Towards Valid Parametric CAD Models

Christoph M. Hoffmann[*] and Ku-Jin Kim[†]
Department of Computer Sciences
Purdue University

February 29, 2000

In variational CAD design, parametric models may fail to regenerate raising the question of which parameter values lead to valid models. The problem is easy to state but difficult to solve. Using a simplification, we present an algorithm that computes for a parametric model valid parameter ranges within which the model will regenerate. We explain also why the general problem is hard.

1 Introduction

Parametric CAD, originally pioneered by Parametric Technology's Pro/Engineer system more than 10 years ago, has become an accepted paradigm for all major CAD systems. Manufacturers use parametric CAD extensively in their product models, and instantiate the resulting generic designs for various parameter and constraint values. Ideally, such model variation may be used to model entire product lines using a single set of generic models.

When parametric design is used for the highly complex models of parts and assemblies, it is found that the instantiation for parameter values may fail. The types of failures are diverse. When they are bugs in the CAD system, future releases hopefully repair the situation. However, more often the failure to regenerate is related to the intrinsic nature of the chosen schema for parameterization of the shape. That is, given a particular constraint and parameter schema, certain value combinations may not define valid shapes. As a simple example, consider the L-shapes planar "solid" of Figure 1. If $d_2 > d_1$, as happens on the right, an improper solid is obtained. The failure to regenerate poses naturally the question

[*]Supported in part by ONR Contract N00014-96-1-0635, by NSF Grant CCR 99-02025, and by ARO Contract 39136-MA.

[†]Partially supported by the Purdue Visualization Center and by Korea Science and Engineering Foundation (KOSEF) Overseas Postdoctoral Fellowship Program in the second half of the year 1998.

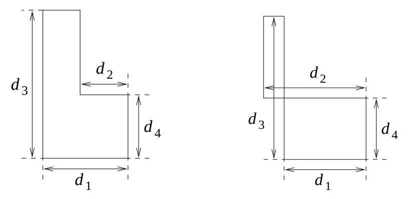


Figure 1: Left: generic L-shape with constraint schema. Right: failure to define a valid shape from chosen parameter values.

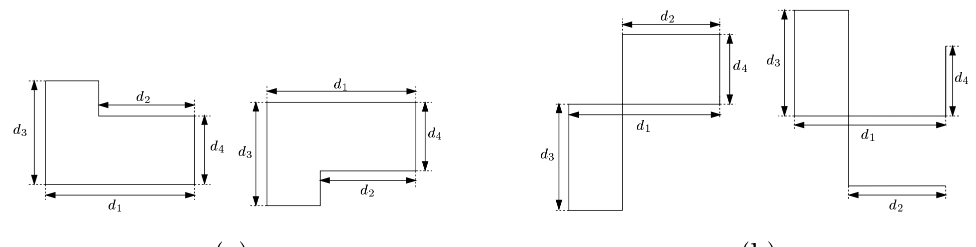
Given a parametric solid and its constraint schema, what are the valid ranges for its dimensional constraints and parameters?

This question is frequently asked by practicing CAD designers but has not been addressed based on constructive constraint range investigations. It is eminently at once a question of intense practical interest as well as of great theoretical depth.

A related problem asks for a sound theoretical definition of a *parametric family of solids*. This question is raised by Shapiro and Vossler in [9]. They show the difficulties defining a parametric family by comparing it to the difficulties maintaining the consistency between constructive solid geometry (CSG) and boundary representation (B-rep) models under the parametric and variational changes in dual system. To keep the consistency between parametric CSG and B-rep models, their approach focuses on the conversion between those two models. However, the natural parametric formulation of the boundary construction, in the sense of industrial CAD systems, coupled with the great difficulty of constructing a CSG model from a boundary representation, makes this approach unattractive. Raghothama and Shapiro [8] present a way to keep the consistency of the two representations without conversion between them. Their approach is to verify each representation's validity at every step of an update.

In [6, 7], Shapiro and Raghothama make a formal definition for a boundary solid. They define a parametric family by a continuity principle that postulates that, in a valid parameter range, small parameter changes lead to small changes of the solid's boundary, employing an algebraic topology formalism. The formulation is illuminating. However, this investigation does not help the practitioner who finds that his parametric model did not regenerate. In that situation, there is no associated CSG model, hence no answer whether the regeneration failure points to an erroneous model definition, an improper regeneration mechanism of the CAD system, or a combination of the two.

In the CAD folklore, it is recommended to adhere to a stringent design methodology which, like structured programming in programming languages, reduces the chances for failure. Clearly, design methodology may help, but cannot avoid the question altogether.



(a) (b)

Figure 2: Example of parameterization for a set of constraints: (a) valid parameterization and (b) invalid parameterization

In the general form described, the question of valid design parameter ranges is too complicated to answer satisfactorily by a constructive algorithm. We therefore narrow the question as follows:

> Given a set of geometric elements in the plane defining a closed, simple contour, and a set of constraints upon them by which it is to be parameterized, what are valid values for the parameters?

Recall that a simple contour is one that does not have self-intersections, thus defines topologically a closed disk without pinches. Moreover, we will call a set of values *valid* if the resulting contours is closed and simple. Figure 2 shows the example of valid and invalid values for parameters under the same set of constraints.

A number of factors make even this restricted, more precise problem difficult to address in this generality. To wit, such a geometric definition of shape comes about, upon solving for the specific constraint values, as solution of a nonlinear system of algebraic equations [4]. Therefore, a specific set of values defines not a unique shape, but possibly several, and the simple additional requirement to select from this set one that is valid, in the sense just defined, leads at once to an NP-complete problem [1]. So, we have to further restrict the question to be addressed in this paper as follows:

> Given a set of geometric elements in the plane defining a closed, simple contour, and a set of constraints upon them by which it is to be parameterized, and a set of values of the dimensional constraints for which a valid solution has been constructed; within which range may we change the dimension values *continuously* such that, for each value combination, a valid shape is obtained from the current one?

That is, we have restricted ourselves to variants that are obtained by continuous deformation of the current solution by the incremental change of dimensional values within certain ranges that we wish to find. We now have a more tractable problem, one that avoids, in particular, the difficulty arising from wanting to select a valid solution from a set of mathematical solutions, a set that may be exponential in size.

In this formulation, we expect that the problem will yield to a formulation using nonlinear optimization. Moreover, the machinery for solving the problem should identify a semi-algebraic set, and will be complicated, as will be a description of the set so found. Therefore, we propose to understand the variation of the dimension values as happening one at a time, so that the description of the solution reduces to a set of intervals, one for each dimensional parameter. This is the problem we investigate in this paper.

It is well known that a fully constrained contour has several geometric realization, a consequence of the nonlinearity of the underlying equations. This multiplicity imposes limitations on any constructive investigation. For example, when allowing line segments at any angle and distance constraints between any two vertices, it has been shown that the related question whether a particular set of parameter values permits a nonintersecting realization is NP-hard [3] .

In the rest of the paper, Section 2 defines the specific problem we are going to solve and explains the key issues that have to be solved. In Sections 3 and 4, we discuss the tree representation and visibility pair computations, respectively, essential tools for solving the problem efficiently. Section 5 provides the overall algorithm and Section 6 gives the examples. Section 7 shows the performance of the algorithm, and Section 8 shows how to apply this algorithm for continuous use.

2 Preliminaries

We consider the following problem:

> **Problem:** Given well-constrained rectilinear polygon P with only vertical and horizontal distance constraints, find for each constraint the range within which the distance may vary without changing the original topology of P.

We consider this problem for the vertical constraints only, since the horizontal constraints can be treated in the same way separately.

In this paper, we differentiate the topology of the rectilinear polygon as follows. See the polygon P of Figure 3(a). When we change the value of d_1 continuously, during $d_1 > d_2$, the topology of the polygon remains same with the polygon P. But, when $d_1 = d_2$ (Figure 3(b)), the topology of the polygon is different from (a). When $d_1 < d_2$, this polygon (Figure 3(c)) also has a different topology from (a) or (b).

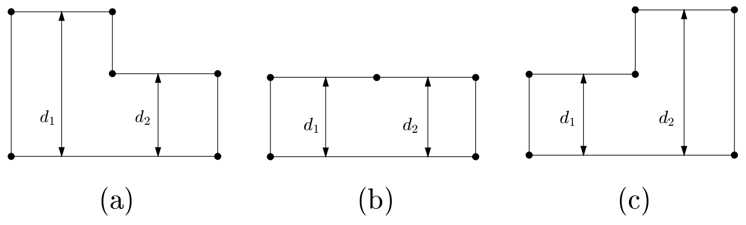


Figure 3: Three different topologies when (a) $d_1 > d_2$, (b) $d_1 = d_2$, and (c) $d_1 < d_2$

To solve above problem, we consider several issues. First, we decide whether the polygon is well-constrained or not. Figure 4(a)–(c) show examples of under-constrained, over-constrained, and well-constrained rectilinear polygons with vertical constraints, respectively. In Figure 4(a), the horizontal segments H_i and H_j are not constrained with respect to the other horizontal segments. When we add the vertical constraints v_k and v_l to it, the polygon is over-constrained (Figure 4(b)). Figure 4(c) shows a case where the polygon is well-constrained.

The second issue we consider is to decide which horizontal segments move together as a group when a particular vertical constraint is varied: When we vary a vertical constraint, the set of horizontal segments S in the polygon is split into two subsets S_0 and S_1, where $S_0 \cup S_1 = S$ and $S_0 \cap S_1 = \emptyset$, and the segments in S_0 and in S_1 move as a group, respectively. The split depends on the constraint being varied. The permissible range of the vertical constraint is governed by the possible range of motion for the sets S_0 or S_1. We have to determine which segments in the two sets define the limits of allowed motion.

Finally, we address a third issue: When we vary the constraint $v_{0,10}$ in Figure 5 and let the horizontal segment H_0 move, then H_0 can move up an arbitrary distance, but moving down is restricted by the horizontal segment H_1. In this case, the possible range for changing the present value of $v_{0,10}$ has the lower limit $H_1.y - H_0.y$ and the upper limit ∞.

In other cases the range may be finite. If we consider the vertical constraint $v_{7,10}$, the segments split into $S_0 = \{H_1, H_2, H_6, H_7, H_8\}$ and $S_1 = \{H_0, H_3, H_4, H_5, H_9, H_{10}\}$. Now the relative motion of H_1 seems to be restricted by H_0 in the positive and by H_2 in the negative direction. But H_1 and H_2 move together; thus, we have to ignore H_2 when determining limits. In fact, H_1's range of motion is restricted by H_0 and H_4. When we move the segments in the set S_0, for all five horizontal segments $\{H_1$, H_2, H_6, H_7, $H_8\}$, we have to compute ranges in this way and select their common intersection.

In this paper, we present a simple and efficient solution for the three issues we have identified. We will use a tree representation of the given rectilinear polygon and visibility pair computations.

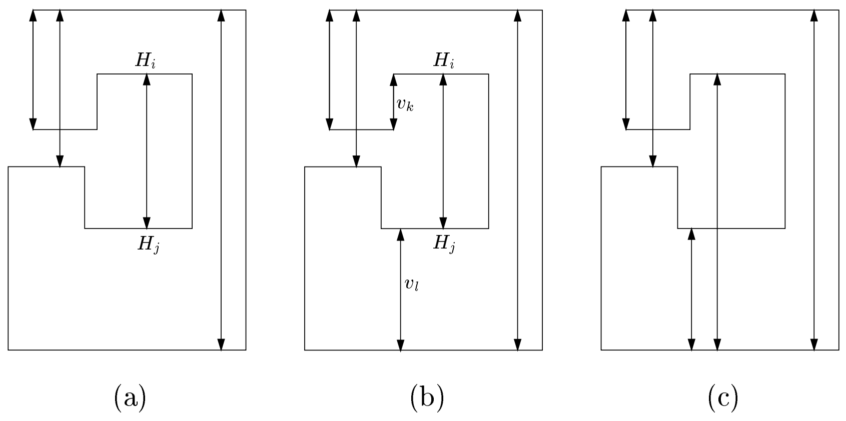


Figure 4: Under-constrained, over-constrained, and well-constrained polygons

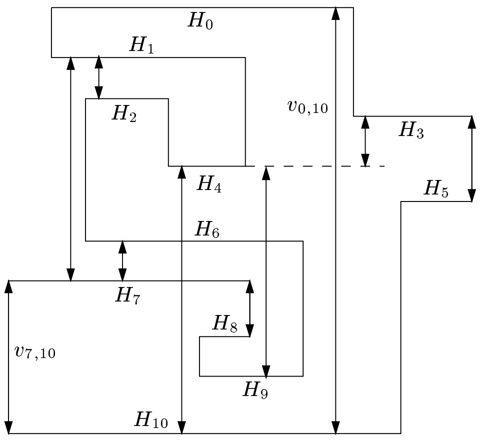


Figure 5: Example for varying one vertical constraint

3 Tree Representation

We represent the given polygon by a graph as follows; see also Figure 6:

- Each horizontal segment H_i of the rectilinear polygon corresponds to a node in the graph N_i.
- The vertical constraint $v_{i,j}$ between two horizontal segments H_i and H_j corresponds to an edge $E_{i,j}$ between two nodes N_i and N_j in the graph.

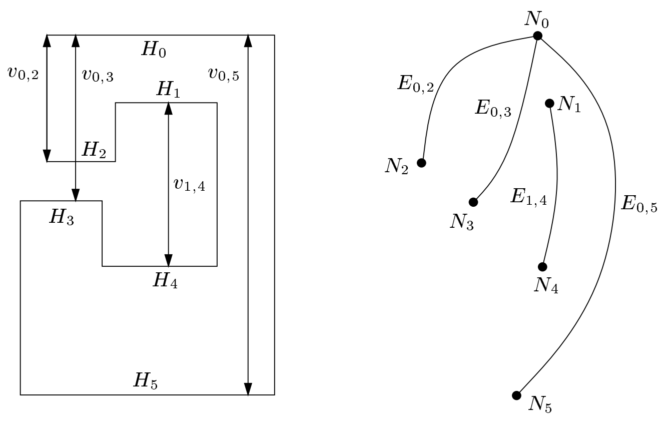
The following facts are obvious: when the polygon is under-constrained, the graph is not connected (Figure 6(a)). If the polygon is over-constrained, the graph has a cycle in it (Figure 6(b)). If and only if the rectilinear polygon is well-constrained, the corresponding graph is a free tree (Figure 6(c)). In the following, we assume that the polygon is well-constrained.

The tree representation is not only for verifying that the polygon is well-constrained, but it also facilitates deciding which horizontal segments move together. Splitting the set of horizontal segments S into S_0 and S_1 can be determined from the tree as follows.

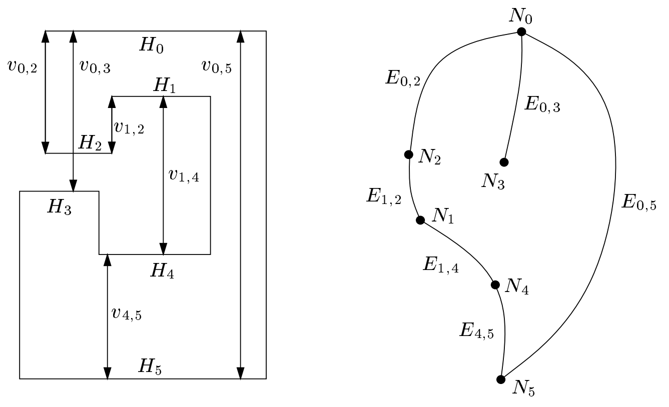
Consider varying the vertical constraint $v_{7,10}$ (Figure 7(a)). The segments H_7 and H_{10} which are connected by $v_{7,10}$ will be in the different subsets because their moving directions are opposite. After we put H_7 into the set S_0, let's repeatedly add a segment H to S_0, where H shares a vertical constraint with an arbitrary segment in S_0. We obtain $S_0 = \{H_1, H_2, H_6, H_7, H_8\}$. S_1 will be $S_1 = S \setminus S_0$. In the tree representation, we cut the edge $E_{7,10}$ corresponding to vertical constraint $v_{7,10}$ (Figure 7(b)). This splits the original tree into two subtrees T_0 and T_1 whose vertices are the sets S_0 and S_1, respectively.

After so determining the subsets, we choose which subset should move. Here, we choose the set of smaller size because moving the smaller set is more efficient than moving the larger set. In preparation for choosing the smaller set, we first identify a suitable root in the tree. The root is chosen such that a node minimizes the size of its largest subtree. This can be done by a simple pruning algorithm as follows:

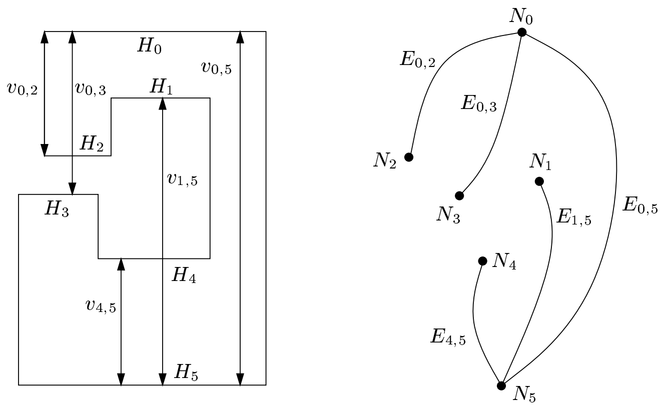
1. Assign 1 to each leaf as a value.
2. If N is a node and all but one adjacent node have a value, N is a candidate.
3. When $N_i \in$ assigned adjacent nodes of N and $v(N_i)$ is the value of N_i, select a candidate N such that $v = 1 + \Sigma_i v(N_i)$ is the minimum.
4. Remove N from the candidate set, and assign the value v to N.
5. Repeat steps 2 – 4 until all the nodes are assigned values. The node last assigned a value is the root.



(a)

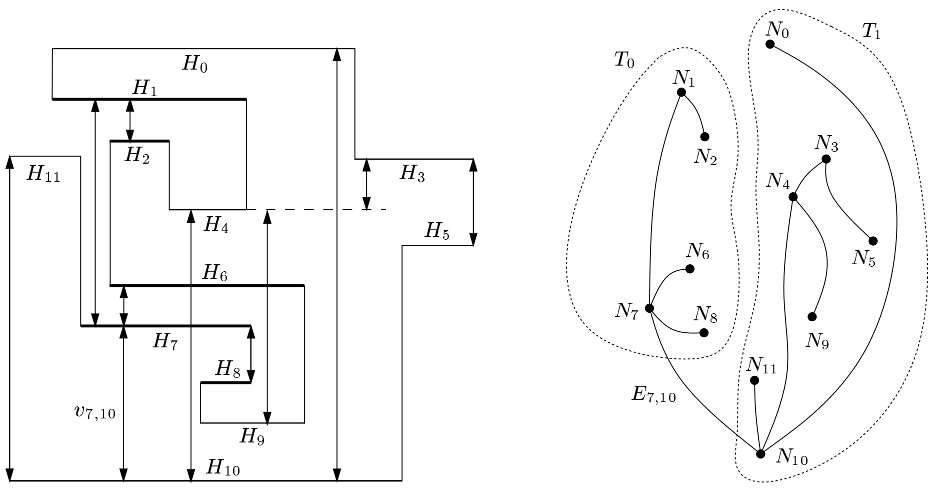


(b)



(c)

Figure 6: Example of tree representations



(a) (b)

Figure 7: Finding subset of horizontal segments to move

Before proving the validity of this algorithm, we consider the characteristic of a root which minimizes the size of maximum subtree. Let's assume that there is a tree with root R which has n children N_i, $0 \leq i < n$, where each subtree with root N_i has the size v_i and $v_j \geq v_{j+1}$ $(0 \leq j < n-1)$ (Figure 8(a)). The node R is a root which minimizes the maximum subtree size, if and only if

$$v_0 \leq 1 + \Sigma_{1 \leq i < n} v_i.$$

Notice that the value assigned to a node N_i is the size of the subtree with root node N_i.

By contradiction, we prove that the root finding algorithm always finds a root which minimizes the maximum subtree as follows. Let's assume that we applied this algorithm to a free tree, and finally we got a tree with root R and subtrees T_i, where $i = 0, \cdots, m$, with $m \geq 1$ (see Figure 8(b)). When we denote the size of the tree T as $S(T)$, without loss of generality, we assume that $S(T_0) \geq S(T_i)$, for each i, where $i = 1, \cdots, m$. When T_0 consists of a root r_0 and subtrees T_{0j}, $j = 0, \cdots, k$, let's assume that $S(T_0) > \Sigma_{i=1}^{m} S(T_i) + 1$. Then we know:

1. R was picked last, i.e. R was picked after r_0 was picked.

2. When r_0 was picked, the trees T_i, $i = 1, \cdots, m$, were assigned values already. If T_i was not assigned value yet, it implies that $S(T_0) < S(T_i)$. This contradicts the assumption.

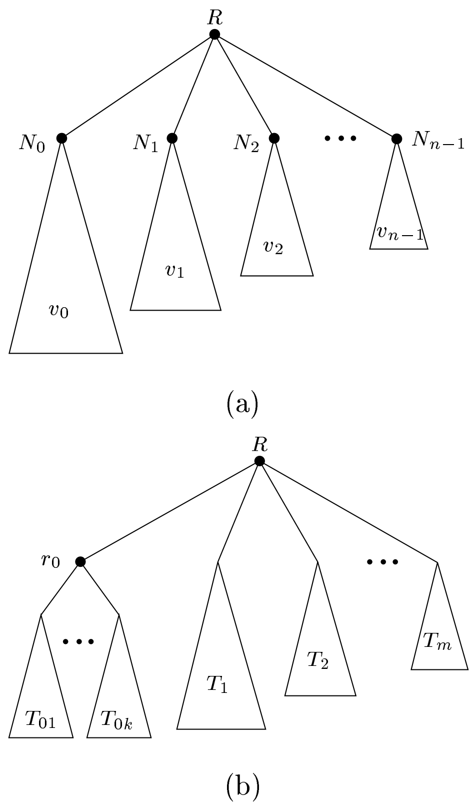


Figure 8: Proof for root finding algorithm

3. When r_0 was picked, we could have picked R instead, because trees T_1, $\cdots$, T_m were finished.

4. Since $S(T_0) > \Sigma_{i=1}^{m} S(T_i)+1$, we must pick R before r_0 and this contradicts that R was picked last.

The last step shows that the assumption $S(T_0) > \Sigma_{i=1}^{m} S(T_i) + 1$ cannot be true; thus, we conclude that the root finding algorithm is valid.

After applying this root finding algorithm, the free tree T is converted to a tree with root node R. When the edge $E_{i,j}$ (between parent node N_i and a child node N_j) is cut, there are two subtrees T_0 and T_1, where T_0 is the subtree whose root node is N_j and $T_1 = T - T_0$. Then, the size of the subtree T_0 is always smaller than or equal to the size of T_1. Assume that the size of subtree T_0 is larger than that of T_1. Then, the root R of the tree T does not minimize the maximum subtree size because if we choose N_j as the root of the tree T, the maximum subtree size is smaller than the case when R is the root. This assumption contradicts that we choose the root of the tree which minimizes the maximum subtree size, so the size of the subtree T_0 is always smaller than or equal to that of the other subtree T_1. Whenever we change the value of vertical constraint $v_{i,j}$, we consider the moving of the horizontal segments which are in the subtree with root node N_j.

In order to determine quickly whether two segments move together or not, we give two associated values (*lval*, *rval*) to each node in the tree:

$$\begin{aligned} lval &= \begin{cases} \text{postorder number}, & \text{if the node is a leaf} \\ lval \text{ of the leftmost child}, & \text{otherwise} \end{cases} \\ rval &= \text{postorder number}. \end{aligned}$$

When a node N in tree T has a node number (l_1, r_1), and a descendant node of N has a node number (l_2, r_2), the values are always $l_1 \le l_2 \le r_2 < r_1$. If there are two subtrees with root nodes numbered (l_i, r_i) and (l_j, r_j), and they don't share any common node, the values are $r_i < l_j$ or $r_j < l_i$. We denote the *lval* and *rval* of a node N by $N.lval$ and $N.rval$, respectively.

The range of motion for horizontal segments in S_0 is determined by the closest distance to other horizontal segments in S_1 that are visible in y direction. Without considering the vertical constraints, let's assume that we move only one horizontal segment H_1 (see Figure 7). Then, the possible upwards limit of H_1 is the distance between H_1 and H_0, and the possible downward limit of H_1 is the distance to H_2. When we consider the vertical constraints, the situation is different. Depending on the vertical constraint which we want to vary, there is a possibility that H_1 and H_0, or H_1 and H_2 will be in the same group to move together. If we vary $v_{1,2}$, H_1 and H_0 will move together, and if we vary $v_{1,7}$, H_1 and H_2 will move together. Let's assume that we move two segments H_1

and H_2 downwards together. When we only consider H_1, H_{11} is not the closest visible segment. But, when we move H_1 and H_2 together, H_{11} will be the closest visible segment for them.

When we move a segment H_i, we consider all the visible segments in y direction from it as the candidates which can give a restriction to H_i. So, for each horizontal segment H_i, we will compute all the visible segments from H_i in y direction, and add them to the visible segment list of H_i. The next section explains the construction of the visible segment list.

4 Visibility Pair Computation

Suppose we have a set $H = \{H_0, H_1, \cdots H_{n-1}\}$ of horizontal segments in the plane. Two segment H_i, and H_j in H are a *visibility pair* if there exists a vertical line L that intersects H_i and H_j, but L does not intersect any other segment of H between the intersections with H_i and H_j. The number of visibility pairs is at most $3n - 6$, and the optimal sequential algorithm to determine them has time complexity $O(n \log n)$ [5].

There are several algorithms for computing visibility pairs; e.g., [2, 5]. We apply the parallel algorithm of Chan and Friesen [2] in a sequential way.

After computing the visibility pairs, we construct two visibility segment lists $VSL(H)_{up}$ and $VSL(H)_{down}$ for each horizontal edge H, where $VSL(H)_{up}$ and $VSL(H)_{down}$ consist of the segments which are visible from H in positive direction and in negative direction, respectively. Each node V in the visibility segment list of H_i has the form: $(seg, dist)$, where $V.seg$ contains the segment H_j which is visible from H_i, and $V.dist$ contains the value $H_j.y - H_i.y$. We sort the nodes in $VSL(H)_{down}$ by decreasing order, and those in $VSL(H)_{up}$ by increasing order of $V.dist$. For each segment H, $VSL(H)_{up}$ and $VSL(H)_{down}$ have pointers $Up(H)$ and $Down(H)$, respectively. Initially, these pointers point to the first node of VSL.

Note that the (horizontal) visibility pairs cannot change when varying the vertical constraints within permitted ranges because we require that the polygon remain topologically unchanged.

5 Algorithm

Algorithm: ComputeRectRange(rectilinear polygon P) shows the overall procedure to compute varying ranges of vertical constraints for a given polygon P. First, we construct a free tree T from the polygon P and its constraint schema. We pick a node as a root of T such that the maximum size of its subtrees is minimized.

We assign the node number to each node in T as explained in Section 3. Then, we compute the visibility pairs, and construct the initial visible seg-

```
Algorithm: ComputeRectRange(rectilinear polygon $P$)
1.  Construct free tree $T$ for given rectilinear polygon $P$;
2.  Determine the root of $T$, $R$;
3.  Assign ($lval$, $rval$) to each node in $T$;
4.  Construct initial $VSL$ for each node in $T$;
5.  for each child $N_i$ of root node $R$ of $T$ do
        ComputeRange($N_i$);
```

Table 1: Algorithm: ComputeRectRange

ment list for each horizontal edges. For each node N in the tree, we call **ComputeRange(**N**)** which computes the variable range of each edge in the subtree with root N.

6 Example

Figure 9 shows an example of running ComputeRange on a polygon P. After applying the first three steps

1. Construction of the free tree T for given polygon P,
2. Determination of the root of T, and
3. Numbering the nodes of the tree T,

we get the tree T of Figure 9(a).

After we compute the visible segment list (VSL) for each node in the tree, we visit the tree nodes in depth-first order, and compute the range of vertical constraints. For convenience of the explanation, we show the VSL_{up} and VSL_{down} for the nodes which are examined during the computation of the range of a specific edge only. First, we compute the range of the vertical constraint $v_{1,2}$ (that is the edge between the node N_1 and N_2 in the tree T). Figure 9(b) shows the $VSL(N_2)$, which is the only list which are examined for computing the range of $v_{1,2}$. The range of $v_{1,2}$ has the upper limit 1 which is the minimum gap value in the $VSL(N_2)_{up}$ and the lower limit -2.5 which is the maximum gap value in the $VSL(N_2)_{down}$. The pointers $Up(N_2)$ and $Down(N_2)$ point to the minimum gap value of $VSL(N_2)_{up}$ and the maximum gap value of $VSL(N_2)_{down}$, respectively.

Figure 9(c) shows the status after we compute the range of $v_{7,1}$. The VSL of N_1 and N_2 are examined, and their Up and $Down$ pointers point the minimum and maximum values of the horizontal segments whose number is not covered

```
Algorithm: ComputeRange (node $N$)

if $N$ is a leaf node then begin
   $N.visit$ := TRUE;
   $E.range := (Up(N), Down(N))$,
      where $E$ is the tree edge between $N$ and its parent;
end
else begin
 for each child node $N_i$ of node $N$ do begin
    if $N_i.visit$ = FALSE then
      ComputeRange($N_i$);
 end
 for each $N_i \in \{N$ and its descendants$\}$ do begin
  while $N.lval \leq V.lval \leq V.rval \leq N.rval$ ($V$ is $Up(N_i).seg$) do
     $Up(N_i) := Up(N_i) \rightarrow next$;
  while $N.lval \leq V.lval \leq V.rval \leq N.rval$ ($V$ is $Down(N_i).seg$) do
     $Down(N_i) := Down(N_i) \rightarrow next$;
 end
 $E.range := (Min_{N_i}(Up(N_i).dist), Max_{N_i}(Down(N_i).dist))$,
     where $N_i \in \{N$ and its descendants$\}$
     and $E$ is the tree edge between $N$ and its parent;
end
```

Table 2: Algorithm: ComputeRange

Operations	Time Complexity
Free tree construction	$O(n)$
Determining the root	$O(n \log n)$
Assignment of the node numbers to the tree	$O(n)$
Construction of initial visible segment list	$O(n \log n)$
Computation of the range of vertical constraints	$O(dn)$

Table 3: Performance of the Algorithm

by the node number $N_1(0, 1)$. The upper limit of the range $v_{7,1}$ is 2 which is the minimum value from the values which *Up* pointers point to. Lower limit is -2.5 which is the maximum value from the values which *Down* pointers point to.

Figure 9(d) shows the status after computing of the range of $v_{7,6}$ and $v_{7,8}$. Figure 9(e) shows the status after computing of the range of $v_{10,7}$.

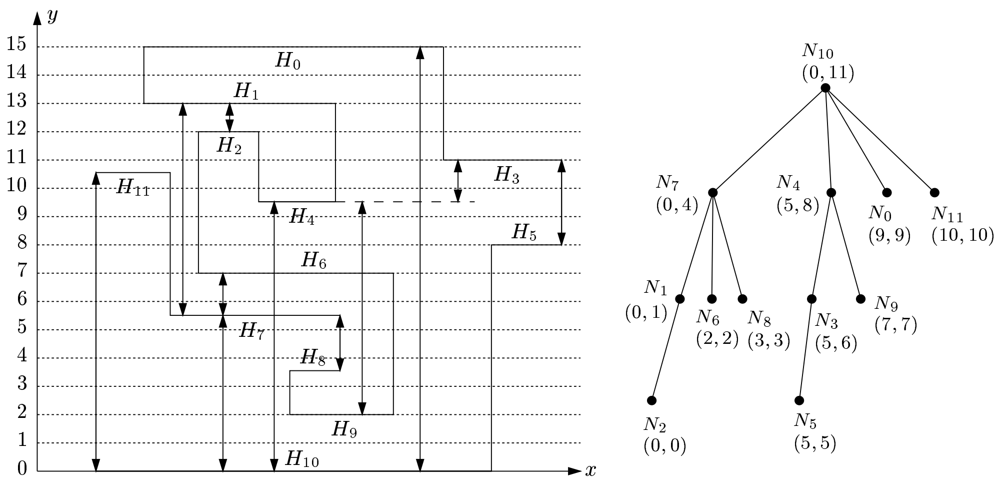
Finally, we will get the tree of Figure 9(f) which contains the ranges of all vertical constraints and visible segment lists for all nodes in the tree.

7 Performance

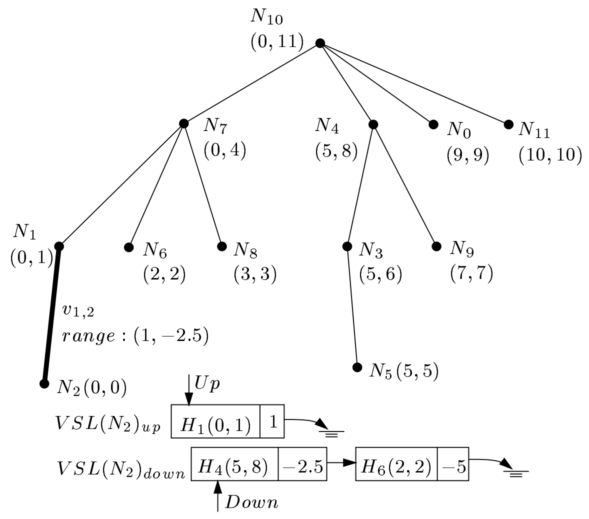
Table 3 shows the performance of the algorithm. When the polygon has n horizontal segments, the time complexity for free tree construction is $O(n)$ because there is a one-to-one correspondence between a horizontal segment (or a constraint) in the polygon and a node (or an edge) in the tree. Determination of the root of the free tree will take $O(n \log n)$ time complexity. We assign the node numbers to the rooted tree by a postorder traversal of the tree, so the time complexity for assigning the node numbers to the rooted tree is $O(n)$. When we construct the initial visible segment list, we sort all horizontal segments by y values, and then apply the divide-and-conquer method to construct the visible segment list for each segment. We need $O(n \log n)$ time for this construction, and $O(dn)$ time for computing the range of each vertical constraint, where d is the height of the tree.

8 Changing Several Vertical Constraints

Given the polygon of Figure 9(a), let's assume that a user changed the value of the vertical constraint $v_{7,1}$ as much as k, $k > 0$ $(k < 0)$. Then, the horizontal

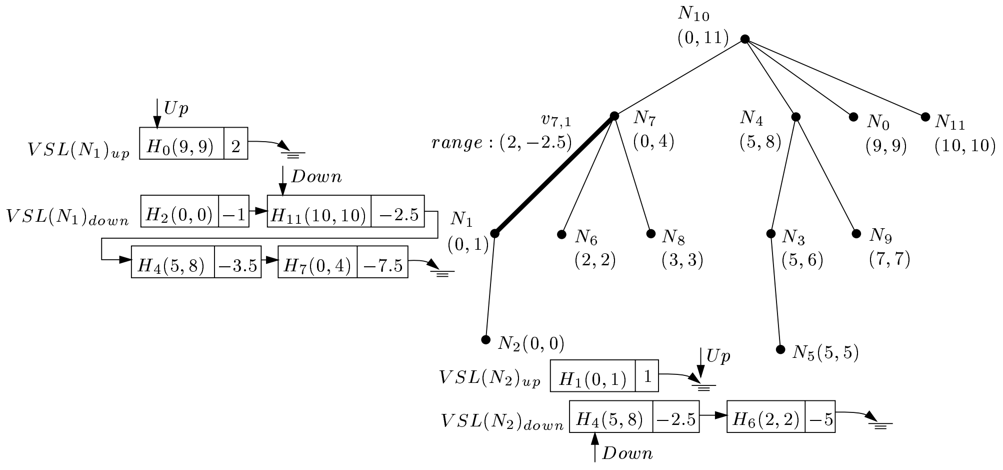


(a)

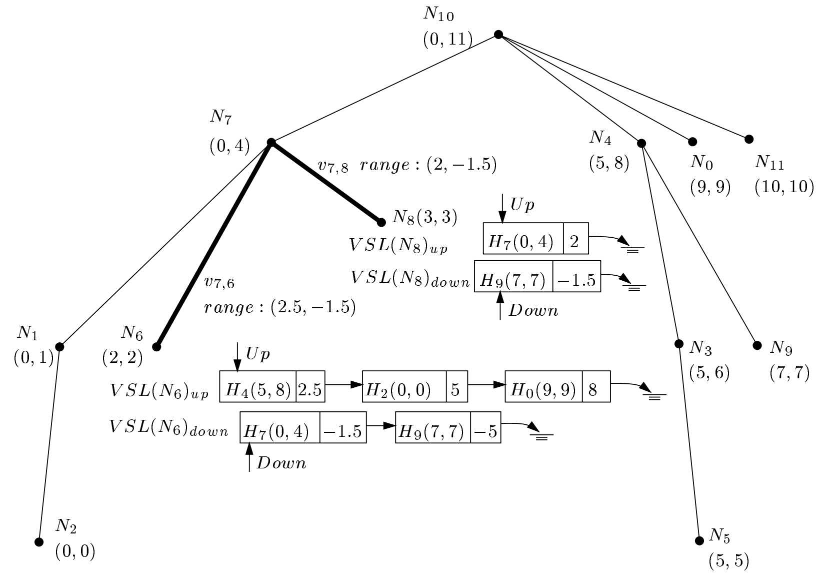


(b)

Figure 9: Example of applying the algorithm

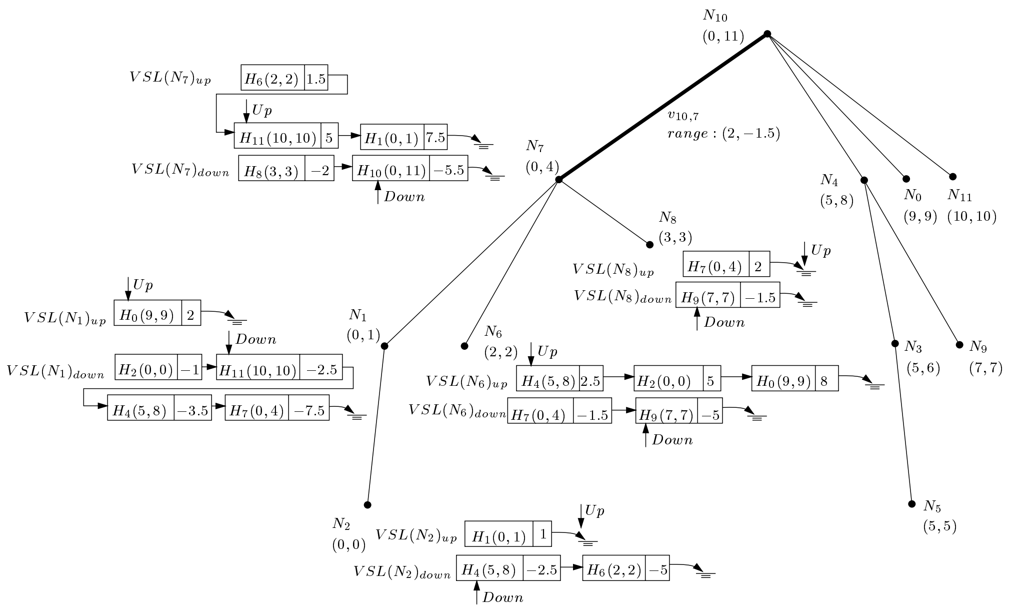


(c)



(d)

Figure 9: (Continued)


(e)

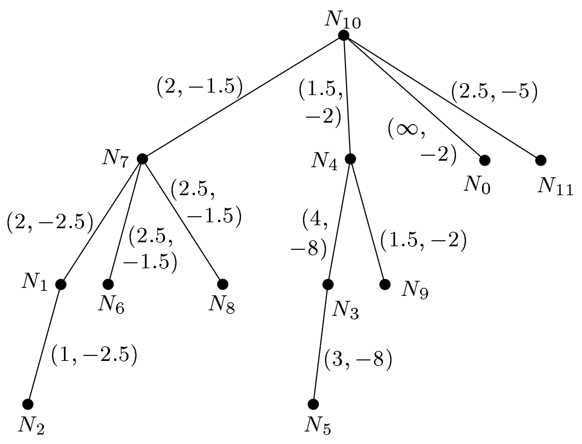

(f)

Figure 9: (Continued)

segments H_1 and H_2 will move downwards (upwards) as much as $|k|$.

The user may want to change the value of another vertical constraint also. The visible segment lists of the nodes whose corresponding horizontal segment were moved are not valid longer because the *dist* values related to the moved horizontal segments are invalid now. But, the field of *seg* in each node of visible segment lists are valid always because we require that the polygon remain topologically unchanged. We only need to update the *dist* value of the visual segment lists for the node N, where N is a node for the horizontal segments in the visible segment list of N_1 or N_2.

The modified lists should be sorted again by new *dist* values in it. We initialize the all pointers *Up* and *Down* to the first element of each list, then we can compute the valid range for each vertical constraint again.

References

[1] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *Computer Aided Design*, 27:487–501, 1995.

[2] I. W. Chan and D. K. Friesen. Parallel algorithm for segment visibility reporting. *Parallel Computing*, 19(9):973–978, September 1993.

[3] I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, April 1997.

[4] C. M. Hoffmann and R. Joan-Arinyo. Symbolic constraints in constructive geometric constraint solving. *J of Symbolic Computation*, 23:287–300, 1997.

[5] E. Lodi and L. Pagli. A VLSI solution to the vertical segment visibility problem. *IEEE Transactions on Computers*, 35:923–928, 1986.

[6] S. Raghothama and V. Shapiro. Necessary conditions for boundary representation variance. In *Proceedings of the 13th International Annual Symposium on Computational Geometry (SCG-97)*, pages 77–86, New York, June 4–6 1997. ACM Press.

[7] S. Raghothama and V. Shapiro. Boundary representation deformation in parametric solid modeling. *ACM Transactions on Graphics*, 17(4):259–286, October 1998.

[8] S. Raghothama and V. Shapiro. Consistent updates in dual representation systems. In Willem F. Bronsvoort and David C. Anderson, editors, *Proceedings of the Fifth Symposium on Solid Modeling and Applications (SSMA-99)*, pages 65–75, New York, June 9–11 1999. ACM Press.

[9] V. Shapiro and D. L. Vossler. What is a parametric family of solids? In *SMA '95: Proceedings of the Third Symposium on Solid Modeling and Applications*, pages 43–54. ACM, May 1995. held May 17-19, 1995 in Salt Lake City, Utah.