Translational polygons containment and minimal enclosure using geometric algorithms and mathematical programming

Translational Polygon Containment and Minimal Enclosure using Geometric Algorithms and Mathematical Programming

Victor J. Milenkovic
Karen Daniels

TR-25-95

November 1995

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Translational Polygon Containment and Minimal Enclosure using Geometric Algorithms and Mathematical Programming

Victor J. Milenkovic* Karen Daniels †

November 1995

Abstract

We present an algorithm for the two-dimensional translational *containment* problem: find translations for k polygons (with up to m vertices each) which place them inside a polygonal container (with n vertices) without overlapping. The polygons and container may be nonconvex. The containment algorithm consists of new algorithms for *restriction*, *evaluation*, and *subdivision* of two-dimensional configuration spaces. The restriction and evaluation algorithms both depend heavily on linear programming; hence we call our algorithm an *LP containment algorithm*. Our LP containment algorithm is distinguished from previous containment algorithms by the way in which it applies principles of mathematical programming and also by its tight coupling of the evaluation and subdivision algorithms. Our new evaluation algorithm finds a local overlap minimum. Our distance-based subdivision algorithm eliminates a "false" (local but not global) overlap minimum and all layouts near that overlap minimum, allowing the algorithm to make progress towards the global overlap minimum with each subdivision.

In our experiments on data sets from the apparel industry, our LP algorithm can solve containment for up to ten polygons in a few minutes on a desktop workstation. Its practical running time is better than our previous containment algorithms and we believe it to be superior to all previous translational containment algorithms. Its theoretical running time, however, depends on the number of local minima visited, which is $O((6kmn + k^2m^2)^{2k+1}/k!)$. To obtain a better theoretical running time, we present a modified (combinatorial) version of LP containment with a running time of

$$O\left(\frac{(6kmn + k^2m^2)^{2k}}{(k-5)!} \log kmn\right),$$

which is better than any previous combinatorial containment algorithm. For constant k, it is within a factor of $\log mn$ of the lower bound.

We generalize our configuration space containment approach to solve *minimal enclosure* problems. We give algorithms to find the minimal enclosing square and the minimal area enclosing rectangle for k translating polygons. Our LP containment algorithm and our minimal enclosure algorithms succeed by combining rather than replacing geometric techniques with linear programming. This demonstrates the manner in which linear programming can greatly increase the power of geometric algorithms.

*University of Miami, Department of Math and Computer Science. Email: vjm@cs.miami.edu. This research was funded by the Textile/Clothing Technology Corporation from funds awarded by the Alfred P. Sloan Foundation, by NSF grant CCR-91-157993, and by a subcontract of a National Textile Center grant to Auburn University, Department of Consumer Affairs.

†Harvard University, Division of Applied Sciences, and University of Miami. Email: daniels@das.harvard.edu. This research was funded by the Textile/Clothing Technology Corporation from funds awarded by the Alfred P. Sloan Foundation and by NSF grants CCR-91-157993 and CCR-90-09272.

1 Introduction.

A number of industries generate new parts by cutting them from stock material: cloth, leather (hides), sheet metal, *etc.* These industries require good solutions to *containment* or *minimal enclosure* problems. *Containment* is the question of whether a given set of part shapes can be fit, without overlapping, into a given container. *Minimal enclosure* is the problem of minimizing the size (area, length, or other measure) of the container for a given set of parts. One might think that a few good layouts might suffice for a given industry, but such is not the case. One large apparel manufacturer "solves" about 1500 minimal enclosure problems *per week*. As illustrated in Figure 1, these are *strip-packing* problems: find an enclosure with fixed width and minimum length. Human employees generate these near-optimal layouts by hand, and we estimate that their total compensation is about three million dollars per year for that one company. Large companies are highly automated, and they could easily substitute software for humans. However, no current software (even our own) can reliably find enclosures within one percent[1] of those generated by expert humans for the number of part shapes they routinely lay out (one to five hundred). (See [4, 9, 11, 34] for a survey of current results.)

Material such as cloth has a grain, and it sometimes has a "nap" (*e.g.* velvet or corduroy) or a colored pattern (*e.g.* stripes or plaid). A part cut out of such material has only one, two, four, or possibly eight allowed orientations, not arbitrary rotation. For such materials, containment can be decomposed into two subproblems: 1) selection of discrete orientations and 2) *translational* containment. *Translational containment*, containment with fixed orientations, is clearly an important subproblem. Textile manufactures use polygonal representations for their shapes because their cutting accuracy is not high enough to require spline curve representations. Therefore, it is reasonable to assume that the shapes are polygons. The focus of this paper is translational containment and translational minimal enclosure of polygonal shapes.

Containment is (of course) NP-hard (Section 1.2).[2] Because continuous rotations make containment nonlinear, it is not clear whether containment is in NP, even for polygonal shapes. Discrete orientation selection (Section 1.2) and translational polygon containment [4, 26] are both "merely" NP-complete. Techniques of combinatorial optimization such as integer or dynamic programming have been successfully applied to NP-complete problems such as the traveling salesman or the layout of rectangles [14, 1, 19]. However, nonconvex shapes appear to be too irregular to permit *direct* application of mathematical programming. Approximating individual nonconvex shapes by rectangles or even their convex hulls leads to very nonoptimal approximating enclosures or solutions to containment.

Unless P=NP, there can be no practical exact algorithm for containment or minimal enclosure when the number of parts is in the hundreds. However, it is our hope that we can eventually develop *approximation* algorithms as has been done for packing rectangles. Towards that end we have focused on developing *exact* algorithms which are practical for the largest possible number of shapes. (The algorithm presented here can handle up to ten shapes quickly.) Also, even a heuristic strategy for hundreds of shapes can benefit from a "subroutine" that can solve containment or find the minimal enclosure for smaller groups of shapes [4]. For example, a common layout strategy is to cluster irregular polygons into a rectangle[3] and then apply rectangle packing algorithms. As we stated in the previous paragraph, replacing *individual* nonconvex shapes by an enclosing rectangle results in poor packing densities. However, replacing an entire group of five to ten pieces by the minimum area rectangular enclosure (for the group under translation) yields a much higher density packing. In our work, we have noted that most applications of containment involve at least as many infeasible inputs as feasible inputs, and thus it is important that the "subroutine" detect infeasibility quickly. In the language of mathematical programming, it is just as important to establish a lower bound (no solution), as it is to establish an upper bound (find a solution).

[1] One percent is about the break-even point of labor vs. material costs.

[2] The reader will recognize that containment is the "feasibility question" for minimal enclosure, and thus the appropriate complexity question is the hardness of containment.

[3] The sides of the rectangle are parallel to the x and y axes.

1.1 New Containment Algorithm.

This paper presents a new algorithm for translational containment based entirely on the principles of mathematical programming:

Restriction : establish lower bounds through relaxation and the solution of linear programs;

Evaluation : establish upper bounds by finding potential solutions;

Subdivision : branch, when necessary, by introducing a cutting plane.

The objective that the containment algorithm minimizes is the overlap among the placed polygons. Naturally, we are seeking a layout with zero overlap. We refer to the first operation as *restriction* since we throw away any region of the solution space for which we establish a non-zero lower bound on the overlap. Hence, we *restrict* the search to regions in the solution space for which the lower bound might still be zero.

The new algorithm is called the *linear programming based containment* algorithm or *LP containment* algorithm. Strictly speaking, we should say *mathematical* programming but, in fact, all mathematical programming in the algorithm is reduced to linear programming.[4] The algorithm employs a new *LP restriction* algorithm for restriction, a new *LP local overlap minimization* algorithm for evaluation, and a new distance-based subdivision algorithm. It also uses a *geometric* restriction algorithm [5, 6, 4] on the initial solution space. The subdivision algorithm does not use linear programming. The evaluation and subdivision methods are tightly coupled. Distance-based subdivision eliminates a "false" (local but not global) overlap minimum and all layouts near that minimum, allowing the containment algorithm to make progress towards the global overlap minimum with each subdivision. We generalize our LP containment approach to find minimal enclosures for a collection of translating polygons.

In our experiments on data sets from the apparel industry, our LP algorithm can solve containment for up to ten polygons in a few minutes on a desktop workstation. Its theoretical running time, however, depends on the number of local minima visited. To improve the theoretical running time of LP containment, we also present a modified (combinatorial[5]) version of LP containment. The combinatorial version does not use geometric restriction on the initial solution space. In the evaluation step, it only searches for a local minimum on a discrete subset of the solution space. It does not use distance-based subdivision. Instead, it uses the central ideas of LP restriction to tightly integrate restriction, evaluation, and subdivision. It uses a modification of the simplex method. Its subdivision method is combinatorial.

As Section 1.5 explains, the LP containment algorithm is not the first to use restriction, evaluation, and subdivision. Nor is it the first containment algorithm to use linear programming. However, it uses new algorithms for restriction, evaluation and subdivision, and it rigorously applies the principles of mathematical programming to show that either the upper bound of the objective (overlap) is exactly zero or the lower bound is greater than zero. Hence, it gives an exact solution to the containment problem. Furthermore, it is the first algorithm that can provide an exact solution to minimum area rectangular enclosure. The practical running time of LP containment is better than our previous containment algorithms and we believe it to be superior to all previous translational containment algorithms. The combinatorial version has a better theoretical running time bound than any other combinatorial containment algorithm. LP containment is not "pure" mathematical programming, and, of course, running times depend greatly on the implementation. Nevertheless, our experiments definitely establish the value of the new mathematical programming techniques.

An implementation of our LP containment algorithm has been licensed by Gerber Garment Technologies, the largest provider of textile CAD/CAM software in the U.S., and they are incorporating it into an existing CAD/CAM software product.

The following section establishes lower bounds on containment and discrete orientation selection. Section 1.3 provides notations and definitions which permit a rigorous theoretical basis for describing and analyzing

[4] Our implementation of the LP containment algorithm uses a commercial linear programming library.

[5] The running time of a *combinatorial* algorithm depends only on the *complexity* of the input: the number of vertices, edges, lines, and so forth. The running time of a *numerical* algorithm also depends on the numerical *values* of the input coordinates and coefficients.

containment algorithms. Section 1.4 describes related work, and Section 1.5 explains how LP containment is related to our previous work on containment algorithms. Finally, Section 1.6 outlines the rest of the paper.

1.2 Lower Bounds.

In previous work, we established that *marker making* is NP-hard [28]: given 1) a set of shapes, 2) allowed orientations for each shape (rotations and reflections), and 3) a rectangle of fixed width and indeterminate length, find the nonoverlapping layout of minimum length. It was shown that *translational* marker making is NP-complete: each shape is allowed only a single orientation. The same proof works for nonrectangular containers [4, 5] to show that translational containment is NP-complete. This section establishes three new results.

1. Containment with unrestricted orientations is NP-hard.
2. Containment with discrete orientations is NP-complete.
3. *Selection* of discrete orientations is NP-complete, even if the resulting translational containment problem can be solved in polynomial time.

Result 1: Containment with unrestricted orientations is NP-hard by reduction from PARTITION. Let $a_1, a_2, \ldots, a_n$ be a list of integers. For $i = 1, 2, \ldots, n$, construct a $1 \times a_i$ rectangular shape which is permitted any orientation. Construct a container with two rectangular components, each with dimensions $1 \times (\sum a_i)/2$. Since each rectangular shape is at least 1 unit long, it can only fit into the container with no orientation change or an equivalent change (rotation by 180 degrees, for example). The rectangles can all be placed if and only if there is a way of partitioning the list into two sublists with equal sum.

Result 2: Clearly, the preceding argument also works if only discrete orientations are allowed. Containment with discrete orientations is in NP since translational containment is in NP: one can nondeterministically choose the set of orientations. Note that this argument does not work for continuous orientations because the "correct" angle for a shape could be irrational and not representable in polynomial space.

Result 3: Even by itself, discrete orientation selection is NP-hard by reduction from PARTITION. Let $a_1, a_2, \ldots, a_n$ be a list of integers. For $i = 1, 2, \ldots, n$, construct a $1 \times a_i$ rectangular shape which is permitted to rotate 90 degrees. Construct a container with two rectangular components: $1 \times (\sum a_i)/2$ and $(\sum a_i)/2 \times 1$. Selecting an angle, 0 or 90 degrees, for each shape is equivalent to partitioning $a_1, a_2, \ldots, a_n$ into two lists with equal sum. Again, since one can nondeterministically choose a set of discrete orientations, discrete orientation is in NP and therefore is NP-complete.

1.3 Notation and Definitions.

1.3.1 Minkowski Sum.

The *Minkowski sum* [30, 18, 32, 33] of two point-sets (of $\mathbf{R}^2$ in the case of this paper) is defined

$$A \oplus B = \{a + b \,|\, a \in A, b \in B\}.$$

For a point-set A, let $\overline{A}$ denote the set complement of A and define $-A = \{-a \,|\, a \in A\}$. For a vector t, define $A + t = \{a + t \,|\, a \in A\}$. Note that $A + t = A \oplus \{t\}$. Let $|A|$ denote the number of vertices of A. It is well-known that $|A \oplus B| = \Theta(|A|^2|B|^2)$ [21].

1.3.2 Configuration Spaces.

This paper presents algorithms for translating k polygonal regions $P_1, P_2, \ldots, P_k$ into a polygonal container C without overlap. If we denote $P_0 = \overline{C}$ to be the complement of the container region, then containment is

equivalent to the *placement* of $k+1$ polygons $P_0, P_1, P_2, \ldots, P_k$ in nonoverlapping positions. For translations t_i and t_j, $P_i + t_i$ and $P_j + t_j$ do not overlap if and only if $t_j - t_i \in U_{ij}$, where

$$U_{ij} = \overline{P_i \oplus -P_j}, \qquad 0 \leq i, j \leq k, i \neq j. \tag{1}$$

The set U_{ij} is the two-dimensional *configuration space* for placing P_j with respect to P_i. Clearly, $U_{ij} = -U_{ji}$. Let $\mathcal{P}$ and $\mathcal{U}$ denote the lists of all P_i and U_{ij}, respectively. A *configuration* of $\mathcal{P}$ is a list ($(k+1)$-tuple) of translations $\langle t_i \,|\, 0 \leq i \leq k \rangle$ where t_0 is arbitrarily set to $(0,0)$ since the container cannot move. The set of all configurations is the solution space of the containment problem. A *valid configuration* of $\mathcal{P}$ satisfies

$$t_j - t_i \in U_{ij}, \qquad 0 \leq i < j \leq k. \tag{2}$$

A valid configuration is an exact solution to a translational containment problem.

Within the context of this paper, a *partial configuration* is a list $\langle t_i \,|\, 0 \leq i \leq k' \rangle$ where $k' < k$. A *valid* partial configuration satisfies Equation 2 for $0 \leq i < j \leq k'$, and it corresponds to a valid placement of $P_1, P_2, P_3, \ldots, P_{k'}$ into the container.

1.3.3 Valid and Invalid Restriction.

A *restriction* of $\mathcal{U}$ replaces one or more U_{ij} in $\mathcal{U}$ by a subset of itself. A *valid restriction* does so in a way that does not eliminate any valid configurations: it only eliminates configurations with nonzero overlap. In previous work [5, 6], we show that the following direct[6] *geometric* restriction

$$U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj}), \qquad 0 \leq h, i, j \leq k, h \neq i \neq j, \tag{3}$$

is always valid. Section 2 defines a new type of valid restriction called *LP restriction*. When $\mathcal{U}$ has been restricted in any fashion, we denote the initial U_{ij} of Equation 1 by U_{ij}^{init}.

Subdivision makes use of invalid restrictions. In particular, the subdivision algorithm performs two different invalid restrictions on $\mathcal{U}$, yielding $\mathcal{U}^+$ and $\mathcal{U}^-$. Taken together, $\mathcal{U}^+$ and $\mathcal{U}^-$ must cover the same set of configurations as $\mathcal{U}$. For example, subdivision can introduce a cutting plane Π which separates the set of all configurations into two half-spaces. One half-space corresponds to $\mathcal{U}^+$, and the other to $\mathcal{U}^-$.

In this paper, the term *restriction* generally refers to valid restriction only, unless it is clear from context that it refers to both valid and invalid restrictions.

1.3.4 Size Analysis.

Our containment/enclosure algorithms and the earlier containment algorithms of Avnaim, Boissonnat, and Devillers (see Section 1.4) all use two-dimensional configuration spaces and both valid and invalid restrictions of these spaces. For any particular algorithm, the allowed restrictions are chosen from a small class of operations, although different algorithms will not necessarily use the same class. For example, our geometric containment algorithm [5, 6, 4] uses two types of operations: 1) Equation 3 and 2) intersection of some U_{ij} with a vertical or horizontal half-plane. By definition, a restriction operation reduces the size of one or more U_{ij} in $\mathcal{U}$. It *usually* but not necessarily reduces the number of vertices. Given a containment algorithm and the class of restriction operations used by that algorithm, we define the following complexity measure s: s is the largest number of vertices of any of the two-dimensional configuration spaces generated by *any* sequence of restriction operations taken from the class and applied to the initial configuration spaces generated by the algorithm. For some pathological classes of restriction operations (such as intersection with arbitrary half-planes), s is clearly infinite. With some theoretical justification [4], we believe that s is finite for the classes of restrictions used by actual containment algorithms.

Expressing the running time in terms of s is what we call *s-analysis* or *size analysis* of the algorithm. As stated in Section 1.3.1, the Minkowski sum can be fourth degree in the size (number of vertices) of the input polygons. However, except in pathological cases, the complexity is between linear and quadratic. In practice,

[6] A *direct* restriction operates on $\mathcal{U}$, not an approximation of $\mathcal{U}$.

we observe that $|U_{ij}^{\text{init}}|$ is no more than quadratic in $|P_i|$ and $|P_j|$. Subsequent restrictions of $\mathcal{U}$ may increase the number of vertices somewhat, but the number is still never more than quadratic in $\max|P_i|$. Hence, in practice, s is somewhere between linear and quadratic in the actual size of input. For this reason, size analysis yields a good measure of the practical running time of a containment algorithm.

1.3.5 Classification.

The paper focuses on the kNN problem: find a valid configuration of k nonconvex polygons in a nonconvex container. Variations replace "N" with either "C" (convex), "R" (rectangle), or "P" (parallelogram). The container C has $|C| = n$ vertices, and $|P_i| = m_i$, $1 \leq i \leq k$. We define $m = \max_{1 \leq i \leq k} m_i$. Almost always $m < n$. In general, we express the running time of a containment algorithm in terms of m, n, s, and k. For our applications: $4 \leq m \leq 100$, $100 \leq n \leq 300$, and $1 \leq k \leq 10$, (and $m \leq s \leq mn$ in practice).

1.4 Related Work.

In general, industrial systems use heuristics which cannot detect infeasibility or determine if they have found the optimal enclosure. These systems use practical efficiency techniques such as inner and outer approximations, and thus the running time increases with the tightness of the fit.[7] We have licensed our software, including the LP containment algorithm described here, and we believe it to be superior to what is currently available.[8]

The set of valid configurations for kNN containment has $\Omega((mn)^{2k})$ components, and so solving kNN requires $\Omega((mn)^{2k})$ time in the worst case [27]. It may not require this much time to generate a single solution. The "practical" algorithm described in this paper only generates a single solution. It can easily be modified to generate "all" solutions ("all" meaning at least one from each component of the solution space), but we do not claim that it would remain practical. The "theoretical" version of the algorithm has the same running time bound whether it generates a single solution or "all" solutions. It is easy to write down a "naive" kNN algorithm that generates "all" solutions in $O((mn + m^2)^{2k} \log mn)$ time, assuming "constant" k, by iterating over all choices of $2k$ contacting pairs among the polygons and container. In terms of size analysis, the running time is something like $O(s^{4k})$. On *all* infeasible inputs, the running time is at least a constant times $(mn)^{2k}$, which is thoroughly impractical, and the "naive" algorithm is impractical for feasible inputs also.

Fortune [13] gives a $O(mn \log mn)$ time solution to 1CN by computing the Minkowski sum using a generalized Voronoi diagram. Avnaim and Boissonnat [3, 2] use the Minkowski sum and convex decomposition to solve 1NN and 2NN in $O(m^2n^2 \log mn)$ and $O(m^4n^4 \log mn)$ (both $O(s \log s)$ under size analysis) time, respectively, and Avnaim [2] gives a $O(m^{14}n^6 \log mn)$ ($O(s^5 \log s)$) time algorithm for 3NN. Devillers [8] gives faster algorithms (in terms of m and n, not s) for 2CN and 3CN with running times $O(m^2n^2 \log m)$ ($O(s^2 \log s)$) and $O(m^3n^3 \log m)$ ($O(s^3 \log s)$). Avnaim and Boissonnat also give a solution to the 3NP problem, three nonconvex polygons in a *parallelogram* container, using time in $O(m^{60} \log m)$ ($O(s \log s)$) [3, 2].

In earlier work, Milenkovic, *et al.* [7] offer three approaches to translational containment. The first is fast algorithms for convex shapes: $O(mn \log n)$ (or $O(s \log s)$) for 2CN and $O(m^3n \log n)$ (or $O(m^2s \log s)$) for 3CN [6, 26]. The second approach is an MIP (mixed integer programming) model for kNN containment [23]. The MIP method takes a minute or two on a typical workstation for $k \leq 3$ polygons, but is slow for $k \geq 4$. The third approach is the aforementioned *geometric* algorithm for kNN which can find a layout with at most 2ϵ overlap using time in $O((\epsilon^{-k} \log \epsilon^{-1})k^6 s \log s)$ [4, 6, 26]. More recently, we presented a kNN algorithm, which we now call the *naive LP* algorithm, with running time

$$O\left(\frac{(2kmn + k^2m^2)^{2k+1}}{k!} \text{LP}(2k, 2kmn + k^2m^2)\right),$$

[7] Incidentally, although we do not currently do so, we also could apply these practical efficiency techniques.

[8] Companies closely guard their products, and so it is difficult to compare our algorithms to others.

where $\mathrm{LP}(a, b)$ is the time to solve a linear program with a variables and b constraints [6, 26]. A version of the naive LP algorithm can solve kCN in $\mathrm{O}((mn)^{k+1})$ time (the constant factor depends on k). Both the geometric and the naive LP algorithms can handle four or five nonconvex polygons in practice (see Tables 1 and 2, page 30).

In her Ph.D. thesis [4], Daniels presents the geometric containment algorithm. She also proposes a number of ways of improving the naive LP algorithm: using *overlap minimization* [23, 29] to improve evaluation and introducing distance-based subdivision (a different algorithm from the one presented in this paper). The thesis summarizes the LP containment algorithm described here, which uses these ideas and others, and compares its running time to the geometric containment algorithm. In a conference paper, Milenkovic proves an $\Omega((mn)^{2k})$ lower bound on the running time for kNN containment, gives an abbreviated description of the combinatorial version of the LP containment algorithm presented here, and shows that it can also solve kCN in $\mathrm{O}((mn)^k \log(mn))$ time and that it can determine minimal convex enclosures with fixed orientation edges [27]. To find the minimum area rectangle enclosing k m-gons, the running times are $\mathrm{O}(m^{k-1} \log m)$ for convex polygons and $\mathrm{O}(m^{4k-4} \log m)$ for nonconvex polygons.

1.5 Contributions of New Algorithm.

In this section we describe the contributions of the new *LP containment* algorithm and compare it to the geometric algorithm, which has running time

$$\mathrm{O}((\epsilon^{-k} \log \epsilon^{-1}) k^6 s \log s),$$

and to the naive LP algorithm, which has running time

$$\mathrm{O}\left(\frac{(2kmn + k^2m^2)^{2k+1}}{k!} \mathrm{LP}(2k, 2kmn + k^2m^2)\right).$$

In comparison, the practical running time of LP containment is much faster than the practical running time of the geometric and the naive LP algorithms. The running time of the combinatorial version of LP containment is

$$\mathrm{O}\left(\frac{(6kmn + k^2m^2)^{2k}}{(k-5)!} \log kmn\right).$$

For all these algorithms, the list $\mathcal{U}$ encodes the entire placement/containment problem, and one can think of it as the input to and current state of the containment algorithm.[9] We refer to $\mathcal{U}$ as the *hypothesized solution space* or *hypothesis*. A solution *within* the current hypothesis consists of $\langle t_0, t_1, t_2, \ldots, t_k \rangle$, where $t_0 = (0, 0)$, satisfying Equation 2 for the current list $\mathcal{U}$. Within the context of these containment algorithms, restriction, evaluation, and subdivision have the following meanings:

(Valid) Restriction Replace each U_{ij} in $\mathcal{U}$ by a subset such that if the initial hypothesis has a solution then the restricted hypothesis does also.

Evaluation Try to find a solution within the given hypothesis. If possible find a valid configuration or, failing that, find a partial configuration or find an overlapping configuration.

Subdivision Divide a hypothesis. 1) Choose some U_{ij}. 2) Partition it into U_{ij}^- and U_{ij}^+. 3) Create two subhypotheses $\mathcal{U}^-$ and $\mathcal{U}^+$ by replacing U_{ij} in $\mathcal{U}$ by either U_{ij}^- or U_{ij}^+.

Given these operations, a containment algorithm is straightforward. 0) Start with the hypothesis $\mathcal{U}$ given by Equation 1. 1) Restrict the current hypothesis. 2) If the result of restriction is not null, evaluate the restricted hypothesis. 3) If the resulting configuration is valid, output it; otherwise, subdivide the hypothesis and recurse (Step 1) on the two subhypotheses.

[9] As shown in Section 5.3.2, $\mathcal{U}$ can also encode *generalized* containment problems such as: "Place the shapes into any rectangle with area A." This is a separate contribution of this paper.

1.5.1 Geometric Containment Algorithm.

The *geometric* algorithm [5, 6, 4] performs restriction and evaluation using Equation 3. Evaluation is done in a greedy fashion: fix some t_i, restrict, and repeat until all t_i's are fixed or until some U_{ij} in the hypothesis is restricted to "null". In the latter case, the list of fixed t_i is a valid partial configuration. The algorithm subdivides by cutting the largest U_{0i}, $1 \leq i \leq k$, into two halves using a vertical or horizontal line. It stops when the largest dimension of any U_{0i}, $1 \leq i \leq k$, in $\mathcal{U}$ is smaller than $\epsilon/\sqrt{2}$.

Since it makes no use of the partial configurations it generates, the geometric algorithm can get "stuck in a rut", subdividing without actually eliminating the bad choices of greedy evaluation. Geometric restriction is often much stronger than LP restriction, particularly when the configuration spaces are highly nonconvex. For this reason, our practical implementation of the LP containment algorithm always applies geometric restriction to the initial hypothesis. In her Ph.D. thesis [4], Daniels observes that, for a test suite of 18 examples,[10] the total number of hypotheses examined by geometric containment is less than the total number of hypotheses examined by LP containment. Daniels also observes that geometric containment visits from 3 to 15 times fewer hypotheses than LP containment on infeasible problems with $k = 4$. These results demonstrate the strength of geometric restriction. Unfortunately, we have yet to implement optimal algorithms for geometric restriction. Currently, for $k = 4$, geometric restriction is about 10 times slower than LP restriction, which uses a highly optimized commercial linear programming package. Daniels also shows that it is possible to apply geometric restriction to each hypothesis visited by the LP containment algorithm without increasing the combinatorial bound on the number of hypotheses. When we implement optimal geometric restriction, we will apply *both* geometric and LP restriction to each hypothesis.

1.5.2 Naive LP Containment Algorithm.

The *naive LP* containment algorithm [6, 26] can be thought of as the simplest possible way to use linear programming to solve containment. It is a predecessor of the much more sophisticated LP containment algorithm presented here. For each hypothesis, it solves a single *constraint linear program* (CLP) which is a relaxation of Equation 2 (see Section 2.1.1). If the CLP is infeasible, it restricts the hypothesis to null, otherwise it does not restrict it at all. If the CLP is feasible, the output is either a valid or invalid configuration. If the configuration $\langle t_0, t_1, t_2, \ldots, t_k \rangle$ is invalid, it selects a pair i, j such that $t_j - t_i \notin U_{ij}$, and it subdivides U_{ij} and generates two subhypotheses.

The naive LP algorithm visits many more hypotheses than the geometric algorithm and the LP algorithm. It also has many "near misses": invalid configurations which have very little overlap. In other work, Li and Milenkovic [23, 29] developed methods to eliminate overlaps in layouts. In her Ph.D. thesis[4], Daniels proposed the idea of applying overlap elimination to each near miss of the naive LP algorithm. This combination could solve feasible problems much faster than naive LP alone. Unfortunately, applying overlap elimination in this manner is of no help in the case of infeasible inputs.

Naive LP subdivision is only loosely coupled with evaluation: it uses the result of evaluation to determine the indices i, j of the U_{ij} to be split, but it does not use the value of $t_j - t_i$ to determine the splitting line. As a consequence, it can fall into the same local overlap minimum over and over again. One of the reasons that its combinatorial running time bound is much worse than the $O((mn+m^2)^{2k} \log mn)$ "naive" algorithm (Section 1.4) is that it can visit the same set of $2k$ contacting pairs $O(kmn + k^2m^2)$ times in the worst case.

1.5.3 New LP Containment Algorithm.

The new LP containment algorithm has new methods for restriction, evaluation, and subdivision. The algorithm employs a new *LP restriction* algorithm for restriction, a new *LP local overlap minimization* algorithm for evaluation, and a new distance-based subdivision algorithm.[11] Unlike naive LP plus separate

[10] We implemented LP containment in December 1994, well in advance of this writing. Values of k range from 4 to 7. The test suite includes five infeasible examples.

[11] Our implementation of LP containment uses distance-based subdivision, introduced by Daniels [4]. In this paper we present a distance-based subdivision algorithm which improves upon the running time of the algorithm given by Daniels.

overlap minimization, the new algorithm is a true application of mathematical programming to the problem of finding the configuration with minimum overlap. In contrast to both the geometric algorithm and the naive LP algorithm, our LP containment algorithm uses the overlap minimization results of evaluation to determine the splitting line. We believe that this tight coupling of evaluation with subdivision is one of the main reasons the LP containment algorithm is faster in practice than the geometric and naive LP algorithms.

The combinatorial version of our LP algorithm uses the central ideas of LP restriction to tightly integrate restriction, evaluation, and subdivision. This tight integration guarantees that it makes progress after each subdivision. In contrast, the naive LP algorithm might require many subdivisions before it eliminates a bad configuration, and thus it visits $O(m(m+n))$ times as many hypotheses as the combinatorial LP containment algorithm.

The combinatorial running time bound matches the naive containment algorithm (for constant k); and it is a factor of $\log mn$ worse than the lower bound. Furthermore, the LP containment algorithm visits only μ local minima. Currently, we can only show that $\mu = O((6kmn + k^2m^2)^{2k+1}/k!)$, but this second bound gives us another mathematical avenue to give an output-sensitive running time bound. Intuitively, we expect μ to depend mostly on the number of planar graphs on k objects and be mostly independent of m and n.

Finally, we mention that the LP restriction is complementary to the geometric restriction of Equation 3: there are hypotheses which geometric restriction "shrinks" more and *vice versa* [4]. It is possible to use both without losing the combinatorial bounds on the number of hypotheses visited by the "pure" LP containment algorithm [4].

1.6 Outline.

Section 2 describes our new LP restriction method. The restriction intersects each U_{ij} with a convex polygon C_{ij} which is computed using $O(|C_{ij}|)$ solutions to a linear program. This section also defines and gives an efficient algorithm for combinatorial LP restriction. Combinatorial LP restriction is weak enough to permit an efficient algorithm yet strong enough to give combinatorial LP containment a near-optimal worst case running time bound.

Section 3 presents the new overlap minimization evaluation method and proves its convergence. It has an exponential worst case but requires only a constant number of linear program solutions in practice. This section also discusses evaluation techniques for the combinatorial version of LP containment.

Section 4 describes our new distance-based subdivision algorithm. It also gives the combinatorial subdivision algorithm used in the combinatorial version of LP containment. Both algorithms move the containment algorithm away from invalid (overlapping) configurations.

The final section of the paper has four parts. Section 5.1 describes our implementation of LP containment, presents running times, and compares the experimental running time of the LP containment algorithm to our previous containment algorithms. Section 5.2 proves that the running time of the combinatorial version of LP containment is

$$O\left(\frac{(6kmn + k^2m^2)^{2k}}{(k-5)!}\log kmn\right).$$

Section 5.3 shows how to extend the LP containment algorithm to solve interesting minimal enclosure problems. Section 5.4 summarize our results and draws conclusions.

2 Restriction.

The LP containment algorithm has three phases: (valid) restriction, evaluation, and subdivision. As indicated in Section 1.3.3, subdivision uses invalid restrictions. Depending on how one counts them, LP containment uses three or four types of restriction. It uses a valid *LP restriction* and two invalid restrictions: *half-plane restriction* and *component restriction*. As Section 2.2 shows, LP restriction is actually a combination of two valid restrictions: *convex hull restriction* and *combinatorial LP restriction*.

Our currently implemented LP containment uses all three types of restriction. Section 2.1 defines and describes all these restrictions. It gives an algorithm for LP restriction that makes use of linear programming. Our implementation calls a commercial linear programming library to solve the linear programs. LP restriction is iterated until it converges to a "steady state". The number of iterations is determined by a numerical, not a combinatorial, condition. Strictly speaking, the LP containment is not a combinatorial algorithm.

As often happens, practice moves ahead of theory. We have been able to prove a combinatorial running time bound for a modified (combinatorial) version of LP containment which does not use all the types of restriction used by LP containment. As shown in here and in Section 5.2, the running time bound of the combinatorial version is within a log factor of optimal. Of course, this is a worst case bound, which is why we use additional types of restriction in practice (even though they make the algorithm noncombinatorial). Section 2.2 shows that LP restriction is actually a combination of two types of restriction: *convex hull restriction* and *combinatorial LP restriction*. Individually, each of these types of restriction converges after one iteration and is combinatorial. When these two types of restrictions are alternated, we know of no combinatorial bound on the number of iterations required for convergence. The combinatorial version of LP containment uses only combinatorial LP restriction and half-plane restriction. It does not use convex hull restriction and component restriction. Section 2.2 shows that combinatorial LP restriction has a running time bound within a log factor of optimal if a specialized version of the simplex method is used to solve the linear programs.

2.1 Restrictions for LP Containment.

This section first summaries our previous algorithm for restriction/evaluation of a hypothesis $\mathcal{U}$, and then it describes how to generalize it into a new restriction method, *LP restriction*. The invalid restrictions used by distance-based subdivision are also given. LP containment uses all these types of restrictions. Our implementation solves the linear programs by calling a commercial package (see Section 5.1).

2.1.1 Naive LP Restriction and Evaluation.

The naive LP method for restriction/evaluation [6, 26] is as follows: 1) For each U_{ij} in $\mathcal{U}$, calculate its convex hull $\mathrm{CH}(U_{ij})$. 2) Set up a *containment linear program* (CLP) with $2k$ variables corresponding to the coordinates of $t_1, t_2, \ldots, t_k$ ($t_0 = (0,0)$, as always) and constraints to enforce $t_j - t_i \in \mathrm{CH}(U_{ij})$ for $0 \leq i < j \leq k$. Note that this requires $|\mathrm{CH}(U_{ij})|$ half-plane constraints for each U_{ij}. 3) If the CLP is infeasible, restrict the hypothesis to "null". Otherwise, the result of evaluation is the configuration generated by the linear program. This configuration may or may not be valid with respect to the "unrelaxed" Equation 2 (page 4).

2.1.2 New LP Restriction Algorithm.

Mathematically, the new *LP restriction* is as follows. Let $\mathcal{F}$ be the set of feasible values for the variables of the CLP. Let C_{ij} be the projection of $\mathcal{F}$ into $\mathrm{CH}(U_{ij})$ under

$$\langle t_0, t_1, t_2, \ldots, t_k \rangle \rightarrow t_j - t_i.$$

In other words, C_{ij} is the set of values for $t_j - t_i$ that permit a solution to the CLP. The new restriction replaces U_{ij} with $C_{ij} \cap U_{ij}$. This entire restriction process can be iterated until $\mathcal{U}$ stops changing ($C_{ij} = \mathrm{CH}(U_{ij})$ for all i, j). In practice, one applies restriction until the percent decrease in area is less than some threshold. Since it is not clear how many iterations are required, LP restriction is not necessarily a combinatorial algorithm.

To calculate C_{ij}, it is not necessary to compute $\mathcal{F}$ (which would require exponential time). Instead, we use the CLP as an oracle to return vertices of C_{ij}. Let v be any vector. To determine the vertex ORACLE$(\mathcal{U}, i, j, v)$ of C_{ij} that maximizes $v \cdot u$ over all $u \in C_{ij}$, simply add the following objective to the CLP: maximize $v \cdot (t_j - t_i)$.

If calling the oracle with $v = (1, 0)$, $v = (0, 1)$, $v = (-1, 0)$, and $v = (0, -1)$ yields the same point each time, then C_{ij} is a single point. Otherwise, it will yield at least two distinct vertices of an inner approximation $Q \subseteq C_{ij}$ (Q might be a degenerate "biangle"). For each edge ab of Q, let v be the outer normal vector to ab, and let $c = \text{ORACLE}(\mathcal{U}, i, j, v)$. If $c \in ab$, then ab is an edge of C_{ij}. Otherwise, replace ab of Q by ac and cb. This algorithm requires $2|C_{ij}|$ calls to the oracle: one for each vertex of C_{ij} and one for each edge. If this many calls is too costly in practice, then for any set V of query directions v, the following intersection of half-planes is a superset of C_{ij}:

$$S(V) = \bigcap_{v \in V} \{(x, y) \mid v \cdot (x, y) \leq v \cdot \text{ORACLE}(\mathcal{U}, i, j, v)\}. \tag{4}$$

Replacing U_{ij} with $S(V) \cap U_{ij}$ is also a valid (albeit less severe) restriction.

2.1.3 Invalid Restrictions.

LP restriction, as described in the previous section, is a valid restriction. Section 1.3.3 indicated that a containment algorithm also uses invalid restriction, particularly when it is performing subdivision. It applies two invalid restrictions to the hypothesis $\mathcal{U}$ to yield two subhypotheses $\mathcal{U}^-$ and $\mathcal{U}^+$. Individually, these restrictions are invalid, but taken together, $\mathcal{U}^-$ and $\mathcal{U}^+$ cover the same set of configurations as $\mathcal{U}$. Section 4.1 gives the details of the distance-based subdivision algorithm for LP containment. This subdivision uses two types of invalid restriction: *component restriction* and *half-plane restriction*.

Component restriction deletes one or more connected components from (exactly) one U_{ij} in $\mathcal{U}$. When subdivision employs component restriction, it sets U_{ij}^- equal to (the disjoint union of) some of the components of U_{ij}, and it sets U_{ij}^+ equal to the remaining components. It creates $\mathcal{U}^-$ and $\mathcal{U}^+$ by replacing U_{ij} by U_{ij}^- or U_{ij}^+. Actually, subdivision sets U_{ij}^- equal to *one* component and U_{ij}^+ equal to all the others. Section 4.1 gives the details on how this component is selected.

Half-plane restriction intersects one U_{ij} in $\mathcal{U}$ with a half-plane. When subdivision employs half-plane restriction, it chooses a line L. It sets U_{ij}^- equal to the subset of U_{ij} to the left side of L, and it sets U_{ij}^+ equal to the subset of U_{ij} to the right side of L. Section 4.1 describes how to select the line L.

2.2 Restrictions for Combinatorial Version.

The restriction algorithms themselves for half-plane restriction and component restriction are both combinatorial. Component restriction appears quite logical: why cut U_{ij} when it already consists of several pieces? However, we have not been able to prove a good theoretical running time bound for LP containment when it uses component restriction. Hence, the combinatorial version of LP containment does not use component restriction.

LP restriction is not necessarily combinatorial because it is not clear how many iterations are required before it converges. This section separates LP restriction into *convex hull restriction* and *combinatorial LP restriction*. Doing this requires a change of notation. By themselves, both of these restrictions are combinatorial and converge in one iteration. The combinatorial version of LP containment uses only combinatorial LP restriction, not convex hull restriction.

Section 2.2.1 gives a new notation for representing a hypothesis. Under this notation, LP restriction can be separated into *convex hull restriction* and *combinatorial LP restriction*. Section 2.2.2 gives a version of the simplex algorithm that is applicable to a CLP and shows how to use this algorithm as part of an algorithm for combinatorial LP restriction. The simplex algorithm uses only logarithmic time per vertex of $\mathcal{F}$, but a vertex may appear in many different hypotheses. Section 2.2.3 shows how to restrict a hypothesis using only logarithmic time per *new* vertex of $\mathcal{F}$. Clearly, this is within a log factor of optimal. This algorithm for combinatorial LP restriction is an important part of the combinatorial version of LP containment.

2.2.1 New Notation.

If the only types of restriction we use are LP restriction and half-plane restriction, then each U_{ij} will be the intersection of a convex region with U_{ij}^{init} (Section 1.3.3). We will call this polygon C_{ij}:

$$U_{ij} = C_{ij} \cap U_{ij}^{\text{init}}, \qquad 0 \leq i, j \leq k.$$

Component restriction does not have this property because the components of U_{ij} might be nonconvex and not separable by convex separating polygons.

In the modified notation, the list $\mathcal{C} = \langle C_{ij} \,|\, 0 \leq i, j \leq k, i \neq j \rangle$ acts as the hypothesis instead of the list of U_{ij} regions. Whenever we need the value of U_{ij}, we intersect C_{ij} with U_{ij}^{init}. LP restriction and half-plane restriction act on the C_{ij} regions in $\mathcal{C}$, replacing one or more by subsets.

Under the modified notation, LP restriction is a combination of *convex hull restriction* and *combinatorial LP restriction*. *Convex hull restriction* replaces C_{gh} by the convex hull of U_{gh}:

$$C_{gh} \leftarrow \text{CH}(C_{gh} \cap U_{gh}^{\text{init}}).$$

Combinatorial LP restriction acts in the same way on C_{gh}, $0 \leq g < h \leq k$, as ordinary LP restriction acts on $\text{CH}(U_{gh})$. Each C_{gh} is replaced by the range of $t_h - t_g$ over all configurations $\langle t_0, t_1, t_2, \ldots, t_k \rangle$ which satisfy $t_j - t_i \in C_{ij}$, $0 \leq i < j \leq k$.

Under this new notation and these definitions, one "iteration" of LP restriction is equivalent to a single application of convex hull restriction followed by a single application of combinatorial LP restriction. Note that if we do not apply convex hull restriction, then combinatorial LP restriction converges in one "iteration": after restriction, C_{gh} is the full range of $t_g - t_h$. Therefore, the constraint $t_g - t_h \in C_{gh}$ is no more restrictive after C_{gh} is restricted than before C_{gh} is restricted.

2.2.2 Simplex Algorithm.

This section gives an algorithm for employing the simplex method to solve the CLP and to apply combinatorial LP restriction to a hypothesis $\mathcal{C}$. Let $\mathcal{F} \subseteq \mathbf{R}^{2k+2}$ be the set of feasible configurations with respect to the CLP:

$$\mathcal{F} = \{ \langle t_0, t_1, t_2, \ldots, t_k \rangle \,|\, t_0 = (0,0) \text{ and } t_j - t_i \in C_{ij}, 0 \leq i < j \leq k \}.$$

Note that even though $\mathcal{F}$ is embedded in $\mathbf{R}^{2k+2}$, it is actually only $2k$-dimensional because t_0 is set equal to $(0,0)$. This section first shows how to step from one vertex of $\mathcal{F}$ to all neighboring vertices using time logarithmic in

$$|\mathcal{C}| = \sum_{0 \leq i < j \leq k} |C_{ij}|.$$

This "simplex step" algorithm is then used to apply combinatorial LP restriction to $\mathcal{C}$.

A vertex $f = \langle t_0, t_1, t_2, \ldots, t_k \rangle$ of $\mathcal{F}$ is the intersection of $2k$ faces. Each face corresponds to (the boundary of) one of the (half-space) constraints of the CLP. We can represent a constraint as a triple $\langle i, j, e \rangle$, meaning $t_j - t_i$ lies to the left of the line $\text{LINE}(e)$ containing edge e of C_{ij} ("left" means on the same side as the interior of C_{ij}). The constraint $\langle i, j, e \rangle$ is a half-plane constraint in the 2D "space" of C_{ij}, and it is a half-space constraint in $\mathbf{R}^{2k+2}$. The half-plane in 2D (half-space in $\mathbf{R}^{2k+2}$) is bounded by the line (hyper-plane) $t_j - t_i \in \text{LINE}(e)$. The face itself is the set of configurations which "project" onto e: $t_j - t_i \in e$. Hence, a vertex of $\mathcal{F}$ projects onto $2k$ edges and lies on $2k$ hyper-planes corresponding to these edges. The constraints corresponding to these edges/hyper-planes are the *critical constraints* of that vertex.

Stepping from vertex $f = \langle t_0, t_1, t_2, \ldots, t_k \rangle$ to a neighboring vertex $f' = \langle t'_0, t'_1, t'_2, \ldots, t'_k \rangle$ of $\mathcal{F}$ is equivalent to replacing exactly one of the critical constraints $\langle i, j, e \rangle$ by a different constraint $\langle i', j', e' \rangle$. For each critical constraint $\langle i, j, e \rangle$ of f, there is a unique constraint $\langle i', j', e' \rangle$ which can replace it. This replacement constraint corresponds to the unique face which touches vertex f' but does not touch vertex f.

Lemma 2.1 *Given a vertex f of $\mathcal{F}$ and a critical constraint $\langle i, j, e \rangle$ of f, the replacement constraint $\langle i', j', e' \rangle$ corresponding to a neighboring vertex f' can be computed in $O(k^3 + k^2 \log |\mathcal{C}|)$ time.*

Proof: Eliminating $\langle i, j, e\rangle$ leaves $2k-1$ constraints. Using standard techniques of linear algebra, the $2k-1$ corresponding hyper-planes (plus the constraint $t_0 = (0,0)$) can be intersected in $O(k^3)$ time. The intersection is a line in $\mathbf{R}^{2k+2}$. We can parameterize this line by s as $\tau(s) = \langle t_0(s), t_1(s), t_2(s), \ldots, t_k(s)\rangle$, where $\tau(0) = f$ and where $t_j(s) - t_i(s)$ is to the left of LINE(e) for $s > 0$.

For $0 \le g < h \le k$, we intersect the ray $t_h(s) - t_g(s)$, $s > 0$, with the boundary of C_{gh}. Using standard techniques of computational geometry, this can be done in $O(\log |C_{gh}|)$ time. Let i', j' be the pair such that $t_{j'}(s) - t_{i'}(s)$ exits $C_{i'j'}$ with the minimum value $s_{\min}$ of s, and let e' be the edge through which the ray exits $C_{i'j'}$. Because $s_{\min}$ is the minimum "exit parameter", $t_h(s_{\min}) - t_g(s_{\min}) \in C_{gh}$ for all other pairs g, h. Therefore, $f' = \tau(s_{\min})$ is feasible; it projects onto edge e' of $C_{i'j'}$; and it projects onto the $2k-1$ edges corresponding to the constraints we did not eliminate.

Total time is $O(k^3)$ plus the sum of $O(\log |C_{gh}|)$ over all pairs g, h. Since $|C_{gh}| < |\mathcal{C}|$, this gives the bound claimed in the lemma. ■

Lemma 2.2 *Given a feasible starting configuration, each C_{ij} in $\mathcal{C}$ can be LP restricted using total time $O(|\mathcal{F}|(k^3 + k^2 \log |\mathcal{C}| + k^2 \log |\mathcal{F}|))$.*

Proof: In the following proof, C_{ij} denotes the unrestricted region and $C^\star_{ij}$ denotes the restricted region. The algorithm is partitioned into several **tasks**.

I. Determining the critical constraints. The feasible starting configuration $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ projects into each C_{ij}. For $0 \le i < j \le k$, we can determine on which, if any, edges of C_{ij}, $t_j - t_i$ lies. Each such edge corresponds to a critical constraint. Since C_{ij} is convex, each edge can be determined in $O(\log |C_{ij}|)$ time using standard techniques of computational geometry.

II. Finding a vertex of $\mathcal{F}$. In case $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ does not project onto $2k$ edges, we need to add more constraints. This can be done using a simple modification of the "simplex step" algorithm of Lemma 2.1. Since there are less than $2k$ constraints, a "feasible" ray $\tau(s) = \langle t_0(s), t_1(s), t_2(s), \ldots, t_k(s)\rangle$, $s > 0$, can be chosen. If there are less than $2k-1$ constraints, this ray is not unique, and so we choose one arbitrarily. Standard techniques of linear algebra accomplish this in $O(k^3)$ time. We do not remove a constraint, but we add the constraint corresponding to the minimum-s "exit edge". This modified simplex step is repeated until there are $2k$ constraints. Note: we only have to repeat it at most $2k$ times.

III. Does vertex f of $\mathcal{F}$ project onto a vertex or edge of $C^\star_{ij}$? For a vertex $f = \langle t_0, t_1, t_2, \ldots, t_k\rangle$ of a convex $2k$-dimensional polytope, determining the $2k$ directions of the edges out of f is equivalent to inverting the $2k \times 2k$ matrix of normal vectors to the $2k$ faces which meet at f. This can be done in $O(k^3)$ time. The projection of the $2k$ edge vectors yields $2k$ vectors in C_{ij} pointing out of $t_j - t_i$, the projection of f. The projection $t_j - t_i$ lies on the boundary of $C^\star_{ij}$ if and only if there is a line L through $t_j - t_i$ such that all the projected vectors lie on or to the left of L. If L can be chosen such that the heads of the vectors all lie strictly to the left of L, then $t_j - t_i$ is a vertex. Consider the two vectors out of $t_j - t_i$ which form the largest angle between them. These point along the two edges of $C^\star_{ij}$ which meet at $t_j - t_i$. If $t_j - t_i$ lies on an edge e of $C^\star_{ij}$, then L will equal LINE(e), and two of the vectors will point along e. The cost of testing this half-plane condition and finding the largest-angle pair of vectors is $O(k)$. Therefore, the total cost over all C_{ij}, $0 \le i < j \le k$, is $O(k^3)$.

IV. Finding a vertex of $C^\star_{ij}$. As each C_{gh}, $0 \le g < h \le k$ is restricted, the algorithm visits new vertices of $\mathcal{F}$. Each new vertex is tested to see if it projects onto a vertex or edge of any $C^\star_{ij}$. Unfortunately, when it comes time to restrict a particular C_{ij}, the algorithm might not yet have seen a point on the boundary of $C^\star_{ij}$. In this case, out of the vertices of $\mathcal{F}$ it has seen, it chooses the vertex f of $\mathcal{F}$ which projects to the *rightmost* point in C_{ij}: $(1,0) \cdot (t_j - t_i)$ is maximal. It then applies the standard simplex technique to find the unseen vertex of $\mathcal{F}$ which project to the rightmost point in C_{ij}: 1) find an edge direction out of f whose projection into C_{ij} has positive x-component 2) move to the vertex f' of $\mathcal{F}$ at the other end of that edge. These two steps are repeated until the rightmost vertex, which will be a vertex of $C^\star_{ij}$, is found. Note: since the algorithm starts at the rightmost "seen" vertex, each vertex of $\mathcal{F}$ it generates has never been seen before.

V. Given a point v on the boundary of $C^\star_{ij}$, find the two neighboring vertices on the boundary of $C^\star_{ij}$. One way or another, v is generated as the projection of a vertex f of $\mathcal{F}$, and we know the two edge directions out of f whose projections point out of v and along the boundary of $C^\star_{ij}$. Pick either of these edge

directions and find the vertex f' at the other end. Its projection into C_{ij} will be a vertex v' of $C^\star_{ij}$ that is a neighbor of v on the boundary of $C^\star_{ij}$.

It is important not to generate each vertex of $\mathcal{F}$ more than once. When it comes time to construct $C^\star_{ij}$, the algorithm looks into C_{ij} and sees what vertices of $C^\star_{ij}$ have already been generated. If there are none, it has to generate one (the rightmost) using Algorithm IV. It computes the convex hull of the existing vertices. Since it knows the directions of the edges out of each vertex, it knows whether two of these vertices are actually neighbors. If $C^\star_{ij}$ is missing vertices, it uses Algorithm V to fill in the missing vertices.

The running time is $O(k^3 + k^2 \log |\mathcal{C}|)$ to generate each vertex of $\mathcal{F}$ and $O(k^3)$ to check if it projects onto the boundary of any $C^\star_{ij}$. Computing the convex hull of existing vertices of $C^\star_{ij}$ adds a cost of $O(\log |C^\star_{ij}|)$ per vertex, and since the vertices of $C^\star_{ij}$ are projections of vertices of $\mathcal{F}$, this cost is $O(\log |\mathcal{F}|)$. Since each vertex of $\mathcal{F}$ can project onto a vertex of $O(k^2)$ regions, the extra cost per vertex is $O(k^2 \log |\mathcal{F}|)$. ■

2.2.3 Reducing the Cost per Vertex.

The simplex algorithm of the previous section reduced the time for restriction to

$$O(k^3 + k^2 \log |\mathcal{C}| + k^2 \log |\mathcal{F}|)$$

per vertex of $\mathcal{F}$. Section 5.2 shows that this bound is $O(k^3 \log kmn)$. Unfortunately, the same configuration may appear as a vertex of the feasible region of a different hypothesis. This means that we may have to spend $O(k^3 \log kmn)$ many times per vertex. This section shows how to restrict $\mathcal{C}$ using logarithmic cost per *new* vertex that is generated. This algorithm is necessary to obtain our near optimal bound for the combinatorial version of LP containment.

Recall that the combinatorial version of LP containment applies only half-plane restrictions and combinatorial LP restriction. It applies neither component restriction nor convex hull restriction. Recall also that, in this version, restriction is tightly integrated with combinatorial subdivision. To accomplish combinatorial subdivision, the algorithm picks some pair i, j and a line L, and it splits C_{ij} into C^-_{ij} and C^+_{ij} (see Section 4.2). The combinatorial version of LP containment recurses on the two subhypotheses $\mathcal{C}^-$ and $\mathcal{C}^+$ which result from replacing C_{ij} in $\mathcal{C}$ by either C^-_{ij} or C^+_{ij}. When it recurses on $\mathcal{C}^-$, the first thing it does is apply combinatorial LP restriction. This replaces C_{gh} by the new range of $t_h - t_g$ given that C_{ij} has been replaced by C^-_{ij}. Let us denote this restriction of C_{gh} as C^-_{gh}. Similarly, when the algorithm recurses on $\mathcal{C}^+$, let the restriction of C_{gh} be C^+_{gh}. Note that this notation is unambiguous with respect to C_{ij}: with respect to the constraints $t_h - t_g \in C_{gh}$, $0 \le g < h \le k$ and $g \ne i$ or $h \ne j$, the range of $t_j - t_i$ is unaffected. Therefore, C^-_{ij} and C^+_{ij} are unaffected by combinatorial LP restriction.

To construct C^-_{gh}, the running time of the simplex LP restriction algorithm of the previous section is, for each vertex:

$$O(k^3 + k^2 \log(|\mathcal{C}| + 1) + k^2 \log |\mathcal{F}^-|).$$

(We add 1 to $|\mathcal{C}|$ because C^-_{ij} might have one more edge than C_{ij}.) However, C^-_{gh} may have many vertices in common with C_{gh} which the simplex algorithm generates a second time. Therefore, the simplex LP restriction algorithm may be far from optimal.

Define C^0_{ij} to be $L \cap C_{ij}$. Since L is a line and C_{ij} is a convex region, C^0_{ij} is a line segment. Define C^0_{gh} to be the result of restricting C_{gh} after C_{ij} has been replaced by C^0_{ij}. As before, this restriction does not shrink C^0_{ij} any further, and so this notation is unambiguous. Let $\mathcal{F}^0$ denote the feasible configurations of the hypothesis $\mathcal{C}^0$ which result from replacing C_{ij} by C^0_{ij} in $\mathcal{C}$. If Π denotes the plane in $\mathbf{R}^{2k}$ corresponding to the constraint $t_j - t_i \in L$, then $\mathcal{F}^0 = \Pi \cap \mathcal{F}$. Each vertex of $\mathcal{F}^0$ is *new*.[12] The following lemma indicates that it is possible to restrict $\mathcal{C}^-$ and $\mathcal{C}^+$ using log time per vertex of $\mathcal{F}^0$.

Lemma 2.3 *Hypotheses $\mathcal{C}^-$ and $\mathcal{C}^+$ can be restricted in time* $O(|\mathcal{F}^0|(k^3 + k^2 \log(|\mathcal{C}|+1) + k^2 \log(|\mathcal{F}| + |\mathcal{F}^0|)))$.

[12] Actually, if Π passes through a vertex of $\mathcal{F}$, then some of the vertices of $\mathcal{F}^0$ might already belong to $\mathcal{F}$. For the sake of simplicity, we assume that L is in *general position* and thus Π does not pass through any vertex of $\mathcal{F}$. Standard techniques show that it suffices to prove a running time bound under the assumption of general position.

Proof: We describe how to construct C^-_{gh}. Constructing C^+_{gh} is analogous.

We apply the simplex algorithm of Lemma 2.1 to restrict $\mathcal{C}^0$. To apply this lemma, it is necessary to generate an initial feasible configuration for $\mathcal{C}^0$. This can be done in $O(k + \log|C_{ij}|)$ time, where C_{ij} is the hypothesis that is split by line L. Using standard techniques, determine an edge e of C_{ij} which L intersects. Since C_{ij} is restricted, each of its vertices corresponds to a vertex of $\mathcal{F}$. The endpoints of e will correspond to vertices on opposite sides of hyper-plane Π. A simple linear interpolation generates a feasible configuration which lies on Π and hence in $\mathcal{F}^0$. The rest of the proof shows how to restrict C_{gh} to C^-_{gh} given C^0_{gh} in the restricted hypothesis $\mathcal{C}^0$.

Since $\mathcal{C}^0$ is a restriction of $\mathcal{C}$, it follows that $C^0_{gh} \subseteq C_{gh}$. For each vertex a of C^0_{gh}, determine if a lies on the boundary of C_{gh}, and if it does, split the boundary of C_{gh} at a. Assuming the vertices of C_{gh} are stored in a balanced search structure, then standard techniques of computational geometry can perform these operations in $O(\log|C_{gh}|)$ time per vertex of C^0_{gh}.

Let a and b be two vertices of C^0_{gh} that lie on the boundary of C_{gh} such that no vertex of C^0_{gh} between a and b also lies on the boundary of C_{gh}. Both C^0_{gh} and C_{gh} have an "arc" joining a to b. Vertices a and b will also lie on the boundary of C^-_{gh}, and the arc ab of C^-_{gh} will either be the arc ab of C^0_{gh} or the arc ab of C_{gh}. To determine which, consider any vertex v of arc ab of C_{gh}. Vertex v equals $t_h - t_g$ for some feasible (for $\mathcal{C}$) configuration $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ which the LP containment algorithm generated when it restricted $\mathcal{C}$. If $t_j - t_i$ lies to the left of L, then the arc ab of C_{gh} belongs to C^-_{gh}. If $t_j - t_i$ lies to the right of L, then the arc ab of C^0_{gh} belongs to C^-_{gh}. Incidentally, if *no* vertices of C^0_{gh} lie on the boundary of C_{gh}, then one can test any vertex v of C_{gh}. If $t_j - t_i$ lies to the left of L, then $C^-_{gh} = C_{gh}$. If $t_j - t_i$ lies to the left of L, then $C^-_{gh} = C^0_{gh}$.

Clearly, there are $O(|C^0_{gh}|)$ pairs of arcs to be considered, and determining the correct arc from each pair requires $O(|C^0_{gh}|)$ time. However, the vertices of C^-_{gh} need to be in the same sort of balanced search structure that was used to store the vertices of C_{gh}. For the appropriate structure, the time to join $O(|C^0_{gh}|)$ arcs is $O(|C^0_{gh}|\log|C^0_{gh} + C_{gh}|)$. Putting together this time with the time to generate C^0_{gh} gives the time bound claimed in the lemma. ■

3 Evaluation.

A solution $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ to a CLP is a potential solution to containment, but it might not satisfy Equation 2 (page 4) because $t_j - t_i \in \mathrm{CH}(U_{ij})$ does not imply $t_j - t_i \in U_{ij}$. If $t_j - t_i \notin U_{ij}$, translated polygons $P_i + t_i$ and $P_j + t_j$ overlap. We define the *amount of overlap* of $P_i + t_i$ and $P_j + t_j$ to be the shortest distance from $t_j - t_i$ to the boundary of U_{ij}. For any given configuration $\langle t_0, t_1, t_2, \ldots, t_k\rangle$, the *overlap* of the configuration is the maximum amount of overlap over all pairs of polygons.

This section gives an algorithm that calculates a *local overlap minimum*: a configuration $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ such that the amount of overlap is at a local minimum. In particular, every configuration $\langle t'_0, t'_1, t'_2, \ldots, t'_k\rangle$ in some open ball about the minimum $\langle t_0, t_1, t_2, \ldots, t_k\rangle$ has larger overlap. Naturally, one would prefer the *global* overlap minimum because that would be a nonoverlapping configuration (if one exists). Unfortunately, finding the global minimum is NP-hard. Still, a local minimum is more likely, at least in a practical sense, to be a valid configuration than an arbitrary solution to the linear program. Thus, this algorithm represents a new method for evaluating a hypothesis (Section 1.5).

Finding an overlap minimum, even a local minimum, is a nonconvex problem. Fortunately, it is possible to rapidly find a local minimum in practice through iterated solution of an *overlap linear program* (OLP). For the problems we encounter in practice, minimization appears to require at most six iterations, and usually it requires only three or four. The following section lays out the general framework for the definition of the OLP. Sections 3.2 and 3.3 flesh out the details. Section 3.4 describes the actual overlap minimization algorithm and proves its convergence and correctness. The LP containment algorithm uses this algorithm. Unfortunately, it does not appear to have constant or logarithmic cost per vertex of the feasible polytope $\mathcal{F}$ of the CLP. In fact, the solution to the OLP does not necessarily lie at a vertex of $\mathcal{F}$. Section 3.5 gives

some alternate methods of overlap minimization, at least one of which has logarithmic cost per vertex of $\mathcal{F}$. These alternate methods are used in the combinatorial version of LP containment.

3.1 The General Framework for Overlap Minimization.

Ideally, we would like to add constraints and an objective to the CLP so that the resulting program would minimize

$$\max_{0 \le i < j \le k} \Delta(t_j - t_i, U_{ij}),$$

where $\Delta(t_j - t_i, U_{ij})$ is the Euclidean distance from $t_j - t_i$ to U_{ij}. Such a program would minimize the maximum overlap and thus find a nonoverlapping layout, if one exists.

Unfortunately, two properties of this objective prevent us from expressing the program as a linear program. First, the Euclidean distance Δ is not linear. Second, U_{ij} is not necessarily convex. We replace Δ with a linearized distance function Δ_8 and replace U_{ij} with a *local convex approximation* $I(t_j^0 - t_i^0, U_{ij})$, where $T^0 = \langle t_0^0, t_1^0, t_2^0, t_3^0, \ldots, t_k^0 \rangle$ is the current configuration. With these substitutions, we can define an overlap linear program (OLP) which minimizes

$$\max_{0 \le i < j \le k} \Delta_8(t_j - t_i, I(t_j^0 - t_i^0, U_{ij})),$$

under the constraints of the CLP. Iterating the solution to this OLP yields a local overlap minimum for the linearized distance Δ_8 and the nonconvex U_{ij}. We do not necessarily find an overlap minimum for the Euclidean distance function, but the Δ_8 distance differs from the Euclidean distance by at most 10%. The only overlap value which really matters is zero overlap, and this is the same for both distance functions.

3.2 The Linearized Distance Function.

To simplify notation, this section and the next consider a specific i, j and refer to $t_j - t_i$ and U_{ij} as t and U, respectively. This section defines the linearized distance function Δ_8 and proves that it is a good approximation to the Euclidean distance. This distance function is a generalization of the standard convex distance function[13] based on the regular octagon. We could generalize any convex distance function in the same manner, but since Δ_8 serves our purposes, we leave these other generalizations as an exercise for the reader. In what follows, we assume that each vertex of U is *simple*: it is an endpoint of exactly two edges. The region U itself can have multiple components and/or holes. In the implementation, if the contour of U passes through the same point more than one, each instance is considered a separate vertex.

For each point u on the boundary of U, define NORMALS(u, U) to be a set of outward pointing unit normal vectors to U at u given as follows. If u lies on an edge, then NORMALS(u, U) contains only one element perpendicular to the edge. If u is a concave vertex of U, then NORMALS(u, U) is empty. Otherwise, if u is a convex vertex of U, let v and w be the unit normal vectors to the edges of U that meet at u. In this case, NORMALS(u, U) contains v, w and all the unit vectors in the angle between v and w (the angle which is less than 180 degrees).

Point t *projects onto* p on the boundary of U if vector $t - p$ points in the same direction as some element of NORMALS(p, U). If p lies on an edge, this is an ordinary perpendicular projection. The usual definition of the Euclidean distance $\Delta(t, U)$ from a point t to a set U is

$$\Delta(t, U) = \min_{u \in U} |t - u|.$$

It is well known that the point $u \in U$ which minimizes $|t - u|$ is a point of projection of t onto U. Hence, $\Delta(t, U)$ is the distance to the nearest point of projection of t onto U. In order to motivate the definition of Δ_8 below, we observe that if p is a point of projection, then the Euclidean distance $|t - p|$ satisfies,

$$|t - p| = \max_{v \in \text{NORMALS}(p,U)} v \cdot (t - p). \tag{5}$$

[13] We are referring to the metric whose unit ball is a given convex polygon instead of a circle.

The vector $v \in \text{NORMALS}(p, U)$ which yields the maximum value of $v \cdot (t - p)$ is the one which points in the same direction as $t - p$.

The set $\text{NORMALS}_8(u, U) \subseteq \text{NORMALS}(u, U)$ is defined as follows. If u lies on an edge, then $\text{NORMALS}_8(u, U)$ equals $\text{NORMALS}(u, U)$. If u lies at a convex vertex of U, then $\text{NORMALS}_8(u, U)$ is defined to contain v and w (the outward unit normals to the edges of U which meet at u) plus all vectors in

$$S_8 = \left\{ \left(\cos \frac{2\pi i}{8}, \sin \frac{2\pi i}{8} \right) \mid i = 0, 1, 2, \ldots, 7 \right\}$$

which lie in the angle between v and w. In other words, take a discrete sample of unit normal vectors every 45 degrees.

If p is a point of projection of t, then define,

$$\Delta_8(t, p, U) = \max_{v \in \text{NORMALS}_8(p,U)} v \cdot (t - p). \tag{6}$$

Define $\Delta_8(t, U)$ to be the Δ_8-distance to the nearest point of projection of t onto U.

Lemma 3.1 $0.91\,\Delta(t, U) \leq \Delta_8(t, U) \leq \Delta(t, U)$.

Proof: For each point of projection p of t onto U,

$$\text{NORMALS}_8(p, U) \subseteq \text{NORMALS}(p, U).$$

Therefore, by Equations 5 and 6, $\Delta_8(t, p, U) \leq |t - p|$. This establishes the upper bound on $\Delta_8(t, U)$.

There must always be some $v \in \text{NORMALS}_8(p, U)$ such that the angle between v and $t - p$ is less than half of $2\pi/8$. Therefore,

$$v \cdot (t - p) \geq |t - p| \cos \frac{\pi}{8} \geq 0.91|t - p|.$$

This establishes the lower bound on $\Delta_8(t, U)$. ■

Finally, the distance function Δ_8 has a nice formulation for convex polygons. We will need this when we define the OLP in Section 3.4.

Lemma 3.2 *If Q is a convex polygon and t lies outside Q, then $\Delta_8(t, Q)$ is the maximum value of*

$$\max_{v \in \text{NORMALS}_8(q,Q)} v \cdot (t - q),$$

over all vertices q of Q.

Proof: Let p be the projection of t onto Q (since Q is convex, the projection is unique), and let $v_p \in \text{NORMALS}_8(p, Q)$ be the normal such that $\Delta_8(t, Q) = v_p \cdot (t - p)$ (Equation 6). Let $v_{\min}, v_{\max} \in \text{NORMALS}(p, Q)$ be the two elements the largest angle between them, where the angle between $v_{\min}$ and $v_{\max}$ is counterclockwise and less than 180 degrees. If p lies in the interior of an edge e of Q, then v_p, $v_{\min}$, and $v_{\max}$ are all equal to the outward normal vector to e.

Let q be any vertex of Q where $q \neq p$ if p is a vertex. Let v_q be any vector in $\text{NORMALS}_8(q, Q)$. We claim that

$$v_q \cdot (t - q) \leq v_p \cdot (t - p).$$

Since $v_q \in \text{NORMALS}(q, Q)$, v_q is an outward normal to a line of support for Q through q. Every point $p \in Q$ lies on the side of this line that satisfies,

$$v_q \cdot (p - q) \leq 0. \tag{7}$$

Next, observe that NORMALS(p, Q) and NORMALS(q, Q) do not overlap. At most, they have either $v_{\min}$ or $v_{\max}$ in common (but not both). Therefore, v_q is not in the interior of NORMALS(p, Q). On the other hand, $t - p$ does point in the same direction as some element of NORMALS(p, Q). Therefore, either $v_{\min}$ or $v_{\max}$ lies in the proper (≤ 180 degree) angle formed by v_q and $t - p$. It follows that

$$v_q \cdot (t - p) \leq \max(v_{\min} \cdot (t - p), v_{\max} \cdot (t - p)).$$

Since $v_{\min}, v_{\max} \in \text{NORMALS}_8(p, Q)$, it follows from Equation 6 that

$$v_q \cdot (t - p) \leq v_p \cdot (t - p). \tag{8}$$

Adding Equations 7 and 8 proves the claim.

If the projection p is a vertex of Q, then the lemma follows directly from the claim: set $q = p$ and $v = v_p$, and for this choice of q and v, $\Delta_8(t, Q) = v \cdot (t - q)$. Hence the maximum is equal to $\Delta_8(t, Q)$. If p lies in the interior of edge e, set q equal to one of the endpoints of this edge and set $v = v_p$. Since v_p is the outward normal to e, $v \in \text{NORMALS}_8(q, Q)$. Since $p - q$ is perpendicular to $t - p$, $v \cdot (t - p) = v_p \cdot (t - q) = \Delta_8(t, Q)$, and again the maximum is equal to $\Delta_8(t, Q)$. ■

3.3 Local Convex Approximation.

This section defines the notion of a *local convex approximation* $I(t, U)$ to U near t. We prove some properties of local convex approximations that are crucial to the correctness and convergence of the overlap minimization algorithm.

3.3.1 Discussion.

We would like to define an inner approximation to be a convex set $Q \subseteq U$ that looks like U near t from the point of view of t. Furthermore, we would like $\Delta_8(t, Q)$ to be an upper bound for $\Delta_8(t, U)$ for all $t \in \overline{U}$. In this way, minimizing the distance from t to Q will also move t closer to U. Unfortunately, the first goal is counterproductive and the second goal is unattainable. Hence, these unrealistic goals need to be modified. This section discusses these goals and their modification into realistic goals.

Imagine that U is a thin crescent and t can see the convex part of the boundary of U. Necessarily, any convex subset $Q \subset U$ can only cover a small portion of U. From the point of view of t, using $Q = \text{CH}(U)$ is a much better convex approximation to U. In particular, if t is near the middle of the convex part of the boundary of U, then only a very large motion of t will enable it to see the "back side" of Q where it differs from the concave "back side" of U. This example shows that the goal that Q be a subset of U can be counterproductive.

Even if $Q \subseteq U$, the second goal is unattainable for the Δ_8 metric. The second goal *is* attainable for the Euclidean metric. Clearly, if $Q \subseteq U$, then $\Delta(t, Q) \geq \Delta(t, U)$ for all $t \in \overline{U}$. Unfortunately, the Δ_8 metric does not have this property. The counterexample is somewhat complicated, and we do not present it here.

The next section defines $I(t, U)$ in a way that satisfies a modified version of these goals. First, $I(t, U)$ will be a subset of U, but only inside some limited region. Second, $\Delta_8(t', I(t, U))$ will be an upper bound for $\Delta_8(t', U)$ for t' sufficiently *near* to t.

3.3.2 Definition.

A *local convex approximation* to a region U *relative* to a point $t \notin U$ is a convex set $I(t, U)$ with two properties:[14] 1) Its boundary must contain the Δ_8-nearest point p to t on U. If the nearest point is not unique (more than one point of projection is at the same minimum Δ_8-distance), then $I(t, U)$ must contain one of these nearest points. 2) For some radius $\delta > 0$ and for all balls B about p with radius less than δ, $B \cap I(t, U) = B \cap U$.

[14] The "I" stands for "inner": from the point of view of t, $I(t,U)$ is a convex subset of U.

Lemma 3.3 *Let $Q = I(t, U)$ be a local convex approximation to U near t. There exists some ball B_t about t such that $t' \in B_t$ implies $\Delta_8(t', U) \leq \Delta_8(t', Q)$. If the Δ_8-nearest point to t on U is* **unique**, *then there exists some ball B_t about t such that $t' \in B_t$ implies $\Delta_8(t', U) = \Delta_8(t', Q)$.*

Proof: By definition, the boundary of Q contains p, a Δ_8-nearest point to t on U. Since Q is convex, the projection of t onto Q is unique, and we denote it as PROJECTION(t, Q). Clearly, p is equal to PROJECTION(t, Q). Let B_p be a ball about p which satisfies the condition of the definition of $I(t, U)$; let B_t be a ball centered at t with radius

$$\min(\Delta(t, U), \text{RADIUS}(B_p))$$

and let t' range over the points in B_t. It is clear that PROJECTION(t', Q) is a continuous function of t', and for some regions in B_t, PROJECTION(t', Q) is fixed at a vertex, and for other regions in B_t, PROJECTION(t', Q) moves along an edge of Q. We claim that PROJECTION(t', Q) never moves faster than t': in the worst case, PROJECTION(t', Q) is moving along an edge and its velocity is equal to the component of the velocity of t' parallel to that edge. As a consequence, $t' \in B_t$ implies PROJECTION$(t', Q) \in B_p$. Furthermore, if PROJECTION(t', Q) is in B_p, then it stays on the boundary of U which is identical to the boundary of Q inside B. Since the definition of projection is a local property, it remains a projection of t' onto U. Since $\Delta_8(t, U)$ is a minimum over all projections onto U, $\Delta_8(t', U) \leq \Delta_8(t', Q)$ for $t' \in B_t$.

Suppose p is a *unique* Δ_8-nearest projection of t onto U. The set of points which project onto a given vertex v or edge e of U is closed, and therefore its complement is open. Therefore if t does not project onto v (or e), there is a ball about t such that t' in the ball does not project onto v (or e). If t *does* project onto v (or e) outside B_p (if e is partially inside B_p, then we ignore it because it is an edge of Q), we observe that the Δ_8 distance of t to v (or e) is continuous. In the case of an edge, it is the true distance. In the case of a vertex, it is the maximum of a collection of continuous functions, which is continuous. This is true within the domain of points which project onto v (or e). Since t is closer to Q than v (or e) and since the Δ_8 distance is continuous, then there exists a ball B_v (or B_e) about t such that for t' in B_v (or B_e), t' either does not project onto v (or e), or t' is still closer to Q than v (or e). We choose B_t to be the ball about t whose radius is

$$\min(\Delta(t, U), \text{RADIUS}(B_p), \text{RADIUS}(B_v), \text{RADIUS}(B_e))$$

as v and e range over all vertices and edges of U outside B_p. For $t' \in B_t$, PROJECTION(t', Q) remains the unique Δ_8-nearest projection of t onto U, and therefore $\Delta_8(t', U) = \Delta_8(t', Q)$. ■

3.3.3 Constructing $I(t, U)$.

There is more than one way one can construct $I(t, U)$ to satisfy the definition of the previous section. As Section 3.3.1 discussed, it may be counterproductive to insist that $I(t, U) \subseteq U$. In fact, it is often useful to allow $I(t, U)$ to be unbounded.

As Figure 2 illustrates, our current implementation generates $I(t, U)$ as follows. It starts with the point p on the boundary of U which is nearest to t under the Δ_8 metric. Next, it "walks" along the boundary of U in both directions. For each direction, as long as it encounters convex vertices, it adds the next edge to the boundary of $I(t, U)$. If it encounters a concave vertex, it extends the current edge to infinity or until it intersects the chain of edges generated in the other direction. It also stops and extends the current edge if the next edge normal is rotated more than 180 degrees with respect to the direction of $t - p$.

Since p either lies in the middle of an edge or at a convex vertex of U, this extension process will always include a neighborhood of the boundary of U visible from t near p. Therefore, the region $I(t, U)$ will satisfy the definition of the previous section.

3.4 Overlap Minimization Algorithm.

This section describes the overlap linear program (OLP) and the overlap minimization algorithm. This algorithm finds a local minimum of the Δ_8-overlap,

$$\max_{0 \le i < j \le k} \Delta_8(t_j - t_i, U_{ij}).$$

It also proves convergence of the algorithm.

3.4.1 The OLP.

Suppose we are currently at an overlapping configuration

$$T^0 = \langle t_0^0, t_1^0, t_2^0, t_3^0, \ldots, t_k^0 \rangle.$$

For each pair of overlapping polygons $P_i + t_i^0$ and $P_j + t_j^0$ the overlap minimization algorithm computes a local convex approximation $I(t_j^0 - t_i^0, U_{ij})$. It sets up an OLP to minimize,

$$\text{APPROX-DIST}(T, \mathcal{U}, T^0) = \max_{0 \le i < j \le k} \Delta_8(t_j - t_i, I(t_j^0 - t_i^0, U_{ij})), \tag{9}$$

under the constraints of the CLP (Section 2.1.1), where $T = \langle t_0, t_1, t_2, \ldots, t_k \rangle$. To do this it adds the constraints,

$$v_{ij} \cdot (t_j - t_i - q_{ij}) \le d,$$

for each overlapping i, j, for each vertex q_{ij} of $I(t_j^0 - t_i^0, U_{ij})$, and for each normal vector

$$v_{ij} \in \text{NORMALS}_8(t_j - t_i, q_{ij}, I(t_j^0 - t_i^0, U_{ij})).$$

It also sets the objective to minimize d. This is the OLP. By Lemma 3.2, the minimum value of d, which is the maximum value of $v_{ij} \cdot (t_j - t_i - q_{ij})$, is the maximum distance from $t_j - t_i$ to $I(t_j^0 - t_i^0, U_{ij})$ for $0 \le i < j \le k$. Therefore, this OLP does indeed minimize APPROX-DIST(T,$\mathcal{U}$,T^0).

3.4.2 Subdividing the Interval.

Let $T^1 = \langle t_0^1, t_1^1, t_2^1, t_3^1, \ldots, t_k^1 \rangle$ be the output of the OLP. It follows that APPROX-DIST(T^1,$\mathcal{U}$,T^0) $\le$ APPROX-DIST(T^0,$\mathcal{U}$,T^0). If the two are equal, the algorithm terminates. Otherwise, it does the following. It sets $T' = T^1$. While the Δ_8-overlap of configuration T' is not less than that of T^0, it sets T' equal to the midpoint of T^0T'. As we show below, this loop terminates, and when it does, the algorithm makes T' the new current configuration and goes back to the OLP step.[15]

3.4.3 Convergence and Correctness

Theorem 3.4 *The OLP algorithm converges to a configuration which is either a local Δ_8-overlap minimum of $\mathcal{U}$ or of a perturbed hypothesis $\mathcal{U}'$ which can be made arbitrarily close to $\mathcal{U}$.*

Proof: Convergence. If T^1 has smaller APPROX-DIST than T^0, then by linearity, every $T' \in T^0T^1$, $T' \neq T^0$ will have smaller APPROX-DIST. Lemma 3.3 implies that for T' sufficiently close to T^0, APPROX-DIST(T',$\mathcal{U}$,T^0) is an upper bound on the Δ_8-overlap of T'. Remember that APPROX-DIST(T^0,$\mathcal{U}$,T^0) *equals* the Δ_8-overlap of T^0. Therefore, after sufficient interval halving, T' will have smaller Δ_8-overlap than T^0.

Local minimum. Suppose the algorithm converges to a configuration T. If for each i, j, $t_j - t_i$ has a unique nearest point on the boundary of U_{ij}, then Lemma 3.3 implies that APPROX-DIST(T',$\mathcal{U}$,T) *equals* the Δ_8-overlap of T' in the vicinity of T. Since T is a (global) minimum of APPROX-DIST, it must at least be a local minimum of the Δ_8-overlap. If for some $t_j - t_i$, U_{ij} does not have a unique nearest point of projection, then we can create an perturbation U'_{ij} which moves the "wrong" points of projection farther away and which is arbitrarily close to U_{ij}. ■

[15] In practice, we stop the algorithm when the Δ_8-overlap diminishes by less than a fixed fraction.

3.5 Evaluation Methods for Combinatorial Version.

In order to establish the desired running time for the combinatorial version of LP containment, it is necessary that evaluation have constant or logarithmic cost per vertex of $\mathcal{F}$, the feasible polytope of the CLP, just as the combinatorial LP restriction algorithm does in Sections 2.2.2 and 2.2.3. Unfortunately, the overlap minimization algorithm in the previous section does not have this property. This section gives some alternatives, one of which has this property.

By computing the *Voronoi diagram* of U_{ij}^{init}, it is possible to create a data structure that allows us to determine the (Euclidean) distance of any point $t_j - t_i$ to U_{ij}^{init} using $O(\log |U_{ij}^{\text{init}}|)$ time. For each vertex f of $\mathcal{F}$ generated by the LP restriction algorithm, we can compute the overlap of the corresponding configuration in $O(k^2 \log |\mathcal{U}^{\text{init}}|)$ time. This cost is logarithmic per vertex. Using $O(k^3 \log |\mathcal{U}^{\text{init}}|)$ time per vertex f, we could also determine if moving along any edge of $\mathcal{F}$ out of f diminishes the overlap. Hence, a modification of the simplex step algorithm of Section 2.2.2 can find a local overlap minimum *relative* to the set of vertices and edges of $\mathcal{F}$. This algorithm would not have to visit vertices that had already been seen. Of course, a true local minimum might be on a face of $\mathcal{F}$ or in its interior. Nevertheless, this disadvantage might be outweighed by the speed of the simplex algorithm on vertices of $\mathcal{F}$.

Another slight disadvantage of this evaluation method is that $t_j - t_i$ might be "attracted" to a portion of U_{ij}^{init} which lies outside C_{ij}. The solution to this problem is to use the Voronoi diagram of $U_{ij} = C_{ij} \cap U_{ij}^{\text{init}}$. It is probably not possible to prove that that one can construct this Voronoi diagram using logarithmic time per vertex of $\mathcal{F}$. Nevertheless, it could be quite fast in practice since it still makes use of the fast simplex algorithm for $\mathcal{F}$.

4 Subdivision.

Subdivision divides the current hypothesis into two subhypotheses. Specifically, subdivision splits one U_{ij} in $\mathcal{U}$ into two parts U_{ij}^- and U_{ij}^+. It creates two subhypotheses by replacing U_{ij} with U_{ij}^- or U_{ij}^+. Subdivision has two goals. The first goal is to make the convex hull $\text{CH}(U_{ij})$ "stick out" of U_{ij}^{init} less (see Section 1.3.3). In particular, if we define the *maximum overlap* of $\text{CH}(U_{ij})$ to be,

$$\max_{u \in \text{CH}(U_{ij})} \Delta(u, U_{ij}^{\text{init}}),$$

then subdivision should make the maximum overlap of $\text{CH}(U_{ij}^-)$ and $\text{CH}(U_{ij}^+)$ smaller. It does not matter whether we use the Euclidean metric Δ or the linearized metric Δ_8. The second goal is to drive the OLP evaluator "away" from the current overlapping configuration. Since the current configuration is at a local overlap minimum, the only way to prevent the evaluator from "falling into" the same local minimum is to move to a configuration outside $\text{CH}(U_{ij}^-)$ and $\text{CH}(U_{ij}^+)$, as far outside as possible.

Note that if $\text{CH}(U_{ij}) \subseteq U_{ij}^{\text{init}}$, then U_{ij} need not be split: if the output of the CLP satisfies $t_j - t_i \in \text{CH}(U_{ij})$, then it will satisfy $t_j - t_i \in U_{ij}^{\text{init}}$. If each U_{ij} in the current hypothesis $\mathcal{U}$ satisfies $\text{CH}(U_{ij}) \subseteq U_{ij}^{\text{init}}$, $0 \leq i < j \leq k$, then the output of the CLP must be a solution to containment [4]. In such a case, we say that $\mathcal{U}$ is *pseudo-convex*. Note that $\mathcal{U}$ can be pseudo-convex even if each U_{ij} in $\mathcal{U}$ is nonconvex. Any algorithm that subdivides down to pseudo-convex hypotheses will be a combinatorial containment algorithm. Furthermore, we can add geometric (Equation 3) or other restrictions to the algorithm without spoiling the combinatorial running time [4]. The running time is, of course, increased by the time to perform the geometric restrictions, but the number of hypotheses is the same in the worst case. Geometric restriction might introduce new concave vertices into the configuration spaces, but it makes some of the U_{ij} smaller, and therefore it will not change a pseudo-convex hypothesis into a non-pseudo-convex hypothesis.

The first goal of subdivision is accomplished by almost any reasonable splitting algorithm. The second goal is the focus of this section. Section 4.1 gives a *distance-based* subdivision algorithm that creates subhypotheses which are the maximum possible *distance* from the current configuration. Because it fragments the regions as little as possible, this algorithm is the one we use in our LP containment algorithm. This subdivision algorithm runs in linear time, but we cannot prove that a containment algorithm that uses it necessarily

has a good theoretical running time bound. Section 4.2 gives a *combinatorial* subdivision algorithm that drives the evaluator to a configuration which is *combinatorially* different. The combinatorial version of LP containment uses this combinatorial subdivision algorithm.

4.1 Subdivision for LP Containment.

Overlap minimization computes a configuration $\langle t_0, t_1, ..., t_k \rangle$ which is a local overlap minimum. If the overlap is zero, this is a valid configuration, and there is no need to subdivide. If the overlap is not zero, the subdivision algorithm selects the pair i, j such that $P_i + t_i$ and $P_j + t_j$ are the most overlapped: $\Delta_8(t_j - t_i, U_{ij})$ is maximal. It then subdivides U_{ij} into U_{ij}^- and U_{ij}^+. Note that U_{ij} may have multiple components and holes. There are three cases:

1. U_{ij} **has multiple components.** It sets U_{ij}^- equal to the component of U_{ij} which is closest to $t_j - t_i$. It sets U_{ij}^+ equal to the union of the other components.

2. U_{ij} **is connected and every ray out of** $t_j - t_i$ **intersects** U_{ij}**.** It selects the vertex u of U_{ij} which is nearest to $t_j - t_i$. It splits U_{ij} into U_{ij}^-, the closure of $H^- \cap U_{ij}$, and U_{ij}^+, the closure of $H^+ \cap U_{ij}$, where H^- and H^+ are the open half-planes to the left and right of LINE$((t_j - t_i)u)$, respectively.[16]

3. U_{ij} **is connected and** $t_j - t_i$ **can "see to infinity" along some ray.** It selects a line L through $t_j - t_i$ which splits U_{ij} into U_{ij}^- and U_{ij}^+. The line is chosen to maximize

$$d(L) = \min(\Delta(t_j - t_i, \mathrm{CH}(U_{ij}^-)),\ \Delta(t_j - t_i, \mathrm{CH}(U_{ij}^+))), \tag{10}$$

   the minimum Euclidean distance from $t_j - t_i$ to the convex hulls of the two "halves" of U_{ij}.

For cases 1 and 2, the closest component or vertex of U_{ij} to $t_j - t_i$ can clearly be located in $O(|U_{ij}|)$ time. The remainder of this section shows how to determine in $O(|U_{ij}|)$ time whether $t_j - t_i$ can "see to infinity" when U_{ij} is connected (case 2 or 3) and, if so, how to compute the line described in case 3. To simplify notation, let t denote $t_j - t_i$, U denote U_{ij} and $L_{\max}$ denote the splitting line which maximizes the expression for $d(L)$ in Equation 10.

To maximize $d(L)$, it suffices to consider lines in the set $\mathcal{L} = \{L | d(L) \neq 0\}$. Each $L \in \mathcal{L}$ contains a ray out of t along which t can "see to infinity." Section 4.1.1 shows how to find $\mathcal{L}$ in $O(|U|)$ time. If $\mathcal{L} = \emptyset$, then U and t satisfy case 2; otherwise they satisfy case 3. Section 4.1.1 also shows how to construct in $O(|U|)$ time an ordered list E of $O(|U|)$ subedges of the boundary of U, where E is the part of the boundary of U which is visible to t. It also shows how to find a sublist G of E such that CLOSURE$(\mathcal{L}) = \{\mathrm{LINE}(tu) | u \in e \text{ for } e \in G\}$.[17]

Section 4.1.2 shows that binary search on the edge index in G can determine the edge in the list which contains $u_{\max}$ such that $L_{\max} = \mathrm{LINE}(tu_{\max})$. It also observes that binary search can be used on any partition G' of G. Section 4.1.3 creates a partition G' of G with the property that, with $O(|U|)$ preprocessing time, we can evaluate a given vertex of any edge in G' in $O(1)$ time within the binary search. It shows that G' contains $O(|U|)$ edges and can be constructed in $O(|U|)$ time. Section 4.1.3 also shows that, if the binary search determines that $u_{\max}$ is in the interior of an edge in G', then $O(|U|)$ preprocessing allows us to retrieve, in $O(1)$ time, the information necessary to calculate $u_{\max}$. Section 4.1.4 shows that, in this case, $u_{\max}$ can be calculated in $O(1)$ time in a real arithmetic model and in $O(\log(\log \epsilon^{-1}))$ time in a rational arithmetic model, where ϵ is the accuracy. Section 4.1.5 concludes that $L_{\max}$ can be computed in $O(|U|)$ time in a real arithmetic model and in $O(|U| + \log(\log \epsilon^{-1}))$ time in a rational arithmetic model.

[16] A point p is on the left of LINE(ab) if $(b - a) \times (p - a) > 0$: *left* is from the point of view of an observer standing at a and facing toward b.

[17] The set of lines through t is a metric space under angle, so closure is well-defined. Taking the closure adds the two lines which "bound" the set of lines in $\mathcal{L}$.

4.1.1 Finding $\mathcal{L}$.

Lemma 4.1 *If U is connected, then in $O(|U|)$ time we can find $\mathcal{L} = \{L | d(L) \neq 0\}$.*

Proof: $L \in \mathcal{L}$ if and only if it contains a ray out of t which does not intersect U. First we show how to find all such rays and then we show they can be found in $O(|U|)$ time. The set of rays $\text{RAYS}(t, U)$ out of t that hit U is connected [18] because U is connected. Therefore, the set of rays $\overline{\text{RAYS}(t, U)} = \text{RAYS}(t) \setminus \text{RAYS}(t, U)$ that do not intersect U is also connected; these are the rays we need. Because t is part of the output of a CLP, $t \in \text{CH}(U)$. Since $t \in \text{CH}(U)$, each ray in $\overline{\text{RAYS}(t, U)}$ must pass through the boundary of $\text{CH}(U)$. The (connected) set $\{\rho \in \overline{\text{RAYS}(t, U)} | (\rho \cap \text{BOUNDARY}(\text{CH}(U))) \neq \emptyset\}$ must be a subset S of a *single* edge e of $\text{CH}(U)$: if the set contained a vertex of $\text{CH}(U)$, then we could move this vertex towards t and make $\text{CH}(U)$ smaller. The edge e is not an edge of U. Furthermore, let H be the component of $\text{CH}(U) \cap \overline{U}$ which contains t, and let V be the visibility polygon of H with respect to t. If V has an edge which is not collinear with t and which is not a subset of an edge of U, then the interior of that edge is S and $\mathcal{L} = \{\text{LINE}(st) | s \in S\}$. Otherwise, S does not exist and $\mathcal{L} = \emptyset$.

Now we establish the running time. Region U is connected, and its outer boundary is simple. The convex hull of the outer boundary equals $\text{CH}(U)$ and can be computed in linear time [25, 10, 16]. The point u nearest to t on the boundary of U can be found in $O(|U|)$ time. To determine the boundary of H, walk along the boundary of U in each direction away from u. Either the two paths will meet, or they will both reach (the same edge of) the boundary of $\text{CH}(U)$. In the latter case, join these two endpoints; the interior of this segment is S. Since U is connected, it has no "islands" inside H, and thus H is a simple polygon. The visibility polygon V for a simple polygon H can be found in $O(|H|)$ ($\leq O(|U|)$) time [15, 22, 20]. ■

The remainder of this section assumes that $\mathcal{L}$ is non-null, so that U and t satisfy case 3. Figure 3 illustrates an example of U, t, and V for which t can see to infinity through S. Note that S is a proper subset of an edge of the convex hull of U. Some of the edges of V other than $\text{CLOSURE}(S)$ might be collinear with t, and these may or may not be subsets of the boundary of U. All the non-collinear edges except $\text{CLOSURE}(S)$ lie on the boundary of U.

Lemma 4.2 *If U and t satisfy case 3, then in $O(|U|)$ time we can construct: 1) an ordered list E of $O(|U|)$ subedges of the boundary of U, where E is the part of the boundary of U that is visible to t and 2) a sublist G of E such that $\text{CLOSURE}(\mathcal{L}) = \{\text{LINE}(tu) | u \in e \text{ for } e \in G\}$.*

Proof: Let V be the visibility polygon from the proof of Lemma 4.1. To form E, remove the single edge $\text{CLOSURE}(S)$ of V which is a subset of the boundary of $\text{CH}(U)$. Remove edges of V which are collinear with t. The result is a list of edges E which we can assume are ordered clockwise with respect to t.[19] The list E can be represented as: $A_1A_2, A_3A_4, \ldots, A_{2n-1}A_{2n}$, where A_1 and A_{2n} are the endpoints of S and $\text{RAY}(tA_{2i})$ equals $\text{RAY}(tA_{2i+1})$, $1 \leq i \leq n-1$. Because U is connected and $t \in \text{CH}(U)$, both $\text{LINE}(A_1t)$ and $\text{LINE}(A_{2n}t)$ intersect E beyond t at least once (and at most twice if A_1t (or $A_{2n}t$) passes through A_{2i} and A_{2i+1} and these are not the same point). If an intersection point is in the interior of an edge in E, split that edge and reindex E. Let the intersection points of $\text{LINE}(A_{2n}t)$ with E (beyond t) be $A_{2\alpha-1}$ and $A_{2\alpha-2}$ (which may be the same point) and the intersection points of $\text{LINE}(A_1t)$ with E (beyond t) be $A_{2\beta}$ and $A_{2\beta+1}$ (which also may be the same point). Let G be the following sublist of E: $A_{2\alpha-1}A_{2\alpha}, \ldots, A_{2\beta-1}A_{2\beta}$. Because $\mathcal{L}$ is connected[20] and, from the proof of Lemma 4.1, $\mathcal{L} = \{\text{LINE}(st) | s \in S\}$, we conclude that $\text{CLOSURE}(\mathcal{L}) = \{\text{LINE}(tu) | u \in e \text{ for } e \in G\}$, as required. The construction time is $O(|U|)$ for two reasons: 1) Lemma 4.1 guarantees that V can be constructed in $O(|U|)$ time and 2) V, being the visibility polygon of the simple polygon H of size $O(|U|)$, has $O(|U|)$ edges. The second reason also guarantees that E has $O(|U|)$ edges. ■

Figure 4 shows gives an example of $A_{2\alpha-1}$, $A_{2\beta}$ and the edges of G.

[18] The union of the rays is a single wedge.

[19] We know the order of these edges because we know the order of the edges of V.

[20] The union of lines in $\mathcal{L}$ is a double wedge through t, bounded by two lines.

4.1.2 Binary Search.

Here we show that binary search on the value of the edge index i in G, $\alpha \leq i \leq \beta$, can determine the edge in G which contains $u_{\max}$ such that $L_{\max} = \text{LINE}(tu_{\max})$. This result also holds for any partition of G. Let $U^-(L)$ denote U^- when U is split by L, and define $U^+(L)$ analogously. Let $d^-(L) = \Delta(t, \text{CH}(U^-(L)))$ and $d^+(L) = \Delta(t, \text{CH}(U^+(L)))$. The functions $d^-(L)$ and $d^+(L)$ are discontinuous where L intersects both A_{2i} and A_{2i+1} but $A_{2i} \neq A_{2i+1}$ (this occurs where L intersects an edge of V which is collinear with t). Unfortunately,

$$d^-(\text{LINE}(tA_{2i})) = d^-(\text{LINE}(tA_{2i+1})) \qquad \text{and} \qquad d^+(\text{LINE}(tA_{2i})) = d^+(\text{LINE}(tA_{2i+1})), \tag{11}$$

so $d^-(L)$ and $d^+(L)$ cannot reveal if $L_{\max}$ occurs at a discontinuity. To overcome this problem we define $d^-(u)$ and $d^+(u)$, for $u \in A_{2i-1}A_{2i}$, as follows:

$$d^-(u) = \Delta(t, \text{CH}(A_1, A_2, \ldots, A_{2i-1}, u)) \qquad \text{and} \qquad d^+(u) = \Delta(t, \text{CH}(A_{2n}, A_{2n-1}, \ldots, A_{2i}, u)).$$

Below we present the binary search and then Lemma 4.3 proves its correctness.

BINARY-SEARCH(t, E, α, β)
    $i_{\text{lower}} \leftarrow \alpha$
    if (($result \leftarrow$ EVALUATE(t, E, i_{lower})) $\neq$ NULL) **return** $result$
    $i_{\text{upper}} \leftarrow \beta$
    if (($result \leftarrow$ EVALUATE(t, E, i_{upper})) $\neq$ NULL) **return** $result$
    while ($i_{\text{upper}} - i_{\text{lower}} > 1$)
        $i \leftarrow (i_{\text{lower}} + \text{upper})/2$
        if (($result \leftarrow$ EVALUATE(t, E, i)) $\neq$ NULL) **return** $result$
        if ($d^-(A_{2i-1}) > d^+(A_{2i-1})$) **then**
            $i_{\text{lower}} = i$
        else
            $i_{\text{upper}} = i$

EVALUATE(t, E, i)
    if ($d^-(A_{2i-1}) > d^+(A_{2i-1})$ **and** $d^-(A_{2i}) < d^+(A_{2i})$) **return** edge $A_{2i-1}A_{2i}$
    if ($d^-(A_{2i}) \geq d^+(A_{2i})$ **and** $d^-(A_{2i+1}) \leq d^+(A_{2i+1})$) **return** vertex A_{2i}
    if ($d^-(A_{2i-2}) \geq d^+(A_{2i-2})$ **and** $d^-(A_{2i-1}) \leq d^+(A_{2i-1})$) **return** vertex A_{2i-1}
    return NULL

Lemma 4.3 BINARY-SEARCH() *yields either the vertex A_j of G such that $d(A_j)$ maximizes $d(L)$ for $L \in \mathcal{L}$ or the edge $A_{2i-1}A_{2i}$ of G such that $d(u)$ maximizes $d(L)$ for $L \in \mathcal{L}$ for some $u \in$ INTERIOR$(A_{2i-1}A_{2i})$.*

Proof: We claim that the search maintains the following invariant:

$$d^-(A_{2i_{\text{lower}}-1}) > d^+(A_{2i_{\text{lower}}-1}) \qquad \text{and} \qquad d^-(A_{2i_{\text{upper}}}) < d^+(A_{2i_{\text{upper}}})$$

and that the "ordering"[21] of $d^+(u)$ and $d^-(u)$ changes exactly once for $i_{\text{lower}} \leq i \leq i_{\text{upper}}$. The first part of the invariant is true at the start of the search because

$$d^-(A_{2\alpha-1}) > d^+(A_{2\alpha-1}) = 0 \qquad \text{and} \qquad 0 = d^-(A_{2\beta}) < d^+(A_{2\beta}).$$

The updates of i_{lower} and i_{upper} for each i maintain this part of the invariant. The second part of the invariant relies on the following monotonicity result by Daniels [4]. Parameterize the line segment A_1A_{2n} by the linear function $\gamma(\tau)$, for $\tau \in [0, 1]$, such that $\gamma(0) = A_{2n}$ and $\gamma(1) = A_1$. Daniels shows that the distance

[21] We cannot guarantee intersection of $d^-(u)$ and $d^+(u)$ because $d^-(u)$ and $d^+(u)$ are discontinuous where edges of the visibility polygon V are collinear with t. Hence, we discuss ordering change instead of intersection.

function $d^-(\text{LINE}(\gamma(\tau)t))$ is non-increasing for τ over $[0,1]$. For $\tau, \tau' \in [0,1]$ and $\tau' > \tau$, $\text{LINE}(\gamma(\tau')t)$ is rotated clockwise from $\text{LINE}(\gamma(\tau)t)$. Her proof shows that, as a consequence, "more" of U^- lies to the left of $\text{LINE}(\gamma(\tau')t)$ than $\text{LINE}(\gamma(\tau)t)$, and thus

$$\text{CH}(U^-(\text{LINE}(\gamma(\tau)t))) \subseteq \text{CH}(U^-(\text{LINE}(\gamma(\tau')t))),$$

which implies that

$$d^-(\text{LINE}(\gamma(\tau)t)) \geq d^-(\text{LINE}(\gamma(\tau')t))$$

and therefore establishes that $d^-(\text{LINE}(\gamma(\tau)t))$ is non-increasing. A similar argument shows that $d^+(\text{LINE}(\gamma(\tau)t))$ is non-decreasing for τ over $[0,1]$.

Given $u \in A_{2i-1}A_{2i}$ in G, it is easily shown that

$$d^-(u) = d^-(\text{LINE}(tu)) \qquad \text{unless} \qquad u = A_{2i+1} (\neq A_{2i}). \tag{12}$$

Furthermore, if $d^-(A_{2i+1}) \neq d^-(\text{LINE}(tA_{2i+1}))$, then $d^-(A_{2i}) \geq d^-(A_{2i+1})$ and $d^-(A_{2i+1}) \geq d^-(u)$ for $u \in A_{2i+1}A_{2i+2}$. These two facts imply that $d^-(u)$ has the same monotonicity property as $d^-(\text{LINE}(tu))$. A similar argument shows that $d^+(u)$ has the same monotonicity property as $d^+(\text{LINE}(tu))$.

If the binary search returns the edge $A_{2i-1}A_{2i}$, then the invariant guarantees that the ordering of $d^-(u)$ and $d^+(u)$ changes exactly once for some $u \in \text{INTERIOR}(A_{2i-1}A_{2i})$. Similarly, if the search returns A_{2i}, then the ordering changes once between A_{2i} and A_{2i+1} and if the search returns A_{2i-1}, then the ordering changes once between A_{2i-2} and A_{2i-1}. Monotonicity guarantees that the minimum of $d^-(u)$ and $d^+(u)$ is maximized where the ordering changes. We claim that the minimum of $d^-(L)$ and $d^+(L)$ is maximized where the minimum of $d^-(u)$ and $d^+(u)$ is maximized. This follows from Equation 11 and Equation 12. ■

Note: Daniels uses her monotonicity result to conclude that $L_{\max}$ can be found to any degree of accuracy by binary search on the value of τ. This leads to an approximate algorithm with running time $O(|U| \log \epsilon^{-1})$.

4.1.3 Preprocessing.

Lemma 4.3 allows us to find the vertex or edge in G associated with $L_{\max}$ using binary search on the value of the index of G. The running time of the binary search is dominated by $O(\log(|G|)X) = O(\log(|U|)X)$, where X is the time required to calculate $d^+(A_j)$ and $d^-(A_j)$ for a given vertex A_j of G. This allows $L_{\max}$ to be found in time $O(|U| + \log(|U|)X + Y)$, where Y is the time to find $u \in \text{INTERIOR}(A_{2i-1}A_{2i})$ such that $d(u)$ maximizes $d(L)$ for $L \in \mathcal{L}$ (if the binary search returns an edge instead of a vertex). This section shows how to preprocess G in $O(|U|)$ time so that $X = O(1)$ and so that $Y = O(1)$ in a real arithmetic model and $Y = O(\log(\log \epsilon^{-1}))$ in a rational arithmetic model with accuracy ϵ. (Finding $u \in \text{INTERIOR}(A_{2i-1}A_{2i})$ such that $d(u)$ maximizes $d(L)$ for $L \in \mathcal{L}$ is addressed in Section 4.1.4.) This allows us to find $L_{\max}$ in $O(|U|)$ time in a real arithmetic model and in $O(|U| + \log(\log \epsilon^{-1}))$ time in a rational arithmetic model. Some of the preprocessing involves partitioning G in $O(|U|)$ time into G' which has $O(|U|)$ edges. Clearly, any partition G' of G also satisfies Lemma 4.3 and, since G' has $O(|U|)$ edges and is constructed in $O(|U|)$ time, the asymptotic running time of the overall algorithm is unaffected by the partitioning.

Our discussion of preprocessing has three parts. The first part introduces a modification of an existing convex hull algorithm. The second part shows how to build the convex hulls we need for the distance calculations from the convex chain which is maintained during the modified convex hull algorithm. This part also involves partitioning G. The third part shows how to quickly update the closest point of the convex hulls to t as the hulls are constructed.

Before treating each of these parts, we introduce more notation. If u is in edge $A_{2i-1}A_{2i}$, of G, $(\alpha \leq i \leq \beta)$, then denote by $C^-(u)$ the portion of $\text{CH}(A_1, A_2, \ldots, A_{2i-1}, u)$ which is visible from t. Let $c^-(u)$ be the closest point of $C^-(u)$ to t. Similarly, define $C^+(u)$ to be the portion of $\text{CH}(A_{2n}, A_{2n-1}, \ldots, A_{2i}, u)$ which is visible from t and define $c^+(u)$ be the closest point of $C^-(u)$ to t.

Figure 5 illustrates $\text{LINE}(tu)$, $C^-(u)$, $C^+(u)$, $c^-(u)$, and $c^+(u)$ for a particular u.

Lemma 4.4 *$d^-(u) = \Delta(t, C^-(u))$, and $d^+(u) = \Delta(t, C^+(u))$.*

Proof: Both the closest point of $\mathrm{CH}(A_1, A_2, \ldots, A_{2i-1}, u)$ to t and the closest point of $\mathrm{CH}(A_{2n}, A_{2n-1}, \ldots, A_{2i}, u)$ to t are visible to t. If either lies in the interior of an edge of the respective convex hull, then the entire edge is visible because the hull is convex. ■

The rest of our discussion uses $C^-(u)$ and $C^+(u)$ instead of $\mathrm{CH}(A_1, A_2, \ldots, A_{2i-1}, u)$ and $\mathrm{CH}(A_{2n}, A_{2n-1}, \ldots, A_{2i}, u)$.

Consider $u \in A_{2i-1}A_{2i}, \alpha \leq i \leq \beta$. The vertices of $C^-(u)$ are a subsequence of $A_1, A_2, \ldots, A_{2i-1}$ plus the last (clockwise most)[22] vertex is u. Similarly, the vertices of $C^+(u)$ are a subsequence of $A_{2n}, A_{2n-1}, \ldots, A_{2i}$ plus the first (counter-clockwise most) vertex is u. Define the *fixed part* $R^-(u)$ of $C^-(u)$ to be the chain which is a subset of $C^-(u)$ and whose vertices form the set $\mathrm{VERTICES}(C^-(u)) \setminus \{u\}$. Let $r^-(u)$ be the closest point of $R^-(u)$ to t. Let $s^-(u)$ be the last vertex of $R^-(u)$. Define the *fixed part* $R^+(u)$ of $C^+(u)$, $r^+(u)$, and $s^+(u)$ analogously. Figure 6 shows gives an example of $R^-(u)$, $R^+(u)$, $r^-(u)$, $r^+(u)$, $s^-(u)$ and $s^+(u)$. For the u in this example, it happens to be the case that $r^-(u) = s^-(u)$ and $r^+(u) = s^+(u)$.

We say that $C^-(p)$ and $C^-(q)$ are *combinatorially equivalent* if $R^-(p) = R^-(q)$. The analogous definition holds for $C^+(p)$ and $C^+(q)$. We say that two points p and q are *hull equivalent* if $C^-(p)$ and $C^-(q)$ are combinatorially equivalent and $C^+(p)$ and $C^+(q)$ are combinatorially equivalent.

Modified Graham Scan.

Here we describe a clockwise scan of $E = A_1A_2, \ldots, A_{2n-1}A_{2n}$ which is a modification of Graham's scan [17]. Graham's scan constructs the convex hull of a set of points in linear time once they are in sorted order about an internal point O. As observed in [31], in Graham's scan, "if a point is not a vertex of the convex hull, it is internal to some triangle Opq where p and q are consecutive hull vertices." If points are ordered clockwise about O, then this implies that the point is in the wedge which is on the right of Op and on the left of Oq and the point is on the right of pq. When the scan encounters a triple of consecutive points prq in the ordering about O such that prq is a left turn, it pops vertices off the current hull until convexity is restored.

We modify Graham's scan so that it correctly constructs the visible part of a convex chain when O (t in our case) is *external* to the chain. In this case, if a point is not a vertex of the convex chain, it is in a wedge which is on the left of pO and Oq and the point is on the *left* of pq. Thus, we need only change the left turn test for a triple prq to a right turn test in order to perform a clockwise scan about t. A counter-clockwise scan about t from A_{2n} to A_1 requires only minor modifications to the scan. In both cases, the modified scan has the same asymptotic running time of $\mathrm{O}(|U|)$ as Graham's scan. To simplify the remainder of our discussion, we insist that the chain which is built during the modified scan is convex at all times. Hence, we do not add a new point to the chain until all vertices which must be removed have been removed. This does not affect the running time or correctness of the algorithm.

Partitioning.

During a given step of one direction of the modified Graham scan, denote by C the current state of the convex chain which is maintained by the scan. We describe a partitioning process which produces a partition G' of G, where the new index of $A_{2\beta}$ is $2\beta'$. The partition can be generated during the modified scan of E by marking edges of $A_{2\alpha-1}A_{2\alpha}, \ldots, A_{2\beta-1}A_{2\beta}$ as vertices are removed from C. In the clockwise scan, suppose we are processing A_{2i}, $\alpha \leq i \leq \beta$, and that $A_gA_hA_{2i}$ is a right turn, where A_gA_h is the last edge of C. Before deleting A_h, extend ray A_gA_h. If it intersects $A_{2i-1}A_{2i}$, then "mark" $A_{2i-1}A_{2i}$ at the intersection point. Note that no marking need be done when processing A_j if j is odd.[23] Use the same marking process during the counterclockwise scan of $A_{2n}A_{2n-1}, \ldots, A_2A_1$. When the second scan is complete, split the edges of G at the marks to produce the partition $G' = A'_{2\alpha-1}A'_{2\alpha}, \ldots, A'_{2\beta'-1}A'_{2\beta'}$.

Lemma 4.5 *G' has $\mathrm{O}(|U|)$ edges and can be constructed in $\mathrm{O}(|U|)$ time.*

Proof: We first argue that $|G'| \in \mathrm{O}(|U|)$. This is true because, during each of the two scans, each vertex of E is added to the convex chain C at most once and is therefore removed at most once. Because $|G'| \in \mathrm{O}(|U|)$ and we spend $\mathrm{O}(1)$ time per vertex of G', G' can be generated in $\mathrm{O}(|U|)$ time. ■

[22] This is clockwise most from the point of view of t. For the standard definition of clockwise and counterclockwise on a convex chain, this is counterclockwise most.

[23] The vertex A_{2i+1} is collinear with $A_{2i}t$, so there is no edge to mark between A_{2i} and A_{2i+1}.

Lemma 4.6 *Immediately after the clockwise (counterclockwise) partitioning/modified Graham scan processes vertex A'_j of edge $A'_jA'_{j+1}$ of G', we can obtain, in* O(1) *time:*

- $C^-(A'_j)$ *(*$C^+(A'_j)$*) from* C *and*
- $R^-(u)$ *(*$R^+(u)$*) and* $s^-(u)$ *(*$s^+(u)$*) from* C*, for* $u \in \text{INTERIOR}(A'_jA'_{j+1})$ *if* j *is odd.*

Proof: W.l.o.g. we treat only the clockwise scan. The first part is immediate because $C = C^-(A'_j)$. To establish the second part of the lemma, let $j = 2i-1$ and consider $u \in \text{INTERIOR}(A'_{2i-1}A'_{2i})$ for $\alpha \le i \le \beta'$. There are no vertices of $A_1A_2, \ldots, A_{2n-1}A_{2n}$ or partitioning points in $\text{INTERIOR}(A'_{2i-1}A'_{2i})$, so C does not change after processing A'_{2i-1} and before processing A'_{2i}. There are two cases. In the first case, A'_{2i-1} is a vertex of G; in the second case it is not. We start with the first case. The last three vertices of C are A_g, A_h, and A'_{2i-1}. The absence of a partitioning point in $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ implies that u is left of $\text{LINE}(A_gA_h)$ and that C up to and including A_h is a subset of $R^-(u)$. If u is on or to the right of $\text{LINE}(A_hA'_{2i-1})$, then $R^-(u)$ is equal to C with A'_{2i-1} removed. Otherwise, u is on the left of $\text{LINE}(A_hA'_{2i-1})$ and therefore $R^-(u) = C$. In the second case, the last two vertices of C are A_g and A_h. The absence of a partitioning point in $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ implies that u is left of $\text{LINE}(A_gA_h)$ and that C up to and including A_h is a subset of $R^-(u)$. Since A'_{2i-1} is not a vertex of G, u is right of $\text{LINE}(A_hA'_{2i-1})$. This implies that $R^-(u) = C$. In both cases, $R^-(u)$ (and $s^-(u)$, the last vertex of $R^-(u)$), can be obtained from C in O(1) time; this establishes the second part of the lemma. ■

Corollary 4.7 *Given* $p, q \in \text{INTERIOR}(A'_{2i-1}A'_{2i})$ *for* $\alpha \le i \le \beta'$*,* p *and* q *are hull equivalent.*

Closest Point Update.

Lemma 4.8 *Let* Q *be a convex polygon and* t *be a point outside* Q*. Parameterize the portion of the boundary of* Q *which is visible to* t *by the piecewise linear function* $\gamma(\tau)$*,* $\tau \in [0,1]$*, where* τ *increases clockwise about* t*. The distance function* $d_Q(\tau) = \Delta(t, Q(\gamma(\tau)))$ *is unimodal.*

Proof: If suffices to prove that there exists $\tau \in [0,1]$ such that, for $\rho > \eta \ge \tau$, $d_Q(\rho) \ge d_Q(\eta)$ and, for $\rho < \eta \le \tau$, $d_Q(\rho) \le d_Q(\eta)$. As the arguments are similar in both cases, we prove the first case. To simplify notation, let $Q(\tau)$ denote $Q(\gamma(\tau))$. Let $Q(\tau)$ be a point where the largest circle S about t (empty with respect to Q) touches Q. As Q is convex, $Q(\tau)Q(\eta)Q(\rho)$ is either collinear or a left turn. Therefore, $Q(\rho)$ is on or to the left of the ray $Q(\tau)Q(\eta)$. Furthermore, $Q(\rho)$ is visible from t, so $Q(\rho)$ is right of $tQ(\eta)$. Together, these two statements about $Q(\rho)$ confine it to a wedge W. Now, let N be the line through $Q(\eta)$ which is perpendicular to the ray $Q(\eta t)$. The emptiness of the circle S implies that $Q(\eta)$ is not closer to t than $Q(\tau)$; this implies that $Q(\tau)$ is on the same side of N as t. This, in turn, implies that W is on the opposite side of N from t. Therefore, $d_Q(\eta) \ge d_Q(\rho)$. ■

Lemma 4.9 *In* $O(|U|)$ *time it is possible to precompute* $r^-(u)$ *and* $r^+(u)$ *for* $u \in \text{INTERIOR}(A'_{2i-1}A'_{2i})$ *of* G'*,* $\alpha \le i \le \beta'$*, as well as* $c^-(u)$ *and* $c^+(u)$ *for each vertex* u *of* G'*.*

Proof: The modified Graham scan repeatedly either adds or deletes a single edge from the current convex chain C. Suppose we know the nearest point of C to t. If we add an edge to C, we can update the nearest point in O(1) time. If we delete an edge from C, Lemma 4.8 implies that we can update the nearest point in O(1) time. Now, the proof of Lemma 4.6 shows that $R^-(u)$ is either equal to C or else equal to C with the last edge removed. Therefore, we can obtain $r^-(u)$ and $r^+(u)$ from the closest point of C to t in O(1) time. The proof of Lemma 4.6 also shows that, for vertex $u = A'_{2i} \in G'$, $C^-(A'_{2i}) = C^-(A'_{2i+1}) = C$. We can therefore obtain $c^-(u)$ and $c^+(u)$ for vertex u in O(1) time. ■

4.1.4 Locating the Intersection Point along an Edge.

Lemma 4.10 *If* $L_{\max}$ *intersects* $\text{INTERIOR}(A'_{2i-1}A'_{2i})$*, then we can find the intersection point in* O(1) *time in a real arithmetic model, or in* $O(\log(\log \epsilon^{-1}))$ *time in a rational arithmetic model with accuracy* ϵ*.*

Proof: Consider $u \in \text{INTERIOR}(A'_{2i-1}A'_{2i})$. The point $c^-(u)$ is either: 1) on $R^-(u)$ (and therefore equal to $r^-(u)$), 2) in the interior of $s^-(u)u$, or 3) equal to u. A similar statement holds for $c^+(u)$. We show that $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ contains a constant number of subsegments, each having constant $c^-(u)$ and $c^+(u)$ type. Consider $c^-(u)$. We treat type (1) first. Corollary 4.7 shows that $r^-(u)$ does not change within $\text{INTERIOR}(A'_{2i-1}A'_{2i})$. Furthermore, the monotonicity of the distance function $d^-(u)$ (see Section 4.1.2), guarantees that once $c^-(u)$ moves beyond $s^-(u)$ onto $\text{INTERIOR}(s^-(u)u)$, then it cannot move back onto $s^-(u)$. Therefore, if type (1) applies to any point in $\text{INTERIOR}(A'_{2i-1}A'_{2i})$, then either $c^-(u)$ is the same for all points u in $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ or else there is one point w along it where $c^-(u)$ changes from $s^-(u)$ to a point on $(s^-(u)u]$. The point w, if it exists, is the intersection of $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ with the line through $s^-(u)$ perpendicular to $s^-(u)t$. The nearest point of $(s^-(u)u]$ to t becomes u when tu is perpendicular to $ts^-(u)$. The locus of points for which tu is perpendicular to $ts^-(u)$ is the circle with diameter $ts^-(u)$. This circle has up to two intersections with $\text{INTERIOR}(A'_{2i-1}A'_{2i})$, and $c^-(u)$ is at u in between the two points of intersection. Thus, changes in the type of $c^-(u)$ partition $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ into at most four pieces. A similar argument shows that changes in the type of $c^+(u)$ partition $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ into at most four pieces. We conclude that $\text{INTERIOR}(A'_{2i-1}A'_{2i})$ contains a constant number of subsegments, each having constant $c^-(u)$ and $c^+(u)$ type.

In O(1) time we can retrieve, for a given i, the information necessary to locate the endpoints of the constant-type subsegments. This relies on Lemma 4.6 and Lemma 4.9. The former gives us $s^-(u)$ and $s^+(u)$ and the latter gives us $r^-(u)$ and $r^+(u)$. From this information, the endpoints of the subsegments can be located in O(1) time for a given i, because they require only a constant number of algebraic operations, such as intersecting a line and/or circle with a line.

Now we discuss finding the intersection point on a constant-type subsegment. Parameterize the subsegment by the linear function $\gamma(\tau)$, for $\tau \in [0,1]$. Consider $c^-(\gamma(\tau))$. By Lemma 4.4, $d^-(\gamma(\tau))$ is the distance from $c^-(\gamma(\tau))$ to t. In case (2),

$$d^-(\gamma(\tau)) = \frac{(t-q) \times (\gamma(\tau)-q)}{|\gamma(\tau)-q|},$$

a continuous algebraic function. In case (3), $d^-(\gamma(\tau)) = |\gamma(\tau)-t|$; this is also continuous. Analogous formulas hold for $d^+(\gamma(\tau))$. We can intersect two pieces using algebraic operations. The case which dominates the running time occurs when both $c^-(\gamma(\tau))$ and $c^+(\gamma(\tau))$ are of type (2). In this case, one can find the intersection point by proceeding algebraically at first and then numerically. Setting $d^+(\gamma(\tau)) = d^-(\gamma(\tau))$, squaring, and clearing fractions produces a fourth-degree polynomial equation in τ. The intersection point for $\tau \in [0,1]$ is the root (in this interval) of the fourth-degree polynomial. Under a real arithmetic model, we can represent the root *exactly* as an algebraic number. This requires the containment algorithm to manipulate algebraic numbers. To avoid this, one can operate on a nearby rational approximation to the root. This can be obtained using, for example, Newton's method. The quadratic convergence of Newton's method adds only $\log(\log \epsilon^{-1})$ to the running time of the algorithm, where ϵ is the accuracy. ■

4.1.5 Running Time and Correctness.

Theorem 4.11 *The line $L_{\max}$ can be computed in $O(|U|)$ time in a real arithmetic model, and in $O(|U| + \log(\log \epsilon^{-1}))$ time in a rational arithmetic model, where ϵ is the accuracy.*

Proof: First we establish the running time. By Lemma 4.2, the edge list E can be constructed in $O(|U|)$ time and contains $O(|U|)$ edges. By Lemma 4.5, G can be partitioned into G' in $O(|U|)$ time and G' contains $O(|U|)$ edges. By Lemma 4.6, $O(|U|)$ preprocessing allows us to obtain $d^-(u)$ and $d^+(u)$ in O(1) time for a vertex u of G'. Lemma 4.3 allows us to find the edge in G' associated with $L_{\max}$ using binary search on the value of the edge index of G'. The running time of the binary search is dominated by $O(\log(|U|))$ due to Lemma 4.6. If the binary search returns an edge instead of a vertex, Lemma 4.10 allows us to find $u \in \text{INTERIOR}(A'_{2i-1}A'_{2i})$ such that $d(u)$ maximizes $d(L)$ for $L \in \mathcal{L}$ in O(1) in a real arithmetic model and $O(\log(\log \epsilon^{-1}))$ in a rational arithmetic model with accuracy ϵ. This allows us to find $L_{\max}$ in $O(|U|)$ time in a real arithmetic model and in $O(|U| + \log(\log \epsilon^{-1}))$ time in the rational arithmetic model.

Now we establish correctness. Lemma 4.2 shows that CLOSURE($\mathcal{L}$) = $\{\text{LINE}(tu) | u \in e \text{ for } e \in G\}$. Partitioning G into G' does not eliminate any $L \in \mathcal{L}$ from consideration. Binary search determines if L passes through a vertex of G' or the interior of an edge in G'. The correctness of the binary search is guaranteed by Lemma 4.3. The distances $d^-(u)$ and $d^+(u)$ are calculated correctly for each vertex u of G' during the binary search because Lemma 4.4 guarantees that $d^-(u) = \Delta(t, C^-(u))$, Lemma 4.6 shows how to obtain $C^-(u)$, and Lemma 4.9 gives $c^-(u)$. If $L_{\max}$ passes through the interior of an edge of G', then the proof of Lemma 4.10 guarantees correct calculation of $L_{\max}$. ■

4.2 Subdivision for Combinatorial Version.

Combinatorial subdivision is used to establish the running time of the combinatorial version of LP containment. It selects some pair i, j such that U_{ij}^{init} has an edge e which intersects the interior of C_{ij}. The subdivision algorithm then splits C_{ij} using LINE(e). If there is no such pair i, j and edge e, then a single evaluation will find a solution because each $C_{ij} \subseteq U_{ij}^{\text{init}}$ for $0 \le i < j \le k$ (see the discussion at the start of Section 4).

It is essential to the combinatorial splitting algorithm that we be able to quickly determine if some part of some edge e of U_{ij}^{init} lies in the interior of C_{ij}. If so, then we can use the line containing e as the next splitting line. When we subdivide $\mathcal{C}$ into $\mathcal{C}^+$ and $\mathcal{C}^-$, the time to answer this *interior edge* query for each C_{gh}^+ and C_{gh}^- in $\mathcal{C}^+$ and $\mathcal{C}^-$ must be at worst linear in $|C_{gh}^0|$ and logarithmic in $|\mathcal{F}|$. This section describes how we can accomplish this goal. It gives an algorithm for finding i, j, and e using time logarithmic in the number of new vertices in $\mathcal{F}$ (the number of vertices in $\mathcal{F}^0$, see Section 2.2.3).

It is possible to preprocess U_{ij}^{init} so that a *ray-shooting* query can be answered in $O(\log |U_{ij}^{\text{init}}|)$ time. A ray-shooting query RAYSHOOT($p, v, U_{ij}^{\text{init}}$) asks the following question: given a point p and vector v, determine the minimum value of $t > 0$ such that $p + tv$ lies on the boundary of U_{ij}^{init}. If there is no intersection, the answer is "none" or $t = \infty$. It does not really matter how much the preprocessing costs because we only have to do it once for each U_{ij}^{init}, $0 \le i < j \le k$. Using standard techniques of computational geometry, it clearly can be accomplished in time polynomial in $|U_{ij}^{\text{init}}|$. Note that $|U_{ij}^{\text{init}}|$ is $O(m^2 n^2)$ or $O(m^4)$, so $\log |U_{ij}^{\text{init}}|$ is $O(\log mn)$. Incidentally, our bound for $\log |C_{ij}|$ is the same as $\log |\mathcal{F}|$, which by Section 5.2 is $O(k \log kmn)$. Therefore, we can substitute $\log |\mathcal{F}|$ for $\log |U_{ij}^{\text{init}}|$ or $\log |C_{ij}|$ in our upper bounds.

Given an edge ab (of C_{ij}), a single ray-shooting query determines if ab intersects the boundary of U_{ij}^{init}: check if RAYSHOOT($a, b - a, U_{ij}^{\text{init}}$) ≤ 1. In particular, we are interested in determining if an edge e of U_{ij}^{init} *crosses into* C_{ij} through ab: e intersects ab and e intersects the interior of C_{ij}. We assume we have resolved the degenerate cases so that RAYSHOOT only captures edges which cross into U_{ij}^{init} and are not, for example, merely collinear with ab.

Assume below in Lemma 4.12 and Algorithm I that U_{ij}^{init} has only one simply connected component. We deal with multiple components and/or nonsimple connectivity later in Corollary 4.13 and Algorithm II. Recall that $C_{ij}^{\text{init}} = \text{CH}(U_{ij}^{\text{init}})$. In the following lemma, $\overline{U_{ij}^{\text{init}}}$ refers to the *regularized* complement of U_{ij}^{init}: the complement plus the boundary of U_{ij}^{init}. We say that the boundary of U_{ij}^{init} *crosses into* C_{ij} if some edge e crosses into C_{ij} through some edge ab.

Lemma 4.12 *If U_{ij}^{init} is simple, then any convex set $C_{ij} \subset C_{ij}^{\text{init}}$ must satisfy one of the following: $C_{ij} \subseteq U_{ij}^{\text{init}}$, $C_{ij} \subseteq \overline{U_{ij}^{\text{init}}}$, or the boundary of U_{ij}^{init} crosses into C_{ij}.*

Proof: Clearly, the boundary of U_{ij}^{init} touches the boundary of $C^{\text{init}} = \text{CH}(U_{ij}^{\text{init}})$. Therefore it touches the boundary of or have some point exterior to $C_{ij} \subseteq C^{\text{init}}$. If the boundary of U_{ij}^{init} intersects the interior of C_{ij}, then it must cross into C_{ij} because it is simple.

Therefore, if the boundary of U_{ij}^{init} does not cross into C_{ij}, it must be a subset of $\overline{C_{ij}}$. The interior of C_{ij} does not intersect the boundary of U_{ij}^{init} and therefore it is a subset of either the interior or exterior of U_{ij}^{init}. Taking the closure, the lemma follows. ■

Algorithm I: For each edge ab of each C_{gh} of each restricted hypothesis $\mathcal{C}$, the algorithm computes if any edge e of U_{gh}^{init} crosses into C_{gh} using RAYSHOOT. The algorithm maintains a linked list of edges of U_{gh}^{init} which have such a crossing edge. For the restricted version of $\mathcal{C}^{\text{init}}$ (the restriction of the root hypothesis), the algorithm computes this information from "scratch". When $\mathcal{C}$ is split and restricted into $\mathcal{C}^+$ and $\mathcal{C}^-$ and thus each C_{gh} of $\mathcal{C}$ is split-restricted into C_{gh}^+ and C_{gh}^-, the algorithm avoids recomputing RAYSHOOT for unaffected edges. It computes RAYSHOOT only for edges of C_{gh}^0 and for new edges which appear in C_{gh}^+ and C_{gh}^-. Just as we split the edges of C_{gh} into chains and reused these chains in C_{gh}^+ and C_{gh}^-, this algorithm splits the linked list of crossed edges and reuses them.

Running Time: Since a RAYSHOOT query uses $O(\log|\mathcal{F}|)$ time, the cost of computing the list of crossed edges from "scratch" for C_{ij} is $O(|C_{ij}|\log|\mathcal{F}|)$. When the algorithm acts on C_{gh}^+ and C_{gh}^-, it only runs RAYSHOOT on each new edge, and there are only $|C_{gh}^0|$ of these. The total time for these calls to RAYSHOOT is $|C_{gh}^0|\log|\mathcal{F}|$. Finally, the time to create, split, and merge linked lists of crossed edges is no more than the time to compute C_{gh}^+ and C_{gh}^-: $O(|C_{gh}^0|\log|\mathcal{F}|)$.

Corollary 4.13 *If each connected component of the boundary of U_{ij}^{init} touches the boundary of $\text{CH}(U_{ij}^{\text{init}})$, then Lemma 4.12 still holds.*

Proof: The only way the proof used the fact that U_{ij}^{init} has a single component was the fact that the boundary of U_{ij}^{init} must touch the boundary of C_{ij}^{init}. ■

Algorithm II: If U_{ij}^{init} has multiple component and/or non-simply connected components, the algorithm modifies each component of the boundary of U_{ij}^{init} that does not touch $\text{CH}(U_{ij}^{\text{init}})$. First, it computes the maximum-x vertex v of that component. Then it chooses one of the edges e with v as an endpoint. It creates an edge e' collinear with e and on the "other side" (greater x) of v. It extends e' until it hits some other component of the boundary of U_{ij}^{init} or it hits the boundary of $\text{CH}(U_{ij}^{\text{init}})$.

Running Time: Clearly, Algorithm II no more than doubles the number of edges of U_{ij}^{init}. Furthermore, e' lies on the same line as e, and therefore it does not introduce any new potential splitting lines. Splitting on e' is equivalent to splitting on e.

5 Running Time, Generalization, and Conclusion.

This section first discusses running time. Section 5.1 gives the practical running times of our implementation of LP containment. Section 5.2 establishes the theoretical running time of the combinatorial version. Section 5.3 generalizes LP containment to solve minimal enclosure problems. Section 5.4 summarizes our results and draws conclusions.

5.1 Practical Running Time of LP Containment.

We implemented the LP containment algorithm using the CPLEX 2.1 linear programming system (CPLEX Optimization, Inc.).[24] Most of the examples we selected are from the apparel industry. We produced each feasible containment example from an existing apparel layout by removing a group of neighboring items. To create an infeasible example, we moved some of the remaining items a small distance into the container. No examples were dropped.

Figure 7 illustrates the effect of our new LP restriction algorithm. The figure shows two two-component U_{0j}s. The boundaries of the unrestricted U_{0j}s are outlined in black, and the restricted U_{0j}s are depicted as black filled regions. Restriction eliminates an entire component in the U_{0j} on the left, and it decreases the area of the U_{0j} region on the right by more than 90%.

[24] Currently, we are using CPLEX 3.0, but the experiments were all run using CPLEX 2.1.

Type	GEOM	NAIVE LP	LP
3NN	19	22	9
3NN	2	4	4
4CC	25	1	2
4CC	21	10	11
4CN	113	84	4
4NN	155	19	33
4NN	80	24	35
4NN	35	43	15
4NN	319	89	10
5CC	126	6	3
5NN	> 300	> 300	24
6CC	75	53	5
6CN	> 300	> 300	184
6NN	> 300	> 300	86
6NN	> 900	> 900	593
7CC	823	162	7
7CN	> 300	> 300	72
7NN	> 600	> 600	176
8CC	> 300	> 300	34
10CC	-	-	14
10NN	-	-	183

Table 1: Running times for feasible examples in seconds

Type	GEOM	NAIVE LP	LP
4NN	47	152	135
4NN	138	3037	493
4NN	> 600	> 600	416
5NN	> 600	> 600	103

Table 2: Running times for infeasible examples in seconds

We compared the performance of our new LP containment algorithm with two of our previous algorithms [5, 6]: the geometric algorithm and the naive LP algorithm (see Section 1.4). All three implementations apply geometric restriction (Equation 3 on page 4) to the initial hypothesis (Equation 1 on page 4). Table 1 gives running times for feasible containment problems. Running times are in seconds on a 50 MHz SPARCstation.[25] Table 2 gives running times for infeasible containment problems. Our new algorithm clearly outperforms both of the other algorithms. For five or more polygons, the difference is dramatic.[26] Table 3 compares the number of hypotheses evaluated by the naive LP containment algorithm and the new LP containment algorithm on some infeasible examples with $k = 4$ and no restriction applied. This experiment isolates the benefits of overlap minimization followed by distance-based subdivision. Table 3 clearly demonstrates that the new evaluation/subdivision combination is much "smarter" than naive subdivision, which simply extends an edge of a U_{ij}. Figure 8 and Figure 9 depict configurations obtained by our algorithm.

[25] SPARCstation is a trademark of SPARC, International, Inc., licensed exclusively to Sun Microsystems, Inc.

[26] Although the running time of LP containment is substantially less than that of geometric containment for $k > 4$, the number of hypotheses examined by geometric containment can be less than the number examined by LP containment, as discussed in Section 1.5.1. That discussion also points out that geometric restriction is currently much slower than LP restriction.

Type	NAIVE LP	LP
4CN	32	27
4CN	42	11
4NN	493	153
4NN	1736	311
4NN	153	103
4NN	436	323

Table 3: Number of hypotheses evaluated for infeasible problems (no hypothesis restriction)

5.2 Theoretical Running Time of Combinatorial Version.

In this section we prove a running time bound for the combinatorial version of LP containment. This version uses the combinatorial LP restriction algorithm of Section 2.2.3, the evaluation algorithm of Section 3.5, and the combinatorial subdivision algorithm given in Section 4.2. The running time is faster than that of naive LP containment (Section 2.1.1) and within a log factor of the $\Omega((mn)^{2k})$ lower bound established by Milenkovic [27].

Theorem 5.1 *The combinatorial version of LP containment has running time in*

$$\mathrm{O}\left(\frac{(6kmn+k^2m^2)^{2k}}{(k-5)!}\log kmn\right).$$

Proof: Recall from Section 1.3 that the container $\overline{P_0}$ has n vertices, P_i, $1 \le i \le k$, has at most m vertices, and the Minkowski sum $A \oplus B$ has $\mathrm{O}(|A|^2|B|^2)$ vertices. For $i = 0$ or $j = 0$, U_{ij}^{init} has $\mathrm{O}(m^2n^2)$ edges. However, each edge corresponds to a translation $P_i + t_i$ such that a vertex of $P_i + t_i$ touches an edge of P_0 or an edge of $P_i + t_i$ touches a vertex of P_0. There are at most $2mn$ ways this can happen. Therefore, the $\mathrm{O}(m^2n^2)$ edges of U_{ij}^{init} lie on (are covered by) only $2mn$ lines. Similarly, for $1 \le i < j \le k$, U_{ij}^{init} has $\mathrm{O}(m^4)$ edges which lie on $2m^2$ lines. Let C_{ij} be the convex hull of the initial U_{ij}^{init}. Each line of U_{ij}^{init} can contribute at most two vertices to C_{ij}. Furthermore, for $1 \le i < j \le k$, U_{ij}^{init} is unbounded, and so C_{ij} is the entire plane. Thus, the initial C_{ij} polygons have a total of $4kmn$ vertices (and edges). Each edge-line contributes a linear constraint to some of the CLPs, hence we claim: in total, the CLPs have as input at most $6kmn + k^2m^2$ (actually $6kmn + k(k-1)m^2$) linear constraints.

Since there are $2k$ degrees of freedom (remember $t_0 = (0,0)$), each CLP-generated vertex corresponds to a choice of $2k$ constraints. Hence, $\mathcal{F}$ (and each C_{ij}) has $\mathrm{O}((6kmn + k^2m^2)^{2k}/k!)$ vertices, and this in turn implies that $\log|\mathcal{F}|$ and $\log|\mathcal{C}|$ are $\mathrm{O}(k \log kmn)$.

Section 3.5 shows that evaluation requires $\mathrm{O}(k^3 \log|\mathcal{U}^{\mathrm{init}}|)$ time per *new* vertex of $\mathcal{F}$, which is $\mathrm{O}(k^3 \log kmn)$ time per new vertex of $\mathcal{F}$. Lemma 2.3 shows that using the simplex method (Section 2.2.2) to solve the CLP allows us to restrict in time $\mathrm{O}(k^3 + k^2 \log|\mathcal{C}| + k^2 \log|\mathcal{F}|)$ per new vertex of $\mathcal{F}$, which is $\mathrm{O}(k^3 \log kmn)$ time per new vertex of $\mathcal{F}$. Now, recall that combinatorial subdivision (Section 4.2) selects some pair i, j such that U_{ij}^{init} has an edge which intersects the interior of C_{ij}. Section 4.2 shows that the time to answer the interior edge query for a given C_{ij} is no more than the time to compute the restricted region C_{ij}. Hence, the total subdivision time is at most as large as the total restriction time.

When we restrict immediately after subdividing, we must compute the C_{gh}^0 polygons. However, all the vertices of all the C_{gh}^0 polygons involve the "splitting" constraint $t_j - t_i \in L$, and this constraint is seen for the first time. Hence all the vertices are "new." Of course, since there are $k(k+1)/2$ C_{gh}^0 polygons, we visit each new vertex $\mathrm{O}(k^2)$ times.

The total time of all the evaluation, subdivision and restriction is therefore:

$$\mathrm{O}\left(k^2 \cdot \frac{(6kmn+k^2m^2)^{2k}}{k!} \cdot k^3 \log kmn\right) = \mathrm{O}\left(\frac{(6kmn+k^2m^2)^{2k}}{(k-5)!}\log kmn\right).$$

■

5.3 Generalizing LP Containment to Find Minimal Enclosures.

Here we generalize LP containment to solve two types of minimal enclosure problems. Section 5.3.1 and Section 5.3.2 show how to obtain the minimal area enclosing square and rectangle, respectively, for k translating polygons. In both cases the sides of the enclosure are parallel to the x and y axes. In both cases we use binary search, but the parameter we search on is different in each case. The area of a square varies monotonically with a single "size" parameter such as the length of a side. Thus, binary search on a size parameter can be used to find the minimal enclosing square. To decide if the polygons fit in a square of a given size, we create a square of that size and then solve containment. A rectangle of arbitrary aspect ratio, however, does not have this monotonicity property and so we cannot search on a size parameter. Instead, we search on the area, asking, for a given area α, if the polygons fit in a rectangle of area α. This question cannot be answered directly by calling containment. However, we generalize LP containment so that it may answer the question.

5.3.1 Minimum Area Square Enclosure.

Given any containment algorithm, we can find the smallest square[27] that encloses translated copies of $P_1, P_2, \ldots, P_k$ [4]. For a numerical algorithm: use binary search on the size of the square. Figure 10 shows the minimal square for four polygons; this was obtained using a numerical algorithm. For a combinatorial algorithm: given a feasible square, use *compaction* [24, 29, 23] to "shrink" the square to a local minimum. Any more shrinking will guarantee either infeasibility (if it is the global minimum) or at least a change in combinatorial structure. One can use symbolic perturbation [12, 35] to shrink the square an infinitesimal amount and force infeasibility or a change in structure. Compact again and repeat. In practice, we would just compact the feasible squares and then continue the (numerical) binary search. Doubtless there is a combinatorial way to do binary search using parametric searching.

5.3.2 Minimum Area Rectangle.

One cannot use containment and binary search to find the minimum area rectangular enclosure (of *arbitrary* aspect ratio). We need to solve the following: given shapes $P_1, P_2, \ldots, P_k$ and area α, is there a rectangle of area α which contains translated copies of the shapes? With this, one can find the minimum α numerically by binary search.

Given $P_1, P_2, \ldots, P_k$, let $P_0 = \{(x, y) \mid x \leq 0 \text{ or } y \leq 0\}$ and let $P_{k+1} = \{(x, y) \mid x \geq 0 \text{ or } y \geq 0\}$. Calculate U_{ij}, $0 \leq i, j \leq k+1$ according to Equation 1 (page 4). Since P_0 and P_{k+1} intersect no matter where you put them, $U_{0,k+1}$ will be empty. "Artificially" set $U_{0,k+1} = \{(x, y) \mid xy \leq \alpha\}$. We claim 1) the hypothesis $\mathcal{U}$ has a valid configuration if and only if there is a rectangle of area α enclosing $P_1, P_2, \ldots, P_k$, and 2) only one minor modification to our algorithm is required. If $t_{k+1} - t_0$ is outside $U_{0,k+1}$, then instead of taking the closest edge, we take the closest point on the hyperbola. We set up the OLP using the tangent line at that point. We have not tested this algorithm, but we expect it to have quadratic convergence once it "homes in" on a configuration.

5.4 Conclusion.

This paper has presented a new algorithm for two-dimensional translational containment based on mathematical programming principles. It also gives algorithms, based on the same approach, for finding the minimal enclosing square and the minimal area enclosing rectangle for a collection of translating polygons. All of these algorithms provide exact solutions.

[27] Or indeed the smallest similar copy of any star-shaped polygon.

In our experiments, our new containment algorithm clearly outperforms purely geometric containment algorithms. For data sets from the apparel industry, it can solve containment for up to ten nonconvex polygons in a nonconvex container in practice. This important practical result is achieved by following our restrict/evaluate/subdivide algorithmic paradigm, whose steps correspond to tightening upper and lower bounds on the amount of overlap in a layout and introducing cutting planes when necessary. Some of our new algorithms for the individual steps in the paradigm use linear programming as well as techniques from computational geometry.

Combining the power of linear programming with techniques from computational geometry gives us a translational containment "tool" which can be used to perform many tasks related to layout, including ones for which each item can have a discrete set of orientations. For example, we can pack multiple containers by first generating groups which fit into each container and then matching groups to containers [4]. This type of problem arises in a variety of settings, such as in the second phase of a two-phase layout problem. In addition to using the containment tool directly, one can apply our *approach* to containment to solve a variety of previously unsolved problems. Our solutions to minimal enclosure problems in this paper and in [4] illustrate this.

We believe that combining mathematical programming with computational geometry is a fruitful approach for tackling many NP-hard geometric optimization problems. Our work with linear programming and our brief experiments so far with mixed-integer programming are just the beginning of our exploration. Recent results in integer programming may well provide a starting point for our future work.

References

[1] M. Adamowicz and A. Albano. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design*, 8:27–33, 1976.

[2] F. Avnaim. *Placement et déplacement de formes rigides ou articulées.* PhD thesis, Université de Franche-Comté, France, 1989.

[3] F. Avnaim and J. Boissonnat. Simultaneous Containment of Several Polygons. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 242–250, 1987.

[4] K. Daniels. *Containment Algorithms for Nonconvex Polygons with Applications to Layout.* PhD thesis, Harvard University, 1995.

[5] K. Daniels and V. J. Milenkovic. Multiple Translational Containment, Part I: An Approximate Algorithm. *Algorithmica, special issue on Computational Geometry in Manufacturing, accepted, subject to revisions.*

[6] K. Daniels and V. J. Milenkovic. Multiple Translational Containment: Approximate and Exact Algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 205–214, 1995.

[7] K. Daniels, V. J. Milenkovic, and Z. Li. Multiple Containment Methods. Technical Report 12–94, Center for Research in Computing Technology, Division of Applied Sciences, Harvard University, 1994.

[8] O. Devillers. Simultaneous Containment of Several Polygons: Analysis of the Contact Configurations. Technical Report 1179, INRIA, 1990.

[9] K. A. Dowsland and W. B. Dowsland. Packing Problems. *European Journal of Operational Research*, 56:2 – 14, 1992.

[10] D.T.Lee. On finding the convex hull of a simple polygon. *Int'l J. Comput. and Infor. Sci.*, 12(2):87–98, 1983.

[11] H. Dyckhoff. A typology of cutting and packing problems. *European Journal of Operations Research*, 44:145–159, 1990.

[12] H. Edelsbrunner and E. P. Mücke. Simulation of Simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9:66–104, 1990.

[13] S. Fortune. A Fast Algorithm for Polygon Containment by Translation. In *Proceedings of the 12th Colloquium on Automata, Languages, and Programming*, pages 189–198. Springer-Verlag, 1985.

[14] P. Gilmore and R. Gomory. Multistage Cutting Stock Problems of Two and More Dimensions. *Operations Research*, 13:94–120, 1965.

[15] H. El Gindy and D. Avis. A Linear Algorithm for Computing the Visibility Polygon from a Point. *Journal of Algorithms*, 2:186–197, 1981.

[16] R. Graham and F. Yao. Finding the convex hull of a simple polygon. *Journal of Algorithms*, 4(4):324–331, 1983.

[17] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Lett.*, 1:132–133, 1972.

[18] L. Guibas, L. Ramshaw, and J. Stolfi. A Kinetic Framework for Computational Geometry. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 100–111, 1983.

[19] M. Haims and H. Freeman. A multistage solution of the template-layout problem. *I.E.E.E. Transactions on Systems Science and Cybernetics*, 6:145, 1970.

[20] B. Joe and R. B. Simpson. Corrections to Lee's Visibility Polygon Algorithm. *BIT*, 27:458–473, 1987.

[21] A. Kaul, M.A. O'Connor, and V. Srinivasan. Computing Minkowski Sums of Regular Polygons. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, Vancouver, British Columbia, 1991.

[22] D. T. Lee. Visibility of a Simple Polygon. *Computer Vision, Graphics, and Image Processing*, 22:207–221, 1983.

[23] Z. Li. *Compaction Algorithms for Non-Convex Polygons and Their Applications.* PhD thesis, Harvard University, Division of Applied Sciences, 1994.

[24] Z. Li and V. J. Milenkovic. The Complexity of the Compaction Problem. In *Proceedings of the 5th Canadian Conference on Computational Geometry*, pages 7–11, Waterloo, Canada, 1993.

[25] D. McCallum and D. Avis. A linear algorithm for finding the convex hull of a simple polygon. *Info. Proc. Lett.*, 9:201–206, 1979.

[26] V. J. Milenkovic. Multiple Translational Containment, Part II: Exact Algorithms. *Algorithmica, special issue on Computational Geometry in Manufacturing, accepted, subject to revisions.*

[27] V. J. Milenkovic. Translational Polygon Containment and Minimal Enclosure using Linear Programming Based Restriction. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (submitted)*, 1996.

[28] V. J. Milenkovic, K. Daniels, and Z. Li. Automatic Marker Making. In *Proceedings of the 3rd Canadian Conference on Computational Geometry*, 1991.

[29] V. J. Milenkovic and Z. Li. A Compaction Algorithm for Nonconvex Polygons and Its Application. *European Journal of Operations Research*, 84:539–560, 1995.

[30] H. Minkowski. Volumen und Oberfläche. *Mathematische Annalen*, 57:447–495, 1903.

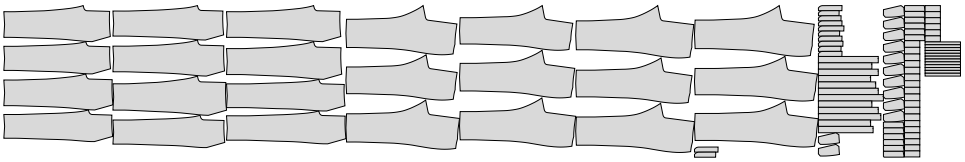
[31] F. Preparata and M. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[32] J. Serra. *Image Analysis and Mathematical Morphology*, volume 1. Academic Press, New York, 1982.

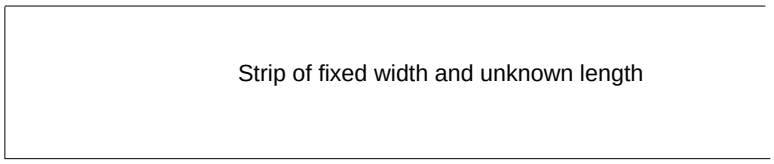
[33] J. Serra, editor. *Image Analysis and Mathematical Morphology*, volume 2: Theoretical Advances. Academic Press, New York, 1988.

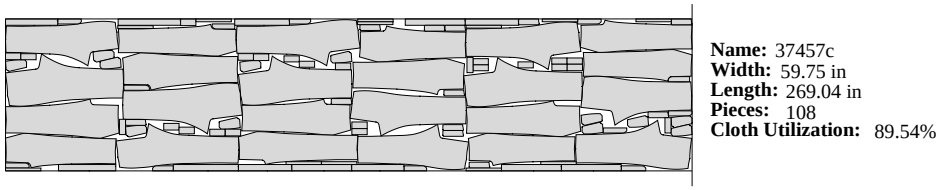
[34] P. E. Sweeney and E. R. Paternoster. Cutting and Packing Problems: A Categorized, Application-Oriented Research Bibliography. *Journal of the Operational Research Society*, 43(7):691–706, 1992.

[35] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. *Journal of Computer and System Sciences*, 40:2–18, 1990.



Set of polygonal parts





Packing with 180 degree rotations and xy-flips allowed

Figure 1: A marker making task in the apparel industry

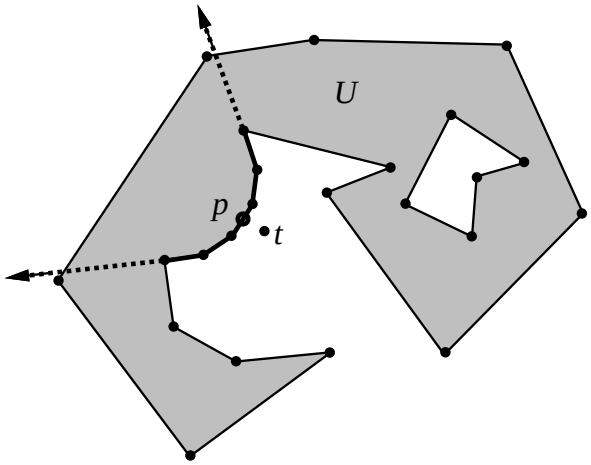


Figure 2: Constructing the boundary of $I(t, U)$

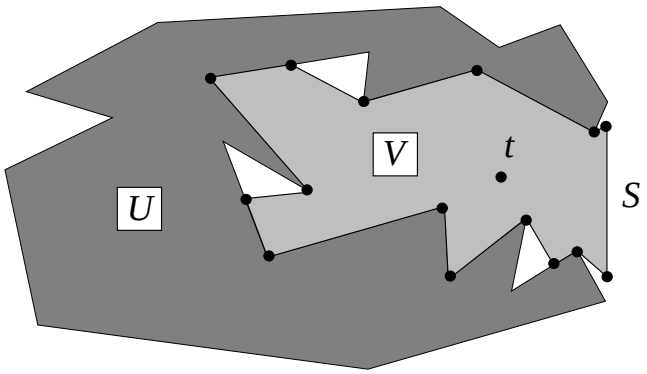


Figure 3: Case 3: U, t, visibility polygon V, and $S \subset \text{BOUNDARY}(\text{CH}(U))$.

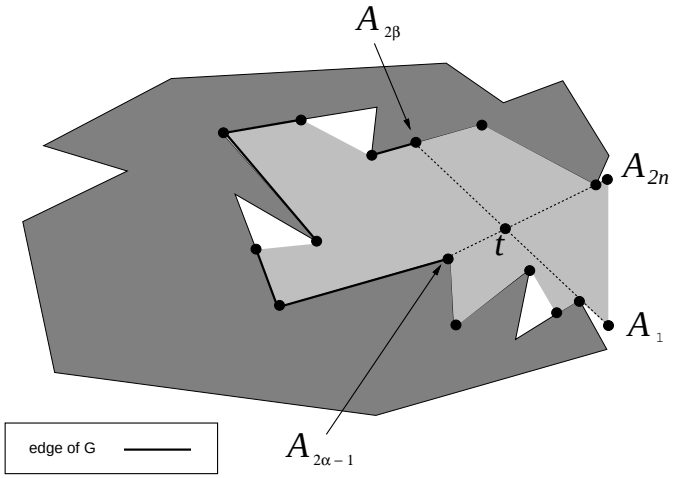


Figure 4: $A_{2\alpha-1}$, $A_{2\beta}$ and edges of G.

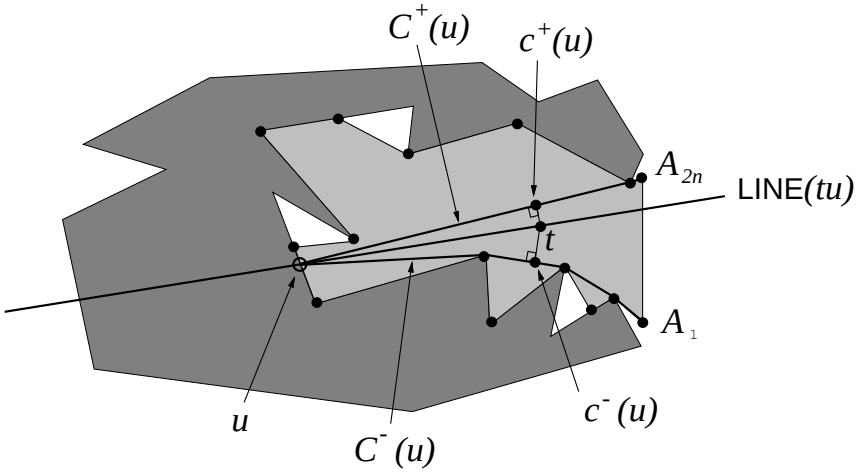


Figure 5: LINE(tu), $C^-(u)$, $C^+(u)$, $c^-(u)$, and $c^+(u)$.

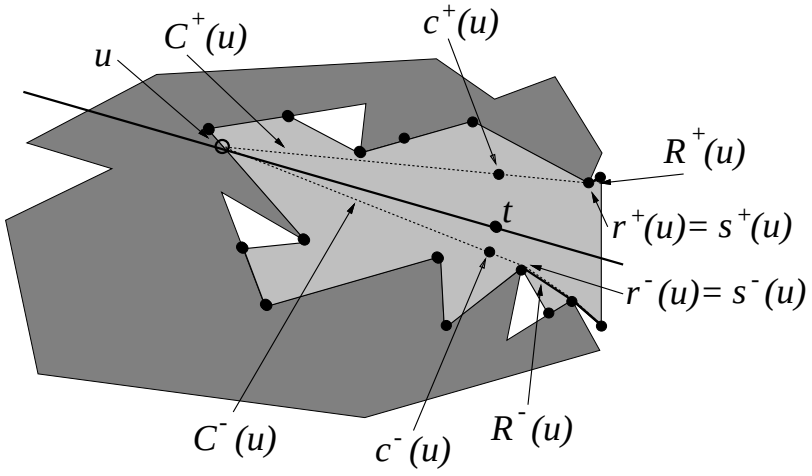


Figure 6: $R^-(u)$, $R^+(u)$, $r^-(u)$, $r^+(u)$, $s^-(u)$ and $s^+(u)$.

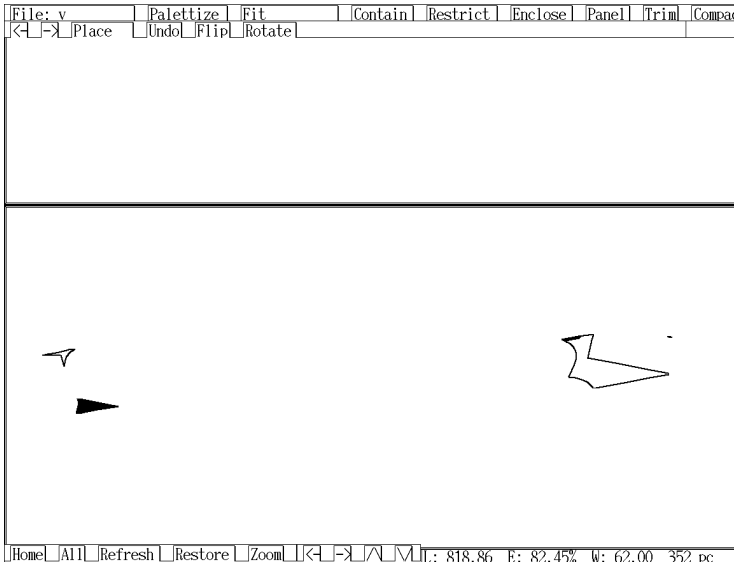


Figure 7: LP restriction of two two-component U_{0j}s

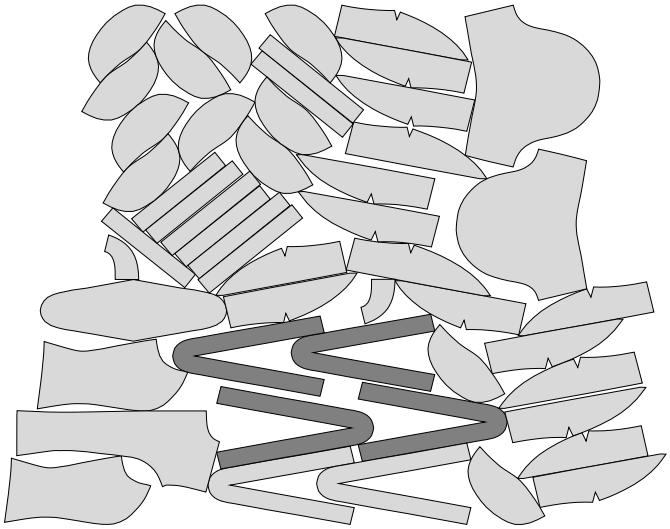


Figure 8: 4NN example: time = 15 seconds

Figure 9: 6NN example: time = 126 seconds

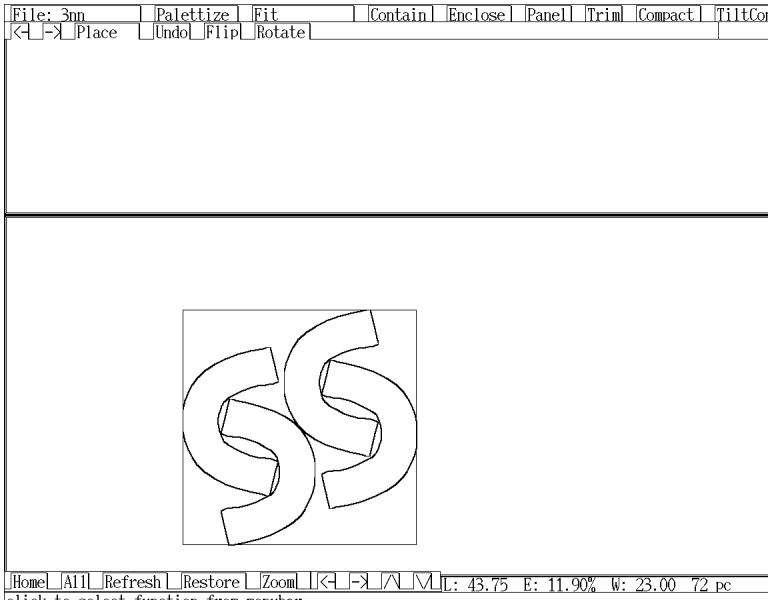


Figure 10: Minimal enclosing square of four polygons, each with 65 vertices: time = 122 seconds (numerical algorithm).