n \log n)$-time algorithm for minimizing $\alpha \cdot W + \beta \cdot L$.

We briefly sketch the approach underlying our algorithms. A *configuration* of a layout assigns to the leftmost edge of every layout component and every wire an x-position in the layout area. We assume that the input is a feasible configuration (i.e., the x-positions associated with the layout components result in a layout satisfying the constraints). From this initial configuration, we generate a configuration minimizing the longest wire length or a tradeoff function over a number of iterations. Each iteration produces a feasible configuration with smaller longest wire length or a smaller tradeoff function value, respectively. The relevant constraints and distances are represented by graphs. Within the area of compaction methods our algorithms are viewed as constraint-graph-based solutions [5]. A new configuration is generated from the previous one by moving layout elements to the right. A crucial parameter in this movement is the speed of a layout element. In the algorithm minimizing the longest wire length, speeds are computed by a longest-path computation and in the tradeoff algorithm by a shortest-path computation. In the longest wire minimizing algorithm the movement to the right reduces the length of longest wires. At the same time, it can increase the length of wires that are originally not the longest ones and it changes distances between layout elements. One iteration stops when a nonlongest wire turns into a longest wire or any further movement would violate constraints. If a further reduction in the longest wire length is possible, the next iteration continues the movement with updated speeds. The scenario for the tradeoff minimizing algorithm is similar. For both compaction problems, a final configuration is found after at most $n_h \cdot n$ iterations. The $O(n_h \cdot n \log n)$-time bound is achieved by using data structures to perform updates and by performing computations on demand.

The best previously known algorithm for minimizing the longest wire length follows from [6]. The algorithm described in that paper uses a different approach and, when translated into our framework, it gives $O(\log L \cdot (n_h + n_r) \cdot n)$ time, where L is the longest wire length in the initial layout. Ignoring constants, our algorithm can be viewed as faster for $\log n < (1 + n_r/n_h) \cdot \log L$. Other compaction algorithms minimizing the layout width, the longest wire length, or the total wire length are described in [2]–[7], [11], [13], and [15]. None of these algorithms can be used to optimize a tradeoff function between the layout width and the longest wire length.

The paper is structured as follows. In Section 2 we state relevant definitions. In Section 3 we discuss our approach for the algorithm minimizing the longest wire length. Section 4 describes how to compute the speeds and how to determine the points in time when new speeds have to be computed. In Section 5 we present an $O(n_h^2 \cdot n^2)$-time algorithm and in Section 6 we show how to reduce the running time to $O(n_h \cdot n \log n)$. In Section 7 we discuss how to generate a configuration minimizing a given tradeoff function. Sections 8 and 9 conclude.

2. Preliminaries and Definitions. In this section we give notation and define the different graphs used by our algorithms. Throughout this paper the wires of a layout are partitioned into horizontal wires and vertical wires according to the following rules. A

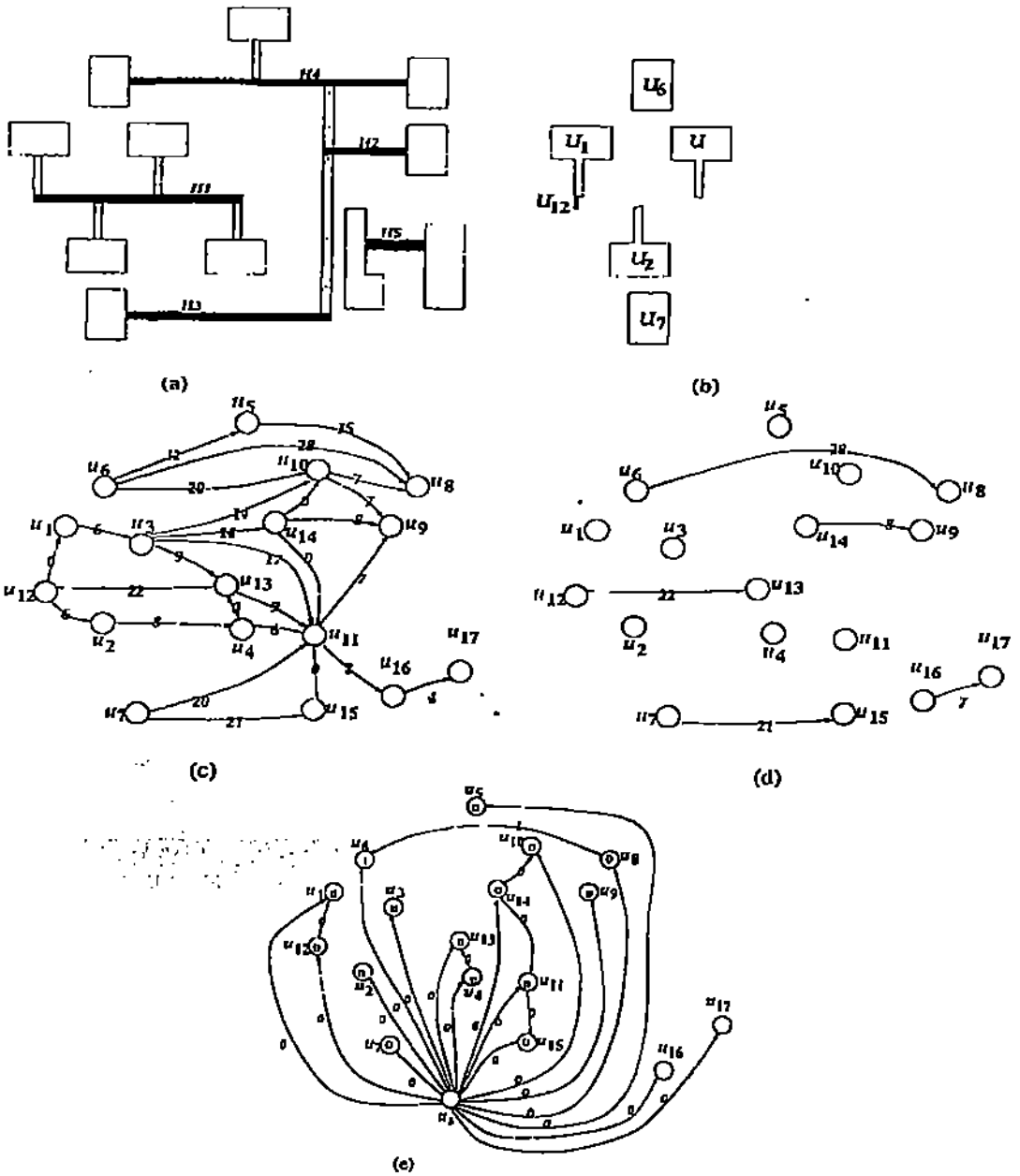

Fig. 1. (a) A layout. (b) The cells of the layout. (c) The distance graph of the layout. (d) The wire graph of the layout. (e) The speed graph of the layout.

horizontal wire is a horizontal segment of maximum length connecting layout components and vertical wires. Every wire segment not belonging to a horizontal wire forms a vertical wire. For example, the layout shown in Figure 1(a) contains five horizontal wires and seven vertical wires. Observe that the endpoint of a vertical wire is incident to either a layout component or a horizontal wire. No endpoint of another wire is incident to any other position of a vertical wire.

We group layout elements into *cells* as follows. Partition the layout components and vertical wires into maximal sets, so that each set represents one rigid object that moves as one entity. One such rigid object induces one cell of the layout. In addition, if an endpoint of a horizontal wire is not connected to a layout component, then this endpoint

induces a rectangular cell. Such a cell has width 0 and its height equals the width of the horizontal wire. Figure 1(b) shows the cells induced by the layout of Figure 1(a). Cells U_{12}, U_{13}, U_{14}, and U_{15} are cells induced by the endpoints of horizontal wires H_1, H_2, and H_3, respectively. Having an endpoint of a horizontal wire that is not connected to a layout component induce a cell simplifies our algorithm.

Let n be the number of cells in configuration C, $n \leq 2n_h + n_v + n_r$. In our algorithms the information about configuration C is represented by two directed, weighted, n-vertex graphs, the *distance graph* $G_d = (V_d, A_d)$ and the *wire graph* $G_w = (V_w, A_w)$. The wire graph records the length of the longest wire between cells. The distance graph models constraints between cells. In related literature this graph is also called the constraint graph. Throughout this paper we assume that a constraint between two cells is induced by visibility and that the weight of the arc is the distance between two cells. When the constraints come from visibility between cells, the distance graph is a planar graph and, as a consequence, it has $O(n)$ arcs. However, no step of our algorithms makes use of the planarity. Our algorithms can handle other inequalities specified by a user or generated by a CAD system. Any constraint graph consisting of $O(n)$ arcs can be used and will give the same running time.

We next define visibility and distance between cells. Since cells move to the right, we only need to capture visibility to the right of a cell. A cell U_j is *visible* from a cell U_i if and only if one can draw a horizontal line connecting U_i and U_j starting at a position x_1 and ending at position x_2, $x_1 \leq x_2$, so that no position between x_1 and x_2 on this horizontal line is occupied by a cell. Every such horizontal line between two cells has a length of $x_2 - x_1$. The *distance* between U_i and U_j is the minimum over all lengths associated with horizontal lines connecting the two cells and inducing visibility between them.

In both the distance and the wire graph the vertices correspond to the cells of configuration C (we thus have $V_d = V_w$). Vertex u_i of either the distance or the wire graph represents cell U_i, $1 \leq i \leq n$. The arcs and weights in the distance graph are formed as follows. When cell U_j is visible from cell U_i, the distance graph contains the arc (u_i, u_j); i.e., the arc from vertex u_i to vertex u_j. Its weight, $d(u_i, u_j)$, is set to the distance between U_i and U_j in configuration C.

In addition, we add the following arcs not corresponding to visibility between cells. These arcs are incident to cells induced by horizontal wires not connected to layout components. Let H be a horizontal wire for which one of the endpoints is not connected to a layout component. Let $V_{up,l}$ and $V_{up,r}$ be the leftmost and rightmost vertical wire incident to H, respectively, and lying above H. Observe that no such vertical wire may exist or that $V_{up,l}$ and $V_{up,r}$ may be identical. Vertical wires $V_{dw,l}$ and $V_{dw,r}$ are defined in the analogous way for the vertical wires lying below H. Assume the left endpoint of H is not connected to a layout component and let U_i be the cell induced by the left endpoint of H. If cell U_j contains vertical wire $V_{up,l}$ (resp. $V_{dw,l}$), we include the arc (u_i, u_j). Its weight $d(u_i, u_j)$ is set to the distance between the left endpoint of H and vertical wire $V_{up,l}$ (resp. $V_{dw,l}$). Observe that at least one of these two arcs has a weight of 0. Arcs (u_{12}, u_1), (u_{12}, u_2), (u_{14}, u_{10}), and (u_{14}, u_{11}) of the distance graph shown in Figure 1(c) are created according to these rules. Assume now that the right endpoint of H is not connected to a layout component and that cell U_i is the cell induced by the right endpoint. Then, if $V_{up,r}$ (resp. $V_{dw,r}$) is contained in cell U_j, we include the arc (u_j, u_i)

and set its weight accordingly. Arcs (u_3, u_{13}), (u_4, u_{13}), and (u_{11}, u_{15}) of Figure 1(c) are created in this way.

The arcs of the wire graph G_w are formed as follows. Arc (u_i, u_j) is in A_w if cell U_j is visible from cell U_i and there exists at least one horizontal wire having its left endpoint incident to U_i and its right endpoint incident to U_j. The weight of this arc, $w(u_i, u_j)$, is set to the length of the longest such wire between U_i and U_j. Figure 1(d) shows the wire graph of the layout in Figure 1(a). G_w contains at most n_h arcs and every arc of the wire graph is also an arc of the distance graph. When layout components are not restricted to be rectangles, the weights associated with the two arcs can be different. In Figure 1 the longest wire connecting cells U_{16} and U_{17} has length 7, but the distance between U_{16} and U_{17} is 4. Throughout, let $H_{i,j}$ represent the longest wire connecting cells U_i and U_j so that the left endpoint of this horizontal wire is incident to U_i and its right endpoint is incident to U_j. From now on we assume that among all wires connecting two cells U_i and U_j, all but the longest one have been removed.

Given a configuration, its distance graph and wire graph can be built in $O(n_h + (n_v + n_r)\log(n_v + n_r))$ time. This can be done, for example, by sorting the layout components according to their vertical edges and using a plane-sweep approach, together with balanced tree operations [12], [14].

As already stated, we generate new configurations by moving cells to the right with certain speeds. The speeds are specified in the speed assignment. A *speed assignment* assigns to every cell U_i a nonnegative real number $speed(U_i)$, $1 \le i \le n$. Our longest wire minimizing algorithm uses the concept of a legal speed assignment. A speed assignment is *legal* for configuration C when:

(i) If $d(u_i, u_j) = 0$, then $speed(U_i) \le speed(U_j)$.
(ii) If $H_{i,j}$ is a longest wire in C and $w(u_i, u_j) \neq 0$, then $speed(U_i) > speed(U_j)$.

Condition (i) guarantees that, when the distance between U_i and U_j is zero, moving cells U_i and U_j according to their legal speeds keeps the relative order between U_i and U_j in the horizontal direction. It also keeps U_j from overlapping with U_i (which would happen if U_i would have a larger speed). Condition (ii) guarantees that the length of a longest wire decreases. Since arcs of the distance graph having weight 0 and arcs of the wire graph corresponding to longest wires determine the legality of a speed assignment, we represent these arcs in a separate graph, the *speed graph*. The speed graph is used for determining a legal speed assignment in the wire minimizing algorithm.

The speed graph $G_s = (V_s, A_s)$ is a directed graph whose arcs have either weight 0 or 1. It has $V_s = V_d \cup \{u_s\}$, where u_s is a source vertex, and arc set A_s is formed as follows. For every vertex u_i in V_d we include the arc (u_s, u_i) of cost 0. For every arc (u_i, u_j) of weight 0 in the distance graph, the speed graph contains the arc (u_i, u_j) of cost 0. If (u_i, u_j) is an arc in the wire graph G_w representing a longest wire in the layout, the speed graph contains the arc (u_j, u_i) having cost 1. Observe that for an arc (u_i, u_j) coming from the wire graph we reverse the direction of the arc in the speed graph. The length of a path from u_s to u_i is the sum of the costs of the arcs on the path.

Assume there exists a legal speed assignment for configuration C and the longest wire length in C is not zero. Moving the cells to the right according to the speeds specified by a legal speed assignment changes wire lengths and distances between cells. Most importantly, the length of the longest wires is reduced. Cells continue moving to the

right until one of the following two events happens:

(i) an arc (u_i, u_j) having a positive weight in distance graph turns into an arc of weight 0, i.e., a nonzero distance between two cells becomes zero, or
(ii) a wire $H_{i,j}$ which is not a longest wire before the movement to the right turns into a longest wire.

We refer to the first event as a *distance event* and to the second one as a *wire event*. The earliest time at which either event occurs is called the *event time* for configuration C. At event time, the speed assignment is no longer legal. However, there could exist another legal speed assignment that continues to reduce the length of the longest wires. If one exists, we find it and keep moving cells to the right according to the new legal speeds. Otherwise, the current configuration is one having the minimum longest wire length.

3. Correctness of Overall Approach. In this section we establish the relationship between a legal speed assignment and a configuration in which the longest wire length is a minimum. This relationship is crucial to the correctness of our algorithm.

LEMMA 3.1. *Let C be a configuration. The longest wire length in C is not minimized if and only if there exists a legal speed assignment for C.*

PROOF. Assume there exists a legal speed assignment for configuration C. By moving the cells according to the associated speeds, the longest wire length is reduced. Thus the longest wire length could not have been minimized in C.

Assume now that the longest wire length is not minimized in C. Let $x_C(U_i)$ be the x-position of the leftmost vertical edge of cell U_i in configuration C. Let C^* be a configuration in which the longest wire length is a minimum and $x_C(U_i) \leq x_{C^*}(U_i)$ holds for every cell U_i. By shifting the cells of a configuration minimizing the longest wire length to the right such a configuration C^* can always be generated. We next show that setting $speed(U_i) = x_{C^*}(U_i) - x_C(U_i)$, $1 \leq i \leq n$, results in a legal speed assignment for configuration C.

Assume that the distance between U_i and U_j is 0 in C. This implies that cell U_i contains a vertical edge e_i and cell U_j contains a vertical edge e_j, such that e_j is visible from e_i and the distance between e_i and e_j is 0. In configuration C^*, e_j is still visible from e_i. The distance between e_i and e_j in C^* is larger than or equal to 0. Hence, $x_{C^*}(U_i) - x_C(U_i) \leq x_{C^*}(U_j) - x_C(U_j)$ and thus $speed(U_i) \leq speed(U_j)$. Let $H_{i,j}$ be a horizontal wire of maximum length in configuration C. Recall that $H_{i,j}$ has its left endpoint incident to cell U_i and its right endpoint incident to U_j. The length of wire $H_{i,j}$ in configuration C is larger than in C^*. Hence, we have $x_{C^*}(U_i) - x_C(U_i) > x_{C^*}(U_j) - x_C(U_j)$ and $speed(U_i) > speed(U_j)$ follows. Both conditions for a legal speed assignment are thus satisfied. □

There can exist many legal speed assignments for a particular configuration C. Our algorithm determines a legal speed assignment by performing a single-source longest-path computation in the speed graph. Assume that the speed graph contains no positive

cycle. Let $\mathcal{L}(u_i)$ be the length of the longest path from source u_s to vertex u_i and let $P = \langle u_s, u_{k_1}, \ldots, u_{k_l} = u_i \rangle$ be the associated path. Intuitively, path P implies that one can "travel" from cell U_{k_1} to cell U_i using $\mathcal{L}(u_i)$ longest wires. By "travel" we mean that there exists a path from u_{k_1} to u_i in the speed graph that corresponds to moving from cell U_{k_1} to cell U_i along $\mathcal{L}(u_i)$ horizontal wires of maximum length, through the interior of cells, and from one cell to another cell as long as the distance between these two cells is 0. Moving every cell U_{k_r} on path P to the right with a speed of $\mathcal{L}(u_{k_r})$, $1 \leq r \leq l$, reduces the length of every longest horizontal wire associated with an arc in path P. The next lemma gives a formal argument showing that setting speeds equal to the longest path entries results in a legal speed assignment.

LEMMA 3.2. *Let G_s be the speed graph of configuration C. If G_s contains no positive cycle, setting speed$(U_i) = \mathcal{L}(u_i)$, $1 \leq i \leq n$, results in a legal speed assignment for C.*

PROOF. Let (u_i, u_j) be an arc of cost 0 in G_s. Since G_s contains no cycle of positive cost, we have $\mathcal{L}(u_i) \leq \mathcal{L}(u_j)$. Setting *speed*$(U_j) = \mathcal{L}(u_j)$ and *speed*$(U_i) = \mathcal{L}(u_i)$ gives *speed*$(U_i) \leq$ *speed*(U_j).

Assume now that (u_j, u_i) is an arc of cost 1 in the speed graph. The arc is induced by a longest wire having a left endpoint incident to cell U_i and a right endpoint incident to cell U_j. Since there exists no cycle of positive length, G_s cannot contain a path from u_i to u_j. We thus have $\mathcal{L}(u_i) \geq \mathcal{L}(u_j) + 1$. Setting *speed*$(U_j) = \mathcal{L}(u_j)$ and *speed*$(U_i) = \mathcal{L}(u_i)$ gives *speed*$(U_i) >$ *speed*(U_j). Hence, both conditions for a legal speed assignment are satisfied. □

In order to use the existence of a positive cycle as the indication that no legal speed assignment exists, we need to prove the following lemma.

LEMMA 3.3. *Let G_s be the speed graph of configuration C. If G_s contains a cycle of positive cost, then no legal speed assignment exists.*

PROOF. Assume $P = \langle u_{i_1}, u_{i_2}, u_{i_3}, \ldots, u_{i_{t-1}}, u_{i_t}, u_{i_1} \rangle$ is a positive cycle in G_s. P contains at least one arc, say $(u_{i_j}, u_{i_{j+1}})$, of cost 1. By definition of G_s, arc $(u_{i_j}, u_{i_{j+1}})$ corresponds to a longest wire with a left endpoint incident to cell $U_{i_{j+1}}$ and a right endpoint incident to cell U_{i_j}. Assume there exists a legal speed assignment for C. Then, we have *speed*$(U_{i_{j+1}}) >$ *speed*(U_{i_j}). This implies *speed*$(U_{i_1}) \leq$ *speed*$(U_{i_2}) \leq \cdots \leq$ *speed*$(U_{i_j}) <$ *speed*$(U_{i_{j+1}}) \leq \cdots \leq$ *speed*$(U_{i_t}) \leq$ *speed*(U_{i_1}), which is not possible. Hence, no legal speed assignment can exist for C. □

We summarize the discussion in the following theorem.

THEOREM 3.1. *There exists a legal speed assignment for C if and only if the speed graph of C contains no positive cycle. If no positive cycle exists, setting speed$(U_i) = \mathcal{L}(u_i)$, $1 \leq i \leq n$, gives a legal speed assignment.*

4. Finding a Legal Speed Assignment and the Event Time. In this section we first give an algorithm for determining a legal speed assignment for a given configuration C and then describe how to determine the event time induced by the legal speed assignment.

As described in the previous section, we determine a legal speed assignment by performing a single-source longest-path computation on the speed graph with vertex u_s as the source. The entries $speed(u_i)$ are computed similar to the Bellman–Ford algorithm [1] for solving a single-source shortest-path problem on a graph with negative weights. We use the technique of *relaxation*, in which the *speed*-entries (and thus the length of the longest paths) are progressively increased. An arc (u_i, u_j) in speed graph G_s is *relaxed* if

$$speed(U_i) \leq speed(U_j) \quad \text{when} \quad cost(u_i, u_j) = 0,$$
$$speed(U_i) < speed(U_j) \quad \text{when} \quad cost(u_i, u_j) = 1.$$

Figure 2 gives a description of the algorithm, to which we refer as algorithm LEGAL-SPEED. Since vertex u_s has an arc of cost 0 to every other vertex in speed graph G_s, we initialize $speed(U_i) = 0$ for every cell U_i and put all arcs of G_s in a first-in-first-out queue Q. When an arc (u_i, u_j) is extracted from queue Q, arc (u_i, u_j) is checked, and updates in *speed*-entries and insertions into queue Q are performed (as done in steps 5–11 of Figure 2). Speed graph G_s contains at most n_h arcs having cost 1, where n_h is the number of horizontal wires in configuration C. Hence, if the length of the longest path from source u_s to a vertex exceeds n_h, speed graph G_s contains a positive cycle and algorithm LEGAL-SPEED terminates without generating a legal speed assignment.

The running time of algorithm LEGAL-SPEED is bounded by $O(n \cdot n_h)$ which is shown as follows. Let c_j be the number of arcs incident to vertex u_j in G_s. When the speed of U_j (or, equivalently, the length of the longest path from u_s to u_j) is increased,

Algorithm LEGAL-SPEED:

Input: A speed graph $G_s = (V_s, A_s)$.
Output: A legal speed assignment.

```
1.  Q ← A_s;
2.  for u_i ∈ V_s do speed(U_i) ← 0;
3.  while Q ≠ ∅ do
    begin
4.      (u_i, u_j) ← dequeue(Q);
        /* relaxation */
5.      speed_old(U_j) ← speed(U_j);
6.      if cost(u_i, u_j) = 0 and speed(U_i) > speed(U_j) then speed(U_j) ← speed(U_i);
7.      if cost(u_i, u_j) = 1 and speed(U_j) ≤ speed(U_i) then speed(U_j) ← speed(U_i) + 1;
8.      if speed(U_i) > n_h then no legal speed assignment exists;
9.      if speed(U_j) > speed_old(U_j) then
10.         for every (u_j, u_k) ∈ A_s do enqueue(Q, (u_j, u_k));
11.         for every (u_k, u_j) ∈ A_s do enqueue(Q, (u_k, u_j));
        /* end of relaxation */
    end of while;
```

Fig. 2. Algorithm LEGAL-SPEED.

all arcs incident to u_j are inserted into Q and are checked in a later iteration. This takes $O(c_j)$ time. The speed of a cell is increased at most n_h times, and thus the total time spent on checking the arcs incident to u_j is $O(c_j \cdot n_h)$. The overall running time of algorithm LEGAL-SPEED is thus $O(\sum_{j=1}^{n} c_j \cdot n_h) = O(n \cdot n_h)$. This time bound uses the assumption that the number of arcs in the speed graph is $O(n)$. For a speed graph containing m arcs, the bound is $O(m \cdot n_h)$.

Assume that a legal speed assignment has been determined for configuration C. The remainder of this section describes how to determine the associated event time in $O(n + n_h)$ time. Let t_d be the earliest time at which a distance event occurs and let t_w be the earliest time at which a wire event occurs. The event time is then $\min\{t_d, t_w\}$. Consider first the computation of t_d. Let (u_i, u_j) be an arc of the distance graph with $d(u_i, u_j) > 0$ and $speed(U_i) > speed(U_j)$. The *zero time* $zt_{i,j}$ of arc (u_i, u_j) is defined as the time at which the distance between U_i and U_j turns 0 when these cells move to the right with speeds $speed(U_i)$ and $speed(U_j)$, respectively; i.e.,

$$zt_{i,j} = \frac{d(u_i, u_j)}{speed(U_i) - speed(U_j)}.$$

Time t_d is determined by computing the zero time for each arc of the speed graph and then selecting the minimum among them; i.e., $t_d = \min_{(u_i,u_j)\in A_d^+}\{zt_{i,j}\}$, where

$$A_d^+ = \{(u_i, u_j) | (u_i, u_j) \in A_d, d(u_i, u_j) > 0 \text{ and } speed(U_i) > speed(U_j)\}.$$

Since G_s contains $O(n)$ arcs, this is done in $O(n)$ time.

Consider now the computation of t_w. Let (u_i, u_j) be an arc of the wire graph representing horizontal wire $H_{i,j}$ in C. We define $length_{i,j}(t)$ to be the linear function

$$length_{i,j}(t) = w(u_i, u_j) + t \cdot (speed(U_j) - speed(U_i)).$$

The value of $length_{i,j}(t)$ represents the length of wire $H_{i,j}$ at time t when cells U_i and U_j move to the right according to speeds $speed(U_i)$ and $speed(U_j)$, respectively. If $speed(U_i) > speed(U_j)$, the length of wire $H_{i,j}$ reduces and thus the slope of function $length_{i,j}(t)$ is negative. On the other hand, if $speed(U_i) < speed(U_j)$, the length of wire $H_{i,j}$ increases and the slope of $length_{i,j}(t)$ is positive. When $speed(U_i) = speed(U_j)$, the length of wire $H_{i,j}$ does not change and the slope is 0. Let $ENV(t)$ be the upper envelope of all *length* functions; i.e.,

$$ENV(t) = \max_{(u_i,u_j)\in A_w} length_{i,j}(t).$$

Then t_w is the minimum of $ENV(t)$. Figure 3 shows the *length* functions of eight horizontal wires. The upper envelope of the *length* functions is indicated by the dashed line. The minimum of $ENV(t)$ can be obtained in $O(n_h)$ time [8]. Since our line segments have a special structure, the minimum of $ENV(t)$ can be determined by a simpler method having the same time bound as follows.

The way legal speed assignments are determined implies that for every horizontal wire $H_{i,j}$ whose length is reduced, we have $speed(U_i) \geq speed(U_j) + 1$. Thus, the slope of $length_{i,j}$ is not larger than -1. Furthermore, there exists at least one longest wire, say $H_{i,j}$, for which $speed(U_i) = speed(U_j) + 1$, and thus the slope of $length_{i,j}(t)$ is -1. This implies that $ENV(t)$ contains only one line segment of negative slope and that

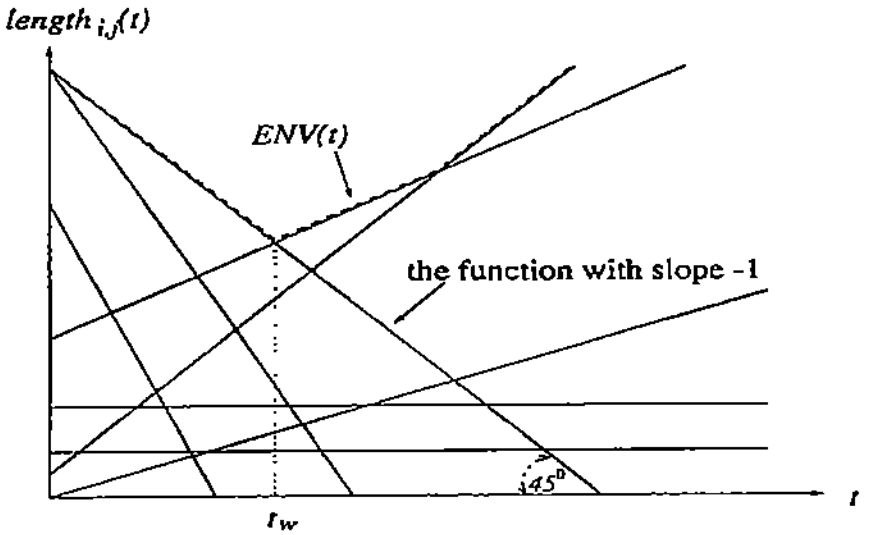

Fig. 3. The $length_{i,j}(t)$ functions.

this line segment has slope -1. Let $L(C)$ be the length of the longest horizontal wire in configuration C and let $length_L(t) = L(C) - t$. The minimum of the upper envelope occurs at the intersection of $length_L(t)$ with a function $length_{p,q}(t)$ of nonnegative slope. We thus consider all wires $H_{p,q}$ with $speed(U_p) \leq speed(U_q)$ and determine for each one the time at which $length_L(t)$ intersects $length_{p,q}(t)$. We refer to this time as the *intersection time* of wire $H_{p,q}$. The intersection time can easily be determined in $O(1)$ time and hence the minimum of the upper envelope $ENV(t)$ is determined in $O(n_h)$ time.

5. A Longest Wire Minimizing Algorithm. In this section we describe an $O(n_h^2 \cdot n^2)$-time algorithm for generating a configuration which minimizes the length of the longest wires. The algorithm performs at most $n_h \cdot n$ iterations, with each iteration generating a configuration having smaller longest wire length. Let C_i be the configuration at the beginning of the ith iteration, $i \geq 1$. Also, let G_d^i, G_w^i, and G_s^i be the distance graph, wire graph, and speed graph of C_i, respectively. For the first iteration this information is generated from the initial configuration C.

The ith iteration performs the following steps. We use algorithm LEGAL-SPEED described in the previous section to determine a legal speed assignment for configuration C_i, if one exists. If no legal speed assignment exists (or the length of the longest horizontal wire is 0), the algorithm terminates with C_i. Assume that a legal speed assignment exists and let $speed_i(\cdot)$ be the computed entries. Using these entries, we compute the event time $t_{e,i}$ for configuration C_i and then move every cell U_j distance $t_{e,i} \cdot speed_i(U_j)$ to the right. We point out that, since the quantities $t_{e,i} \cdot speed_i(U_j)$ are not guaranteed to be integers, compaction is not done on an integer grid. The movement to the right results in a new configuration C_{i+1}. We complete the ith iteration by determining graphs G_d^{i+1}, G_w^{i+1}, and G_s^{i+1}. From the bounds given in the previous section, it follows that one iteration is completed in $O(n_h \cdot n)$ time. The remainder of this section shows that after at most $n_h \cdot n$ iterations no further reduction in the longest wire length is possible.

First, we show that the speed of a cell does not decrease from one iteration to the next. Assume that neither G_s^i nor G_s^{i+1} contain a positive cycle. Let $P = \langle u_s, u_{k_1}, \ldots, u_{k_l} = u_j \rangle$ be a longest path from u_s to u_j in speed graph G_s^i. One can thus travel in C_i from cell U_{k_1} to cell U_j by traversing $speed_i(U_j)$ longest horizontal wires. In Lemma 5.1 we show that every arc on such a longest path P is also an arc in speed graph G_s^{i+1}. This implies that the *speed*-entries do not decrease from one iteration to the next. Observe that path P may not be a longest path from u_s to u_j in G_s^{i+1} (an even longer path many now exist).

LEMMA 5.1. *If neither G_s^i nor G_s^{i+1} contain a positive cycle, then $speed_i(U_j) \leq speed_{i+1}(U_j)$ for every cell U_j, $1 \leq j \leq n$.*

PROOF. Let $P = \langle u_s, u_{k_1}, \ldots, u_{k_l} = u_j \rangle$ be a longest path from u_s to u_j in G_s^i. We first show that every arc on P is also an arc in G_s^{i+1}. Assume (u_a, u_b) is an arc of cost 1 in P. Arc (u_a, u_b) in G_s^i implies that there exists a longest wire $H_{b,a}$ connecting cell U_b to cell U_a in C_i. From algorithm LEGAL-SPEED and the fact that P is a longest path it follows that $speed_i(U_b) = speed_i(U_a) + 1$. If cells U_b and U_a are moved for t time units, the length of wire $H_{b,a}$ reduces by exactly t. $H_{b,a}$ remains a longest wire and thus arc (u_a, u_b) is an arc of cost 1 in G_s^{i+1}.

Assume now that (u_a, u_b) is an arc of cost 0 in P. The existence of arc (u_a, u_b) in G_s^i implies that the distance between cells U_a and U_b in configuration C_i is 0. From algorithm LEGAL-SPEED and the fact that P is a longest path it follows that $speed_i(U_a) = speed_i(U_b)$. Hence, after moving U_a and U_b to the right, the distance between U_a and U_b remains 0. Thus, arc (u_a, u_b) is still an arc of cost 0 in G_s^{i+1}. Speed graph G_s^{i+1} is obtained from G_s^i by arc additions and arc deletions. Since no arc on a longest path is deleted and the addition of arcs cannot decrease the length of the longest path from u_s to a vertex, it follows that $speed_i(U_j) \leq speed_{i+1}(U_j)$. □

We show that the algorithm terminates in at most $n_h \cdot n$ iterations by showing that in every iteration there exists at least one cell U_j with $speed_i(U_j) < speed_{i+1}(U_j)$. Since the speed of a cell is bounded by n_h, the claimed bound of $n_h \cdot n$ follows immediately.

LEMMA 5.2. *Assume there exists a legal speed assignment for configuration C_i and C_{i+1}, respectively. Then there exists at least one cell U_q such that $speed_i(U_q) < speed_{i+1}(U_q)$.*

PROOF. The ith iteration terminates when either a distance event or a wire event occurs. Assume a distance event occurred. Then there exists one arc, say (u_p, u_q), that is not in speed graph G_s^i, but is an arc of cost 0 in G_s^{i+1}. This implies that $speed_i(U_p) > speed_i(U_q)$. By Lemma 5.1, we have $speed_i(U_p) \leq speed_{i+1}(U_p)$. Since the $(i+1)$st iteration generates a legal speed assignment and the cost of (u_p, u_q) is 0 in G_s^{i+1}, we have $speed_{i+1}(U_p) \leq speed_{i+1}(U_q)$. Combining these three inequalities gives $speed_i(U_q) < speed_{i+1}(U_q)$.

Assume now that a wire event terminated the ith iteration. In this case there exists an arc (u_p, u_q) that is not in G_s^i, but which is an arc of cost 1 in G_s^{i+1}. The arc corresponds to a horizontal wire $H_{q,p}$ which is not a longest wire in the ith iteration, but is a longest wire in the $(i+1)$st iteration. Thus, $speed_i(U_q) \leq speed_i(U_p)$. By Lemma 5.1, we have

$speed_i(U_p) \leq speed_{i+1}(U_p)$, and in the $(i+1)$st iteration we have $speed_{i+1}(U_p) < speed_{i+1}(U_q)$. The inequality $speed_i(U_q) < speed_{i+1}(U_q)$ follows. □

We summarize the above discussion in the following theorem.

THEOREM 5.1. *Given a configuration and its distance graph and wire graph, a configuration minimizing the longest wire length can be generated in $O(n_h^2 \cdot n^2)$ time, where n_h is the number of horizontal wires and n is the number of cells in the layout.*

6. Improving the Running Time. In this section we describe our $O(n_h \cdot n \log n)$-time longest wire minimizing algorithm. The algorithm performs, as the algorithm sketched in the previous section, up to $n_h \cdot n$ iterations. However, it updates, rather than recomputes, the data structures and information for the ith iteration from the ones used in the $(i-1)$st iteration.

We assume the ith iteration starts at time $t_{e,i-1}$ and ends at time $t_{e,i}$, with $t_{e,0} = 0$. The quantity $t_{e,i-1}$ is thus added whenever zero times and intersection times are computed in the ith iteration. We maintain zero times giving the next distance event and intersection times giving the next wire event in heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$, respectively. The speed graph used in an iteration is generated from the one used in the previous iteration by performing arc deletions and arc additions. Every arc change is checked as to whether it causes a change in the *speed* entries. Let $d_i(u_j, u_k)$ and $w_i(u_j, u_k)$ be the weight of arc (u_j, u_k) in distance graph G_d^i and wire graph G_w^i, respectively. We do not explicitly generate all the d_i and w_i entries. When the weight of an arc is needed, we compute it in $O(1)$ time.

The ith iteration generates two arc sets, D_i and A_i. Arc set D_i contains arcs which are in speed graph G_s^i, but not in speed graph G_s^{i+1}. A_i contains arcs which are not in G_s^i, but in G_s^{i+1}. Thus, speed graph G_s^{i+1} is obtained from G_s^i by deleting the arcs in set D_i and adding the ones in set A_i. At the beginning of the ith iteration, $i \geq 2$, we have the following information:

- a legal speed assignment for G_s^{i-1},
- heap $\mathcal{H}_Z$ storing the zero times and heap $\mathcal{H}_I$ storing the intersection times for the $speed_i$ entries, and
- event time $t_{e,i-1}$ (i.e., the time at which the $(i-1)$st iteration terminated) and arc sets A_{i-1} and D_{i-1}.

In the first iteration we build G_d^1 and G_w^1 from the initial configuration C. Speed graph G_s^1 is obtained from G_d^1 and G_w^1. We use the algorithm described in Section 4 to determine the $speed_1$ entries in $O(n_h \cdot n)$ time. Heap $\mathcal{H}_Z$ containing the zero times is created in $O(n)$ time. Let $L(C)$ be the length of the longest wire in configuration C and let $length_L(t) = L(C) - t$. For every wire $H_{j,k}$ we determine the time at which function $length_{j,k}(t)$ and function $length_L(t)$ intersect. These intersection times are the entries of heap $\mathcal{H}_I$. Heap $\mathcal{H}_I$ is created in $O(n_h)$ time. Event time $t_{e,1}$ and arc sets A_1 and D_1 can easily be determined within the $O(n_h \cdot n)$ time allowed for the first iteration. Assume $i \geq 2$. In the ith iteration we perform the following steps:

1. Generate speed graph G_s^i.

2. Determine a legal speed assignment for G_s^i.
3. Update heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ to reflect the new speed assignment.
4. Determine event time $t_{e,i}$ terminating the ith iteration and arc sets A_i and D_i.

In the following we describe each one of the four steps in more detail. Speed graph G_s^i is created from speed graph G_s^{i-1} by adding the arcs that are in A_{i-1} and deleting the ones that are in D_{i-1}.

Consider next the computation of the $speed_i$ entries. From Lemma 5.1 we know that the speed of a cell cannot decrease. An increase in the speed of a cell is caused by an arc in A_{i-1}. Every arc in A_{i-1} is considered and its effect on the *speed* entries is determined. This is done by performing relaxation on arcs of the speed graph. Observe that deleting the arcs in D_{i-1} from speed graph G_s^{i-1} does not cause a change in the *speed* entries. We start the updating of the *speed* entries by putting the arcs in A_{i-1} into a first-in-first-out queue Q. Assume we initialize $speed_i(U_j) = speed_{i-1}(U_j)$, $1 \leq j \leq n$. Of course, this initialization is not performed explicitly. For each arc (u_j, u_k) extracted from Q we perform steps 5–11 of algorithm LEGAL-SPEED (given in Figure 2). The computation of the $speed_i$ entries is completed once Q is empty.

Once the legal speed assignment for G_s^i has been determined, heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ are updated in the third step of the ith iteration. Consider first heap $\mathcal{H}_Z$. Every element of $\mathcal{H}_Z$ corresponds to an arc of the distance graph inducing a zero time. Recall that the zero time is defined for an arc in the distance graph whose weight decreases during the ith iteration. The zero time represents the time the arc weight turns 0. Let (u_j, u_k) be an arc of the distance graph. If the speed of cell U_j or that of cell U_k is increased in the second step of the ith iteration, then the zero time associated with arc (u_j, u_k) may need to be updated. Assume (u_j, u_k) is such an arc with $d_i(u_j, u_k) > 0$ and $speed_i(U_j) > speed_i(U_k)$. We delete arc (u_j, u_k) and its old zero time from $\mathcal{H}_Z$ (if it is present in the heap $\mathcal{H}_Z$), and then insert the arc with a new zero time of

$$zt_{j,k}^i = t_{e,i-1} + \frac{d_i(u_j, u_k)}{speed_i(U_j) - speed_i(U_k)}$$

into $\mathcal{H}_Z$, where $d_i(u_j, u_k)$ is the distance between cells U_j and U_k in the beginning of ith iteration. In order to compute $zt_{j,k}^i$ we need the value of $d_i(u_j, u_k)$. As already stated, we do not explicitly compute all d_i and w_i entries in the ith iteration. The value of $d_i(u_j, u_k)$ is computed in $O(1)$ time, when needed, as follows. Assume that the pth iteration, where $p < i$, was the last iteration in which either the speed of cell U_j or U_k was increased. Whenever the speed of a cell increases, we update the d_p entries of all the arcs incident to this cell. This implies that at the end of the pth iteration we did compute and record the entry $d_{p+1}(u_j, u_k)$. Recall that $d_{p+1}(u_j, u_k)$ represents the distance between cells U_j and U_k at the beginning of the $(p+1)$st iteration. During iterations $p+1, \ldots, i-1$ the speed of neither U_j nor U_k was increased. Then

$$d_i(u_j, u_k) = d_{p+1}(u_j, u_k) + (speed_p(U_k) - speed_p(U_j)) \times (t_{e,i-1} - t_{e,p}).$$

The weights of the wire graph are determined in an analogous way.

Consider now heap $\mathcal{H}_I$. It records, for every wire $H_{j,k}$ having a nondecreasing wire length, the intersection time; i.e., the time when functions $length_{j,k}(t)$ and $length_L$ intersect. Analogous to heap $\mathcal{H}_Z$, the intersection time of a wire $H_{j,k}$ can change only

when the speed of U_j or U_k increases. If this happens, we check whether an entry in $\mathcal{H}_I$ needs to be updated and, if so, update the corresponding intersection time. The length $w_i(u_j, u_k)$ of wire $H_{j,k}$ needed for computing the intersection time is obtained in $O(1)$ time as already described above. This concludes the description of the third step of the ith iteration.

The last step of the ith iteration determines the event time $t_{e,i}$ and arc sets A_i and D_i. Event time $t_{e,i}$ is computed by determining the minimum in each heap and choosing the minimum among the two. Arc set A_i contains the arcs to be added to G_s^i in order to obtain G_s^{i+1}. We claim that set A_i is formed by the arcs in heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ causing event time $t_{e,i}$. Assume (u_j, u_k) is an arc in A_i. First we consider the case in which (u_j, u_k) is in A_i because the distance between cells U_j and U_k is positive in the ith iteration, but turns 0 at the end of the ith iteration. Arc (u_j, u_k) induces a distance event terminating the ith iteration. This implies that the zero time associated with arc (u_j, u_k) is a minimum in heap $\mathcal{H}_Z$. On the other hand, if (u_j, u_k) is in A_i because wire $H_{k,j}$, which is not a longest wire in the ith iteration, turns into a longest wire at the end of the ith iteration, then (u_j, u_k) induced a wire event terminating the ith iteration. Hence, the intersection time associated with $H_{k,j}$ is a minimum in heap $\mathcal{H}_I$. Set A_i is thus formed by the arcs in heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ causing event time $t_{e,i}$. We delete these arcs from the heaps and place them into A_i.

Arc set D_i contains the arcs to be deleted from G_s^i and is obtained as follows. Assume (u_j, u_k) is an arc in D_i. Assume first that (u_j, u_k) has cost 0 in speed graph G_s^i. For the arc to be in D_i, we need to have $speed_i(U_j) < speed_i(U_k)$. Arc (u_j, u_k) may or may not have been an arc in speed graph G_s^{i-1}. If it was in G_s^{i-1}, we had $speed_{i-1}(U_j) = speed_{i-1}(U_k)$; if it was not, we had $speed_{i-1}(U_j) > speed_{i-1}(U_k)$. In either situation, in order to have $speed_i(U_j) < speed_i(U_k)$, the speed of one of the cells must have been increased in the ith iteration. Consider now the case when (u_j, u_k) has cost 1 in G_s^i. Since wire $H_{k,j}$ is no longer a longest wire at the start of the $(i+1)$st iteration, we have $speed_i(U_k) \geq speed_i(U_j) + 2$. At the end of the $(i-1)$st iteration we had $speed_{i-1}(U_k) \leq speed_{i-1}(U_j) + 1$. Hence, the speed of at least one of the cells was increased in the ith iteration. Arc set D_i can thus be found during the second step of the ith iteration. (For the sake of clarity, we place the discussion of finding D_i into the fourth step.) Whenever the speed of a cell increases, we check whether an arc in G_s^i incident to the corresponding vertex of the cell is to be deleted from the speed graph. This concludes the description of the last step of the ith iteration.

The following theorem summarizes the above discussion.

THEOREM 6.1. *Given a configuration, its distance graph, and its wire graph, a configuration minimizing the longest 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 the layout.*

PROOF. The algorithm is described above and it remains to be shown that it achieves the claimed running time. From Section 5 it follows that the algorithm performs at most $n_h \cdot n$ iterations. We show that the total work done in all iterations is bounded by $O(n_h \cdot n \log n)$. The work done in the first iteration is obviously bounded by $O(n_h \cdot n)$. The total work done in all the remaining iterations is determined as follows.

Assume that the algorithm terminates after m iterations. If an arc (u_j, u_k) is added

into D_i, the speed of at least one of U_j or U_k increased. Since the speed of a cell cannot exceed n_h, we have $\sum_{i=1}^{m} |D_i| \leq n_h \cdot n$. If (u_j, u_k) is an arc included in A_i, it was either not in G_s^l or it got deleted in some earlier iteration. Hence, $\sum_{i=1}^{m} |A_i| = O(n_h \cdot n)$. Speed graph G_s^i is generated in $O(|A_{i-1}| + |D_{i-1}|)$ time. Thus, the total work done for generating the speed graphs is bounded by $O(\sum_{i=2}^{m} |A_{i-1}| + |D_{i-1}|) = O(n_h \cdot n)$.

Consider the total work done for updating the *speed* entries. Let c_j be the maximum number of arcs incident to vertex u_j in a speed graph (during all iterations). When the speed of u_j increases, the arcs incident to u_j are checked for increases in the *speed* entries. Since the speed of a cell is bounded by n_h and the total number of arcs in the speed graph is $O(n)$, the total work done for updating the *speed* entries is bounded by $O(\sum_{j=1}^{n} c_j \cdot n_h) = O(n \cdot n_h)$.

When the speed of a cell U_k increases, entries in $\mathcal{H}_Z$ and/or $\mathcal{H}_I$ may need to be updated. Using an argument identical to the one above, heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ are updated $O(n_h \cdot n)$ times throughout all iterations. The total work for updating the heaps is thus bounded by $O(n_h \cdot n \cdot (\log n + \log n_h)) = O(n_h \cdot n \log n)$. After completing the updating of heaps $\mathcal{H}_Z$ and $\mathcal{H}_I$ in an iteration, event time $t_{e,i}$ is found in $O(1)$ time. Finally, the total work done for generating all D_i-sets is $\sum_{i=1}^{m} |D_i| = O(n_h \cdot n)$. The total work done for generating all A_i-sets is $\sum_{i=1}^{m} |A_i| \cdot \log n = O(n_h \cdot n \log n)$, where the $\log n$ comes from the min-deletions performed on the heaps. This completes our discussion of the $O(n_h \cdot n \log n)$ running time. □

Let C_f be the configuration generated by our algorithm and let $L(C_f)$ be the length of the longest wire in C_f. Configuration C_f may not have minimum width among all configurations having minimum longest wire length. There exist a number of algorithms for generating a layout of minimum width from a given feasible configuration, subject to not exceeding a given upper bound on the horizontal wire length. For example, by adding arcs corresponding to the upper bound of wire length to the distance graph and performing a compaction which positions cells as far to the left as possible, a configuration minimizing the width can be generated in an additional $O(n \log n)$ time.

7. A Tradeoff Between Wire Length and Layout Width. In this section we consider how to generate a layout minimizing a tradeoff function between the width and the longest wire length. For any configuration C, let $W(C)$ be its width and let $L(C)$ be its longest wire length. Given an initial configuration and two constants α and β, $\alpha, \beta > 0$, we are to determine a configuration C^* such that $\alpha \cdot W(C^*) + \beta \cdot L(C^*) \leq \alpha \cdot W(C') + \beta \cdot L(C')$, for any other configuration C'. We call a configuration C^* minimizing the tradeoff function an *optimal configuration*. We present an algorithm for generating an optimal configuration in $O(n_h \cdot n \log n)$ time. The overall approach is similar to the one used for minimizing the longest wire length. We generate C^* over a number of iterations, with each iteration computing speeds and moving cells according to the speeds. The resulting configuration has a smaller tradeoff function value. After at most $n_h \cdot n$ iterations, C^* is generated.

From now on, instead of minimizing $\alpha \cdot W(\cdot) + \beta \cdot L(\cdot)$, we minimize $\gamma \cdot W(\cdot) + L(\cdot)$, where $\gamma = \alpha/\beta$. We start by describing some of the differences in the definitions and data structures. First, we add to the initial configuration two fictitious cells U_0 and U_{n+1}. These two cells have a height equal to the height of the layout and a width of 0. Cell

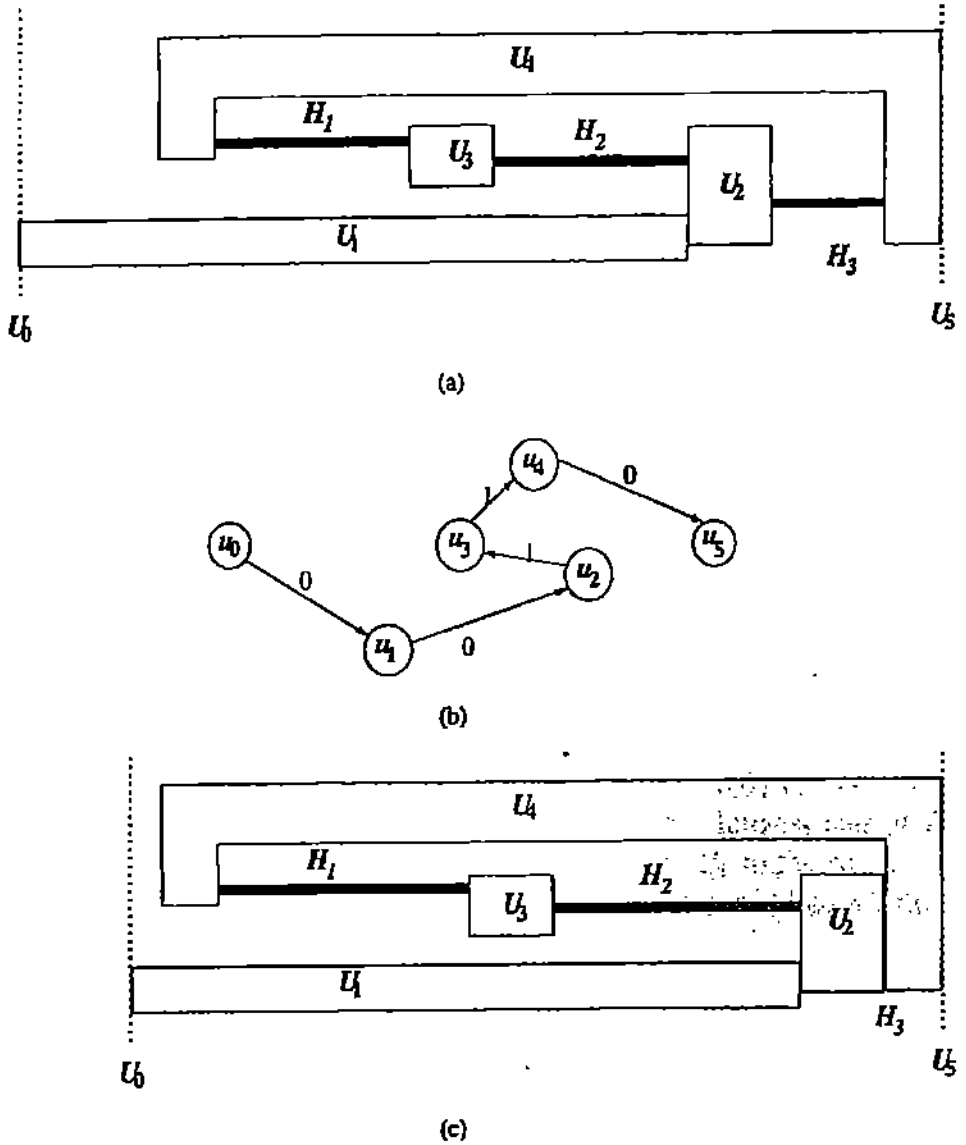

Fig. 4. (a) A min-width configuration. (b) The speed graph of the layout shown in (a). (c) The next min-width configuration generated by our algorithm.

U_0 is positioned immediately to the left of the leftmost cell, and U_{n+1} is positioned immediately to the right of the rightmost cell. The distance between U_0 and U_{n+1} is the width of the configuration. When we refer to a configuration, we always mean a configuration containing U_0 and U_{n+1}.

Distance graph and wire graph remain as defined in Section 2. The speed graph differs from the one defined in Section 2 in that we do not add a source vertex u_s. Instead, the vertex associated with cell U_0 is used as the source. Observe that vertex u_0 has arcs only to vertices corresponding to cells that are distance 0 from cell U_0. Figure 4(b) shows the speed graph of the layout shown in Figure 4(a).

We call a configuration C a *min-width configuration* if width $W(C)$ is a minimum among all configurations having a longest wire length of at most $L(C)$. Min-width configurations are crucial to our tradeoff function algorithm because they give us the

following information: given a min-width configuration C, if we want to reduce $\gamma \cdot W(C) + L(C)$ by decreasing the layout width, the length of a longest wire *must* increase. For example, Figure 4(a) shows a min-width configuration in which H_1 and H_2 are the longest wires. If we want to decrease the layout width of this configuration, the length of either H_1 or H_2 or both must increase.

Our tradeoff function algorithm generates first a configuration having minimum longest wire length and minimum width (among all configurations having minimum longest wire length). Let C_f be this configuration. C_f is, by definition, a min-width configuration. Starting with C_f, we generate C^* over a number of iterations, with each iteration generating a configuration having a smaller tradeoff function value. We achieve this reduction by increasing the longest wire length and reducing the layout width. New configurations are generated by moving cells to the right. The movement to the right is now controlled by a value-reducing speed assignment. A speed assignment is *value-reducing* for configuration C when:

(i) If $d(u_i, u_j) = 0$ in distance graph G_d, then $speed(U_i) \leq speed(U_j)$.
(ii) $speed(U_0) > speed(U_{n+1})$; i.e., the layout width decreases.
(iii) Let C' be the configuration generated from C by moving cell U_i distance $t \cdot speed(U_i)$ to the right, $0 \leq i \leq n+1$. There exists a t such that $\gamma \cdot W(C) + L(C) > \gamma \cdot W(C') + L(C')$.

If C is a min-width configuration, the existence of a value-reducing speed assignment for C implies that another feasible configuration with a smaller tradeoff function value can be generated by moving the cells of C to the right according to the associated value-reducing speeds.

We determine a value-reducing speed assignment by performing a single-source shortest-path computation in the speed graph. Let $S(u_i)$ be the length of the shortest path from u_0 to u_i in the speed graph. Intuitively, the existence of a shortest path from u_0 to u_i implies that in order to travel from U_0 to U_i, one has to go through *at least* $S(u_i)$ longest wires. When decreasing the layout width, the increase in the length of the longest wire should be as small as possible. We thus distribute the increase evenly among the $S(u_i)$ longest wires on the shortest path from u_0 to u_i. For example, Figure 4(a) shows a min-width configuration in which H_1 and H_2 are the longest wires. If we want to decrease the layout width of this configuration, the length of either H_1 or H_2 or both must increase. The increase is minimized by increasing H_1 and H_2 evenly. Figure 4(c) shows the min-width configuration generated by evenly distributing the increase between H_1 and H_2.

Before describing our algorithm we prove the following relevant lemmas concerning min-width configuration.

LEMMA 7.1. *C is a min-width configuration if and only if there exists a path from u_0 to u_{n+1} in the speed graph of C.*

PROOF. Let C be a min-width configuration. Assume there exists no path from u_0 to u_{n+1}. We can then partition the vertex set of G_s into two subsets, V_0 and V_{n+1}, such that V_0 contains u_0, V_{n+1} contains u_{n+1}, and there exists no arc from a vertex in V_0 to a vertex in V_{n+1} in speed graph G_s. The partition of the vertices into V_0 and V_{n+1} implies that the

layout width can be reduced by moving the cells corresponding to the vertices in V_0 to the right. Doing so does not increase the length of the longest wires. This contradicts our assumption that C is a min-width configuration. Hence, if C is a min-width configuration, there exists a path from u_0 to u_{n+1} in speed graph G_s. On the other hand, if C is not a min-width configuration, we can partition the cells in C into two groups S_0 and S_{n+1} such that S_0 contains U_0, S_{n+1} contains U_{n+1}, and moving the cells in S_0 to the right reduces the layout width without increasing the length of the longest wires. This implies that there is no arc from a vertex in the set of vertices induced by the cells in S_0 to a vertex in the set of vertices induced by the cells in S_{n+1}. Hence, if C is not a min-width configuration, there exists no path from u_0 to u_{n+1} in G_s. □

In our longest wire minimizing algorithm we used the existence of positive cycles in the speed graph as the terminating condition of our algorithm. The termination condition for the tradeoff problem is based on a relationship between the shortest path from cell U_0 to cell U_{n+1} and γ, the quantity from the tradeoff function. The following two lemmas give this condition and state how a value-reducing speed assignment is obtained.

LEMMA 7.2. *Let C be a min-width configuration. If $\gamma \cdot S(u_{n+1}) \leq 1$, no value-reducing speed assignment exists for C.*

PROOF. Assume a value-reducing speed assignment exists when $\gamma \cdot S(u_{n+1}) \leq 1$. Let P be the shortest path from u_0 to u_{n+1} in the speed graph G_s of configuration C. By Lemma 7.1 we know that such a path exists in a min-width configuration. Entry $S(u_{n+1})$ represents the minimum number of longest wires that need to be used when traveling from cell U_0 to cell U_{n+1}. Let t be a value satisfying the third condition of a value-reducing speed assignment and let C^t be the associated configuration. In C^t the layout width decreases and the longest wire length increases. The sum of the increases in length of the wires corresponding to the arcs on P is at least $W(C) - W(C^t)$. This implies that there exists at least one wire corresponding to an arc in P whose the length is increased by at least $W(C) - W(C^t))/S(u_{n+1})$. Such a minimum increase would be obtained when the decrease in the width (and thus the sum of the increases in length of the wires corresponding to the arcs on P) is evenly distributed between the $S(u_{n+1})$ longest wires on path P. Hence,

$$\gamma \cdot W(C) + L(C) - (\gamma \cdot W(C^t) + L(C^t)) > 0,$$

$$\gamma \cdot (W(C) - W(C^t)) > L(C^t) - L(C) \geq \frac{W(C) - W(C^t)}{S(u_{n+1})},$$

$$\gamma > \frac{1}{S(u_{n+1})},$$

which contradicts the assumption $\gamma \cdot S(u_{n+1}) \leq 1$. □

Next we show that, given a min-width configuration C with $\gamma \cdot S(u_{n+1}) > 1$, a value-reducing speed assignment can be determined by performing a single-source shortest-path computation in the speed graph.

LEMMA 7.3. *Let C be a min-width configuration. If $\gamma \cdot S(u_{n+1}) > 1$, a value-reducing speed assignment for C is obtained by setting $speed(U_j) = n_h - S(u_j)$ if there exists a path from u_0 to u_j in the speed graph of C, and by setting $speed(U_j) = 0$ otherwise.*

PROOF. Assume that (u_j, u_k) is an arc in speed graph G_s with $cost(u_j, u_k) = 0$. Since $S(u_j)$ and $S(u_k)$ represent the length of the shortest path from u_0 to u_j and u_k, respectively, we have $S(u_j) \geq S(u_k)$. Thus, $speed(U_j) = n_h - S(u_j) \leq n_h - S(u_k) = speed(U_k)$, and the first condition of a value-reducing speed assignment is satisfied. Since $S(u_{n+1}) > 0$, we have $speed(U_0) = n_h - 0 > n_h - S(u_{n+1}) = speed(U_{n+1})$ and satisfy the second condition of a value-reducing speed assignment.

Assume the cells in configuration C are moved to the right according to computed speeds. Let t be the first moment in time in which a nonzero distance between cells becomes zero or a nonlongest wire turns into a longest wire. Let C' be the configuration generated from C by moving every cell U_i distance $speed(U_i) \cdot t$ to the right. The layout width decreases by $(speed(U_0) - speed(U_{n+1})) \cdot t$. The longest wire length increases by

$$L(C') - L(C) = \max_{H_{i,j} \text{ is a longest wire in } C} \{(speed(U_j) - speed(U_i)) \cdot t\} \leq t.$$

It is easy to see that for every longest wire $H_{i,j}$ we have $speed(U_j) - speed(U_i) \leq 1$. Moreover, if $H_{i,j}$ is a longest wire corresponding to an arc on the shortest path from u_0 to u_{n+1} in G_s, then $speed(U_j) - speed(U_i) = 1$. We thus have

$$\begin{aligned} \gamma \cdot W(C) + L(C) - (\gamma \cdot W(C') + L(C')) &= \gamma \cdot (speed(U_0) - speed(U_{n+1})) \cdot t \\ &\quad + L(C) - L(C') \\ &\geq \gamma \cdot (speed(U_0) - speed(U_{n+1})) \cdot t - t \\ &\geq (\gamma \cdot S(u_{n+1}) - 1) \cdot t \\ &> 0. \end{aligned}$$

Hence, the third condition of the value-reducing speed assignment is satisfied. □

We are now ready to describe the algorithm. Its first step generates, from an initial configuration C, a configuration of minimum longest wire length using the algorithm described in Section 6. We then minimize the width of this configuration. Let C_f be the resulting min-width configuration. Next we add cells U_0 and U_{n+1} to C_f. A final configuration is generated from C_f over a number of iterations, with each iteration generating a configuration having a smaller layout width, a larger longest wire length, and a smaller tradeoff function value. Let C_i be the configuration at the beginning of the ith iteration, $i \geq 1$. Initially, $C_1 = C_f$. In the ith iteration, we perform a single-source shortest-path computation on G_s^i. If $\gamma \cdot S(u_{n+1}) \leq 1$ or $S(u_{n+1}) = 0$, C_i is the final configuration. Otherwise, we compute value-reducing speeds as described in Lemma 7.3. Using these speeds, we compute event time $t_{e,i}$ and move cell U_j distance $t_{e,i} \cdot speed_i(U_j)$ to the right, $0 \leq j \leq n + 1$. The movement results in a new configuration C_{i+1}.

Again, the event time is the earliest time at which either a distance or wire event occurs. The definition of a distance event is as for the wire length minimizing algorithm; i.e., it is the earliest time at which a nonzero distance between cells becomes zero.

The longest wires get longer during the movement to the right. The wire event is the point in time at which a nonlongest wire turns into a longest wire. Let, as in Section 4, $length_{i,j}(t) = w(u_i, u_j) + t \cdot (speed(U_j) - speed(U_i))$ for horizontal wire $H_{i,j}$. Different from Section 4, we let $length_L(t) = L(C) + t$. The intersection time of wire $H_{i,j}$ is the time at which functions $length_{i,j}$ and $length_L(t)$ intersect. The minimum intersection time gives the next wire event.

In the following we prove that our tradeoff algorithm does generate an optimal configuration. It is easy to see that an optimal configuration is also a min-width configuration. (In an optimal configuration it is not possible to decrease the longest wire length without increasing the width.) Let C^* be an optimal configuration. If there exist different combinations of width and longest wire length resulting in the same minimum tradeoff function value, we choose C^* so that the longest wire length is a minimum among all these configurations. Assume our tradeoff algorithm terminates after k iterations and generates configuration C_{k+1}. We first prove that every C_i generated by our tradeoff algorithm is a min-width configuration, and then prove that $L(C_{k+1}) = L(C^*)$. A straightforward consequence of these two lemmas is that $W(C_{k+1}) = W(C^*)$ and thus C_{k+1} is an optimal configuration.

LEMMA 7.4. *Every configuration C_i generated by our tradeoff algorithm is a min-width configuration, $1 \le i \le k+1$.*

PROOF. We prove this lemma by induction on i. Configuration C_1 is generated by the algorithm described in Section 6 and is a min-width configuration. Assume that C_i is a min-width configuration. Let P be a shortest path from u_0 to u_{n+1} in speed graph G_s^i. According to the speed-assignment method used in our tradeoff algorithm, if longest wire $H_{a,b}$ is associated with an arc (u_b, u_a) of cost 1 on path P, then $H_{a,b}$ is still a longest wire in C_{i+1}. Thus (u_b, u_a) is also an arc of cost 1 in speed graph G_s^{i+1}. On the other hand, if the distance between two cells U_a and U_b is 0 in C_i and (u_a, u_b) is an arc of cost 0 on P, then $speed_i(U_a) - speed_i(U_b) = 0$ and thus (u_a, u_b) is also in G_s^{i+1}. Hence, P is still in G_s^{i+1}, although it is possible that P is no longer the shortest path from u_0 to u_{n+1} in G_s^{i+1}. Thus, by Lemma 7.1, C_{i+1} is a min-width configuration. □

Recall that, in addition to every configuration C_i being a min-width configuration, we know the following about C_i and C_{i+1}: the width of C_i is larger than that of C_{i+1} and the length of the longest wires in C_i is smaller than that in configuration C_{i+1}.

LEMMA 7.5. *Assume the tradeoff algorithm terminates after k iterations and generates min-width configuration C_{k+1}. Then $L(C_{k+1}) = L(C^*)$.*

PROOF. First we show that if $L(C_{k+1}) < L(C^*)$, then there exists a value-reducing speed assignment for C_{k+1}. This contradicts our assumption that C_{k+1} is the final configuration generated by the algorithm. Let $x_{C_{k+1}}(U_j)$ and $x_{C^*}(U_j)$ be the x-positions of the leftmost vertical edge of cell U_j in configurations C_{k+1} and C^*, respectively. Without loss of generality, we can assume, similar to Lemma 3.1, that $x_{C_{k+1}}(U_j) \le x_{C^*}(U_j)$ for every cell U_j. We now show that setting $speed(U_j) = x_{C^*}(U_j) - x_{C_{k+1}}(U_j)$ results in a value-

reducing speed assignment for configuration C_{k+1}. Showing that the first condition of a value-reducing speed assignment is satisfied is done as in the proof of Lemma 3.1 and is omitted. Since both C_{k+1} and C^* are min-width configurations and $L(C_{k+1}) < L(C^*)$, we have

$$W(C_{k+1}) = x_{C_{k+1}}(U_{n+1}) - x_{C_{k+1}}(U_0) > x_{C^*}(U_{n+1}) - x_{C^*}(U_0) = W(C^*)$$

and $speed(U_0) > speed(U_{n+1})$. The second condition of a value-reducing speed assignment is thus satisfied. Finally, if every cell in C_{k+1} moves distance $speed(U_j) = x_{C^*}(U_j) - x_{C_k}(U_j)$ to the right, configuration C^* is generated. Hence, setting $t = 1$ results in a configuration of a smaller tradeoff function value and the third condition of a value-reducing speed assignment is satisfied. Hence, C_{k+1} has a value-reducing speed assignment, and thus $L(C_{k+1}) < L(C^*)$ is not possible.

Assume that $L(C_{k+1}) > L(C^*)$. We prove that our tradeoff algorithm would have generated C^* before generating C_{k+1}, contradicting the optimality of C^*. Let C_i and C_{i+1}, $1 \leq i \leq k$, be two configurations generated by our tradeoff algorithm such that $L(C_i) < L(C^*) < L(C_{i+1})$. Note that $L(C_i) = L(C^*)$ is not possible. If this would be the case, C_i and C^* would have the same tradeoff function value (since both are min-width configurations). The tradeoff function value associated with C_{i+1} is smaller than that of C_i, contradicting the optimality of C^*. Let C' be the configuration generated from C_i by moving every cell U_j distance $speed_i(U_j) \cdot t$ to the right. Assume that the ith iteration starts at time t_i and terminates at time t_{i+1}. The method used for computing the speeds implies that, for any time t with $t_i \leq t \leq t_{i+1}$, C' is a min-width configuration with $L(C') = L(C) + t$. This implies that there exists a t such that $L(C') = L(C^*)$ and thus $W(C') = W(C^*)$. Since $speed_i$ is a value-reducing speed assignment specified in the ith iteration, C_{i+1} can be generated from C' by moving cells to the right according to the speeds specified by $speed_i$. Hence, we have $\gamma \cdot W(C_{i+1}) + L(C_{i+1}) < \gamma \cdot W(C') + L(C')$. In summary, we have

$$\gamma \cdot W(C_{i+1}) + L(C_{i+1}) < \gamma \cdot W(C') + L(C') = \gamma \cdot W(C^*) + L(C^*),$$

which contradicts the optimality of C^*. This concludes the argument showing $L(C_k) = L(C^*)$. □

Using arguments similar to the ones used in the proofs of Lemmas 5.1 and 5.2, it can be shown that the speed of a cell does not decrease and that the speed of at least one cell increases from one iteration to the next. The speed of each cell is again bounded by n_h, and thus the tradeoff algorithm terminates in $O(n \cdot n_h)$ iterations. By determining the speed graph, the shortest path information, and the next event time for each iteration without using data from previous iterations, we can generate C_{i+1} from C_i in $O(n \log n)$ time. This results in an $O(n_h \cdot n^2 \log n)$-time algorithm. By using an approach similar to the one described in Section 6, we can perform the iterations by generating the graphs and shortest-path entries through updates, rather than recomputations. Using this approach allows us to state the following theorem.

THEOREM 7.1. *Given a configuration, its distance, wire, and speed graph, a configuration minimizing the tradeoff function $\gamma \cdot W(\cdot) + L(\cdot)$ can be generated in $O(n_h \cdot n \log n)$ time.*

As already stated, if different optimal configurations exist, our algorithm generates an optimal configuration having smallest longest wire length. The described tradeoff algorithm terminates when $\gamma \cdot S(u_{n+1}) \leq 1$. If we change the termination rule to $\gamma \cdot S(u_{n+1}) < 1$, an optimal configuration having minimum width among all optimal configurations is generated. The $O(n_h \cdot n \log n)$ time bound does not change.

8. Extensions. In this section we briefly discuss a number of extensions and describe how our algorithms can be modified to handle them.

First, given a configuration C and a bound W, our algorithm minimizing the longest wire length can be used to determine a configuration C^* having width at most W and whose longest wire length is a minimum among all such configurations. Let C_{left} be the configuration in which every cell is positioned as far to the left as possible. C_{left} can easily be generated in $O(n \log n)$ time. If $W(C_{\text{left}}) > W$, then there exists no configuration having width less than or equal to W. Assume that $W(C_{\text{left}}) \leq W$. We add to C_{left} a $\sqcup$-shaped cell that "encloses" all other cells. The two vertical segments of this $\sqcup$-shaped cell have a height equal to the height of the layout and a width of 0. The horizontal segment of this $\sqcup$-shaped cell has a height of 0 and a width equal to W. We then apply our longest wire minimizing algorithm to the configuration containing the $\sqcup$-shaped cell. The generated configuration has width W and minimizes the length of the longest wires. Removing the $\sqcup$-shaped cell from this configuration gives C^*.

Another natural extension is to associate with each horizontal wire an upper and lower bound and to require that the length of every wire in the final configuration lies between these two bounds. Both of our algorithms can be used to solve this version of the compaction problem. We describe the necessary changes only for the longest wire minimizing algorithm. The definition of a legal speed assignment and the construction of the speed graph are changed to capture the upper and lower bounds as follows. If the length of wire $H_{i,j}$ equals its upper bound, then we add the condition $speed(U_i) \geq speed(U_j)$. This guarantees that the length of the wire does not continue to increase. If the length of the wire equals its lower bound, we add the condition $speed(U_i) \leq speed(U_j)$ in order to guarantee that the wire length does not continue to decrease. A wire $H_{i,j}$ whose length equals its upper bound induces arc (u_j, u_i) with weight 0 in the speed graph. A wire $H_{i,j}$ whose length equals its lower bound induces arc (u_i, u_j) with weight 0 in the speed graph. Conditions on the upper and lower bounds can induce events which stop movement to the right. Lower bounds on the wire length can cause a distance event. Upper bounds on the wire length can cause a wire event. The algorithm then operates as described. Assume we have n_u upper bounds and n_l lower bounds. If these bounds are only on horizontal wires, we have $n_u + n_l = O(n_h)$ and the overall time bound remains $O(n_h \cdot n \log n)$. If upper and lower bound constraints exist also between cells and the constraint graph contains m arcs (with every upper and lower bound including one arc), the time bound is $O((n_u + n_l + n_h)m \log m)$.

During the longest wire length and the tradeoff algorithms, the length of the horizontal wires changes, but the algorithms do not insert 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. More precisely, 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 can be introduced. Of course, doing this does not guarantee any bound on the total number of jogs introduced. The overall running time is now $O(n_v(n_v + n_r)n \log n)$ with $n = O(n_v(n_v + n_r) + n_h)$.

9. Conclusions. In this paper we presented a new approach for minimizing the longest wire length during one-dimensional compaction. This approach is based on assigning speeds to cells. Moving the cells to the right according to the computed speeds reduces the longest wire length. Longest path entries were used to compute the speeds, and characterizations were given as to when the movement has to be stopped. We used a similar approach to determine a combination of layout width and longest wire length that minimizes a given tradeoff function between these two quantities. An algorithm that is able to trade longest wire length for layout width is novel in layout compaction.

Acknowledgments. We would like to thank the referees for their valuable and helpful comments.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press; Cambridge, MA, 1990.

[2] S. Gao, M. Kaufmann, and F. M. Maley. Advances in homotopic layout compaction. *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 273–282, 1989.

[3] S. E. Hambrusch and H. Y. Tu. Minimizing total wire length during 1-dimensional compaction. *INTEGRATION, the VLSI Journal*, 14(2):113–144, 1992.

[4] J. F. Lee and C. K. Wong. A performance-aimed cell compactor with automatic jogs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(12):1495–1507, December 1992.

[5] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley, New York, 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] D. Marple. A hierarchy preserving hierarchical compactor. *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 375–381, 1990.

[8] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 12(4):759–775, November 1983.

[9] D. A. Mlynski and C. H. Sung. Layout compaction. In T. Ohtsuki, editor, *Layout Design and Verification*, pages 199–235. Elsevier, Amsterdam, 1986.

[10] A. R. Newton. Symbolic layout and procedural design. In G. DeMicheli, A. Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits*, pages 65–112. Martinus Nijhoff, Boston, MA, 1987.

[11] A. Onozawa. Layout compaction with attractive and repulsive constraints. *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 369–374, 1990.

[12] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[13] W. L. Schiele. Improved compaction by minimized length of wires. *Proceedings of the 20th ACM/IEEE Design Automation Conference*, pages 121–121, 1983.

[14] 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. JAI Press, Greenwich, CT, 1984.

[15] B. X. Weis and D. A. Mlynski. A graph-theoretic approach to the relative placement problem. *IEEE Transactions on Circuits and Systems*, 35(3):286–293, 1988.