RECTANGLE LAYOUT OPTIMIZATION

Kai Tang

Stephen Pollock

Department of
Industrial & Operations Engineering
University of Michigan
Ann Arbor, MI 48109-2117

Technical Report 90-26

October 1990

Rectangle Layout Optimization

Kai Tang
Stephen Pollock

Dept. of Industrial and Operations Engineering
The University of Michigan
Ann Arbor, MI 48105

Oct. 14, 1990

1. INTRODUCTION

Floor plan layout is an early stage of **VLSI** design. It is based on the following simplified but very realistic assumption [MEAD]: each piece of circuitry or module can be modeled as a rectangle. In order to obey certain necessary electronic and physical properties (e.g., two modules must be a certain distance away from each other to prevent possible magnetic interference or a module can not be made arbitrarily small due to manufacturing feasibility), a rectangle must have lower-bound limits on its length and width, and each rectangle must obey some positional constraints relative to some other rectangles. For example, the width of a wire must be greater than some value λ in order to keep the wire resistance below some threshold; a cut might be required to be totally inside a diffusion so that they can function properly.

Since the bigger a module is in a **VLSI** layout the more material it needs, there is a cost function associated with each module, which can be reasonably simplified as a linear function of the area of that module. Therefore, one optimization objective in **VLSI** floor plan layout design might be to minimize the overall cost of all these modules. Specifically, suppose there are n modules to be designed in the layout and let $\alpha_1, \alpha_2, \ldots, \alpha_n$ be their final designed areas, then we would like $\sum \omega_i \alpha_i$ (i=1,2,...,n) to be as small as possible, where each ω_i is a non-negative "cost per unit area" constant.

Another objective in **VLSI** layout design might require the rectilinear hull of the final layout to be minimized. The rectilinear hull of a set of rectangles is just the smallest rectangle that covers the set. (See Figure 1.1.) This requirement can be easily justified by noticing that a layout will often become a module in a higher level layout. For instance, the **VLSI** layouts of a micro processor and a memory module usually are designed independently; their rectilinear hulls are then taken as two individual modules to a higher stage layout design with some constraints between the two rectilinear hulls only. As an illustration, the layout design in Figure 1.2 can be viewed as a two level hierarchy design: first design the layouts of Processor and Memory, and then design the layout of their rectilinear hulls and two other modules M_1 and M_2.

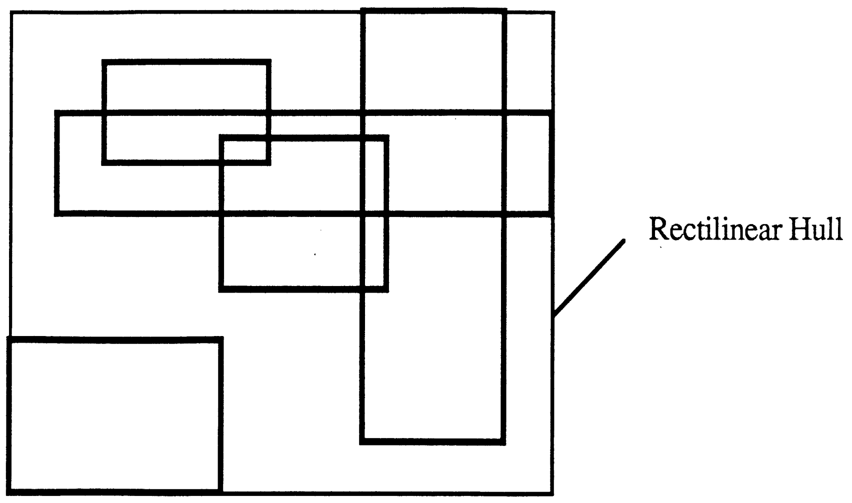


Figure 1.1. Rectilinear Hull of five rectangles

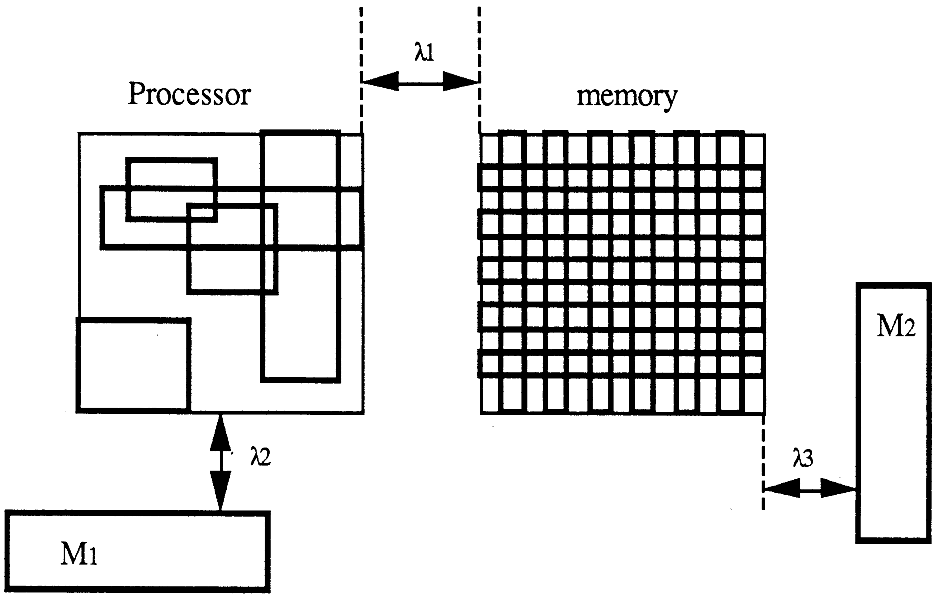


Figure 1.2. Multi-stage layout design

We now generalize this **VLSI** layout design to a broader optimization problem: the *Rectangle Layout Optimization* (RLO) problem. Study of this problem sheds light on related applications such as facility layout design in industrial engineering, as well as exposure to some interesting algorithmic aspects of graph theory.

2. PROBLEM FORMULATION

In this section, we give a rigorous formulation of the RLO problem. First, we present some definitions and terminology.

2.1. Constraint and Relations

Definition 2.1. A constraint (λ_1, λ_2) is an interval on the real positive line; λ_1 and λ_2 are called the lower and upper bounds of the constraint respectively.

Each rectangle in the layout is associated with two constraints (w_1,w_2) and (h_1,h_2). The first constraint (w_1,w_2) bounds the width of the rectangle. More precisely, it specifies that the width of the rectangle must be no less than w_1 and no greater than w_2. The second constraint (h_1,h_2) bounds the height of the rectangle.

The following three types of constraint are distinguished.

Definitions 2.2. A constraint (λ_1, λ_2) is said to be *tight* if $\lambda_1=\lambda_2$. The constraint (λ_1, ∞) is an *upper-free* constraint. A constraint is said to be *ordinary* if it is neither tight nor upper-free.

In terms of the design, a rectangle with a tight width constraint (λ, λ) means that its width must be of fixed length λ. An upper-free height constraint $(\lambda, +\infty)$ indicates that its height must be no shorter than λ but can be arbitrarily long.

Two rectangles are *related* to each other if there are any constraints among their relative positions. Figure 2.1 shows an example of two related rectangles O_i and O_j, with the following relations: (a) the left side of O_j must be to the right of the left side of O_i and their distance must be in the range (λ_1, λ'_1); (b) the top side of O_j should be below that of O_i by a distance bounded by (λ_2, λ'_2); and (c) the top side of O_j must be above the bottom side of O_i such that their distance is within the range (λ_3, λ'_3).

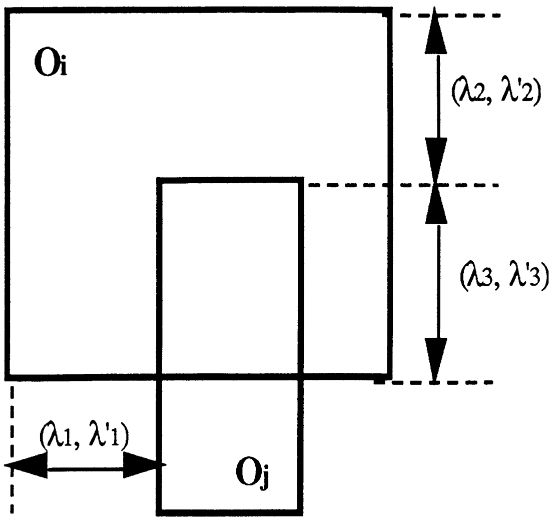


Figure 2.1. Constraints between two rectangles

The three constraints (λ_1, λ'_1), (λ_2, λ'_2) and (λ_3, λ'_3) in Figure 2.1 are called the *binding constraints* between O_i and O_j. A binding constraint is further said to be *homogeneous* if the two sides it binds are in a same category, i.e., either both are top sides, or bottom sides, or left sides, or right sides. Notice that by imposing the different types on them, various physical meanings are manifested. For example, if (λ_3, λ'_3) is a tight constraint (0,0), then the designing rule specified by it states that the top side of O_j and the bottom side of O_i must be coincident.

We shall assume that a constraint between two rectangles can relate to two parallel sides only. More precisely, a horizontal (vertical) side of a rectangle can only be related to a horizontal (vertical) side of another rectangle. This disjunction between the horizontal and vertical sides is directly due to our definition of a constraint, i.e., it is only a distance measure. It is not hard to see that between a pair of rectangles there can be at most 16 constraints.

2.2. Feasible Layout

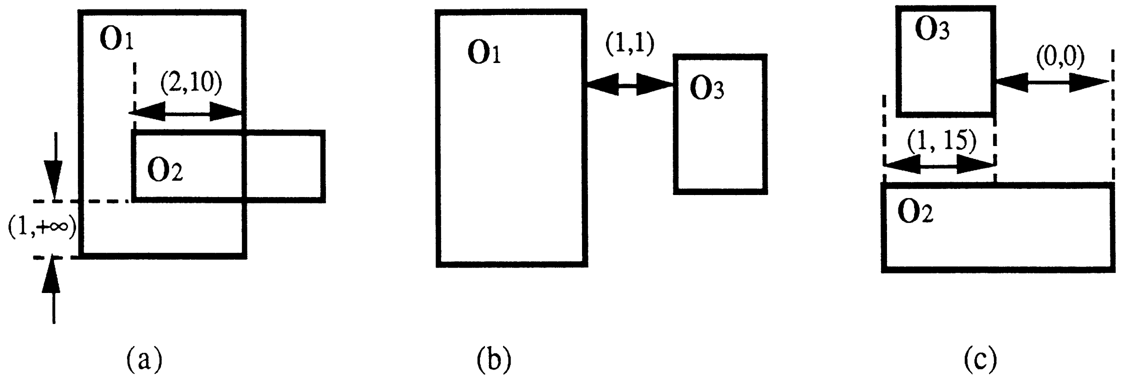
Let $O=\{O_1, O_2, ..., O_n\}$ represent the n rectangular objects required in the layout. Let O_i have width constraint (w_i,w'_i) and height constraint (h_i, h'_i), $i=1,2,...,n$. Let C_{ij} represent the set of constraints binding O_i and O_j, e.g., $C_{ij} = \{(\lambda_1, \lambda'_1), (\lambda_2, \lambda'_2), (\lambda_3, \lambda'_3)\}$ in Figure 2.1. Note that C_{ij} could be an empty set if the two rectangles are not related. The set $S=\{(w_i,w'_i), (h_i, h'_i): i=1,2,...,n\}$ and $C=\{C_{ij} : 1\leq i\leq n, 1\leq j\leq n, \text{ and } i\neq j\}$

will be referred to as the *self constraint set* and *mutual constraint set* respectively. Together, (S, C) form the *constraint description* of the n rectangles. Then, a *feasible layout* of O subject to S and C, is the determination of the following two data lists:

(1). $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$

(2). $(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)$

such that when the lower-left corner of the rectangle O_i is placed at the location (x_i, y_i) and O_i is assigned width a_i and height b_i, $i=1,2,...,n$, all the constraints in S and C are satisfied.



$(w_1,w'_1) = (3,10), (h_1, h'_1) = (5,10)$
$(w_2,w'_2) = (1,10), (h_2, h'_2) = (1,10)$
$(w_3,w'_3) = (2,10), (h_3, h'_3) = (3,10)$

Figure 2.2. A constraint description of three rectangles

As an illustration, consider finding a feasible layout of the three rectangles shown in Figure 2.2. Figure 2.3 shows a feasible layout whose satisfaction of all the constraints specified in Figure 2.2 can be easily verified. On the other hand, with minor change to some constraints in Figure 2.2, we might have no feasible layout solution. For instance, if in Figure 2.2(c) the constraint (1,15), which relates the left side of O_2 and the right side of O_3, is changed to (1,1.5), then there will be no feasible layout since this constraint (1,1.5) conflicts with the other three constraints (2,10), (1,1) and (w_3,w'_3). Actually, it can be easily shown that as long as the upper bound in that constraint is less than $2+1+w_3$, no feasible layout can exist. We will call a set of constraints S and C *invalid* if there is no feasible layout associated with them; otherwise, (S,C) is *valid* or *feasible.*

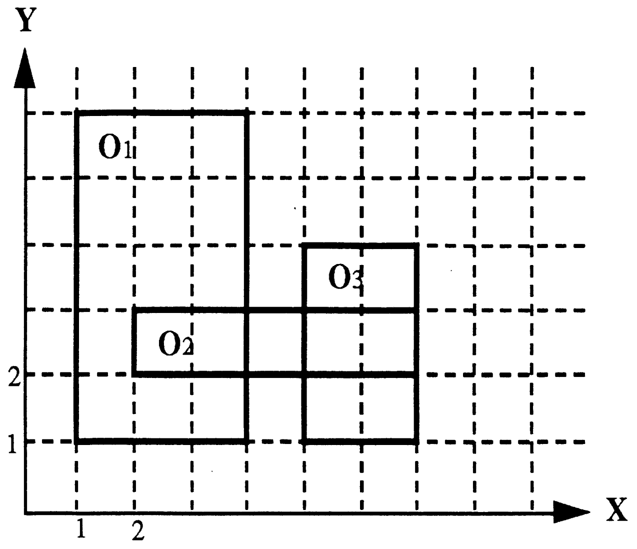


Figure 2.3. A feasible layout of the rectangles shown in Figure 2.2.

It is thus imperative to verify the validity of a set of constraints, prior to the search for a feasible layout and for various optimization objectives (which will be discussed later). To facilitate this verification process, we adopt the concept of a *binding graph.* Let T_i, B_i, L_i and R_i denote the top,bottom, left and right sides of the rectangular object O_i, i=1,2,...,n. The *vertical binding graph* of the n rectangles $O_1, O_2, ..., O_n$ is a labeled directional graph with 2n nodes; each of these nodes represents a top or bottom side of a rectangle; an edge is drawn from a node to another node if the former is required to be above the latter by a constraint, and the constraint itself is the label of this edge. The *value* of a node v, denoted as Val(v), is defined to be the Y coordinate of the side represented by v. The *horizontal binding graph* is defined in a similar manner on the left and right sides of the rectangles. Figure 2.4 shows the horizontal and vertical binding graphs of the rectangles and their constraints in the Figure 2.2.

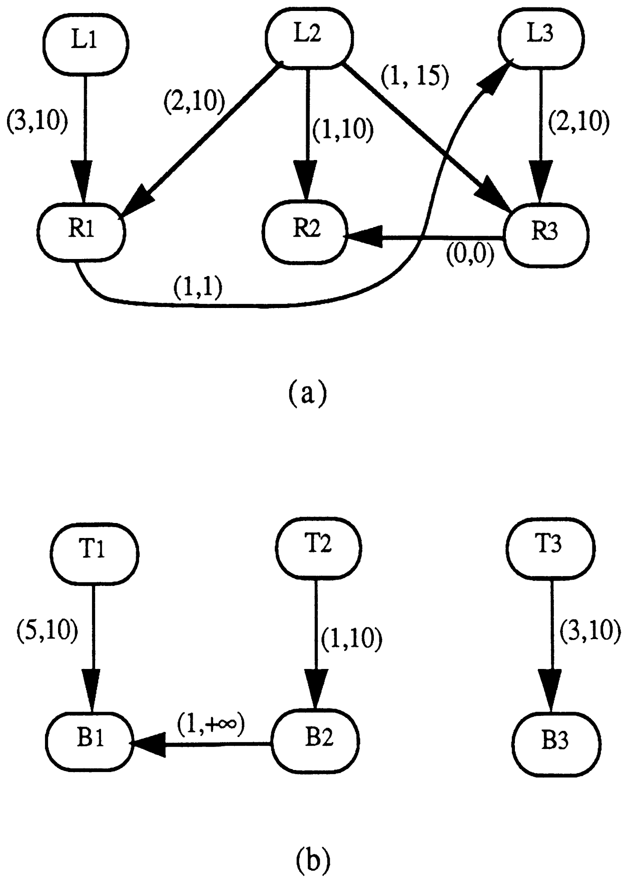


Figure 2.4. (a) Horizontal and (b) vertical binding graphs of the constraints in Figure 2.2

The horizontal and vertical binding graphs of a set of n rectangles and their associated constraints possess all the information for verifying the validity of the constraints. The two graphs themselves can be obviously constructed in O(n+m) time and space from the constraint description (S, C) where m is the total number of the constraints. Finding a fast algorithm to check for the validity, however, is expected to be very challenging. We will discuss this in the following.

2.3. Optimization Objectives

Given a constraint description (S,C) with at least one feasible layout, we can impose certain optimization objectives. Particularly, the following three optimization objectives are in order.

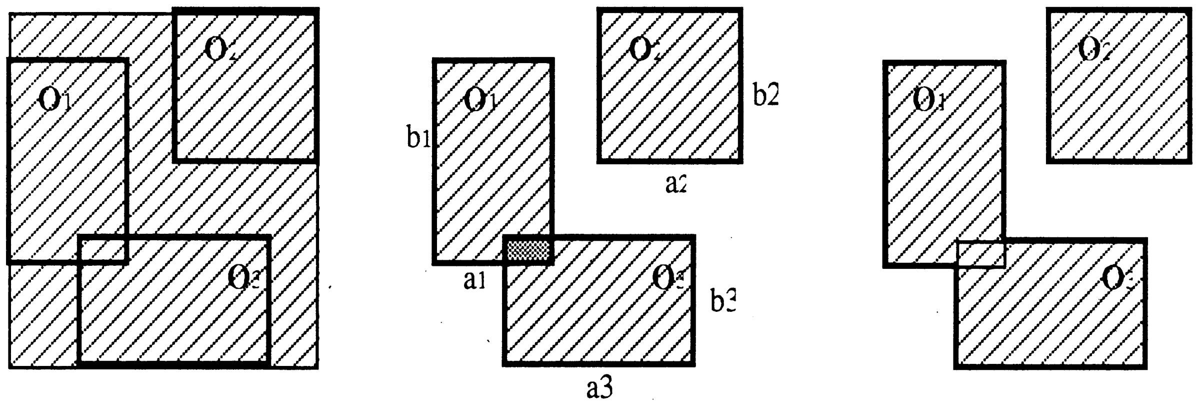
<u>Minimization of Rectilinear Hull (MRH)</u>. Given {S,C}, a feasible layout ({(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}, {(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)}) satisfies the minimization of rectilinear hull if the following product is minimized:

$$(\mathbf{Max}\{x_i + a_i : 1 \le i \le n\} - \mathbf{Min}\{x_i : 1 \le i \le n\}) * (\mathbf{Max}\{y_i + b_i : 1 \le i \le n\} - \mathbf{Min}\{y_i : 1 \le i \le n\}).$$

<u>Minimization of Overall Cost (MOC)</u>. Let $\omega_1, \omega_2, ..., \omega_n$ be n non-negative real *cost coefficients*. A feasible layout ({(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}, {(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)}) is said to have the minimum overall cost if the following summation is minimized:

$$\sum \omega_i\, a_i\, b_i \ (i=1,2,...,n).$$

<u>Minimization of Area Measure (MAM)</u>. The area measure of a set of rectangles is the area of the union of the rectangles. A feasible layout ({(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)}, {(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)}) has a minimum area measure if the area measure of the n rectangles in the layout is minimized.



$$C = \omega_1 a_1 b_1 + \omega_2 a_2 b_2 + \omega_3 a_3 b_3$$

(a) Rectilinear Hull (MRH) (b) Overall Cost (MOC) (c) Area Measure (MAM)

Figure 2.5. Definitions of Rectilinear Hull, Overall Cost, and Area Measure

Figure 2.5 illustrates these three definitions. When the cost coefficients ω_i's are all equal, the MOC problem reduces to the problem of minimizing the summation of the areas of n rectangles. We will call this reduced MOC problem the *Minimization of Overall Area* problem, or simply MOA. The inequivalence between MOA and MAM is apparent. For the example shown in Figure 2.5, if the areas of O_1, O_2 and O_3 are 2, 1.8, and 2 respectively and the overlapping area between O_1 and O_3 is 0.15, then the overall area is 2+1.8+2, whereas the area measure will be 2+1.8+2-0.15.

3. RLO Under Upper-free Constraints

In this section, we investigate a special case of the RLO problem when the constraint description (S,C) contains upper-free constraints only. As defined in the preceding section, an upper-free constraint is a semi-open interval $[\lambda,+\infty)$ on the positive real axis. For simplicity, we will hereafter use the lower-bound λ only to represent an upper-free constraint, with the understanding that its upper-bound is free.

As discussed in the preceding section, given a constraint description (S,C), solving the RLO problem consists of two stages: first verifying the validity of the vertical and horizontal binding graphs; then (if both graphs are valid) finding a feasible layout which meets the optimization objective.

3.1. Verifying the Validity of a Binding graph

A binding graph, from the way it is defined, is equivalent to a set of inequalities, each two of which corresponds to a single constraint. For example, the set of the inequalities corresponding to the horizontal binding graph in Figure 2.4(a) is listed below:

$$\begin{aligned} &R1 - L1 \geq 3, && R1 - L1 \leq 10 \\ &R2 - L2 \geq 1, && R2 - L2 \leq 10 \\ &R3 - L3 \geq 2, && R3 - L3 \leq 10 \end{aligned}$$

$$\begin{aligned} &R1 - L2 \geq 2, && R1 - L2 \leq 10 \\ &R3 - L2 \geq 1, && R3 - L2 \leq 15 \\ &L3 - R1 \geq 1, && L3 - R1 \leq 1 \\ &R2 - R3 \geq 0, && R2 - R3 \leq 0. \end{aligned}$$

The first group corresponds to the self constraints S, and the second group is due to the mutual constraints C.

Thus, the verification of the validity of a binding graph is equivalent to a linear programming problem with no optimization objective. More precisely, whether or not a binding graph is valid can be answered by verifying the feasibility of the set of inequalities implied by the binding graph. Such a linear programming approach, however, needs to be considered with caution. Although in practice the average running time of most current linear programming algorithms is almost linear in the number of variables and constraints, its upper bound is exponential with the input size [MURTY]. Thus, resorting to a general linear programming method might not be appropriate.

If all the constraints are upper-free, only "≥" inequalities exist. For example, if all the constraints of the binding graph in Figure 2.4(a) are upper-free, the corresponding inequalities are the following seven:

$$R1 - L1 \geq 3$$
$$R2 - L2 \geq 1$$
$$R3 - L3 \geq 2$$

$$R1 - L2 \geq 2$$
$$R3 - L2 \geq 1$$
$$L3 - R1 \geq 1$$
$$R2 - R3 \geq 0.$$

It is therefore desirable to have a worst case linear time algorithm that checks the validity of a set of "≥" inequalities. Let the lower-bound of the upper-free constraint associated with an edge in a binding graph be called the *weight* of that edge, and correspondingly the summations of the weights of the edges along a path the weight of that path. The following lemma provides a method for linear-time feasibility checking.

Lemma 3.1. Let G=(V,E) be a binding graph with upper-free constraints only. G is invalid if and only if there exist two vertices v and v' in V such that there is a cycle passing through v and v' whose weight is not zero.

Proof. Suppose there is a path $\pi(v,v')$ from v to v' and another path $\pi'(v',v)$ from v' to v; the summations of the weights of $\pi(v,v')$ and $\pi'(v',v)$ are W_π and $W_{\pi'}$ respectively. $\pi(v,v')$ and $\pi'(v',v)$ imply two inequalities:

$$\mathrm{Val}(v) - \mathrm{Val}(v') \geq W_{\pi} \quad \text{and} \quad \mathrm{Val}(v') - \mathrm{Val}(v) \geq W_{\pi'}. \qquad (3\text{-}1)$$

Clearly, such two inequalities are solvable if and only if both W_{π} and $W_{\pi'}$ are zeros.

Conversely, assume there is no non-zero weight cycle in G. We claim G must be valid. Suppose G is invalid and let v and v' be two offending vertices, i.e., they violate the two inequalities in (3-1). Only two cases are possible as the result of the induction of the constraints: (1) Val(v) - Val(v') ≥A and Val(v') - Val(v) ≥B, or (2) Val(v) - Val(v') ≥ A and Val(v) - Val(v') ≤B, where A and B are some non-negative real numbers and at least one of them is non-zero, and in case (2) B is less than A. The first case would imply a cycle between v and v' with total non-zero weight A+B, which violates our assumption. The second case contradicts the assumption that all the constraints are upper-free, because for any path from v to v', we can always arbitrarily increase its total weight. Q.E.D.

Lemma 3.1 implies that when the weights of all the edges of a binding graph are greater than zero, (in which case the graph is said to be in *regular form*), its validity can be verified by merely testing the acyclicity of the graph, i.e., the graph is valid if and only if it has no cycles. Since the test for acyclicity of a directed graph can be achieved by a depth-first search in time and space linear with the number of vertices and the edges of the graph [AHO], we immediately conclude that the verification of the validity of a regular binding graph can be done in linear time and space.

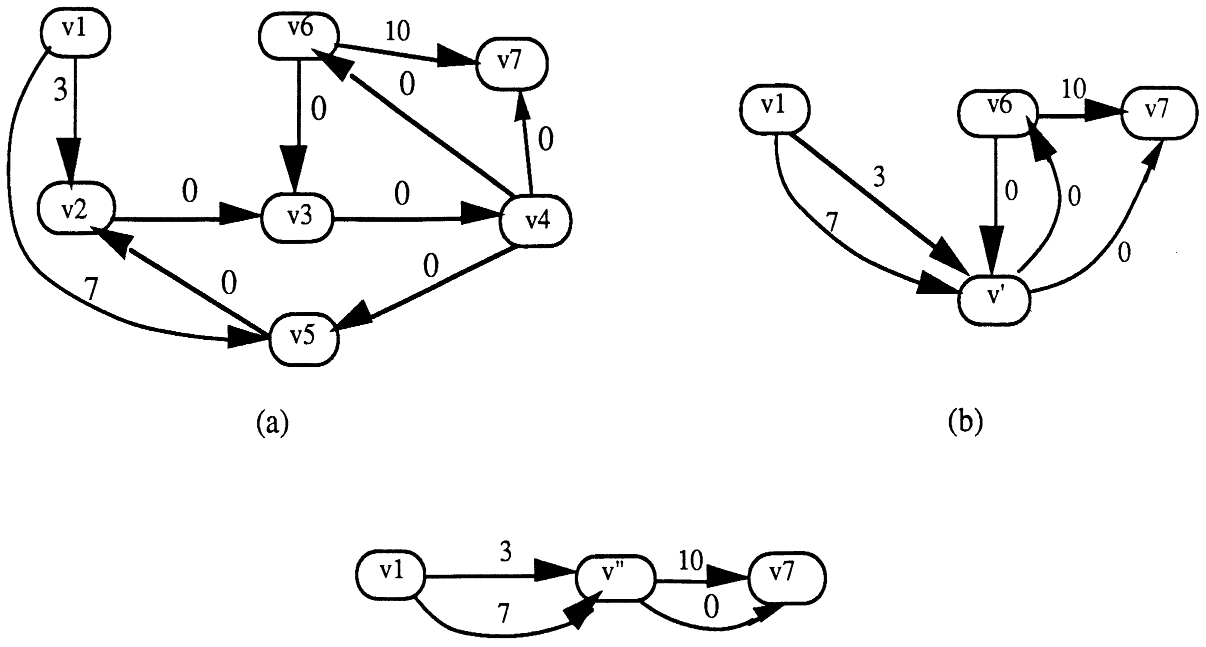
However, when a binding graph is not in a regular form, (i.e., it has some zero weight edges), the depth-first search might fail, because a zero cycle does not contribute to the invalidity. To account for this, we can use the concept of a *transitive closure.* The transitive closure of a directed graph G is also a directed graph G*, which has the same vertex set as G, but has an edge from a vertex v to a vertex v' of weight W if and only if there is a path of weight W from v to v' in G. Once having the transitive closure G* of a binding graph G(V,E), we can then perform a *trimming operation.* Specifically, if there is more than one edge from a vertex v to a vertex v' in G*, only the one with the maximum weight will be kept after the trimming operation. Calling the "trimmed" graph G**, it is straightforward to show that the set of inequalities for G** is equivalent to that for G*. Also, although we do not give a proof, such a trimming operation can be easily embedded into a transitive closure construction algorithm without changing its asymptotic time complexity. (See, for example,

[AHO].) Since the transitive closure of a directed graph with n vertices can be found in $O(n^3)$ time, so can the trimmed one. After G** is found, we then perform the following simple comparison between each pair of vertices v and v' on G**: if there is an edge of weight W from v to v' and another edge of weight W' from v' to v, and W and W' are not both zero, then G is invalid. Clearly, these comparisons can be done in $O(n^2)$ time. Therefore, the validity of a binding graph with n vertices can be verified in $O(n^3)$ time.

One disadvantage of the transitive closure method is that the $O(n^3)$ time bound is tight; that is, it is independent of the number of the edges in the graph. In many applications, the number of edges in a graph is much less than n(n-1). To take advantage this, we propose an algorithm whose time bound is sensitive to the number of the edges. The algorithm is based on the simple idea of merging the vertices of a zero weight cycle into a single vertex. We define a merging operation that transforms a binding graph with zero weight cycles into an equivalence graph of regular form. The term equivalence here is defined in the sense that the "≥" inequalities of the two binding graphs are essentially the same. As already shown in the proof of Lemma 3.1, a zero weight cycle enforces all the vertices on this cycle to have the same value (or coordinate) in any feasible solution. The merging operation thus replaces these vertices by a single vertex and correspondingly deletes the associated zero weight cycle from the binding graph. To keep the equivalence, each edge coming into or out of any vertex in this cycle is assigned to the replacing vertex. Figure 3.1 shows an example. The original binding graph in Figure 3.1(a) has a zero weight cycle v2-v3-v4-v5-v2; it is replaced by the single vertex v', as in Figure 3.1(b). Since v'-v6-v' is a zero-weight cycle; it is in turn replaced by a new vertex v". The result is the equivalent binding graph of regular form as shown in Figure 3.1(c).

It turns out that a maximal set of vertices, such that between any two vertices in this set there exists a zero weight path, is a strongly connected component* of the binding graph with all the non-zero weights excluded. This suggests a straightforward algorithm for implementing the merging operation, which is presented in Appendix 1, where we also show that the time complexity of the algorithm is O(|V|+|E|), an obvious improvement over the transitive closure method.

* A subset V' of V of a directed graph G(V,E) is a strongly connected set if for any pair of vertices in V' there is a cycle passing through them.



$v'' - v1 \geq 3$
$v'' - v1 \geq 7$
$v7 - v'' \geq 10$
$v7 - v'' \geq 0$

(c)

Figure 3.1. Merging the vertices of zero weight cycles

3.2. Minimization of the Rectilinear Hull

In this section, we present an algorithm that finds a feasible layout of a constraint description (S,C) that satisfies the MRH property, i.e., the area of its rectilinear hull is minimized. The algorithm depends crucially on the (assumed) independence between vertical and horizontal directions. Let $(S,C)_V$ and $(S,C)_H$ denote the vertical and horizontal constraints in (S,C) respectively. The terms feasible layout, MRH, MCO, and MMA can be analogously transplanted to $(S,C)_V$ and $(S,C)_H$. For example, the MCO problem of $(S,C)_V$ is to find two real number lists $(y_1, y_2, ..., y_n)$ and $(b_1, b_2, ..., b_n)$, such that when the lower end point of the vertical line segment i is placed at the height y_i and with the length b_i, $i=1,2,...,n$, all the constraints in $(S,C)_V$ are met and the sum $\sum\omega_i b_i$ $(i=1,2,...,n)$ is minimized, where ω_i $(i=1,2,...,n)$ are some given non-negative constants.

A Rectangle Layout Optimization problem of (S,C) is said to satisfy the separation condition if it can be solved by independently solving the RLO problems of $(S,C)_V$ *and* $(S,C)_H$, respectively.

If only a feasible layout is sought, because of the lack of dependence between vertical and horizontal constraints, the positions of the top and bottom sides (left and right sides) of the rectangles depend only on the vertical (horizontal) constraints. As a result, they satisfy the separation condition, and hence can be determined by the vertical and horizontal binding graphs respectively. We should be careful, however, to confirm this independence if an optimization objective is pursued. This is particularly important in our case since all the three optimization objectives MRH, MOC, and MMA involve areas which are two dimensional. Fortunately, as we are to show, the separation condition is applicable to both MRH and MOC.

Suppose we have a valid constraint description (S,C). After the merging process, its vertical and horizontal binding graphs become two *partial order trees*, i.e., directed graphs without cycles. As a convention, let those vertices in a tree that have no "IN" incident edges be called *source vertices* and those that have no "OUT" incident edges be called *sink vertices.* * A vertex with no incident edges, which is both a source and sink vertex, will be referred as an *isolated vertex*. For a non-trivial partial order tree **, clearly there should be at least one source vertex and one sink vertex. A *longest path* in a partial order tree is a path from a source to a sink whose weight is the maximum (there could be more than one). A longest path of a binding graph with weight W requires that its sink vertex must be below (to the right of) its source vertex by at least a distance W. Therefore, the width and the height of the minimum rectilinear hull of a constraint description (S,C) must be at least the weights of the longest paths of the horizontal and vertical binding graphs, respectively. If we can find a feasible layout whose rectilinear hull width and height are no greater than these two weights, it is certainly a MRH layout.

To facilitate the design of the algorithm, a minor augmentation is needed for the binding graph G(V,E). We add a dummy source vertex ROOT to V and assign an edge of zero weight from ROOT to every source vertex in V. It is easy to see the equivalence between

* An edge e is an "IN" edge of a vertex v if v is the head of this edge; e is an "OUT" edge of v if v is its tail vertex.

** A partial order tree is trivial if it has no edges.

the original G(V,E) and this augmented graph. We will also assume that a binding graph always has a unique single source vertex. The algorithm **PO_TREE** given in Appendix 2 assigns real numbers to the vertices of a partial order tree such that these numbers will satisfy the constraints implied by the edges of the tree. The algorithm is essentially carried out by a recursive procedure **ASSIGN**. We assume that the partial order tree is represented by the adjacency list data structure [AHO]. Associated with each vertex v are three pieces of data: Val(v) is a real number variable which will eventually store the value assigned to the vertex v; Count(v) is an integer variable that is initially the number of the "IN" edges incident at v; List(v) is a list of pointers that point to the child vertices of v. (A vertex w is a child of a vertex v if there is an edge from v to w.)

Refer to the algorithms **PO_TREE** and **ASSIGN**. At step 1 of **PO_TREE**, we initialize the Val of every vertex. ROOT is assigned 0 value since it is the only source vertex. All the other vertices are initialized with $+\infty$ that will later be updated by the procedure **ASSIGN**. In **ASSIGN**(v), we first check to see if Count(v) is zero. If yes, this means all the parent vertices of v have been assigned the final Val, and the following steps are taken for every child w of v: At step 1, the weight λ of the edge from v to w is retrieved. Since w must be less than Val(w) by at least λ, step 2 will assign Val(w) the number Val(v)-λ if it is less than the current Val(w). At step 3, we decrease Count(w) by one, which means one "IN" edge (constraint) of w has been processed. Finally, at step 4, the recurrence **ASSIGN**(w) is called. That the Val's of all the vertices (after **PO_TREE** terminates) satisfy all the constraints can be easily verified by the minimization of step 2 in **ASSIGN**. More importantly, since step 2 of **ASSIGN** always tries to assign the Val(w) as large as possible, (notice Val(w) is initially $+\infty$), and by the fact that the partial order tree G(V,E) has only one source vertex ROOT and no isolated vertices, we conclude that the number **Min**$\{\text{Val}(v): v \in V\}$ is maximized. That is, the number -**Min**$\{\text{Val}(v): v \in V\}$ is equal to the weight of the longest path of G(V,E).

<u>Lemma 3.2</u>. Algorithm **PO_TREE** runs in O(|E|) time and space.

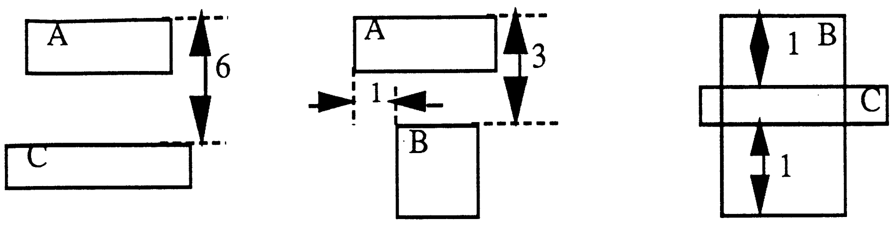
Proof. Notice that step 1 through step 4 in **ASSIGN**(v) will be executed only when Count(v) becomes zero. This guarantees that the "**OUT**" edges incident at v will be processed exactly once throughout the recurrence of **ASSIGN**. By introduction from ROOT which has no "IN" edges, we conclude that each edge in E will be processed only once. Therefore, **PO_TREE** runs in O(|E|) time. The proof for the linearity of the space requirement is trivial. Q.E.D.

For a given constraint description (S,C), by calling **PO_TREE** twice, once for the vertical binding graph and once for the horizontal binding graph, we can obtain the Y coordinates of the top and bottom sides and the X coordinates of the left and right sides of the rectangles. The induced layout of these rectangles is a feasible layout meeting the MRH optimization objective. Since constructing the binding graphs take time linear in the number of the constraints, it follows that the MRH problem of a valid (S,C) can be solved in linear time and space. We summarize this in a theorem.

Theorem 3.1. The Minimum Rectilinear Hull problem of a valid constraint description (S,C) can be solved in time and space linear in the number of the constraints.

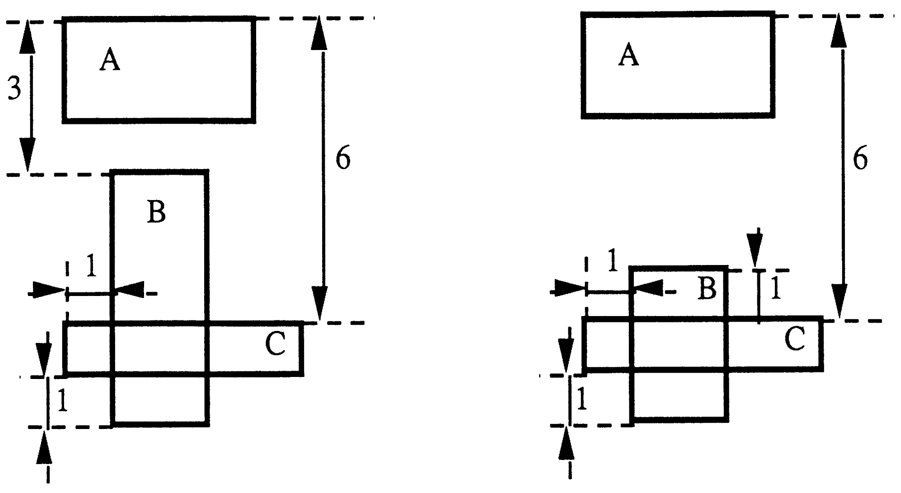
3.3. Minimization of Overall Cost: a Heuristic Procedure

Although the algorithm **PO_TREE** can be used to achieve the minimization of rectilinear hull of a feasible layout, we can not expect it to be directly applicable to the MOC problem. Figure 3.2(a) shows the (S,C) of three rectangles A, B and C. The feasible MRH layout after applying the algorithm **PO_TREE** to its vertical and horizontal binding graphs is shown in Figure 3.2(b). Figure 3.2(c) shows another feasible layout. Obviously, the sum (a∗area(A) + b∗area(B) +c∗area(C)) of the layout in Figure 3.2(b) is always strictly larger than that of the layout in Figure 3.2(c), for any cost coefficients a,b,c > 0.



T(A) - B(A) ≥ 2, R(A) - L(B) ≥ 4
T(B) - B(B) ≥ 2, R(B) - L(B) ≥ 2
T(C) - B(C) ≥ 1, R(C) - L(C) ≥ 5

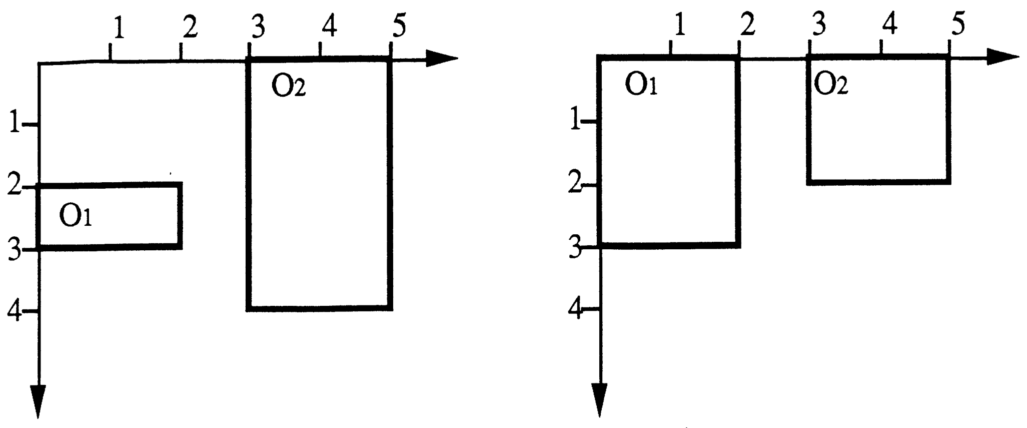
(a)



(b) (c)

Figure 3.2. Inequivalence between MRH and MOC problems

The difficulty in having a general solution to the MOC problem stems from the dependence of the objective function on the cost coefficients. Suppose there are only two rectangles O_1 and O_2, and the constraints are: $B(O_1)$-$T(O_1)\geq1$, $R(O_1)$-$L(O_1)\geq2$, $B(O_2)$-$T(O_2)\geq1$, $R(O_2)$-$L(O_1)\geq2$, $B(O_1)$-$T(O_2)\geq3$, $B(O_2)$-$T(O_1)\geq2$. Figure 3.3(a) and Figure 3.3(b) show the MOC layouts when the cost coefficients (ω_1, ω_2) are (100,1) and (1,100) respectively. Moreover, instances can be designed to show that the MOC problem does not satisfy the separation condition, thus precluding even using a linear programming method.



(ω1, ω2)=(100,1)
(a)

(ω1, ω2)=(1,100)
(b)

Figure 3.3. Dependence of MOC on the cost coefficients

Instead of searching for an exact solution to the MOC problem, we turn the attention getting a "good", but possibly sub-optimal solution. The main idea is "shrinking". To reduce the height (width) of a rectangle in a layout means to shrink its top (left) and bottom (right) sides toward each other as much as possible. Examining a layout output from the algorithm **PO_TREE**, we see that every vertex v achieves its possible maximum coordinate, i.e., the value Val(v) is maximized. Because of this maximization, no bottom (right) sides of the rectangles in a layout output from **PO_TREE** can be moved up (to the left). The top (left) sides, however, can still be possibly moved down (to the right), such as T(B) in Figure 3.2(b).

To facilitate such a top (left) side shrinking operation, the mechanics of the complement graph is needed. The complement graph of a directed graph G(V,E) is another graph, denoted as $G^c(V,E^c)$, such that there is an edge in E^c from a vertex v to a vertex w if and only if there is an edge in E from w to v. Figures 3.4(a) and (b) depict the vertical binding graph of the (S,C) shown in Figure 3.2(a) and its complement graph. Obviously, if G(V,E) is a partial order tree, so must be its complement graph $G^c(V,E^c)$. To abide with the convention that a partial order tree always has a single source vertex, a minor modification on augmenting is needed. This time, in addition to adding the vertex ROOT, a sink vertex SINK is also added such that for every sink vertex we assign a zero weight edge from it to SINK. Figures 3.5(a) and 3.5(b) show such an augmentation of the partial order tree of Figures 3.4(a) and 3.4(b). The equivalence between an original partial order tree and its augmented one is obvious.

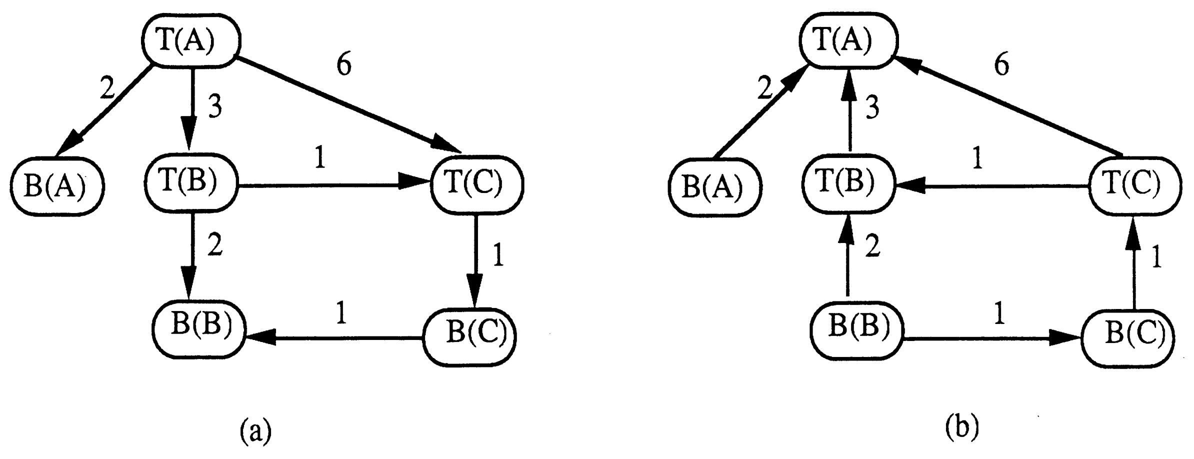


Figure 3.4. Complement of a graph

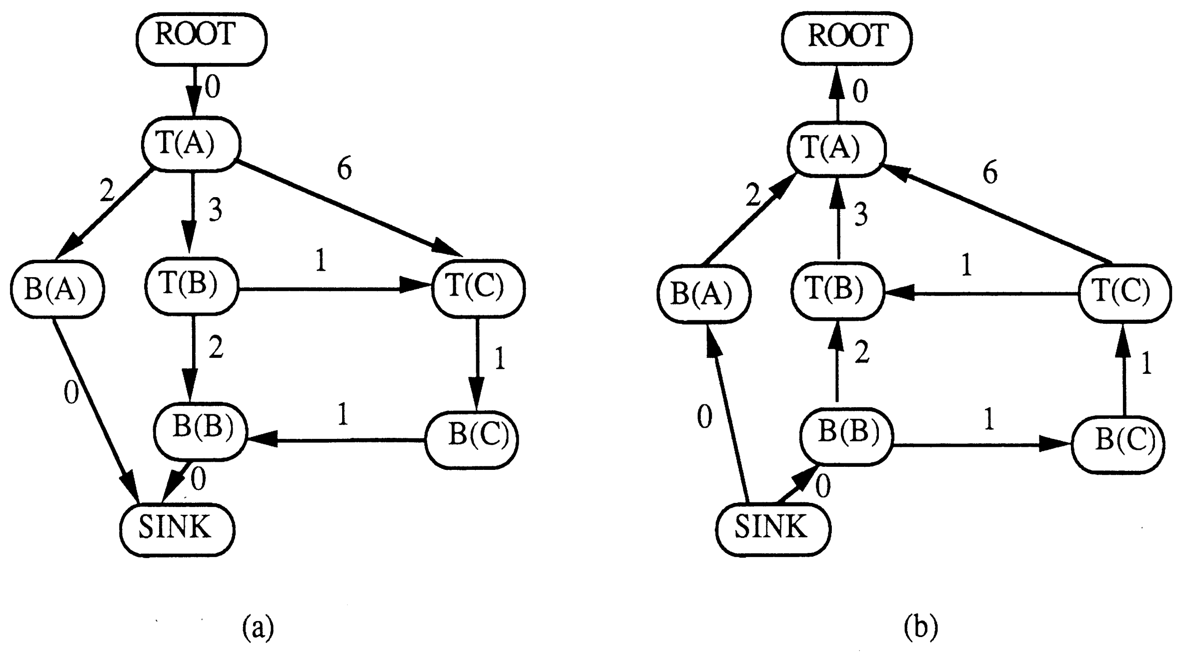


Figure 3.5. Augmenting a partial order tree and its complement

With the help of the complement graph, an algorithm for performing the shrinking becomes quite straightforward. It consists of three phases as given below:

Phase 1: On the binding graph G(V,E) (which is $(S,C)_H$ and $(S,C)_V$), perform an execution of **PO_TREE** that assigns values to the vertices in G(V,E).

Phase2: for every top (left) side vertex v in G(V,E) do

```
    Val(v) <-- $-\infty$
```

```
          end do

Phase 3:  Compute the complement graph $G^c(V, E^c)$ and call the procedure
          **SHRINK** on the SINK vertex of the $G^c(V,E)$, where **SHRINK** is
          essentially the same as **ASSIGN** except that step 2 is changed to
          "Val(w)<-- **Max**{Val(w), Val(v)+$\lambda$}".
```

The reader can easily verify that the layout in Figure 3.2(c) can be obtained by these three phases.

The behavior of this heuristic algorithm is currently being explored.

References

[AHO] A.V. Aho, J.E. Hopcroft, and J.D. Ullman: The Design and Analysis of Algorithms, Addison-Wesley, Mass., 1974.

[MEAD] C. Mead and L. Conway, Introduction to VLSI systems, Addison-Wesley, Mass, 1980.

[MURTY] K.G. Murty: Linear and Combinatoria Programming, Robert E Krieger Publication Co., Florida, 1985.

Appendix 1.

```
Algorithm MERGE (G(V,E), G'(V',E'))
/*  Merge the vertices of zero weight cycles in a binding graph G(V,E) and delete the zero
    weight cycles. The resultant binding graph is G'(V',E').
*/
begin
    step 1.     T <-- G(V,E) with its non-zero weight edges excluded

    step 2.     {$S_1$, $S_2$, ..., $S_m$} <-- the strongly connected components in T

    step 3.     for each edge e in E do
    step 3.1.           $S_i$ <--  the strongly connected component that the tail vertex of e
                                belongs
                        $S_j$ <--  the strongly connected component that the head vertex of e
                                belongs
    step 3.2.           if (the weight W of edge e is not zero) then
                                assign an edge of weight W from $S_i$ to $S_j$ and store this edge
                                data to E'
                        end if
                end do

    step 4.     return  (V'={$S_1$, $S_2$, ..., $S_m$}, E')

end MERGE
```

Lemma. Algorithm MERGE runs in $O(|V|+|E|)$ time.

Proof. Step 1 needs only $O(|V|+|E|)$ time. Step 2 also requires $O(|V|+|E|)$ time [AHO]. During step 2, we can easily, in linear time, establish a unique mapping from V to $\{S_1, S_2, ..., S_m\}$. Because of this mapping, step 3 can be done in $O(|E|)$ time.

Q.E.D.

Appendix 2.

```
Algorithm PO_TREE(G(V,E))
/* Assign numbers to the vertices of a partial order tree G(V,E) such that these numbers
   will satisfy the constraints implied by the edges of the tree.
*/
begin
   step 1.     Val(ROOT) <-- 0
               for every vertex v in V do
                      Val(v) <-- $+\infty$
               end do

   step 2.     call ASSIGN(ROOT)
end PO_TREE
```

```
procedure ASSIGN(v)
/* Assign numbers to v and all the decedents of v in a partial order tree.          */

begin
               if Count(v)=0 then
                      For each child w of v do
   step 1.                   $\lambda$ <-- the weight of the edge from v to w
   step 2.                   Val(w) <-- Min {Val(w), Val(v) - $\lambda$}
   step 3.                   Count(w) <-- Count(w) - 1
   step 4.                   call ASSIGN(w)
                      end do
               end if
end ASSIGN
```