1990

1-D Compaction

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

Hung-Yi Tu

Report Number:
90-999

TR #999

1-D Compaction in the Presence of Forbidden Regions

Susanne Hambrusch *
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Hung-Yi Tu†
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

October 18, 1990

Abstract

In this paper we consider the one-dimensional compaction problem when the layout area contains forbidden regions and the layout components are allowed to move across these regions. Given n layout components and k forbidden regions, each of rectangular shape, we show how to solve this compaction problem in $O(\Delta)$ time with $O((n+k)\log k + \delta \log \delta)$ preprocessing, where Δ and δ are measures for the interaction between layout components and forbidden regions, $\Delta < n^2k$, $\delta \leq nk$. We also consider special cases of the forbidden regions. For example, when every forbidden region is of length h, where h is the height of the layout, the compaction problem can be solved in $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho)$ time, with $O(\rho + n \log n + k)$ preprocessing, where ρ is the number of edges in transitive closure of visibility graph induced by the layout components, $\rho < n^2$.

*Research supported in part by ONR under contracts N00014-84-K-0502 and N00014-86-K-0689, and by NSF under Grant MIP-87-15652.

†Research supported in part by NSF under Grant MIP-87-15652 and ONR under contract N00014-84-K-0502.

1 Introduction

A one-dimensional (1-D) compacter takes as an input a VLSI layout and generates a layout of smaller area by sliding the layout components in one direction [1, 2, 4]. W.l.o.g., let it be the horizontal direction. Early compaction algorithms have not been used as widely as expected. One of the reasons given attributes it to the limitations inherent to these compaction systems; e.g., they could not handle additional constraints on where to place or not to place certain layout components [5]. In this paper we take a step towards incorporating additional constraints into the compaction process. We consider compaction when the layout area contains forbidden regions. The forbidden regions can represent, for example, pre-positioned layout components or holes in the layout area. The positions of the forbidden regions cannot be altered during the compaction process, but layout components are allowed to "slide over" the forbidden regions. We assume that both forbidden regions and layout components are of rectangular shape.

Given are n rectangles, R_1,R_2,...,R_n, and k forbidden regions, B_1, B_2, ..., B_k, with the edges of the rectangles and forbidden regions parallel to the coordinate axes. A *configuration* of the layout assigns to every lower left corner of a rectangle a position of the layout area. A configuration is called *feasible* if it keeps the relative order of the rectangles in the horizontal direction and no two rectangles and no rectangle and forbidden region overlap. A feasible configuration of minimum area is called a *minimum configuration*.

Let h be the height of the layout. Since compaction is done in the horizontal direction, h is determined by the forbidden regions and the rectangles, and is not altered during compaction. We first consider the compaction

problem when every forbidden region has height h. We refer to this problem as the *k-partition problem.* We develop characterizations of a minimum configuration that allow us to determine a minimum configuration of the k-partition problem in $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho + n \log n + k)$ time, where ρ is the number of edges in transitive closure of visibility graph induced by the n rectangles, $\rho < n^2$. We then generalize the approach used for the k-partition problem to handle the general problem, the *forbidden region problem.* Let δ be the number of pairs (i, j) such that rectangle R_i could overlap with forbidden region B_j (if we slid R_i horizontally), $\delta \leq nk$. We again characterize a set of feasible configurations and show that a minimum configuration is among them. The number of feasible configurations considered is at most $n+1$ for the k-partition problem and at most δ for the forbidden region problem. In both algorithms we generate the configurations in an order that allows us to update changes in the positions of the layout (and thus the width of the layout associated with each configuration) efficiently. The running time for the forbidden region problem is $O(\Delta)$ with $O((n+k) \log k + \delta \log \delta)$ preprocessing, where Δ is another measure for the interaction between the layout components and the forbidden regions, $\Delta < n^2 k$.

The k forbidden regions in the k-partition problem can be viewed as a position in the layout where a vertical cut can be made. In certain environments one may need to make k cuts, but does not have the positions of the cuts pre-determined. Rather, the vertical cuts should be made so that the maximum distance between two consecutive cuts is a minimum. For example, in a multi-layer environment minimizing the maximum space between two cuts corresponds to minimizing the volume of the 3-dimensional

layout. This is the objective in the *minmax k-partition problem*. For the case when only three cuts are made (i.e., the rectangles are compacted onto 2 layers) we present an $O(n)$ time algorithm. For arbitrary k we present an $O(\rho + n \log n)$ time algorithm, where ρ equals the number of edges in the transitive closure of the visibility graph. In this algorithm we also identify a set of feasible configurations. However, it is now possible to employ a binary search technique for identifying a minimum configuration.

This paper is structured as follows. In Section 2 we describe our algorithm for the k-partition problem. Section 3 addresses the forbidden region problem. In Section 4 we consider the minmax k-partition problem.

2 k-partition problem

In this section we present our algorithm for the k-partition problem. Recall that in this problem the height of every forbidden region is equal to the height of the layout area. We first present an $O(n^2 \log k + k)$ time algorithm. Using properties of minimum configurations and relationships between configurations, we then reduce the time to $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho + k + n \log n)$, where ρ is the number of edges in the transitive closure of the visibility graph of the rectangles.

Assume that the width of every forbidden region is zero. Straightforward modifications to the algorithm can handle forbidden regions with arbitrary widths. For convenience we introduce two fictitious rectangles R_0 and R_{n+1} of height h and width zero, which are initially positioned to the left and to the right of the other rectangles and the forbidden regions, respectively. Figure 1(a) shows an initial configuration of a 7-partition problem for $n = 8$.

Let $x(B_i)$ be the position (i.e., the x-coordinate of the left side) of forbidden region B_i. Let S_i denote the area of width d_i, called the *slot*, between B_i and B_{i+1} for $1 \leq i \leq k-1$. For $i = 0$, S_i is the area available to the left of B_1, and for $i = k$, S_i is the area to the right of B_k, respectively. In any configuration C, let $(x_C(R_i), y_C(R_i))$ be the position of the lower left corner of rectangle R_i. Since the width of any minimum configuration is at least $x(B_k) - x(B_1)$, we only consider feasible configurations in which R_0 is to the left of position $x(B_1)$ and R_{n+1} is to the right of position $x(B_k)$. The width of configuration C is then the distance between R_0 and R_{n+1}; i.e., $x_C(R_{n+1}) - x_C(R_0)$. A feasible configuration C is called *left-compressed* if for any other feasible configuration C' in which $x_{C'}(R_0) = x_C(R_0)$ we have $x_C(R_i) \leq x_{C'}(R_i)$ for $1 \leq i \leq n$. Intuitively, in a left-compressed configuration all rectangles are positioned as far to the left as possible. Figure 1(b) shows the left-compressed version of the configuration shown in Figure 1(a). It is easy to see that performing a left-compression on a configuration cannot increase its width.

Two rectangles R_i and R_j are *visible from* each other if one can draw a horizontal line segment connecting R_i and R_j without intersecting any other rectangles. The *visibility graph* induced by the rectangles is the directed graph $G = (V, E)$ in which each vertex corresponds to a rectangle and the edges reflect the visibility between the rectangles. More precisely, an edge is directed from i to j if rectangle R_j is to the right of R_i and R_i and R_j are visible from each other. Throughout this paper the vertices of the visibility graph have a weight associated with them. The weight of the vertex corresponding to R_i is w_i, the width of R_i. Figure 2 shows the

visibility graph for the rectangles of Figure 1(a). The length of a path from R_i to R_j is the sum of the weights of the vertices on this path. The length of the longest path from R_0 to R_i in a visibility graph G is denoted by l_i.

Depending on the layout system used to generate the initial layout, the visibility information between rectangles may or may not be available [6]. If it is not available, it can easily be determined in $O(n \log n)$ time. Throughout the paper we assume that the visibility graph G is available. Furthermore, we assume that the forbidden regions $B_1, B_2, \ldots, B_k$ have been sorted in increasing order according to the x-coordinate, and that the rectangles $R_0, R_1, \ldots, R_{n+1}$ are arranged in a topological order induced by G. All these pre-processing steps can be accomplished in $O(n \log n + k \log k)$ time and their running time will no longer be explicitly stated. The following property of a left-compressed minimum configuration relates some longest path to the position of R_0 as follows.

Property 2.1 *Let C be a left-compressed minimum configuration. Then, there exists a rectangle R_i, $0 \leq i \leq n$, such that*

$$x_C(R_0) + l_i = x(B_1).$$

There are at most $n+1$ left-compressed feasible configurations satisfying Property 2.1. An immediate algorithm for the k-partition problem is to generate these $n+1$ configurations and to determine the minimum one among them as follows. Let C_i be the left-compressed configuration in which the width of slot S_0 is l_i, $0 \leq i \leq n$. The width of configuration C_i is determined by first setting $x_{C_i}(R_0) = x(B_1) - l_i$. Then, process the rectangles in order $R_1, R_2, \ldots, R_{n+1}$. Recall that the rectangles are sorted in a topological order

induced by G. When processing rectangle R_j we determine the rectangle R_{max} which is a predecessor of R_j in G and whose value $x_{C_i}(R_{max}) + w_{max}$ is a maximum among all such values. Assume that R_{max} is in slot S_l. In order to determine the position of R_j, the following query is answered. Let d'_l be the width still available in slot S_l; i.e., $d'_l = x(B_{l+1}) - (x_{C_i}(R_{max}) + w_{max})$. Given slots $S_l, S_{l+1}, \ldots, S_{k-1}, S_k$ with widths $d'_l, d_{l+1}, \ldots, d_{k-1}, +\infty$, respectively, and rectangle R_j with width w_j, determine the smallest index a such that $d_a \geq w_j$. If $a = l$, we set $x(R_j)$ to $x_{C_i}(R_{max}) + w_{max}$; otherwise, we set $x(R_j)$ to $x(B_a)$.

By using a balanced tree, which we call the *space tree*, this query can easily be answered in $O(\log k)$ time. Thus, configuration C_i can be generated in $O(n \log k)$ time and the minimum configuration can be determined in $O(n^2 \log k)$ time. The generation of the space tree costs $O(k)$ preprocessing time. We briefly describe the operations performed on the space tree. The space tree initially stores in the leaves the entries $(+\infty, 0), (d_1, 1) \ldots, (d_{k-1}, k-1)$, and $(+\infty, k)$. Every interior node v records a pair of entries $(value(v), index(v))$. In this pair $value(v)$ records the largest leaf entry found in the subtree rooted at v and $index(v)$ records the index of the slot with width $value(v)$. Assume we access v's left child and right child through $lchild(v)$ and $rchild(v)$, respectively. To answer a query we first check whether $w_j \leq d'_l$. If yes, we put rectangle R_j into slot S_l. Otherwise, from the leaf containing d_l, we search upwards for the first interior node v with $value(rchild(v)) \geq w_j$ and $index(rchild(v)) > l$. When this node v has been found, we search downwards for the desired index a in the subtree rooted at $rchild(v)$. Figure 3(a) shows the space tree for the 7-partition problem of Figure 1(a). The dashed

lines indicate the searching for determining the position of rectangle R_7. For R_7 we have $R_{max} = R_5$ and $l = 2$. Since the remaining space in slot S_2 is not wide enough for R_7 ($w_7 = 4$ and $d'_2 = 3$), R_7 ends up in slot S_4.

We now show how to improve the running time to $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho + k + n \log n)$. Since $\rho = O(n^2)$, we not only make the running time input-sensitive, but also improve its worst-case performance. Assume now that the rectangles are topologically ordered so that $l_0 \leq l_1 \leq \ldots \leq l_{n+1}$ and that the configurations are generated in the order $C_n, C_{n-1}, \ldots, C_1, C_0$. In configuration C_n every rectangle is assigned a position in slot S_0 and is to the left of forbidden region B_1. Obviously, if we generate the configurations in this order, the positions assigned to rectangles cannot decrease; i.e., $x_{C_i}(R_j) \leq x_{C_{i-1}}(R_j)$ for all j. In order to efficiently determine the correct slots for rectangles, we change the space tree from a basic balanced binary tree to a *level-linked finger tree* [3], which we call finger space tree. A finger tree allows fast searching in the vicinity of a finger. Figure 3(b) shows the finger space tree for the 7-partition problem of Figure 1(a).

While generating the configurations we maintain for every rectangle R_p a variable $x'(R_p)$ which contains the correct position of R_p in configuration C_i when R_p is to the right of B_1. When R_p is to the left of B_1, we have $x'(R_p) = -\infty$ and R_p's position is determined by the longest path; i.e., $x_{C_i}(R_p) = x(B_1) - l_i + l_p - w_p$. Assume now that the width of configuration C_i has been determined. In order to generate C_{i-1}, the rectangles in slot S_0 are pushed $l_i - l_{i-1}$ positions to the right. This pushing leaves all rectangles, except R_i, to the left of B_1 in slot S_0. Rectangle R_i is pushed to across B_1. During this process a rectangle R_p finds itself in one of three possible

situations.

Case 1. Rectangle R_p is reachable from rectangle R_i. Since R_i moves from being immediately to the left of forbidden region B_1 in C_i to being to the right of B_1 in C_{i-1}, rectangle R_p may need a new position. We use the finger space tree to determine its new position and we record it in entry $x'(R_p)$.

Case 2. Rectangle R_p is to the left of forbidden region B_1 in C_i and $p \neq i$. Then, in configuration C_{i-1}, R_p is shifted $l_i - l_{i-1}$ positions to the right (i.e., $x_{C_{i-1}}(R_p) = x_{C_i}(R_p) + (l_i - l_{i-1})$). Since configurations are generated by decreasing longest paths from R_0, no rectangle for which Case 2 applies can overlap with forbidden region B_1.

Case 3. Rectangle R_p is to the right of forbidden region B_1 in C_i and it is not reachable from R_i in visibility graph G. Then, the position of R_p in C_{i-1} is as in configuration C_i.

For rectangles for which Case 1 applies (including rectangle R_i) we use the finger space tree as follows. Assume we are determining a new position for rectangle R_p in configuration C_{i-1}. Assume R_p is assigned a position in slot S_m in configuration C_i. Let R_{max} be defined for rectangle R_p in configuration C_{i-1} as before. Assume R_{max} is in slot S_l. If $l < m$, then R_p remains in slot S_m. Hence, assume that $l \geq m$ and let d'_l be again the width available in slot S_l for R_p in configuration C_{i-1}. Assume $w_p > d'_l$ (i.e., S_l now is not big enough for R_p.) From the leaf containing d_l, we start traversing the path towards the root. Let v be the first node we meet on this path, and $rneighbor(v)$ be v's right neighbor. If $w_p \leq value(rneighbor(v))$, we perform a downward searching in the subtree rooted at $rneighbor(v)$ to

determine the slot for rectangle R_p; otherwise, we continue with v's parent. Figure 3(b) shows how the new position for R_7 is determined (the dashed lines indicate the links traversed.) Assume R_p's new position is in slot S_a. Then, it takes $O(1 + \log(a - l))$ time to determine this slot. We point out that for the rectangles for which Case 2 applies no updating is done and necessary. Their actual position can be, when needed, determined in $O(1)$ time.

We summarize the main steps of our improved algorithm. The preprocessing includes building the finger space tree, generating the topological order with $l_0 \leq l_1 \leq \ldots \leq l_{n+1}$, and constructing for each rectangle R_i list L_i containing the rectangles reachable from R_i (in a topological order). This requires $O(k + n \log n + \rho)$ time. We then generate the configurations $C_n, C_{n-1}, \ldots, C_1, C_0$. We generate the width of C_{i-1} from C_i by computing new positions for only the rectangles reachable from R_i, as described above. Once we have the index i resulting in the left-compressed minimum configuration, we re-build configuration C_i in $O(n \log k)$ time by setting $x_{C_i}(R_0) = x(B_1) - l_i$ and left-compressing the n rectangles.

We now show that this algorithm achieves the claimed time bound. Let r_i be the number of rectangles that can reach rectangle R_i (we assume that a rectangle can reach itself.) Initially, rectangle R_i is to the left of B_1. At some point R_i moves across B_1 and is assigned a new position in a slot to the right of B_1. From this point on, every time a rectangle that can reach R_i moves across B_1, rectangle R_i may get re-positioned. Its new position is always to the right of its old position. Let $f_{i,j}$ be the number of slots rectangle R_i moves to the right when R_i is re-positioned for the j-th time,

$1 \leq j \leq r_i$. By using the finger space tree to determine the new slots, the total time needed to re-position rectangle R_i is

$$\sum_{j=1}^{r_i} (1 + \log f_{i,j})$$

which is less than

$$r_i \log \frac{k}{r_i} + r_i.$$

Let T_p be the total time needed to re-position all rectangles. Then,

$$\begin{aligned} T_p &\leq \sum_{i=0}^{n} \left(r_i \log \frac{k}{r_i} + r_i \right) \\ &\leq \sum_{i=0}^{n} r_i \log k - \sum_{i=0}^{n} r_i \log r_i + \sum_{i=0}^{n} r_i \\ &\leq \rho \log k - \sum_{i=0}^{n} r_i \log r_i + \rho, \end{aligned}$$

where $\sum_{i=0}^{n} r_i = \rho$. A straightforward computation shows that

$$\sum_{i=0}^{n} r_i \log r_i > \rho \log \frac{\rho}{n}.$$

Using this lower bound on $\sum_{i=0}^{n} r_i \log r_i$, we get

$$T_p < \rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho.$$

Hence, it takes $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho)$ time to compute the widths of configurations $C_n, C_{n-1}, \ldots, C_0$. Note that the $O(n \log k)$ time needed for re-building the minimum configuration is bounded by $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho)$. We conclude this section with the following result.

Theorem 2.1 *Given n rectangles and k forbidden regions, the k-partition problem can be solved in $O(\rho \log \left\lceil \frac{nk}{\rho} \right\rceil + \rho)$ time with $O(k + n \log n + \rho)$ preprocessing time.*

3 Forbidden region problem

In this section we generalize the approach developed in the previous section to solve the forbidden region problem. We again use two fictitious rectangles R_0 and R_{n+1}, each being of width 0 and of height h. Rectangle R_0 is always positioned to the left and R_{n+1} always to the right of all forbidden regions, respectively. Our algorithm will only generate left-compressed configurations. Clearly, left-compressing a configuration cannot increase its width.

Every rectangle R_i has now a set of slots, $\mathcal{S}_i$, associated with it. Set $\mathcal{S}_i$ is determined as follows. We say forbidden region B_j and rectangle R_i are *related* if we can draw a horizontal line intersecting both B_j and R_i. Assume rectangle R_i and forbidden region B_j are related and let b_j be the width of B_j. Consider the rectangular region of maximal width that has position $(x(B_j)+b_j, y(R_i))$ as its lower left corner, has a height equal to the height of R_i and does not intersect any other forbidden region. If the width of this region is at least w_i (i.e., R_i can be placed into it), then this region represents a slot in set $\mathcal{S}_i$. We also include into set $\mathcal{S}_i$ two special slots of infinite width. Namely, the slot whose right border coincides with the left border of the leftmost forbidden region and the slot whose left border coincides with the right border of the rightmost forbidden region related to R_i.

Our first property is a generalization of Property 2.1 of the k-partition problem.

Property 3.1 *Let C be a left-compressed minimum configuration. Then,*

there exists a rectangle R_i, $0 \leq i \leq n$, *and a forbidden region* B_j, $1 \leq j \leq k$, *such that*

$$x_C(R_0) + l_i = x(B_j) \text{ and } x_C(R_i) + w_i = x(B_j).$$

Let q_i be the number of forbidden region rectangle R_i is related to (i.e., $q_i = |S_i|$), and let $\delta = \sum_{i=0}^{n} q_i \leq kn$. Property 3.1 states that the minimum configuration is one among δ configurations. Property 3.1 can easily be proven by contradiction and its proof is omitted. Observe that the requirement that R_0 gets positioned to the left of all forbidden regions is necessary to make Property 3.1 true. It is possible that $i = 0$ is the only index in the minimum left-compressed configuration for which the property holds.

Our algorithm generates the δ configurations in an order that allows us to update the necessary information about new positions of the rectangles efficiently. The positions of rectangle R_0 in the δ configurations are determined in $O(\delta)$ time (by using the l_i's and the $x(B_j)$'s). We then order these positions of rectangle R_0 by increasing x-values. At this time we also discard any configurations in which $x(R_0)$ is greater than the x-position of the leftmost forbidden region (obviously, these configurations are not feasible). Let $C_1, C_2, \cdots, C_{\delta-1}, C_\delta$ be the left-compressed configurations with $x_{C_a}(R_0) < x_{C_{a+1}}(R_0)$. Not every one of these δ configurations represents necessarily a feasible configuration. Let C_a be a configuration in which the position of R_0 is dictated by R_i and B_j. If C_a is feasible, then every rectangle R_t on the longest path from R_0 to R_i is positioned at $x_{C_a}(R_0) + l_t - w_t$. We say that the path from R_0 to R_i is *tight*. Configuration C_a is not feasible when forbidden regions block the tight path from R_0 to R_i. The test of whether configuration C_a is indeed feasible is done during the algorithm.

We next describe how to generate the configurations. It is clear that, when generating the configurations in the order of increasing x-value of the position of rectangle R_0, a rectangle can only move to the right. Recall that set $\mathcal{S}_i$ contains the possible slots rectangle R_i can be positioned in. Our algorithm organizes set $\mathcal{S}_i$ as a linear list containing the slots with increasing x-positions. Each list $\mathcal{S}_i$ will be traversed at most once during the algorithm. The initial configuration C_1 is generated by positioning R_0 at the associated position and performing a left-compression. Assume now that we have decided whether configuration C_a is feasible and, if it is, have determined its width. While generating the configurations we maintain for every rectangle R_i again a variable $x'(R_i)$. Unlike to the k-partition problem, $x'(R_i)$ may contain the correct position of R_i in some configuration C_a, but not in a later one. The actual position of R_i in C_a can be, when needed, determined as follows.

$$x_{C_a}(R_i) = \max\{x'(R_i), x_{C_a}(R_0) + l_i - w_i\}$$

In some sense, $x'(R_i)$ contains the correct position of R_i whenever the path from R_0 to R_i in C_a is not tight.

Let rectangle R_{i1} and forbidden region B_{j1} be the pair that dictates the position of R_0 in C_a (i.e., $x_{C_a}(R_0) = x(B_{j1}) - l_{i1}$). Let R_{i2} and B_{j2} be the pair that dictates the position of R_0 in C_{a+1}. Let $\epsilon = x_{C_{a+1}}(R_0) - x_{C_a}(R_0)$. We check in $O(1)$ time whether having rectangle R_{i2} at position $x(B_{j2}) - w_{i2}$ results in a feasible configuration as follows. We compute $x_{C_a}(R_{i2})$, the position of R_{i2} in configuration C_a. If $x_{C_a}(R_{i2}) + \epsilon = x(B_{j2}) - w_{i2}$, then the path from R_0 to R_{i2} in C_{a+1} is tight and C_{a+1} is feasible. We next describe how to compute the width of configuration C_{a+1}. The crucial insight into

computing the width of C_{a+1} efficiently lies in the fact that, in order to compute the width, we only need to explicitly re-position the rectangles reachable from R_{i1}. When going from configuration C_a to configuration C_{a+1}, a rectangle R_p finds itself in one of four possible situations.

Case 1. Rectangle R_p is reachable from rectangle R_{i1}. Since R_{i1} moves from being immediately to the left of forbidden region B_{j1} to being to the right of it, rectangle R_p may need a new position. Using the example shown in Figure 4, rectangle R_{p1} is reachable from R_{i1} and gets a new position assigned while rectangle R_{p2} keeps its position. We determine the new position by using set S_p and we record the new position in entry $x'(R_p)$.

Case 2. Rectangle R_p is not reachable from R_{i1} and the path from R_0 to R_p in configuration C_a is tight. Then, in configuration C_{a+1} rectangle R_p moves ϵ positions to the right. This situation applies to rectangle R_{p3} of Figure 4. The change in position is not explicitly recorded since doing so would be too time consuming. Recall that we are able to compute the position of any rectangle on a tight path on $O(1)$ time. It remains to be shown that moving rectangle R_p ϵ positions to the right always results in a feasible configuration (i.e., this does not cause R_p to overlap with any other rectangle or forbidden region). Since we are considering left-compressed configurations, R_p can obviously not overlap with another rectangle. Assume now that R_p overlaps with a forbidden region B_t. If there exists more than one rectangle with this property, choose R_p such that no predecessor of R_p that was moved ϵ positions overlapped with a forbidden region. The distance between the right side of R_p in C_a and $x(B_t)$ is less than ϵ. This implies that there exists

a configuration C dictated by R_p and B_l with $x_C(R_0) = x(B_l) - l_p$ such that $x_{C_a}(R_0) < x_C(R_0) < x_{C_{a+1}}(R_0)$. Such a configuration C cannot exist and thus R_p cannot overlap with a forbidden region.

Case 3. Rectangle R_p is not reachable from R_{i1}, the path from R_0 to R_p in C_a is not tight, and the path from R_0 to R_p in C_{a+1} is tight. The amount R_p moves to the right is now determined by ϵ minus the amount of "non-tightness" on the path from R_0 to R_p in configuration C_a. This situation applies to rectangle R_{p4} of Figure 4. Note that in configuration C_a $x'(R_p)$ contained the correct position of R_p, while in C_{a+1} the correct position is determined by $x_{C_{a+1}}(R_0) + l_p - w_p$. The argument that moving R_p to the right results in a feasible configuration is as given for Case 2.

Case 4. Rectangle R_p is not reachable from R_{i1} and the path from R_0 to R_p in neither tight in C_a nor C_{a+1}. In this situation rectangle R_p does not change its position when going from configuration C_a to configuration C_{a+1}. An example for this situation is rectangle R_{p5} in Figure 4.

We are now ready to give a complete description of our algorithm. The preprocessing step includes computing the x-position of rectangle R_0 in the δ configurations, arranging the configurations according to increasing x-value of R_0, and constructing the set S_i for every rectangle R_i, $1 \leq i \leq n$. These steps take $O(\delta \log \delta + (n+k) \log k)$ time. We then generate the configurations $C_1, \cdots, C_{\delta-1}, C_\delta$. When generating C_{a+1} from C_a we determine the new positions for rectangles reachable from R_{i1} as follows. Let R_p be a rectangle reachable from R_{i1} so that all predecessors of R_p for which Case 1 applies have been handled. Let R_m be the immediate predecessor of R_p for which

Case 1 applies (there exists at least one) and for which $x_{C_{a+1}}(R_m) + w_m$ is a maximum. This index m is determined while the immediate predecessors of R_p are re-positioned. Rectangle R_p needs a new position if the condition

$$x_{C_{a+1}}(R_m) + w_m > max\{x'(R_p), x_{C_{a+1}}(R_0) + l_p - w_p\}$$

is satisfied. Note that the right-hand side of the condition does not correspond to the position of R_p in C_a. The quantity $x_{C_{a+1}}(R_0) + l_p - w_p$ already takes the shift to the right from position $x_{C_a}(R_0)$ to $x_{C_{a+1}}(R_0)$ into account. If the condition is true, rectangle R_p overlaps with another rectangle (not necessarily R_m). In order to determine R_p's new position, we locate, using $\mathcal{S}_p$, the leftmost position $\geq x_{Ca+1}(R_m) + w_m$. This position is located by a linear scan which starts at the slot containing the old position of R_p. When all rectangles reachable from R_{i1} have been handled, we compute the width of configuration C_{a+1}. This width is determined by $x_{C_{a+1}}(R_{n+1}) - x_{C_{a+1}}(R_0)$ and is hence computed in $O(1)$ time. After the widths of all configurations have been computed, we re-build the left-compressed configuration giving minimum width in $O(\delta)$ time.

We now establish the claimed time bound and start with the time required for re-positioning the rectangles. Let r_i' be the number of rectangles which can be reached by rectangle R_i. Each time rectangle R_i is pushed across a forbidden region, we may have to re-position all the rectangles which can be reached from R_i. Rectangle R_i is pushed across at most q_i forbidden regions and in each time we may re-position r_i' rectangles. Hence the total number of times R_i causes a re-positioning is at most $q_i r_i'$. Overall, we re-position at most $\Delta = \sum_{i=1}^{n} q_i r_i'$ rectangles. In order to find new positions for all rectangles the lists $\mathcal{S}_i$, $1 \leq i \leq n$, are traversed. This costs

an additional $O(\sum_{i=1}^{n} q_i) = O(\delta)$ time. Since $r_i' \geq 1$ (i.e., a rectangle can be reached by itself), we have $\Delta \geq \delta$. We can thus state the following theorem.

Theorem 3.1 *Given n rectangles and k forbidden regions, the forbidden region problem can be solved in $O(\Delta)$ time with $O((n+k)\log k + \delta \log \delta)$ preprocessing time.*

We conclude this section by observing that the algorithm we presented can also be used to solve a slightly different, somewhat more general problem. Assume every rectangle R_i has its own set of forbidden regions associated with it. A minimum configurations is now a configuration in which no rectangles overlaps with its own forbidden regions and the area induced by the rectangles and all the forbidden regions is a minimum. Since our algorithm associates with every rectangle its own list of slots the rectangle can be placed in, changing how the lists are generated results in an algorithm solving this problem.

4 Minmax k-partition problem

When each of the k forbidden regions has height h, the forbidden regions model positions in the layout area where a vertical cut can be made. In certain environments one may need to make k cuts, but does not have the positions of the cuts pre-determined. Rather, the cuts should be made so that the maximum distance between two consecutive cuts is minimized. We refer to this problem as the minmax k-partition problem and present an algorithm to solve it in $O(\rho + n \log n)$ for arbitrary k, where ρ represents the number of edges in the transitive closure of the visibility graph induced by

the rectangles. For the case when the layout components get separated by only one cut we present an $O(n)$ time algorithm.

We start by giving a more formal definition of the minmax k-partition problem. Given are again n rectangles, $R_1, R_2, \ldots, R_n$, where R_i has width w_i. We are to determine the position of k vertical cuts so that all rectangles are to the right of the first cut and to the left of the k-th cut, respectively, and no rectangle intersects a cut (i.e., the rectangles are partitioned into $k-1$ groups). Let $B_1, B_2, \ldots, B_k$ be the cuts, $S_1, S_2, \ldots, S_{k-1}$ be the slots (i.e., the area between two consecutive cuts), and let $d_1, d_2, \ldots, d_{k-1}$ be the distances between two consecutive cuts. Let $d^* = \max_{1 \leq i < k-1} d_i$. A minimum configuration for the minmax k-partition problem is one in which d^* is a minimum. We also refer to d^* as the *width* of the partition. Observe that the statement of the problem requires $k \geq 2$. For $k = 2$, it corresponds to the standard compaction problem and for $k = 3$ it corresponds to compacting the rectangles onto 2 layers.

The following property characterizes a relationship between the width of a minimum configuration and the length of the longest path between two rectangles. It is the basis for reducing the search space containing the minimum configuration.

Property 4.1 *Let C be a minimum configuration of width d^*. Then, there exist rectangles R_i and R_j and two consecutive cuts B_a and B_{a+1} such that*

$$d^* = d_a = l_{i,j}$$

where $l_{i,j}$ is the length of the longest path from R_i to R_j in G.

This property states that the width of a partition is equal to the length

of the longest path between two rectangles. Our algorithm first generates the necessary $l_{i,j}$'s. For every rectangle R_i we determine the length of the longest paths from R_i to all rectangles reachable from R_i. Using G, these values are generated in $O(\rho)$ time.

We then employ a binary search strategy to find d^*. Our algorithm uses as a procedure that, given a value d, determines the minimum number of cuts needed to achieve a partition of width d. Let MIN_CUT(d) be this procedure. Using G, MIN_CUT(d) generates the number of cuts needed in $O(n)$ time as follows. Assume we have an infinite number of cuts $B_1, B_2, \ldots$ with the distance between two consecutive cuts being d. We now process the rectangles in a topological order induced by G. Let R_i be the rectangle currently being processed, and R_{max} be the rectangle that is a predecessor of R_i in G and for which $x(R_{max}) + w_{max}$ is a maximum. Assume R_{max} is located in slot S_l. If $x(R_{max}) + w_{max} + w_i \leq l \cdot d$, then R_i is assigned position $x(R_{max}) + w_{max}$ in slot S_l. Otherwise it is assigned position $x(B_{l+1})$ in slot B_{l+1}. It is straightforward to see that MIN_CUT(d) generates the number of cuts needed for a given d in $O(n)$ time. Note that when d is less than the width of one of n rectangles, MIN_CUT(d) returns zero.

Assume now that some $l_{i,j}$ is the input for MIN_CUT. If the minimum number of cuts returned is larger than k, a larger width is needed in a configuration making k cut. Otherwise, a minimum configuration making k cuts can possibly achieve a smaller width. Hence, using a binary search, we can find the optimal d^* in $O(\rho + n \log n)$ time.

We next describe an algorithm that solves the minmax problem in $O(n)$ time for $k = 3$. In this case we only have two slots available and a rectangle

R_j is either assigned to slot S_1 or slot S_2. The minimum width d^* is now determined by either $l_{0,j}$ or $l_{j,n+1}$ for some j. We thus only determine $l_{0,j}$ and $l_{j,n+1}$ for every j. This is done in $O(n)$ time. An obvious lower bound on the width of a minimum configuration is $\frac{1}{2}l_{0,n+1}$. Let $d_{opt} = \frac{1}{2}l_{0,n+1}$ and let R_j be any rectangle. If $l_{0,j} \leq d_{opt}$, then any left-compressed configuration of minimum width assigns R_j to slot S_1. Otherwise, the decision on where to put R_j is based on the following rule: If $l_{0,j} \leq l_{j,n+1}$, rectangle R_j is assigned to slot S_1; otherwise R_j is assigned to slot S_2. Using these conditions, it is straightforward to develop an $O(n)$ time algorithm. First, we compute $l_{0,j}$ and $l_{j,n+1}$ for each rectangle R_j. We divide the n rectangles into two sets according to the rules stated above and then compute the resulting width d^*. The following theorem is a consequence of the above discussion.

Theorem 4.1 *The minmax k-partition problem can be solved in $O(n)$ time for $k = 3$ and in $O(\rho + n \log n)$ time for $k \geq 4$.*

References

[1] D.G. Boyer. Symbolic layout compaction review. In *Proceedings of 25th ACM/IEEE Design Automation Conference*, pages 383–389, January 1988.

[2] Y.E. Cho. Subjective review of compaction. In *Proceedings of 22th ACM/IEEE Design Automation Conference*, pages 396–404, January 1985.

[3] K. Melhlhorn. *Data Structures and Algorithm 1: Sorting and Searching.* Springer-Verlag, 1984.

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

[5] A.R. Newton and A.L. Sangiovanni-Vincentelli. Computer-aided design for vlsi circuits. *Computer*, 19:38–63, 1986.

[6] 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.

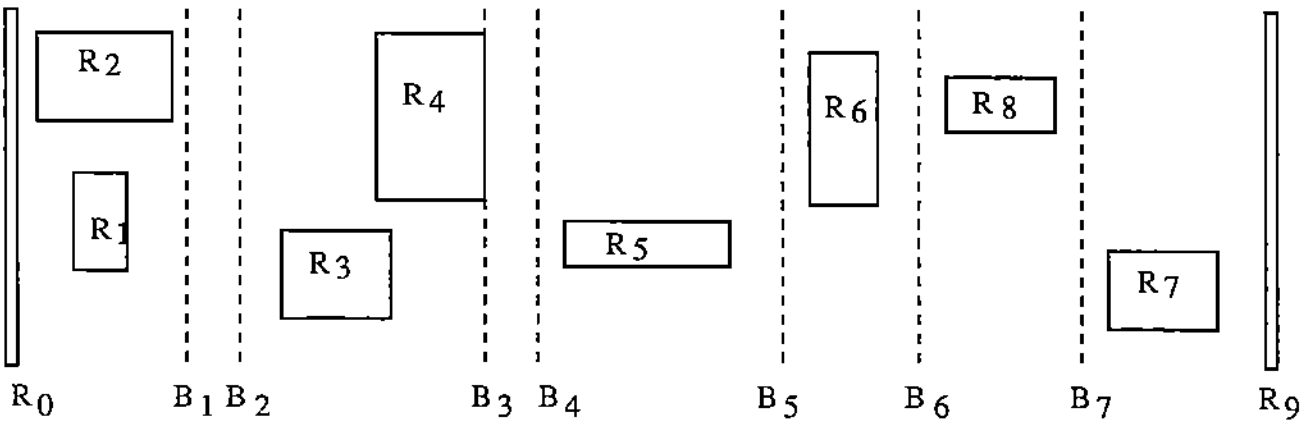

Fig. 1(a) Initial configuration of a 7-partition problem with 8 rectangles.

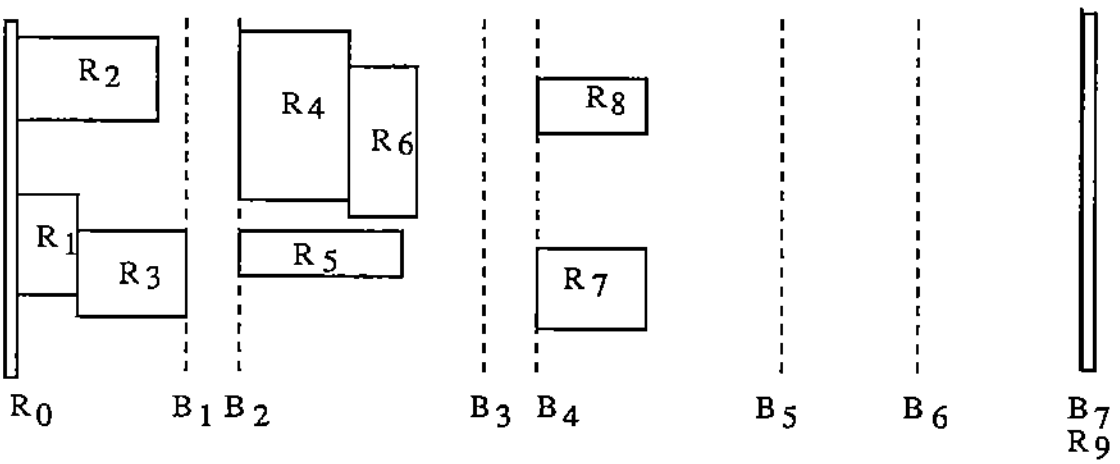

Fig. 1(b) The left-compressed version of the configuration in Fig. 1(a).

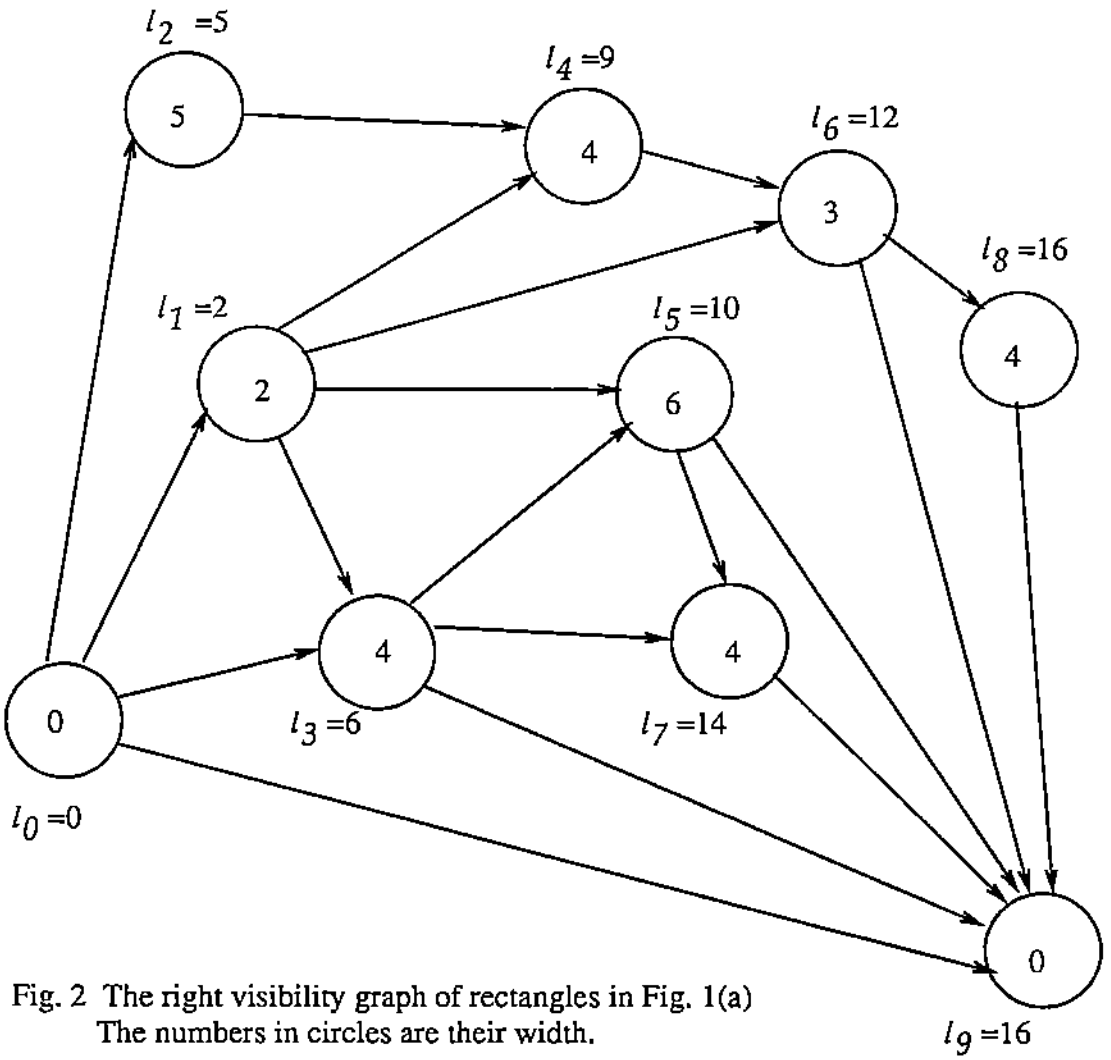

Fig. 2 The right visibility graph of rectangles in Fig. 1(a)
The numbers in circles are their width.

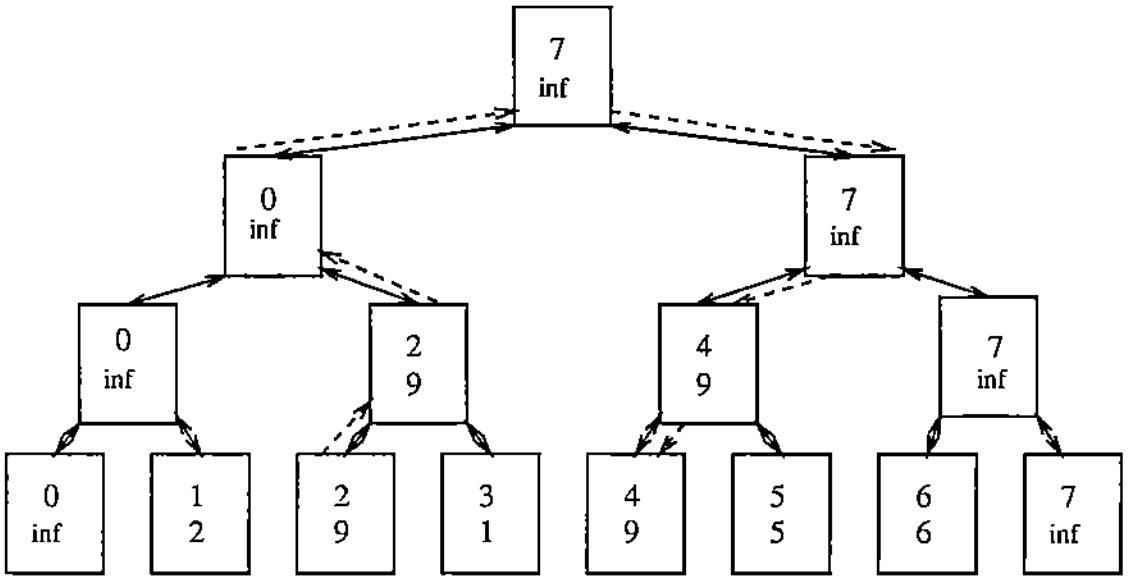

Fig. 3(a) The binary space tree for the forbidden region in Fig. 1(a)
The upper number and lower number in each node are its index and value, respectively.

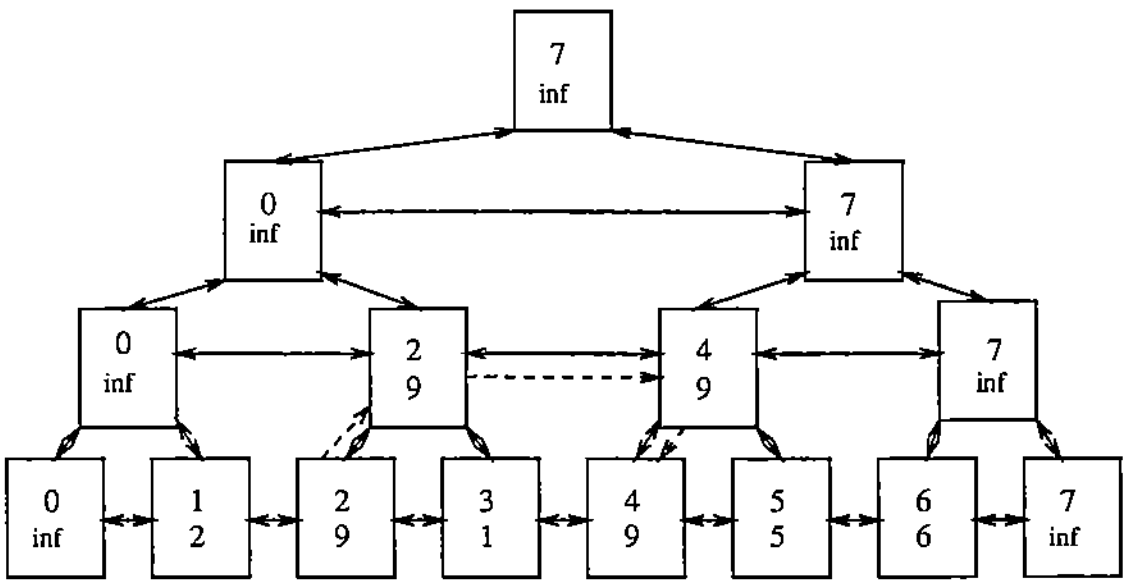

Fig. 3(b) The finger space tree for the forbidden region in Fig. 1(a).
The upper number and lower number in each node are its index and value, respectively.

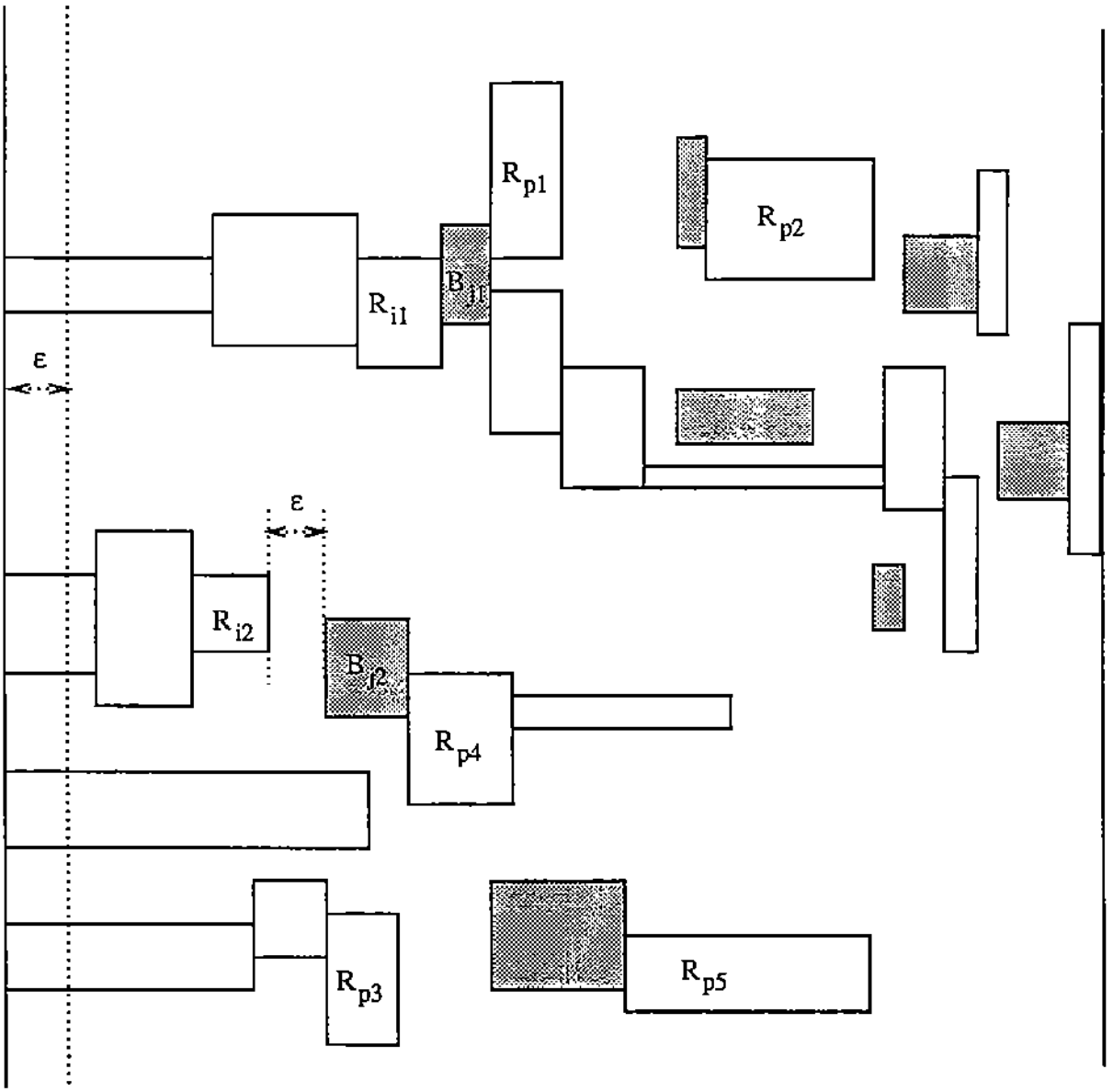

Fig.4 Four cases in generating configuration C_a from C_{a+1}

C_a is defined by R_{i1} and B_{j1}.
C_{a+1} is defined by R_{i2} and B_{j2}.