Submitted to *Algorithmica* July, 1994. In press.

Multiple Translational Containment
Part II: Exact Algorithms

Victor Milenkovic *
Department of Mathematics and Computer Science
University of Miami, Coral Gables, FL 33124
vjm@cs.miami.edu

Abstract

We present exact algorithms for finding a solution to the two-dimensional translational *containment* problem: find translations for k polygons which place them inside a polygonal container without overlapping. The term kCN denotes the version in which the polygons are convex and the container is nonconvex, and the term kNN denotes the version in which the polygons and the container are nonconvex. The notation (r,k)CN, (r,k)NN, and so forth refers to the problem of finding all subsets of size k out of r objects that can be placed in a container. The polygons have up to m vertices, and the container has n vertices, where n is usually much larger than m.

We present exact algorithms for the following: 2CN in $\mathrm{O}(mn \log n)$ time, $(r,2)$CN in $\mathrm{O}(r^2 m \log n)$ time (for $r >> n$), 3CN in $\mathrm{O}(m^3 n \log n)$ time, kCN in $\mathrm{O}(m^{2k} n^k \log n)$ or $\mathrm{O}((mn)^{k+1})$ time, and kNN in

$$\mathrm{O}((mn)^{2k+1}\mathrm{LP}(2k, 2k(2k+1)mn + k(k-1)m^2))$$

time, where $\mathrm{LP}(a,b)$ is the time to solve a linear program with a variables and b constraints. All these results are improvements on previously known running times except for the last. The algorithm for kNN is slower asymptotically than the naive $\mathrm{O}((mn)^{2k} \log n)$ algorithm, but is expected to be much faster in practice. The algorithm for 2CN is based on the use of *separating line orientations* as a means of characterizing the solution. The solution to 3CN also uses a separating line orientation characterization leading to a simple and robust "carrousel" algorithm. The kCN algorithm uses the idea of disassembling the layout to the left. Finally, the kNN algorithm uses the concept of subdivision trees and linear programming.

Keywords: Computational geometry, Containment, Nesting, Layout, Marker making

---

*This research was funded by the Textile/Clothing Technology Corporation from funds awarded to them by the Alfred P. Sloan Foundation and by NSF grants CCR-91-157993 and CCR-90-09272.

1 Introduction

The translational *containment* problem is that of determining a set of translations for k polygons that place them in a nonoverlapping configuration inside a polygonal container. Part I, *An Approximate Algorithm* [3], describes how our research into *containment* problems grew out of the need to solve an NP-hard layout problem in the apparel industry. This problem is called *marker-making*: lay out polygonal apparel pattern pieces on cloth inside a rectangular sheet of stock material of fixed width and minimum length. One might denote marker making as kN(nonconvex)R(rectangle) or kNR containment: place k nonconvex polygons into a rectangular container. Assuming P $\neq$ NP, any exact algorithm for marker making would have running time exponential in k. Hence, we must find some way to simplify the layout problem.

Many types of garments consist of a few large pieces plus many small pieces. For instance, pants consist of a large front and back *panel* (one each for each leg) plus a number of other *trim* pieces corresponding to the waistband, belt loops, pocket facings, and so forth. An adaptation of heuristics for bin packing and partition yields a practical algorithm for laying out the large panels [12], as shown in Figure 1. Our hope is to apply containment algorithms to the problem of filling in the smaller trim pieces into the gaps of unused material between the panel pieces, as shown in Figure 2. Other garment types would require different heuristics to place the large pieces, or it may be possible to place the large pieces by adapting layouts previously generated by expert humans. This technique is called *database-driven* marker making [10] [11].

For pants markers, most of the gaps of unused material can hold no more than a few trim pieces. About half the gaps can hold no more than three. Also, for pants markers, the trim pieces are convex. What is needed to fill in the gaps is an efficient algorithm for what we call $(r, 2)$C(convex)N(nonconvex) ($(r, 2)$CN) or $(r, 3)$CN containment: out of the r trim pieces ($r \approx 100$) find all groups of two or three convex trim pieces that fit into a particular nonconvex gap. As it turns out, industry sometimes permits discrete 90 or 180 degree rotations of the trim pieces and reflection about the x-axis. One would also have to iterate over the permissible orientations. Hence the effective value of r is closer to 400. Once we know which groups of trim pieces can fit into each gap, we can reduce trim placement to a set cover or matching problem. These problems would still be NP-hard, but they would be entirely nongeometric and much more amenable to heuristics. There are many variations on this approach, but it is clear that the faster one can solve containment for small k, the better.

Note that we only seek a single "yes-no" answer to each containment problem; we do not require the set of all possible solutions. We need a fast solution since we must solve the containment problem many times while setting up the matching model.

1.1 Notation

Throughout the paper, we use the following notation:

- C, a polygonal "container" with n vertices;
- P_i, $1 \leq i \leq k$, a polygon with m_i vertices to be placed in the container;
- V_i, $1 \leq i \leq k$, the set of translations of P_i that places it in the interior of C;
- U_{ij}, $1 \leq i < j \leq k$, the set of displacements between P_i and P_j such that they do not overlap each other ($U_{ij} = -U_{ji}$.)

We refer to V_i as the *valid region* of P_i in C.

A *valid configuration* for $P_1, P_2, \ldots, P_k$ inside C is a list of translations $\langle t_1, t_2, t_3, \ldots, t_k \rangle$ such that

$$t_i \in V_i,\ 1 \leq i \leq k, \qquad \text{and} \qquad t_j - t_i \in U_{ij},\ 1 \leq i < j \leq k. \tag{1}$$

We define the kNN *problem* to be that of finding a valid configuration for k nonconvex polygons inside a nonconvex container. Similarly, the kCN *problem* is that of finding a valid configuration for k convex polygons inside a nonconvex container.

We define the (r, k)NN *problem* to be that of finding all subsets of size k out of a set of r nonconvex polygons such that the k polygons have a valid configuration inside a given nonconvex container. We similarly define the (r, k)CN *problem*.

We define $m = \max_{1 \leq i \leq k} m_i$. Typically m is less than n, and we will assume this to be the case when we write down running times. In general, when n appears in a running time, it actually represents "$\max(m, n)$". We express the running time of a containment algorithm in terms of m, n, k, and r. For our applications: $4 \leq m \leq 100$, $100 \leq n \leq 300$, $1 \leq k \leq 10$, $100 \leq r \leq 400$.

1.2 The Minkowski Sum

The *Minkowski sum* [13, 8, 15, 16] 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 $A + t = \{a + t \,|\, a \in A\}$ and $A - t = \{a - t \,|\, a \in A\}$.

Lemma 1.1 ([8, 15]) *$A + s$ and $B + t$ overlap if and only if $t - s \in A \oplus -B$.*

Figure 3 illustrates an example of A, B, $-B$, and $A \oplus -B$, and Figure 4 illustrates this lemma. One can think of $A \oplus -B$ in two ways. First, fix A and "smear" a copy of $-B$ over all translations that put its origin within A. Second, fix A, and "slide" B around A such that it maintains contact but does not overlap A. The origin of B will trace out the boundary of the Minkowski sum $A \oplus -B$. Hence, a translated copy of B overlaps A if and only if the origin of B is inside $A \oplus -B$.

Section 1.1 gave the various parts of the containment problem: C; P_i, $1 \leq i \leq k$; V_i, $1 \leq i \leq k$; and U_{ij}, $1 \leq i < j \leq k$. Expressed in terms of set operations and Minkowski sums,

$$V_i = \overline{\overline{C} \oplus -P_i},\ 1 \leq i \leq k, \qquad \text{and} \qquad U_{ij} = \overline{P_i \oplus -P_j},\ 1 \leq i < j \leq k.$$

These will be the values of the symbols unless otherwise stated.

For convex P_i, each V_i has $O(mn)$ vertices and requires $O(mn \log mn)$ time to compute [7]. Each U_{ij} has $O(m)$ vertices and requires $O(m)$ time to compute [8]. For nonconvex P_i, each V_i has $O(m^2n^2)$ vertices and requires $O(m^2n^2 \log mn)$ time to compute. Each U_{ij} has $O(m^4)$ vertices and requires $O(m^4 \log m)$ time to compute. In the context of the paper, $m < n$, and therefore $O(\log mn) = O(\log n)$. The following section details the known results that lead to these complexities.

1.3 Related Work

Part I [3] of this paper presents a "naive" kNN algorithm that runs in $O((mn)^{2k} \log n)$ time, assuming "constant" k and $n >> m$ (as is typical in our applications). The idea behind this algorithm is that there are $2k$ degrees of freedom, two for each polygon, and that each vertex-edge contact in the layout removes one degree of freedom. Each contact implies a linear equation on the $2k$ coordinates of the configuration. It is shown that if there is a valid configuration, then there is one with exactly $2k$ *linearly independent* contacts.[1] The naive algorithm iterates over all choices of $2k$ vertex-edge contacts among the polygons and container. Each set of choices of $2k$ contacts implies a set of $2k$ linear equations in $2k$ unknowns. These can be solved in $O(k^3)$ time to generate a potential configuration for the polygons (sets of linearly dependent equations are rejected). Testing a potential configuration $\langle t_1, t_2, \ldots, t_k \rangle$ requires k point-in-polygon tests of t_i with respect to V_i and $k(k-1)/2$ point-in-polygon tests of $t_j - t_i$ with respect to U_{ij}. Assuming appropriate preprocessing, each point-in-polygon test requires only $O(\log n)$ time.

[1] Linear independence is essential: a vertex-vertex contact appears to be four vertex-edge contacts, but it yields only two linearly independent equations.

To solve the (r,k)NN problem, we need to (naively) solve $\binom{r}{k}$ kNN problems, so it is essential that the containment algorithm run quickly even if the answer is "no." The naive algorithm iterates over all $(mn)^{2k}$ sets of contacts whenever the answer is "no". One method to avoid excessive iteration in the "no" case is to develop specialized algorithms for convex polygons; another is to explicitly compute configuration spaces for the polygons.

Fortune [7] gives a solution to 1CN by computing the *Minkowski sum* using a generalized Voronoi diagram. Avnaim and Boissonnat [2, 1] use the Minkowski sum and convex decomposition to solve 1NN and 2NN, and Avnaim gives an algorithm for 3NN. Devillers [5] gives faster algorithms for 2CN and 3CN. These running times are summarized in the following table.

k	kCN		kNN	
1	$O(mn \log mn)$	[7]	$O(m^2n^2 \log mn)$	[2, 1]
2	$O(m^2n^2 \log m^2)$	[5]	$O(m^4n^4 \log m^2n)$	[2, 1]
3	$O(m^3n^3 \log m^3)$	[5]	$O(m^{14}n^6 \log m^2n)$	[1]

Avnaim and Boissonnat also give a solution to the 3NP problem, three nonconvex polygons in a *parallelogram* container, using time in $O(m^{60} \log m^3)$ [2, 1].

Milenkovic, *et al.* [4] offer three approaches to translational multiple containment. The first method operates on convex polygons. It iterates over combinations of separating line orientations. This approach yields an algorithm for three convex polygons and a nonconvex container with running time in $O(m^3 n \log n)$, an improvement of a factor of n^2 over previous algorithms. The second method uses a MIP (mixed integer programming) model for k nonconvex polygons and a nonconvex container. This approach is described in detail in [9]. It builds constraints based on convex covers of V_i, $1 \leq i \leq k$ and U_{ij}, $1 \leq i < j \leq k$, then looks for a feasible solution. It takes a minute or two on a typical workstation for two or three polygons, but it is slow for four or more polygons. The third method is an approximation algorithm for k nonconvex polygons and a nonconvex container. Part I of this paper presents a more advanced form of this algorithm.

This Part II improves and simplifies the exact algorithms of [4] for 2CN and 3CN and presents an exact algorithm for kCN. The following table gives the running times for these new results.

2CN	(r,2)CN	3CN	kCN
$O(mn \log n)$	$O(r^2 m \log n)$	$O(m^3 n \log n)$	$O(m^{2k} n^k \log n)$ or $O((mn)^{k+1})$

Finally, we give an exact algorithm for kNN with running time in

$$O((mn)^{2k+1} \mathrm{LP}(2k, 2k(2k+1)mn + k(k-1)m^2)),$$

where $\mathrm{LP}(a, b)$ is the time to solve a linear program with a variables and b constraints.

All these results are improvements on previously known running times except for the last. The algorithm for kNN is slower asymptotically than the naive $O((mn)^{2k} \log n)$ algorithm, but is expected to be much faster in practice. Note that this property is similar to that of Avnaim's algorithm for 3NN containment. The asymptotic running time of Avnaim's algorithm is worse than the naive algorithm, but it is likely to be much better in practice.

1.4 Overview

Section 2 gives fast algorithms for 2CN and $(r,2)$CN based on the notion of *separating lines* for convex polygons. Section 3 continues the use of separating lines and presents a simple, robust and efficient "carrousel" algorithm for 3CN. Section 4 gives a linear programming based algorithm for the most general kNN translational containment problem. This algorithm does not use separating lines, but it does use a generalization of a multiple resolution technique to select separating lines. Finally, Section 5 introduces the notion of "disassembly to the left" as a way of practically solving kCN. Part of this algorithm uses a specialization of the kNN algorithm to y-monotone configuration spaces.

2 2CN Containment

The 2CN containment problem takes as input two convex polygons P_1 and P_2 and a nonconvex container C. Polygons P_1 and P_2 each have at most m vertices, and C has n vertices. Previous solutions to this problem have had running times essentially equal to the product of the times for solving two 1CN problems, in particular, $O(m^2n^2 \log m)$. This section establishes the somewhat surprising fact that 2CN containment requires no more time than 1CN asymptotically. Furthermore, solving $(r, 2)$CN requires $O(rmn \log n)$ "preprocessing" time plus $O(r^2 m \log n)$ time. In other words, the time is the same as r 1CN problems plus r^2 queries taking $O(m \log n)$ time.

The idea behind fast solution of 2CN containment is that of a *separating line*. There always exists a line which separates two nonoverlapping convex polygons. Furthermore, it is only necessary to consider $O(m)$ possible orientations for this line. After preprocessing equivalent to solving 1CN problems for each of the input polygons, testing a particular line orientation requires only $O(\log n)$ time. This leads to efficient algorithms for 2CN and $(r, 2)$CN.

2.1 Separating Lines

This section reviews known results about the Minkowski sum of two convex polygons. Then it gives a lemma about separating lines of convex polygons. This lemma is essential for the 2CN algorithm.

Lemma 2.1 ([8]) *If P_1 and P_2 are convex polygons with m_1 and m_2 vertices, respectively, then $P_1 \oplus -P_2$ has $m_1 + m_2$ vertices, and it can be computed in $O(m_1 + m_2)$ time.*

Of course, Guibas *et al.* [8] did not use this notation, but this is the form that we need here. As P_2 moves about P_1, its origin traces out the boundary of $P_1 \oplus -P_2$. Each edge of $P_1 \oplus -P_2$ is the result of a vertex of P_1 contacting an edge of P_2 or *vice versa*. As illustrated in Figure 5, each edge is either $e_1 + -v_2$, where e_1 is an edge of P_1 and v_2 is a vertex of P_2, or $-e_2 + v_1$, where v_1 is a vertex of P_1 and e_2 is an edge of P_2. Furthermore, only *one* translated copy of each edge appears in the boundary of $P_1 \oplus -P_2$ (unless P_1 and P_2 have a pair of parallel edges). This follows from the fact that each edge can come in contact with only one vertex of the other polygon. (Note: a vertex can contact multiple edges of the other polygon.) For instance, consider the edge e_1 of P_1 with outward normal vector w_1. Edge e_1 can only contact the vertex v_2 of P_2 which is *extreme* with respect to $-w_1$: $-w_1 \cdot v_2$ is maximal ($w_1 \cdot v_2$ is minimal) over all vertices of P_2. The extreme is not unique in the case that P_2 has an edge parallel to e_1. Guibas *et al.* give the general argument.

If $P_1 + t_1$ (P_1 translated to put its origin at t_1) and $P_2 + t_2$ do not overlap, then Lemma 1.1 states that $t_2 - t_1$ is outside $P_1 \oplus -P_2$. Furthermore, the set of lines that separate $P_1 + t_1$ from $P_2 + t_2$ is in one-to-one correspondence with the set of lines separating $t_2 - t_1$ from $P_1 \oplus -P_2$ (Figure 6). We exploit this fact to prove the following lemma.

Lemma 2.2 *Let P_1 and P_2 be convex polygons and let t_1 and t_2 be translations such that $P_1 + t_1$ and $P_2 + t_2$ do not overlap ($t_2 - t_1 \in U_{12} = \overline{P_1 \oplus -P_2}$). Then there exists a line parallel to an edge of P_1 or an edge of P_2 that separates $P_1 + t_1$ from $P_2 + t_2$.*

Proof: This lemma is a fairly well-known fact about convex polygons. As an illustration of the Minkowski sum, we give the following proof. For convex P_1 and P_2, $P_1 \oplus -P_2$ is convex, and the interior of a convex polygon is the intersection of the "inward" half planes defined by its edges. By Lemma 1.1, if (the interiors of) $P_1 + t_1$ and $P_2 + t_2$ do not overlap, then $t_2 - t_1$ is outside (the interior of) $P_1 \oplus -P_2$. Therefore for some edge ab of $P_1 \oplus -P_2$ $t_2 - t_1$ is *not* in the inward half plane of ab. Let w be the outward normal vector to edge ab. There must exist a line L with normal w which separates $t_2 - t_1$ from $P_1 \oplus -P_2$. We claim that the line $L + t_1 + v_2$ separates $P_1 + t_1$ from $P_2 + t_2$, where v_2 is a vertex of P_2 which minimizes $w \cdot v_2$. Every edge of $P_1 \oplus -P_2$ is parallel to an edge of P_1 or P_2 and thus so is the line $L + t_1 + v_2$.

A little bit of analytic geometry can prove the claim. Let $p_1 \in P_1$ and $p_2 \in P_2$ be arbitrary points. By the definition of Minkowski sum, $p_1 - p_2 \in P_1 \oplus -P_2$. Let q be any point on L. Since $t_2 - t_1$ lies on the "positive" side of L and $p_1 - p_2$ lies on the "negative" side,

$$w \cdot (t_2 - t_1) \geq w \cdot q, \quad (2)$$

$$w \cdot (p_1 - p_2) \leq w \cdot q. \quad (3)$$

Also, since v_2 is extreme with respect to p_2:

$$w \cdot p_2 \geq w \cdot v_2. \quad (4)$$

Since $p_2 \in P_2$ is arbitrary, we can substitute v_2 for p_2 in Equation 3 and add $w \cdot (t_1 + v_2)$ to both sides:

$$w \cdot (p_1 + t_1) \leq w \cdot (q + t_1 + v_2). \quad (5)$$

Adding Equation 2 to Equation 4:

$$w \cdot (p_2 + t_2) \geq w \cdot (q + t_1 + v_2). \quad (6)$$

Since $p_1 \in P_1$ and $p_2 \in P_2$ are arbitrary, Equations 5 and 6 imply that $P_1 + t_1$ and $P_2 + t_2$ lie on opposite sides of $L + t_1 + v_2$. ∎

Recall from Section 1.1 that V_i is the set of values of t_i such $P_i + t_i$ is a subset of the container C. The following lemma implies that we can choose a solution to 2CN from among the *vertices* of V_1 and V_2.

Lemma 2.3 *If convex polygons P_1 and P_2 can be placed inside C without overlapping, then there exist vertices v_1 of V_1 and v_2 of V_2 such that $t_1 = v_1$ and $t_2 = v_2$ is a valid configuration.*

Proof: Suppose there exist translations t'_1 and t'_2 such that $P_1 + t'_1$ and $P_2 + t'_2$ lie inside C without overlapping. By Lemma 2.2 there is a line L separating $P_1 + t'_1$ from $P_2 + t'_2$. Let w be the normal to this line such that $P_1 + t'_1$ lies on the $-w$ "side" and $P_2 + t'_2$ lies on the $+w$ "side". Let v_1 be the vertex of V_1 that minimizes $w \cdot v$ for $v \in V_1$. Let v_2 be the vertex of V_2 that maximizes $w \cdot v$ for $v \in V_2$. Moving P_1 to $P_1 + t_1$, where $t_1 = v_1$, moves it farther from line L, and similarly for moving P_2 to $t_2 = v_2$. Since they remain on opposite sides of the separating line, they do not overlap. ∎

Corollary 2.4 *Under the conditions of Lemma 2.3, v_1 and v_2 can be chosen to be 1) on the convex hulls of V_1 and V_2, respectively, and 2) extreme with respect to $-w$ and w, respectively, where w is an outward normal vector of some edge of P_1 or $-P_2$ (or equivalently of $P_1 \oplus -P_2$).*

Proof: By the proof of Lemma 2.2, we can choose L be parallel to some edge of P_1 or $-P_2$. For any polygon, the extreme vertex with respect to a vector is always a vertex of the convex hull of that polygon. ∎

2.2 Computing a Solution

In order to compute v_1 and v_2, we only need to know the normal vector w. Hence, we only need to know the orientation, not the position, of the separating line L. This observation leads to the following algorithm for 2CN containment. This algorithm simply iterates over all possible orientations for a separating line and checks each orientation for a solution.

For each vector w outwardly normal to an edge ab of $P_1 \oplus -P_2$:
1. compute v_1, the vertex of V_1 that minimizes $w \cdot v$, $v \in V_1$;
2. compute v_2, the vertex of V_2 that maximizes $w \cdot v$, $v \in V_2$;
3. if $v_2 - v_1$ lies to the positive w side of line ab $(w \cdot (v_2 - v_1) \geq w \cdot a)$,
then return the solution $t_1 = v_1$, $t_2 = v_2$.

Otherwise, report no solution exists.

By Corollary 2.4, if a solution to 2CN exists, then this algorithm will find w, v_1, and v_2 that work. Note that we can restrict our search in (1) and (2) to the convex hulls $\mathrm{CH}(V_1)$ and $\mathrm{CH}(V_2)$ of V_1 and V_2.

Naively, steps 1 and 2 in the algorithm require time in $\Omega(|V_1|)$ and $\Omega(|V_2|)$, respectively. However, if one precomputes the convex hulls $\mathrm{CH}(V_1)$ and $\mathrm{CH}(V_2)$, then one can find v_1 and v_2 by binary search on the normals. The minimizing vertex v_1 for V_1 is the minimizing vertex for $\mathrm{CH}(V_1)$, and similarly for v_2, V_2, and $\mathrm{CH}(V_2)$. The binary searches require time in $\mathrm{O}(\log|V_1|)$ and $\mathrm{O}(\log|V_2|)$ which are both in $\mathrm{O}(\log n)$. (From Section 1.2, $|V_i|$ is $\mathrm{O}(mn)$, and $\mathrm{O}(\log mn) = \mathrm{O}(\log n)$ for $m < n$.) Total time for $\mathrm{O}(m)$ orientations is $\mathrm{O}(m \log n)$.

Alternately, if we visit the normals to $P_1 \oplus -P_2$ in counterclockwise order, then the minimizing vertex of $\mathrm{CH}(V_1)$ and $\mathrm{CH}(V_2)$ for the normal moves around these polygons in counterclockwise order. Therefore we only have to check each vertex of $\mathrm{CH}(V_1)$ and $\mathrm{CH}(V_2)$ once for total time in $\mathrm{O}(|V_1| + |V_2|)$. This might be faster than repeated binary search if $|V_1|$ and $|V_2|$ are much smaller than the worst case of mn.

Lemma 2.5 *The 2CN containment algorithm uses time in* $\mathrm{O}(m \log n)$ *plus the amount of time required to compute the convex hulls of* V_1 *and* V_2*, which is in* $\mathrm{O}(mn \log n)$*.*

Note that the time it takes to compute $\mathrm{CH}(V_1)$ and $\mathrm{CH}(V_2)$ is of the same order as the time to compute V_1 and V_2. These dominate the $\mathrm{O}(m)$ time to compute U_{12}.

Corollary 2.6 *For* $r \geq n$ *(which is often true for our applications),* $\binom{r}{2}$ *applications of the 2CN algorithm solves the* $(r,2)$*CN problem using time in* $\mathrm{O}(r^2 m \log n)$*. (If* $r < n$*, then the cost* $\mathrm{O}(rmn \log n)$ *of computing the convex hulls of* $V_1, V_2, \ldots, V_k$ *dominates.)*

For our applications, $4 \leq m \leq 20$,[2] $100 \leq n \leq 300$ and $r \approx 400$, and so this is within the realm of practicality on current computers. It is also numerically stable. Finally, using the multiple resolution techniques of Section 3.5, one can hope to reduce the time for $(r,2)$CN in practice to something like $\mathrm{O}(r^2 \log m \log n)$.

3 3CN Containment

The 3CN containment problem has as input three convex polygons P_1, P_2, and P_3, and a nonconvex container C. As in the case of 2CN, we consider separating line orientations. In the case of 3CN, there are three separating line orientations, one for each pair of polygons. We refer to a choice of orientations as an *orientation triple*. Our 3CN algorithm iterates over possible separating line orientation triples. For each triple, it checks if there is a configuration that has separating lines with these orientations.

In the 2CN algorithm, we had to consider $\Theta(m)$ orientations. Naively, one must consider $\Omega(m^3)$ orientation triples to solve 3CN. Section 3.1 shows that we need only consider $\mathrm{O}(m^2)$ triples.

The 2CN algorithm only needed to test the *vertices* of the valid regions V_1 and V_2 for a solution. In the 3CN case, we are not so fortunate. However, the solution can always be moved to the *boundaries* of the respective valid regions. Furthermore, for a given choice of orientation triple, we need only consider a special subset of the boundary. Section 3.2 discusses this subset, called the angle-visible envelope.

Even though we can restrict the solution to a one-dimensional envelope of each valid region, it does not appear that we can find the solution for each orientation in logarithmic time, as we do for 2CN. Section 3.3 gives a "carrousel" algorithm that searches the envelopes for a solution in $\mathrm{O}(|V_1| + |V_2| + |V_3|)$ time, and Section 3.4 proves its correctness.

Finally, Section 3.5 gives a multiple resolution method that can considerably reduce the number of orientation triples in practice. This method can also be applied in the case of 2CN.

[2] In our application, the convex pieces have only up to 20 vertices, unlike the nonconvex pieces which can have up to 100 vertices.

3.1 Orientation Triples

We know from Lemma 2.2 that if convex polygons P_i and P_j are nonoverlapping, then there is a line separating them. We can choose this line so that its normal vector w_{ij} is also an outward normal to some edge $a_{ij}b_{ij}$ of $P_i \oplus -P_j$. Here (i, j) equals $(1, 2)$, $(2, 3)$, or $(3, 1)$.

The boundary of $P_i \oplus -P_j$ corresponds to the set of relative positions that put P_i in *contact* with P_2. The minimum set of separating line orientations corresponds to the set of edges of $P_i \oplus -P_j$. Thus Lemma 2.2 tells us that if we choose a set of separating line orientations that "takes care" of all the contacting placements of P_i and P_j, then this set of orientations also "takes care" of all nonoverlapping placements of P_i and P_j. The following lemma about orientation triples for 3CN generalizes this interpretation of Lemma 2.2.

Lemma 3.1 *Three nonoverlapping convex polygons P_1, P_2 and P_3 can be separated by lines whose orientations arise in the three-contact case (all three pairs of polygons in contact) or the two-contact case (two pairs of polygons in contact). Furthermore, for the two-contact case, we need only consider two pairs of contacting* **vertices**. *The three-contact case has $O(m)$ triples and the two-contact case has $O(m^2)$ triples.*

Proof: Pick points c_1, c_2 and c_3 internal to P_1, P_2 and P_3, respectively. Expand (scale) the polygons about their internal points until two come into contact, w.l.o.g, P_1 and P_2. (Always use the same scaling factor for all three polygons.) Continue expansion of P_3 about c_3 and of P_1 and P_2 about their point of contact c_{12} until a second contact occurs (see Figure 7). At this point, the expanded polygons P_1', P_2', and P_3' are similar to the original polygons with the same scaling factor. Furthermore, the expanded polygons contain their unexpanded counterparts. Therefore any set of separating lines for the expanded polygons also separates the original polygons. Since the scale factors are equal, the *orientations* of these lines are equal to those of separating lines for a similar placement of the original polygons with two or three contacts.

If there are two contacts, assume w.l.o.g. that P_1' and P_3' have vertex-edge contacts with P_2'. Note that the vertex-edge contact between P_1' and P_2' might involve a vertex of P_1' and an edge of P_2' or *vice versa*. Let L_{12} and L_{23} be the lines separating P_2' from P_1' and P_3', respectively. Slide P_1' and P_3' along L_{12} and L_{23} *towards* each other, *i.e.* towards the intersection of L_{12} and L_{23}. Stop P_1' at the position where it is about to lose contact with P_2', and similarly stop P_3'. Also, stop the motion if P_1' and P_3' come in contact. We refer to the displaced copies of P_1' and P_3' as P_1'' and P_3'' (see Figure 8).

If P_1'' and P_3'' are in contact, then there is a triple of separating lines L_{12}, L_{23}, and L_{31} whose orientations correspond to a three-contact configuration of P_1, P_2, and P_3. If P_1'' and P_3'' are not in contact, then each has a vertex-vertex contact with P_2'. Hence we can choose separating lines L_{12}, L_{23}, and L_{31} whose orientations correspond to two vertex-vertex contacts. (Note that there may be more than one possibility for L_{31}, but we only have to consider one of them.) In both the two-contact and three-contact cases, the separating lines will still work if we move P_1'' and P_3'' back to P_1' and P_3'. Hence these separating lines will work if we shrink back to P_1, P_2, and P_3.

There are only $O(m^2)$ ways that two pairs of polygons can have a vertex-vertex contact because a vertex-vertex contact between P_i and P_j corresponds to a vertex of $P_i \oplus -P_j$, and the Minkowski sum has only $O(m)$ vertices. To prove that there are only $O(m)$ orientation triples corresponding to triple contact, we have to revisit the proof that $P_i \oplus -P_j$ has only $|P_i| + |P_j|$ edges. Imagine P_1, P_2, and P_3 "orbiting" each other while maintaining a triple contact (three pairs of contacts). Each point of contact makes an orbit around the boundaries of the pair of polygons in contact. The number of times that a contact point moves to a different edge is $2(|P_1| + |P_2| + |P_3|)$. Since the edges determine the separating line orientations, there are exactly $2(|P_1| + |P_2| + |P_3|)$ orientation triples. (The factor of two arises from the fact that there are two contact points on each polygon.) ■

It is a fairly simple exercise to create algorithms to compute the orientation triples for the two and three contact cases in time $O(m^2)$ and $O(m)$, respectively.

3.2 Angle-Visible Envelopes

Let us suppose we have a valid configuration t_1, t_2, t_3 of P_1, P_2, and P_3 inside a container C. Polygons $P_1 + t_1$, $P_2 + t_2$, and $P_3 + t_3$ are pairwise disjoint and all lie inside C. Translations t_1, t_2, and t_3 lie inside the respective valid regions V_1, V_2, and V_3. This section shows that t_i can be moved to the boundary of V_i, $i = 1, 2, 3$. Furthermore, for a specific choice of separating line orientation triple, t_i can be moved to a specific subset of the boundary of V_i, $i = 1, 2, 3$, called the *angle-visible envelope*.

Definition 3.1 *Suppose we are given a polygon V and vectors w and w' such that $(w \times w')_z > 0$ (w' is rotated less than 180 degrees counterclockwise from w). Point $v \in V$ is* **angle-visible** *with respect to w and w' if there exists no point v' of V that satisfies,*

$$w \cdot v' \geq w \cdot v \qquad \text{and} \qquad w' \cdot v' \leq w' \cdot v.$$

In other words, v can "see to infinity" along any direction ν that satisfies,

$$w \cdot \nu \geq 0 \qquad \text{and} \qquad w' \cdot \nu \leq 0.$$

The **angle-visible envelope** *of V with respect to w and w' is the set of angle-visible points of V with respect to w and w'.*

As Figure 9 illustrates, the angle-visible envelope of V is the set of points that can be touched by the vertex of an angle whose sides are perpendicular to w and w'.

Lemma 3.2 *The angle-visible envelope of polygon V with respect to w and w' consists of $\mathrm{O}(|V|)$ line segments. The envelope is monotonic with respect to w and w' (and any vector in a direction between w and w'). The envelope can be computed in time $\Theta(|V|)$ for a simple polygon and in time $\Theta(|V| \log |V|)$ for a general polygon.*

Proof: Clearly, the envelope is a subset of the boundary of $|V|$, and so it consists of a set of line segments. The endpoint of a segment of the envelope is either a vertex of V or there exists a vertex of V that blocks its visibility in a direction perpendicular to w or w'. Each vertex of V can block at most two endpoints, and so the envelope has at most three times as many vertices as V.

The monotonicity of the envelope follows immediately from the visibility condition. For any choice of $\alpha, \alpha' \geq 0$ (not both zero), suppose the envelope were not monotonic with respect to the vector $w'' = \alpha w + \alpha' w'$. In other words, there are two points v and v' in the envelope such that $w'' \cdot v = w'' \cdot v'$. Since v and v' are not the same point, either $w \cdot v \neq w \cdot v'$ or $w' \cdot v \neq w' \cdot v'$. W.l.o.g., $w \cdot v > w \cdot v'$. Therefore $w \cdot (v - v') > 0$. But,

$$0 = w'' \cdot (v - v') = \alpha w \cdot (v - v') + \alpha' w' \cdot (v - v').$$

Therefore, $w' \cdot (v - v') < 0$ which implies $w' \cdot v < w' \cdot v'$. Thus, v blocks the visibility of v' in the direction $v - v'$, a contradiction.

An algorithm somewhat akin to the convex hull algorithm for simple polygons [14] can compute the angle-visible envelope for a simple polygon in linear time. For a general polygon with multiple components, computing the envelope is at least as hard as choosing one vertex from each component and sorting them in the direction w (or w'). A simple sweep algorithm achieves the optimal asymptotic running time. ■

Lemma 3.3 *Suppose P_1, P_2, and P_3 are convex with a valid configuration t_1, t_2, t_3 inside C. Let w_{ij} be a separating line orientation, $(i, j) \in \{(1, 2), (2, 3), (3, 1)\}$, for $P_i + t_i$ and $P_j + t_j$. Then there exists a valid configuration t'_1, t'_2, and t'_3 such that t'_i lies on the angle-visible envelope of V_i with respect to w_{hi} and w_{ij}, $(h, i, j) \in \{(1, 2, 3), (2, 3, 1), (3, 1, 2)\}$.*

Proof: Let a_{ij} be a vertex of $P_i \oplus -P_j$ which is extreme with respect to w_{ij} (maximizes $w_{ij} \cdot a_{ij}$).[3] Since w_{ij} is the orientation of a separating line, $t_j - t_i$ lies on the $+w_{ij}$ side of a line through a_{ij} (see Lemma 2.2):

$$\begin{aligned} w_{12} \cdot (t_2 - t_1) &\geq w_{12} \cdot a_{12}, & (7)\\ w_{23} \cdot (t_3 - t_2) &\geq w_{23} \cdot a_{23}, & (8)\\ w_{31} \cdot (t_1 - t_3) &\geq w_{31} \cdot a_{31}. & (9) \end{aligned}$$

Inequalities 7, 8, and 9, express the condition that each pair of polygons has a separating line of the appropriate orientation. Let Δt_1 be a vector that satisfies,

$$w_{31} \cdot \Delta t_1 \geq 0, \qquad \text{and} \qquad w_{12} \cdot \Delta t_1 \leq 0. \tag{10}$$

If we replace t_1 by $t_1 + \Delta t_1$, then inequalities 7 and 9 remain true. Inequality 8 is unaffected by changes in t_1. Clearly, if $t_1 \in V_1$ is *not* on the angle-visible envelope of V_1 with respect to w_{31} and w_{12}, then such a Δt_1 exists. In fact, it is easy to see that there exists a point t_1' on the angle-visible envelope such that $\Delta t_1 = t_1' - t_1$ satisfies Equation 10. Therefore, we can always move a solution to 3CN to the respective angle-visible envelopes of the valid regions V_1, V_2, and V_3. ■

We will denote the angle-visible envelope of V_i with respect to w_{hi} and w_{ij} as a table of N_i segments $A_i[1]B_i[1], A_i[2]B_i[2], A_i[3]B_i[3], \ldots, A_i[N_i]B_i[N_i]$. These are ordered monotonically with respect to any positive linear combination $w = \alpha w_{hi} + \beta w_{ij}$, $\alpha, \beta > 0$,

$$\begin{aligned} w \cdot A_i[I] &< w \cdot B_i[I], \text{ for } I = 1, 2, 3, \ldots N_i,\\ w \cdot B_i[I] &\leq w \cdot A_i[I+1], \text{ for } I = 1, 2, 3, \ldots N_i - 1. \end{aligned}$$

3.3 Carrousel Algorithm

Once we have computed the angle-visible envelopes for a particular orientation triple, the algorithm for 3CN is conceptually very simple. (Recall from Section 3.1 that we must iterate over $O(m^2)$ orientation triples.) Each polygon P_i is placed at its "first" envelope point $A_i[1]$. Polygon P_1 "bumps" P_2 forward to the first point on its envelope that satisfies the separating line constraint for P_1 and P_2. Similarly, P_2 "bumps" P_3, and P_3 "bumps" P_1, and so forth. The phrase, "Round and round it goes–where it stops, nobody knows," is apropos. Actually, roulette is the wrong amusement park analogy. The correct analogy is a combination of bumper cars and a merry-go-round. In the interests of dignity, we refer to this as a "carrousel" algorithm. The carrousel algorithm solves 3CN for a given separating line orientation triple.

In the following pseudo-code, I_i is the index of the current envelope segment of V_i, $i = 1, 2, 3$, and T_i is a scalar indicating the position of t_i on segment $A_i[I_i]B_i[I_i]$, $0 \leq T_i \leq 1$. Initially, $I_1 = I_2 = I_3 = 1$ and $T_1 = T_2 = T_3 = 0$.

The following algorithm fixes P_i at its current position t_i and "bumps" P_j to the nearest point t_j on its envelope such that $P_i + t_i$ and $P_j + t_j$ have a separating line with normal w_{ij}. If a_{ij} is the extreme point of $P_i \oplus -P_j$ in the direction w_{ij}, this condition is $w_{ij} \cdot (t_j - t_i) \geq w_{ij} \cdot a_{ij}$ or $w_{ij} \cdot (t_j - t_i - a_{ij}) \geq 0$. BUMP($i$,$j$) returns "FAILURE" if no such point exists. We assume that BUMP has access to $w_{ij}, a_{ij}, I_i, T_i, I_j, T_j$, the envelopes, N_i and N_j (the number of edges on each envelope). is

BUMP (i, j, w, a)
  $t_i \leftarrow (1 - T_i) \cdot A_i[I_i] + T_i \cdot B_i[I_i]$
  $t_j \leftarrow (1 - T_j) \cdot A_j[I_j] + T_j \cdot B_j[I_j]$
  if $w_{ij} \cdot (t_j - t_i - a_{ij}) \geq 0$ **then**
    return "SUCCESS"

  repeat
    $\beta \leftarrow w_{ij} \cdot (B_j[I_j] - t_i - a_{ij})$

[3]In this paper, we always choose w_{ij} to be normal to an edge $a_{ij}b_{ij}$ of $P_i \oplus -P_j$. In this case, both a_{ij} and b_{ij} are extreme.

if $\beta < 0$ **then** $I_j \leftarrow I_j + 1$
if $I_j > N_j$ **then return** "FAILURE"
until $\beta \geq 0$

$\alpha \leftarrow w_{ij} \cdot (A_j[I_j] - t_i - a_{ij})$
if $\alpha \geq 0$ **then**
$T_j \leftarrow 0$
else
$T_j \leftarrow \dfrac{-\alpha}{\beta - \alpha}$
$t_j \leftarrow (1 - T_j) \cdot A_j[I_j] + T_j \cdot B_j[I_j]$
return "SUCCESS"

Note that in the final "else" clause, we have determined that $P_j + A_j[I_j]$ overlaps the separating line but $P_j + B_j[I_j]$ does not. Therefore we use a simple linear interpolation to determine the first valid point on the segment $A_j[I_j]B_j[I_j]$.

Carrousel Algorithm The carrousel algorithm simply applies BUMP(1,2), BUMP(2,3), BUMP(3,1), and so forth. A "turn" is three consecutive bumps. There are several cases.

- If any BUMP returns "FAILURE", then the carrousel algorithm returns "FAILURE".
- If the carrousel makes a complete "turn" without any t_i changing, then it returns "SUCCESS".
- If the carrousel makes two turns without incrementing any I_j, then it is "stuck".

To "unstick" a carrousel, we do the following. First, advance t_i to $B_i[I_i]$, $i = 1, 2, 3$: set $T_1, T_2, T_3 = 1$. Second, run the carrousel *backwards*: BUMP(1,3), BUMP(3,2), BUMP(2,1), and so forth. Third, if the carrousel does not find a solution within two turns, then stop it and start running it forwards again! (In fact, we can stop the backwards carrousel after it has bumped two of the t_i away from $B_i[I_i]$ but just before it moves the third t_i away from the corresponding $B_i[I_i]$.)

Since the carrousel algorithm makes only a constant number of turns before incrementing some I_i, it has running time in $\mathrm{O}(|V_1| + |V_2| + |V_3|)$. Since $|V_i| \in \mathrm{O}(mn)$ (Section 1.3), iterating over all $\mathrm{O}(m^2)$ separating line orientation triples takes time in $\mathrm{O}(m^3 n)$ for simple V_i and $\mathrm{O}(m^3 n \log n)$ for general polygonal V_i.

3.4 Proof of Correctness and Running Time

Definition 3.2 (Carrousel Invariant) *The carrousel position, I_i, T_i, and*

$$t_i = (1 - T_i) \cdot A_i[I_i] + T_i \cdot B_i[I_i],$$

satisfies the **carrousel invariant** *if every valid configuration lies "past" t_1, t_2, t_3. In particular, let t'_1, t'_2, t'_3 be a valid configuration for the current separating line orientation triple w_{12}, w_{23}, w_{31} on the original angle-visible envelopes. Then,*

$$t'_i \in \{t_i B_i[I_i], A_i[I_i+1]B_i[I_i+1], A_i[I_i+2]B_i[I_i+2], \ldots, A_i[N_i]B_i[N_i]\}, \qquad i = 1, 2, 3.$$

Lemma 3.4 (Bumping Lemma) BUMP(i, j) *preserves the carrousel invariant.*

Proof: We prove the lemma for $i = 1$ and $j = 2$. According to Lemma 3.2, the envelopes

$$\{A_1[1]B_1[1], \ldots, A_1[N_1]B_1[N_1]\} \text{ and } \{A_2[1]B_2[1], \ldots, A_2[N_2]B_2[N_2]\}$$

are monotonic with respect to w_{12}. Let t'_1, t'_2, t'_3 be any valid configuration, then by the invariant,

$$w_{12} \cdot t'_1 \geq w_{12} \cdot t_1.$$

Because of the separating line constraint,

$$w_{12} \cdot (t'_2 - t'_1) \geq w_{12} \cdot a_{12}.$$

Therefore,

$$w_{12} \cdot (t'_2 - t_1) \geq w_{12} \cdot a_{12}.$$

BUMP(1,2) moves t_2 to the nearest point t_2^{bump} on the envelope that satisfies the separating line constraint:

$$w_{12} \cdot t_2^{\text{bump}} \geq w_{12} \cdot t_2 w_{12} \cdot (t_2^{\text{bump}} - t_1) \geq w_{12} \cdot a_{12}, \text{ and } w_{12} \cdot t_2^{\text{bump}} \text{ is minimal.}$$

Therefore, t'_2 is "farther" than t_2^{bump}:

$$w_{12} \cdot t'_2 \geq w_{12} \cdot t_2^{\text{bump}}.$$

Since the envelope is monotonic with respect to w_{12}, this means that

$$t'_2 \in \{t_2^{\text{bump}} B_2[I_2], A_2[I_2+1]B_2[I_2+1], A_2[I_2+2]B_2[I_2+2], \ldots, A_2[N_2]B_2[N_2]\}.$$

Since t_1 and t_3 have not moved, the invariant is still satisfied. ∎

Lemma 3.5 *If there is a valid configuration t_1, t_2, t_3 with separating line triple w_{12}, w_{23}, w_{31} such that*

$$t_i \in A_i[I_i]B_i[I_i], \qquad i = 1, 2, 3,$$

then there is a valid configuration t'_1, t'_2, t'_3 such that one of the t'_i is at an endpoint: $T_i = 0$ or $T_i = 1$ for some $i \in \{1, 2, 3\}$.

Proof: Choose the valid configuration such that the separation $w_{31} \cdot (t_1 - t_3)$ between t_1 and t_3 is maximized. Clearly, t_1, t_2, and t_3 have one degree of freedom each. The separating line constraints for P_1, P_2 and for P_2, P_3 imply two linear constraints. There are six endpoint constraints. Since, there are three degrees of freedom, we can assume that three constraints are "equal". Therefore, some t_i must equal an endpoint.

∎

Theorem 3.6 *If there is a valid configuration for a given orientation triple, the carrousel algorithm finds a valid configuration.*

Proof: Lemma 3.3 says that if a valid configuration exists for a given orientation triple, then there is a valid configuration on the angle-visible envelopes. The carrousel algorithm initializes t_i to the "first" point $A_i[1]$, $i = 1, 2, 3$, on its envelope. This initial setting trivially satisfies the carrousel invariant. By Lemma 3.4, each application of BUMP preserves the invariant.

The only question is the method used to "unstick" the carrousel. Clearly, it advances one of the t_i to the endpoint $B_i[I_i]$ of its current segment $A_i[I_i]B_i[I_i]$, and the next forward "turn" of the carrousel will therefore increment I_i. Thus the carrousel makes only a constant number of turns before incrementing some I_i, which implies that the algorithm terminates. If there is a solution on the "sticky" segments, then Lemma 3.5 implies that there is a solution with one t'_i at an endpoint. Therefore, either the forwards carrousel or the backwards carrousel will find it. The only question is whether unsticking maintains the carrousel invariant in the case that there is no valid configuration for those three line segments.

Assume, w.l.o.g., that t_3 is the last t_i to leave its endpoint $B_i[I_i]$ as the result of the backwards carrousel. Just before it does this, the backwards carrousel has executed BUMP(3,2) followed by BUMP(2,1). Polygon $P_1 + t_1$ is "in contact" with $P_2 + t_2$, and polygon $P_2 + t_2$ is "in contact" with $P_3 + t_3$. Call this configuration $\hat{t}_1, \hat{t}_2, \hat{t}_3$. We want to show that we could have reached this state through some hypothetical invariant-preserving *forward* motion.

Here is how to construct the hypothetical motion. First, modify BUMP, so that t_1 is not permitted to move past $\hat{t}_1$. Bumping the full amount satisfies the invariant, and therefore bumping only part of the way definitely satisfies the invariant. The only question is whether this modified forward carrousel will *reach*

$\hat{t}_1, \hat{t}_2, \hat{t}_3$. However, bumping is a simple linear operation. If t_1 does not reach $\hat{t}_1$, then it must converge to a limit, and this limit would imply a valid configuration. Since we are assuming at this point that there is no valid configuration on these segments, there can be no limit. Thus, t_1 reaches $\hat{t}_1$, and the next two bumps move t_2 to $\hat{t}_2$ and t_3 to $\hat{t}_3$. ■

Theorem 3.7 *The total cost of the* 3CN *algorithm is* $O(m^3 n)$ *for simple* V_i *and* $O(m^3 n \log n)$ *for general polygonal* V_i.

Proof: Since the carrousel algorithm makes only a constant number of turns before incrementing some I_i, it has running time in $O(|V_1| + |V_2| + |V_3|)$. Since $|V_i| \in O(mn)$ (Section 1.3), iterating over all $O(m^2)$ separating line orientation triples takes time in $O(m^3 n)$. By Lemma 3.1, the algorithm must compute $O(m)$ envelopes for the triple contact cases and $O(m^2)$ envelopes for the double contact cases. By Lemma 3.2, the cost of computing each envelope is $O(mn)$ for simple V_i and $O(mn \log n)$ for general polygonal V_i. Multiplying times $O(m^2)$ orientation triples give the bounds stated in the lemma. ■

3.5 Practical Improvements to Running Time

The carrousel algorithm runs in time linear in the size of the valid regions V_1, V_2, and V_3. Furthermore, it has a nice behavior when the answer is "no." If the current positions of the polygons are nowhere near a solution, then each "bump" will tend to push each polygon far forward. To take advantage of this observation, we could replace the inner "do-while" by a binary search on the remaining segments of the envelope of V_j.

Unfortunately, the 3CN algorithm must still iterate over $\Omega(m^2)$ separating line triples, particularly when the answer is "no." In terms of s-analysis (see Part I: *An Approximate Algorithm* [3]), the running time is $m^2 s$. Here is where a multiple resolution technique can be very helpful in practice.

Inscribe an inner approximating polygon into $P_i \oplus -P_j$ by joining four of its vertices. We choose the vertices which maximize distance in the directions $(1,1)$, $(-1,1)$, $(-1,-1)$, $(1,-1)$. Generate all 4^3 orientation triples for the edges of these inner approximations. For each triple, employ the carrousel algorithm to test for a solution. If no solution exists, reject the triple. If there is a solution, check to see if it is a solution with respect to the original Minkowski sums. If so, return it. Otherwise, increase the resolution in the following manner. Out of the inner approximation edges, choose the one which is the worst approximation and refine it. We measure badness by either counting the number of skipped vertices or by measuring the area of the "ear" that the edge lops off the Minkowski sum. Refinement consists of choosing a vertex that is skipped by the edge and replacing the edge by two edges. For instance, if the current edge skips from vertex 17 to vertex 29 of the Minkowski sum, then we might replace it with edge 17,23 and edge 23,29. We currently choose the vertex whose index is midway between the endpoints of the edge. We might instead choose the vertex that is farthest from the edge. When an edge is split, each orientation triple which uses that edge are split into two triples.

Since it is difficult to extend Lemma 3.1 to inner approximations, it is safest to run the multiple resolution method as described. In the worst case, this method generates $\Omega(m^3)$ triples, which is more than the $O(m^2)$ method. In practice, the multiple resolution method checks one to two orders of magnitude fewer triples. It appears to eliminate at least a factor of m.

4 kNN Containment

This section presents an algorithm for kNN: place k nonconvex polygons into a nonconvex container. We present this algorithm before the algorithm for kCN (k convex polygons), because the latter algorithm uses a one dimensional version of the kNN algorithm.

For kNN containment, the polygons are nonconvex, and separating lines do not necessarily exist. Yet, in a sense, the multiple resolution technique for handling the separating lines in the case of 3CN provides the inspiration for the kNN algorithm.

The multiple resolution technique covers the exterior of $P_i \oplus -P_j$, which is U_{ij}, with a larger and larger set of half-planes. As the number of half-planes increases, the cover becomes a better approximation to U_{ij}. The kNN algorithm, described below, covers each V_i and each U_{ij} with a binary tree of *convex* sets. At each level of the tree, the cover is a closer approximation to the actual set. The algorithm uses linear programming to evaluate each choice of convex cover polygons.

Section 4.1 presents an algorithm for kNN. Section 4.2 proves this algorithm correctly solves the kNN containment problem in time

$$\mathrm{O}((mn)^{2k+1}\mathrm{LP}(2k, 2k(2k+1)mn + k(k-1)m^2)),$$

where $\mathrm{LP}(a,b)$ is the cost of solving a linear program with a variables and b constraints. The algorithm requires the assumption that the set of valid configurations is not "flat": it contains an open ball. In other words, none of the polygons are "snug". The analysis requires the assumption that certain sets of planes in $\mathbf{R}^{2k}$ are in *general position*: no point lies on $2k+1$ of these planes. Section 4.3 describes how to remove these two assumptions, both in theory and in practice.

4.1 Algorithm

We are given polygons $P_1, P_2, P_3, \ldots, P_k$ and a container polygon C. We define $P_0 = \overline{C}$ and

$$U_{ij} = \overline{P_i \oplus -P_j}, 0 \le i < j \le k.$$

Under this definition, for $1 \le i < j \le k$, U_{ij} equals the U_{ij} defined in Section 1.1. For $1 \le i \le k$,

$$U_{0i} = \overline{P_0 \oplus -P_i} = \overline{\overline{C} \oplus -P_i} = V_i.$$

We define $t_0 = (0,0)$ (the container is fixed). The list of vectors $\langle t_1, t_2, t_3, \ldots, t_k\rangle$ is a *valid configuration* (solution to containment) if

$$t_j - t_i \in U_{ij},\ 0 \le i < j \le k, \tag{11}$$

We define $\mathcal{U} = \langle U_{ij}, 0 \le i < j \le k\rangle$ to be the list of all U_{ij} polygons. The function KNN takes as input the list $\mathcal{U}$ plus a list $\mathcal{C} = \langle C_{ij}, 0 \le i < j \le k\rangle$ of *convex* polygons. The function EVALUATE tests for a solution $\langle t_1, t_2, \ldots, t_k\rangle$ to kNN such that each $t_j - t_i$ lies inside C_{ij}. It returns YES, NO, or MAYBE. The function SUBDIVIDE "splits" $\mathcal{C}$ into $\mathcal{C}^-$ and $\mathcal{C}^+$, and KNN calls itself recursively with these two convex lists. The following is the pseudo-code for KNN:

KNN($\mathcal{U}$, $\mathcal{C}$)
  ans ← EVALUATE($\mathcal{U}$, $\mathcal{C}$)
  if *ans* = YES or *ans* = NO **return** *ans*
  $\mathcal{C}^-$,$\mathcal{C}^+$ ← SUBDIVIDE($\mathcal{U}$, $\mathcal{C}$)
  if KNN($\mathcal{U}$, $\mathcal{C}^-$) = YES **return** YES
  if KNN($\mathcal{U}$, $\mathcal{C}^+$) = YES **return** YES
  return NO

The following section describes how to generate an initial list $\mathcal{C}^{\mathrm{init}}$ of convex polygons. Section 4.1.2 defines SUBDIVIDE, and Section 4.1.3 defines EVALUATE.

4.1.1 Initial Convex List

To solve containment, we must compute an initial list $\mathcal{C}^{\mathrm{init}}$ and call KNN($\mathcal{U}, \mathcal{C}^{\mathrm{init}}$). Let $\mathrm{CH}(V_i)$ denote the convex hull of V_i. We set,

$$C_{0i}^{\mathrm{init}} = \mathrm{CH}(V_i),\ 1 \le i \le k, \tag{12}$$

$$C_{ij}^{\mathrm{init}} = \mathrm{CH}(V_j) \oplus -\mathrm{CH}(V_i),\ 1 \le i < j \le k, \tag{13}$$

The following lemma asserts that Equations 12 and 13 yield a good initial set of C_{ij}.

Lemma 4.1 *If $\langle t_1, t_2, \ldots, t_k \rangle$ is a valid configuration, then $t_j - t_i \in C_{ij}^{\text{init}}$, for $0 \le i < j \le k$.*

Proof:

$$\begin{array}{lll} \text{For} & 1 \le i \le k, & t_i - t_0 \in U_{0i} = V_i \subseteq \mathrm{CH}(V_i) = C_{0i}^{\text{init}}. \\ \text{For} & 1 \le i < j \le k, & t_j - t_i = (t_j - t_0) + -(t_i - t_0) \in \mathrm{CH}(V_j) \oplus -\mathrm{CH}(V_i) = C_{ij}^{\text{init}}. \end{array}$$

■

4.1.2 Subdivision

The definitions of SUBDIVIDE and EVALUATE depend on each other, and it is simpler to define SUBDIVIDE first. KNN only calls SUBDIVIDE($\mathcal{U}, \mathcal{C}$) if EVALUATE($\mathcal{U}, \mathcal{C}$) returns MAYBE, which implies three conditions.

- EVALUATE has computed $t_1, t_2, t_3, \ldots, t_k$ which satisfy $t_j - t_i \in C_{ij}$, for all i, j, $0 \le i, j \le k$.
- For all i, j, $0 \le i, j \le k$, interior$(C_{ij}) \cap U_{ij} \neq \emptyset$.
- For *some* i, j, $0 \le i, j \le k$, $t_j - t_i \notin U_{ij}$.

For the i, j in the third condition, $t_j - t_i \in C_{ij} \cap \overline{U_{ij}}$. Since U_{ij} is closed, $\overline{U_{ij}}$ is open, and therefore interior$(C_{ij}) \cap \overline{U_{ij}} \neq \emptyset$, even if $t_j - t_i \in$ boundary(C_{ij}). Therefore, U_{ij} must have some part of its boundary interior to C_{ij}.

As illustrated in Figure 10, SUBDIVIDE computes the nearest edge e of interior$(C_{ij}) \cap U_{ij}$ that is visible from $t_j - t_i$. It extends e into a line L. Let H^- and H^+ be the half planes bounded by L such that $t_j - t_i \in H^-$. It cuts C_{ij} into $C_{ij}^- = H^- \cap C_{ij}$ and $C_{ij}^+ = H^+ \cap C_{ij}$. Then it makes two copies $\mathcal{C}^-$ and $\mathcal{C}^+$ of $\mathcal{C}$. In $\mathcal{C}^-$ it replaces C_{ij} by C_{ij}^-. In $\mathcal{C}^+$ it replaces C_{ij} by C_{ij}^+.

The process of SUBDIVISION defines a mathematical *subdivision tree* with root $\mathcal{C}^{\text{init}}$. The internal nodes are lists $\mathcal{C}$ such that EVALUATE returns MAYBE. The leaves are lists for which EVALUATE returns YES or NO. If $\mathcal{C}$ is an internal node, its children are $\mathcal{C}^-$ and $\mathcal{C}^+$.

As far as the asymptotic running time bound is concerned, it does not matter that e is the nearest edge to $t_j - t_i$, but it does matter that it intersects the interior of C_{ij}. As one can see from the figure, if $t_j - t_i$ lies in a concave "bay" of nonconvex U_{ij}, cutting U_{ij} along the line L of the nearest edge e will tend to separate U_{ij} into regions with smaller bays in the vicinity of $t_j - t_i$. Thus, choosing the nearest edge helps gets rid of "bad" bays. Incidentally, in our practical (implemented) version of the algorithm, if U_{ij} has multiple components, then subdivision chooses C_{ij}^- to be the convex hull of the nearest component and C_{ij}^+ to be the convex hull of the rest of the components. This technique avoids unnecessary cutting if, for example, all the components of U_{ij} are convex.

4.1.3 Evaluation

This section defines a function EVALUATE that evaluates a list of configuration spaces $\mathcal{U} = \langle U_{ij}, 0 \le i < j \le k \rangle$ with respect to a list of convex sets $\mathcal{C} = \langle C_{ij}, 0 \le i < j \le k \rangle$. EVALUATE returns YES, NO, or MAYBE. For later reference, we number the three ways EVALUATE can say NO.

1) If interior$(C_{ij}) \cap U_{ij} = \emptyset$ for some i, j, $0 \le i < j \le k$, then EVALUATE returns NO.

If $\mathcal{C}$ is the "left" child $\mathcal{C}^-$ of its parent in the subdivision tree, then EVALUATE returns MAYBE. The configuration $\langle t_1, t_2, t_3, \ldots, t_k \rangle$ which satisfies the three conditions of Section 4.1.2 with respect to the parent will also satisfy the conditions with respect to $\mathcal{C}$.

We consider the coordinates of $\langle t_1, t_2, \ldots, t_k \rangle$ to be $2k$ variables, and hence $\langle t_1, t_2, \ldots, t_k \rangle \in \mathbf{R}^{2k}$. Because a convex polygon is the intersection of the half planes determined by its edges, the condition $t_j - t_i \in C_{ij}$ is the intersection of $|C_{ij}|$ *edge constraints* of the form $t_j - t_i \in \mathrm{H}(e)$, where $\mathrm{H}(e)$ is the half plane containing

C_{ij} with e on its boundary. We can also consider an *edge constraint* to be a half *space* constraint on $\langle t_1, t_2, \ldots, t_k \rangle \in \mathbf{R}^{2k}$. In what follows, $\mathcal{H}(\mathcal{C})$ denotes the set of half spaces in $\mathbf{R}^{2k}$ corresponding to all the edge constraints of all the C_{ij} in $\mathcal{C}$.

A half space is equivalent to a linear inequality on $\langle t_1, t_2, \ldots, t_k \rangle$. We define the *constraint linear program* CLP($\mathcal{C}$) to be the linear program with the inequalities implied by $\mathcal{H}(\mathcal{C})$ and the following objective. If $\mathcal{C}$ is the "right" child $\mathcal{C}^+$ of its parent, the objective is to maximize $d \cdot (t_j - t_i)$ where d is the inward pointing normal to H^+. If $\mathcal{C}$ is the root $\mathcal{C}^{\text{init}}$, then we can use an arbitrary objective.

Using any linear programming algorithm, EVALUATE solves CLP($\mathcal{C}$). 2) If it is infeasible, EVALUATE returns NO. If the solution $\langle t_1, t_2, \ldots, t_k \rangle$ satisfies $t_j - t_i \in U_{ij}$, for $0 \leq i < j \leq k$, then EVALUATE returns YES (and presumably reports this solution to kNN). 3) If $\mathcal{C}$ is a "right" child and if the solution $\langle t_1, t_2, \ldots, t_k \rangle$ lies on the boundary of H^+, EVALUATE returns NO. Otherwise, EVALUATE returns MAYBE.

The following modification improves the running time in practice without spoiling the theoretical running time. If EVALUATE is about to return MAYBE, it uses linear programming to solve,

$$t_j - t_i \in \text{CH}(C_{ij} \cap U_{ij}), 0 \leq i < j \leq k.$$

It can use any objective function. If this program is infeasible, it returns NO. If the solution also satisfies Equation 11, it returns YES. Otherwise, it returns MAYBE.

4.2 Analysis

This section analyzes the kNN algorithm. Section 4.2.1 proves correctness. Section 4.2.2 counts lines and edges. Section 4.2.3 proves the running time bound.

In what follows, if $\mathcal{H}$ is a set of half spaces, we denote $\cap\mathcal{H} = \bigcap_{H \in \mathcal{H}} H$.

4.2.1 Correctness

Lemma 4.2 *If the set of valid configurations contains an open ball in* $\mathbf{R}^{2k}$*, then* KNN($\mathcal{U}, \mathcal{C}^{\text{init}}$) *returns* YES.

Proof: By Lemma 4.1, $\cap\mathcal{H}(\mathcal{C}^{\text{init}})$ contains *all* valid configurations, and therefore it contains an open ball of configurations. When SUBDIVIDE splits a list, clearly if $\cap\mathcal{H}(\mathcal{C})$ contains an open ball of valid configurations, then either $\cap\mathcal{H}(\mathcal{C}^-)$ $(= \cap\mathcal{H}(\mathcal{C}) \cap H^-)$ or $\cap\mathcal{H}(\mathcal{C}^+)$ $(= \cap\mathcal{H}(\mathcal{C}) \cap H^+)$ or both contain an open ball of valid configurations.

It suffices to check that EVALUATE does not return NO for any $\mathcal{C}$ such that $\cap\mathcal{H}(\mathcal{C})$ contains an open ball B of valid configurations.

1) Since B is open, $B \subseteq \text{interior}(\cap\mathcal{H}(\mathcal{C}))$, and therefore for each $\langle t_1, t_2, \ldots, t_k \rangle \in B$, $t_j - t_i \in \text{interior}(C_{ij})$, $0 \leq i < j \leq k$. Thus $\text{interior}(C_{ij}) \cap U_{ij} \neq \emptyset$. 2) $\text{interior}(\cap\mathcal{H}(\mathcal{C})) \neq \emptyset$, and thus CLP($\mathcal{C}$) is feasible. 3) If $\mathcal{C}$ is a "right" child, then CLP($\mathcal{C}$) must contain a point not on the boundary of H^+.

KNN terminates because there are only a finite number of edges that SUBDIVIDE can use to cut each convex polygon. ■

Clearly, the running time is bounded by the number of nodes in the subdivision tree times the time to solve CLP($\mathcal{C}$) for each node $\mathcal{C}$. The following section establishes some lemmas needed to count the number of nodes, and Section 4.2.3 proves the running time bound on KNN.

4.2.2 Counting Lines and Edges

Lemma 4.3 *The edges of* V_i*,* $1 \leq i \leq k$*, can be covered by* $2mn$ *lines, and the edges of* U_{ij}*,* $1 \leq i < j \leq k$ *can be covered by* $2m^2$ *lines.*

Proof: Recall that V_i has $\mathrm{O}(m^2n^2)$ edges and that U_{ij} has $\mathrm{O}(m^4)$ edges (for nonconvex P_i polygons). However, each edge of V_i corresponds to a vertex of P_i contacting an edge of C or *vice versa*. For any given vertex-edge contact, the set of valid positions of P_i may correspond multiple ($\mathrm{O}(mn)$) edges of V_i, but all these edges will lie on a single line. The number of ways of selecting a vertex from P_i and an edge from C, or *vice versa*, is $2mn$. The argument for U_{ij} is similar. ∎

Corollary 4.4 *$|\mathrm{CH}(V_i)| \leq 4mn$ and $|\mathrm{CH}(V_j) \oplus -\mathrm{CH}(V_i)| \leq 8mn$.*

Proof: We assume that when we count the edges of $\mathrm{CH}(V_i)$, we consider two collinear edges to be a single edge. The edges and vertices of V_i lie on no more than $2mn$ lines. Only the two extreme vertices on each line can lie on $\mathrm{CH}(V_i)$, hence the lemma. The second part follows from the fact that $|A \oplus B| \leq |A| + |B|$ if A and B are convex polygons. ∎

Lemma 4.5 *The* **subdivision tree** *has height at most $2kmn + k(k-1)m^2$.*

Proof: If a node $\mathcal{C}$ has children, then for some i, j, $0 \leq i < j \leq k$, SUBDIVIDE splits a member C_{ij} of $\mathcal{C}$ into C_{ij}^- and C_{ij}^+. SUBDIVIDE splits C_{ij} using a line L containing some edge e of U_{ij}. At least part of e lies in the interior of C_{ij}.

Line L does not intersect the interiors of C_{ij}^- and C_{ij}^+. Nor does it intersect the interior of C'_{ij} for any descendant $\mathcal{C}'$ of $\mathcal{C}$. Therefore, no edge of U_{ij} which lies on L can appear as a "splitting edge" in the subtree below $\mathcal{C}$. Thus, for every path from root to a leaf of the subdivision tree, the line L covering edge e of U_{ij}, $0 \leq i < j \leq k$, appears at most once. By Lemma 4.3, there are at most

$$k(2mn) + \frac{k(k-1)}{2}(2m^2) = 2kmn + k(k-1)m^2$$

such lines. ∎

4.2.3 Running Time Bound

Theorem 4.6 *Under the assumption that k is constant and $m < n$,* KNN *has running time*

$$\mathrm{O}((mn)^{2k+1}\mathrm{LP}(2k, 2k(2k+1)mn + k(k-1)m^2)),$$

where $\mathrm{LP}(a, b)$ is the cost of solving a linear programs with a variables and b constraints.

This section provides the proof of Theorem 4.6. For simplicity, the proof relies on the assumption that half spaces in $\mathcal{H}(\mathcal{C})$ are in *general position*: no point in $\mathbf{R}^{2k}$ lies on the boundary of more than $2k$ half spaces. Section 4.3 explains how to remove this assumption from the proof. One more lemma is needed to prove the theorem.

Given a set of half spaces $\mathcal{H}$ and a vertex v of $\cap\mathcal{H}$, $\mathrm{FACES}(v, \mathcal{H})$ denotes the set of half planes $H \in \mathcal{H}$ such that H corresponds to a face of $\cap\mathcal{H}$ that touches v. Assuming general position, $|\mathrm{FACES}(v, \mathcal{H})| = 2k$.

Lemma 4.7 *Let $\mathcal{C}'$ and $\mathcal{C}''$ be "cousins" in the subdivision tree (neither is an ancestor of the other) such that* EVALUATE *returns* MAYBE *on each. Let v' and v'' be vertices of $\cap\mathcal{H}(\mathcal{C}')$ and $\cap\mathcal{H}(\mathcal{C}'')$, respectively. Then* $\mathrm{FACES}(v', \mathcal{H}(\mathcal{C}')) \neq \mathrm{FACES}(v'', \mathcal{H}(\mathcal{C}''))$.

Proof: Suppose $\mathcal{C}'$ and $\mathcal{C}''$ belong to the left and right subtrees, respectively, of their lowest common ancestor $\mathcal{C}$. Then $\cap\mathcal{H}(\mathcal{C}') \subseteq \cap\mathcal{H}(\mathcal{C}) \cap H^-$ and $\cap\mathcal{H}(\mathcal{C}'') \subseteq \cap\mathcal{H}(\mathcal{C}) \cap H^+$. Assume that $\mathrm{FACES}(v', \mathcal{H}(\mathcal{C}')) = \mathrm{FACES}(v'', \mathcal{H}(\mathcal{C}''))$. But the location of a vertex is uniquely determined by its neighboring faces, and therefore $v' = v''$ and both lie on the common boundary of H^+ and H^-. By the assumption of general position, boundary(H^-) is one of the $2k$ planes that pass through v'. Since, $H^- \in \mathcal{H}(\mathcal{C}')$, $H^- \in \mathrm{FACES}(v', \mathcal{H}(\mathcal{C}'))$. But $H^- \notin \mathcal{H}(\mathcal{C}'')$ and thus $H^- \notin \mathrm{FACES}(v'', \mathcal{H}(\mathcal{C}''))$. This contradicts the assumption that $\mathrm{FACES}(v', \mathcal{H}(\mathcal{C}')) = \mathrm{FACES}(v'', \mathcal{H}(\mathcal{C}''))$. ∎

Here is the proof of Theorem 4.6.

Proof: Each "YES" node and "NO" node in the subdivision tree is the child of a "MAYBE" node. The total number of nodes is at most three times the number of "MAYBE" nodes.

We can label each "MAYBE" node $\mathcal{C}$ in the subdivision tree by choosing an arbitrary vertex of $\cap\mathcal{H}(\mathcal{C})$ and labeling $\mathcal{C}$ with the set $\mathrm{FACES}(v, \mathcal{H}(\mathcal{C}))$. By Lemma 4.3, there are $\mathrm{O}(mn)$ choices of half spaces. The set $\mathrm{FACES}(v, \mathcal{H}(\mathcal{C}))$ has $2k$ elements, and thus there are $\mathrm{O}((mn)^{2k})$ possible labels. By Lemma 4.7, a label can only be repeated on one path from the root to a leaf, and by Lemma 4.5, this path has length $\mathrm{O}(mn)$, and thus there are $\mathrm{O}((mn)^{2k+1})$ "MAYBE" nodes.

The cost of solving $\mathrm{CLP}(\mathcal{C})$ dominates the cost of evaluating $\mathcal{C}$. Each linear program has $2k$ variables. By Corollary 4.4 and Lemma 4.5,

$$|\mathcal{H}(\mathcal{C})| \leq k(4mn) + \frac{k(k-1)}{2}(8mn) + 2kmn + k(k-1)m^2 = 2k(2k+1)mn + k(k-1)m^2.$$

Hence the running time of the KNN algorithm is

$$\mathrm{O}((mn)^{2k+1}\mathrm{LP}(2k, 2k(2k+1)mn + k(k-1)m^2)).$$

■

4.3 Dealing with Degenerate Cases

There are two types of degeneracy that can arise in the solution of kNN containment. First, the planes of the half spaces might not be in general position. Second, the set of valid configurations might be "flat" and not contain an open ball. Dealing with the first degeneracy requires a change in the *proof.* Dealing with the second degeneracy requires a change in the *algorithm.*

4.3.1 Nongeneral Position

A technique called *symbolic perturbation* permits one to assume general position for a set of planes, particular for intersection of half spaces [6, 17].

Let $\mathcal{H} = \{H_1, H_2, \ldots, H_N\}$ be a set of half spaces. Let $H_i(\epsilon_i) \subset H_i$ denote a half space whose boundary is parallel to that of H_i but ϵ_i distant. The theory of symbolic perturbation ensures that we can choose β_i, $1 \leq i \leq N$,

$$0 < \beta_1 << \beta_2 << \beta_3 << \cdots << \beta_N << 1,$$

such that for each choice of ϵ_i, $0 < \epsilon_i < \beta_i$, the intersection,

$$\cap\mathcal{H}(\overline{\epsilon}) = \bigcap_{i=1}^{N} H_i(\epsilon_i),$$

has the following properties.

- Each $\cap\mathcal{H}(\overline{\epsilon})$ has the same combinatorial structure.
- Each vertex of $\cap\mathcal{H}(\overline{\epsilon})$ has exactly $2k$ neighboring faces.
- If $\cap\mathcal{H}$ touches the boundary of H_i, then $H_i(\epsilon_i)$ appears as a face of $\cap\mathcal{H}(\overline{\epsilon})$.

In the proof of KNN, we use an arbitrary vertex of $\cap\mathcal{H}(\overline{\epsilon})$ to generate a label.

Note that if $\cap\mathcal{H}$ is "flat" ($\mathrm{interior}(\mathcal{H}) = \emptyset$), then $\cap\mathcal{H}(\overline{\epsilon}) = \emptyset$, so we cannot select a vertex! Fortunately, we do not have to generate a label in this case because EVALUATE returns NO when $\cap\mathcal{H}(\mathcal{C})$ is "flat": the solution to $\mathrm{CLP}(\mathcal{C})$ lies on the boundary of H^+.

4.3.2 "Flat" Sets of Valid Configurations

A modified version of the kNN algorithm can find a solution if the set of valid configurations is "flat": it does not contain an open ball. First, we expand each C_{ij}^{init} slightly to make sure all solutions lie in the *interior* of $\mathcal{C}^{\text{init}}$. Next, we modify SUBDIVISION to split C_{ij} into *three* subsets: $\mathcal{C}^-$, $\mathcal{C}^+$, and $\mathcal{C}^0 = L \cap C_{ij}$. Finally, when EVALUATE checks the condition interior$(C_{ij}) \cap U_{ij} = \emptyset$, it uses the *relative* interior of C_{ij}. If C_{ij} is a line segment, its interior is the segment minus its endpoints.

As one can imagine, the subdivision tree becomes "tri-ary" and the node labels include hyperplanes in addition to half spaces. The proof is similar but more complicated.

In practice, we can "shrink" each P_i slightly, and that will ensure that the set of valid configurations includes an open ball. Of course, shrinking might lead to some slightly overlapped solutions for the original P_i, but the overlap can be made to be less than the cutting tolerance for the machines that cuts the polygon shapes out of stock material. If Q is a square or any small "round" polygon containing $(0,0)$, we can replace P_i by

$$P_i' = P_i \ominus Q = \overline{\overline{P_i} \oplus -Q}.$$

If $\langle t_1, t_2, \ldots, t_k \rangle$ is a valid configuration for $P_1, P_2, \ldots, P_k$, then for any $t_i' \in t_i + Q$, $1 \leq i \leq k$, $t_1', t_2', \ldots, t_k'$ is a valid configuration for $P_1', P_2', \ldots, P_k'$.

5 kCN Containment

This section examines the kCN containment problem: place k convex polygons into a nonconvex container. As k increases beyond 3, the containment problem becomes more difficult. Recall that the valid region V_i is the set of translations such that $t_i \in V_i$ implies $P_i + t_i \subseteq C$. Section 2 showed that a solution t_1, t_2 to 2CN can be found at the vertices of V_1 and V_2, if a solution exists. Section 3 showed that a solution t_1, t_2, t_3 to 3CN can be found on the boundary of V_1, V_2, and V_3. Unfortunately, there are instances of 4CN for which no solution has all t_i on the boundary of the respective V_i, $i = 1, 2, 3, 4$.

Fortunately, there are still at least two ways in which kCN is less complex than kNN (placing k *nonconvex* polygons). First, for kCN there are only $\mathrm{O}(m^{2k} n^k)$ "potential" solutions, as opposed to the $\mathrm{O}((mn)^{2k})$ potential solutions for kNN. Second, it is always possible to "disassemble" a set of nonoverlapping convex polygons in any given direction d. In other words, it is possible to index the polygons such that P_1 can be removed in the direction d, followed by P_2, P_3, and so forth. For nonconvex polygons, there may be no direction d for which disassembly is possible.

Pick a direction d, such as $d = (-1, 0)$. Assuming we have the correct indexing for $P_1, P_2, P_3, \ldots, P_k$, we can move P_1 to the left (direction $(-1,0)$) until it hits (the boundary of) the container C. Its translation vector t_1 now lies on the "left envelope" of its valid region V_1: the set of points of V_1 that can "see to infinity" in the direction $(-1, 0)$. Move P_2 left until it hits $P_1 + t_1$ or C: its translation vector t_2 now lies on the boundary of $t_1 + U_{12}$ or on the left envelope of V_2. Similarly, for $i = 3, 4, \ldots, k$, move P_i left until it hits one of $P_h + t_h$, $1 \leq h < i$, or it hits C: t_i lies on the boundary of $t_h + U_{hi}$ or on the left envelope of V_i. Thus, if we know the y-coordinates of $t_1, t_2, \ldots, t_k$, then we can quickly determine the x-coordinates corresponding to a solution to kCN.

This observation leads to the following idea for a kCN algorithm: sweep t_1 upwards, and for each possible y-coordinate of t_1, recursively solve $(k-1)$CN for $P_2, P_3, \ldots, P_k$. For convex P_i, each valid region has $|V_i| \in \mathrm{O}(mn)$ vertices and each set of valid displacements has $|U_{ij}| \in \mathrm{O}(m)$ vertices (Section 1.2). Thus, as each t_i sweeps upward, it "sees" $\mathrm{O}(mn + km)$ combinatorial changes to its left. One might expect a running time in $\mathrm{O}((mn + km)^k)$, which is $\mathrm{O}((mn)^k)$ for $n > m$ and constant k. Essentially, this idea works, but it requires several steps to achieve it, and the actual running time is $\mathrm{O}((mn)^{k+1})$. The following section outlines these steps and Sections 5.2 to 5.4 carry them out.

5.1 Outline

Section 5.2 characterizes layouts of k convex polygons. It proves the $O(m^{2k}n^k)$ bound on the number of combinatorial cases, and it gives a "naive" algorithm for kCN with running time $O(m^{2k}n^k \log n)$. This section also proves the disassembly lemma for convex polygons.

Section 5.3 gives an algorithm for kCN that assumes that each V_i and U_{ij} is *y-monotone*. By definition, the intersection of any horizontal line with a *y-monotone* polygonal region has zero or one connected component: the empty set, a single point, line segment, ray, or the entire line. Polygons with multiple components can be y-monotone as long as the components do not overlap in y. This algorithm is a modification of the kNN algorithm but using only horizontal cutting lines. It has running time $O((mn)^{k+1})$.

Finally, Section 5.4 shows how to use the algorithm of Section 5.3 to create an algorithm for kCN that does not require y-monotone V_i and U_{ij}. This kCN algorithm has the same running time bound $O((mn)^{k+1})$. This running time is better than the naive kCN algorithm if $n < m^{k-1}$. For $k >= 4$, $m^{k-1} \geq m^3$. For n and m in practice, it is safe to assume that $n < m^3$.

5.2 Characterizing Solutions to kCN Containment

This section characterizes layouts of k convex polygons in a nonconvex container. Section 5.2.1 shows that there are up to $O(m^{2k}n^k)$ combinatorially distinct layouts. This leads immediately to a naive kCN algorithm for kCN with running time in $O(m^{2k}n^k \log n)$. Section 5.2.2 proves that a layout of convex polygons can be "disassembled" in any given direction d.

5.2.1 Counting Combinatorial Cases

As in the case of the naive kNN algorithm (Section 4), we can assume that if a solution to kCN exists, then there exists a solution with exactly $2k$ linearly independent contacts among the polygons and the container.

Lemma 5.1 *There are $O(m^{2k}n^k)$ combinatorially distinct sets of $2k$ contacts for kCN.*

Proof: Single and double contacts between convex polygons P_i and P_j correspond to the edges and vertices, respectively, of $U_{ij} = \overline{P_i \oplus -P_j}$, which has $O(m)$ edges if P_i and P_j are convex. Similarly, single and double contacts between the container C and a convex polygon P_i correspond to the edges and vertices, respectively, of V_i, which has $O(mn)$ edges.

A polygon cannot have *three* linearly independent contacts with the container, because the polygon has only two degrees of freedom. Two polygons cannot have a three linearly independent contacts because contact between the two polygons can only remove two out of the four degrees of freedom shared by the two polygons (the two-polygon assembly can still translate in x or y).

Suppose we choose k_{VE} edges and k_{VV} vertices from among the V_i polygons and we choose k_{UE} edges and k_{UV} vertices from among the U_{ij} polygons. Since there are $2k$ contacts, $k_{VE} + 2k_{VV} + k_{UE} + 2k_{UE} = 2k$, and thus $k_{VE} + k_{VV} + k_{UE} + k_{UE} < k$. Also, $k_{VE} + k_{VV} \leq k$ because we cannot choose more than one vertex or edge from each V_i. Given the number of vertices and edges, the number of ways of making this choice is

$$O((mn)^{k_{VE}}(mn)^{k_{VV}}m^{k_{UE}}m^{k_{UV}}) = O(m^{k_{VE}+k_{VV}+k_{UE}+k_{UV}}n^{k_{VE}+k_{VV}}) \leq O(m^{2k}n^k).$$

■

We can easily modify the naive kNN algorithm to create an algorithm for kCN with running time $O(m^{2k}n^k \log n)$. Simply iterate over all $O(m^{2k}n^k)$ choices of sets of contacts. For each set of contacts, check if they are linearly independent, and if so, solve for the unique configuration $\langle t_1, t_2, \ldots, t_k \rangle$ which has this set of contacts. These two steps can be done in time $O(k^3)$ using standard techniques of linear algebra. Finally, check if the configuration is valid. This can be done in $O(\log n)$ time, as before.

Remember that the naive kCN algorithm is very impractical because it *must* iterate over all $\Omega(m^{2k}n^k)$ contacts if the input has no solution.

5.2.2 Characterizing Solutions to kCN Containment

The following lemma is almost certainly in the robotics literature. For completeness, we give its statement and proof.

Lemma 5.2 *Given a set of nonoverlapping convex polygons $P_1, P_2, P_3, \ldots, P_k$ and a direction vector d, one of the polygons can move to infinity in the direction d without "running into" any of the others.*

Proof: Without loss of generality, we assume that $d = (-1, 0)$. Stand to the left and "above" (greater y-coordinate) the set of polygons. Look to the right and walk "downward" (in the direction of decreasing y-coordinate). The first thing that interrupts your line of sight is the top-most vertex of the top-most polygon. Eventually, you will see the bottom-most vertex of the bottom-most polygon. Consider the *first* polygon P_i whose bottom-most vertex you see. We claim that you must have seen the entire left side of P_i. Suppose not. Then your view of P_i was interrupted for a time. The moment just before you see P_i for a second time, you must have just seen the bottom-most vertex of some other polygon P_j that was blocking your view of P_i. This implies that the bottom-most vertex of P_j is higher than the bottom-most vertex of P_i, a contradiction. Since the entire left side of P_i is visible, it can be moved away from the other polygons in the direction $(-1, 0)$. ■

We call P_i the *left-visible* polygon of the set. If the set of polygons is in a container C, then P_i can move leftward until it touches C. In other words, its position t_i can move leftward until it is on the left envelope of V_i.

We reindex the polygons so that P_1 is left-visible. If P_1 is removed, then P_2 is left-visible, and so forth. Next we can reshape U_{ij} for $1 \leq i < j \leq k$. Since P_j cannot block the leftward motion of P_i for $i < j$, $t_j - t_i$ cannot lie in the portion of U_{ij} to the left of $\overline{U_{ij}}$. Remove this set of points from U_{ij}. More precisely, define $X^- = \{(x, 0) \mid x \leq 0\}$, the negative part of the x-axis. The altered version of U_{ij} is, $U_{ij}^- = \overline{P_i \oplus -P_j \oplus X^-}$. See Figure 11. Incidentally, if P_i were to the *right* of P_j, then we would replace U_{ij} with

$$U_{ij}^+ = -U_{ji}^- = -\overline{P_j \oplus -P_i \oplus X^-}.$$

Since P_i must be either right or left of P_j, $U_{ij} = U_{ij}^- \cup U_{ij}^+$.

5.3 "Y-Monotone" Algorithm for kCN

This section gives an algorithm for kCN that requires V_i, $1 \leq i \leq k$, and U_{ij}, $1 \leq i < j \leq k$, to be y-monotone. As we have seen in the previous section, choosing a particular left to right ordering for the P_i implies that we can replace U_{ij} by U_{ij}^-. Polygon U_{ij}^- is y-monotone. Unfortunately, V_i is not necessarily y-monotone. Section 5.4 extends the algorithm of this section to nonmonotone V_i. For the rest of Section 5.3, we will just assume that V_i and U_{ij} are y-monotone.

The y-monotone algorithm Y-MONOTONE-KCN is a modification of KNN in Section 4.1. Each $\mathcal{C}$ is a list of *rectangles*, and SUBDIVIDE uses only horizontal cutting lines. Sections 5.3.1 through 5.3.3 detail the modifications. Section 5.3.4 proves that the running time is $O((mn)^{2k})$.

5.3.1 Initial List

Let $\mathrm{BBox}(V_i)$ denote the "bounding box" of V_i: the smallest rectangle containing V_i with sides parallel to the x and y axis. We set,

$$\begin{aligned} C_{0i}^{\text{init}} &= \mathrm{BBox}(V_i),\ 1 \leq i \leq k, & (14)\\ C_{ij}^{\text{init}} &= \mathrm{BBox}(V_i) \oplus -\mathrm{BBox}(V_j),\ 1 \leq i < j \leq k, & (15)\end{aligned}$$

Since, $\mathrm{CH}(V_i) \subseteq \mathrm{BBox}(V_i)$, $\mathcal{C}^{\mathrm{init}} = \langle C_{ij}^{\mathrm{init}}, 0 \leq i < j \leq k \rangle$ is a "good" initial list (Lemma 4.1).

5.3.2 SUBDIVIDE

Each C_{ij}, $0 \leq i < j \leq k$, is a rectangle. SUBDIVIDE finds a vertex v of some $C_{ij} \cap U_{ij}$, $0 \leq i < j \leq k$, such that v does not lie on the top or bottom of C_{ij}. SUBDIVIDE creates a horizontal line L through v and cuts C_{ij} into C_{ij}^- and C_{ij}^+ using L. It generates $\mathcal{C}^-$ and $\mathcal{C}^+$ as usual. Figure 12 illustrates the choice of L.

5.3.3 EVALUATE

EVALUATE is the same as the function in Section 4.1.3 with the following modifications. First, since the U_{ij} are y-monotone, horizontal cuts will eventually cut them into trapezoids. If $C_{ij} \cap U_{ij}$ is a trapezoid T_{ij} for every i, j, $0 \leq i < j \leq k$, then EVALUATE uses linear programming to solve $t_j - t_i \in T_{ij}$ with any objective. If there is a solution, it returns YES. If it is infeasible, it returns NO.

Second, since the C_{ij} polygons are rectangles, the constraint $t_j - t_i \in C_{ij}$ consists of independent constraints on the x and y coordinates. EVALUATE solves $\mathrm{CLP}(\mathcal{C})$ independently for the x and y coordinates. In fact, since SUBDIVIDE makes no vertical cuts, the solution for the x coordinates is the same as for $\mathrm{CLP}(\mathcal{C}^{\mathrm{init}})$.

Third, the modification to improve the running time in practice uses bounding boxes in the places of convex hulls (otherwise it will spoil the running time bound).

5.3.4 Analysis

Lemma 5.3 *Assuming $m < n$ and k is constant,* Y-MONOTONE-KCN *has running time* $\mathrm{O}((mn)^{k+1})$.

Proof: Each C_{ij}, $0 \leq i < j \leq k$ implies two constraints on the y coordinates of $t_1, t_2, \ldots, t_k$. Therefore solving $\mathrm{CLP}(\mathcal{C})$ for the y coordinates requires time $\mathrm{LP}(k, 2k(k+1))$. This is a constant. We only have to solve for the x coordinates once, and this also has constant cost.

Except for $\mathcal{C} = \mathcal{C}^{\mathrm{init}}$, solving $\mathrm{CLP}(\mathcal{C})$ requires solving a linear program on the k y-coordinates with horizontal cutting line constraints. Since the y coordinates have only k degrees of freedom, the label $\mathrm{FACES}(v, \mathcal{H})$ of Section 4.2.3 has only k elements. Thus, there are only $\mathrm{O}((mn)^k)$ ways to choose a label. Since the height of the subdivision tree is still $\mathrm{O}(mn)$, the total running time is $\mathrm{O}((mn)^{k+1})$. ■

5.4 General kCN Algorithm

This section gives an algorithm for kCN without the assumption of y-monotone configuration spaces. This algorithm subdivides the V_i and U_{ij} regions into y-monotone subsets and then applies Y-MONOTONE-KCN to each selection of subsets. Despite the fact that this algorithm calls Y-MONOTONE-KCN $\Omega(n^k)$ times, the running time bound of this kCN algorithm is the same as the Y-MONOTONE-KCN algorithm: $\mathrm{O}((mn)^{k+1})$.

5.4.1 Subdivision of V_i and U_{ij}

The KCN algorithm must subdivide V_i, $1 \leq i \leq k$, and U_{ij}, $1 \leq i < j \leq k$, into y-monotone polygons. This section establishes the lemmas required for this purpose.

Lemma 5.4 *If polygon P is convex with m vertices and polygon C is bounded with n vertices, then $V = \overline{C} \oplus -P$ has $\mathrm{O}(n)$ connected components.*

Proof: Fortune [7] proved that V has $\mathrm{O}(mn)$ edges by constructing the Voronoi diagram of polygon C under the convex distance function constructed from P. The diagram has $\mathrm{O}(n)$ vertices and edges. For each component $W \subset V$, consider the largest translated copy $Q + t_{\max}$ of P inside C such that $t_{\max} \in W$. Since Q is bigger than P, $t_{\max}$ is *internal* to W. Polygon $Q + t_{\max}$ must have at least a triple contact with the boundary of C; otherwise, it could be made larger. (There are three degrees of freedom: the two coordinates of $t_{\max}$ and the size of Q.) Therefore $t_{\max}$ is a vertex of the Voronoi diagram. Since there are $\mathrm{O}(n)$ Voronoi vertices, there are $\mathrm{O}(n)$ connected components. ■

Lemma 5.5 *Polygon V of Lemma 5.4 can be cut into $\mathrm{O}(n)$ y-monotone polygons.*

Proof: We first separate V into individual connected components. Next, as illustrated in Figure 13, we cut each component through each reflex vertex v of V that is also an extreme (Section 2.1) vertex of $\overline{V}$ with respect to $(0,1)$ or $(0,-1)$. Such a v is a local maximum or minimum in y of the boundary of V, and it prevents V from being y-monotone. The cut extends from v vertically into the interior of V until it reaches the boundary again.

For a given direction d, each extreme vertex of $\overline{V} = \overline{C} \oplus -P$ is the sum of an extreme vertex c of $\overline{C}$ and p of $-P$. If this were not so, then we could move c or p in a way that increased $d \cdot c$ or $d \cdot p$ and thus increased $d \cdot (c+p) = d \cdot v$. However, $-P$ is convex and has only one extreme in each direction. (Actually, it can have two extreme vertices if some edge of $-P$ has normal equal to d.) It follows that there are $\mathrm{O}(n)$ extreme vertices with respect to $(0,1)$ and $(0,-1)$. Since the number of cuts and the number of components are both linear in n, the lemma is proven. ■

5.4.2 High Level Subdivision

The following is the general algorithm for kCN. Functions EVALUATE and SUBDIVIDE are defined below.

KCN($\mathcal{U}$)
  $ans \leftarrow$ EVALUATE($\mathcal{U}$)
  if ans = YES or ans = NO **return** ans
  $\mathcal{U}'$,$\mathcal{U}''$ $\leftarrow$ SUBDIVIDE($\mathcal{U}$)
  if KCN($\mathcal{U}'$)= YES **return** YES
  if KCN($\mathcal{U}''$)= YES **return** YES
  return NO

EVALUATE first checks to see if all the V_i $(= U_{0i})$ and U_{ij} polygons in $\mathcal{U}$ are y-monotone. If they are, it returns Y-MONOTONE-KCN($\mathcal{U}, \mathcal{C}^{\text{init}}$) for $\mathcal{C}^{\text{init}}$ defined in Section 5.3.1.

If the U_{ij} are not all y-monotone, then EVALUATE returns MAYBE. At this point EVALUATE could apply the same technique as the Y-MONOTONE-KCN evaluation function in Section 5.3.3 to make the algorithm more practical. In practice we might even use the technique described in Section 4.1.3, but that would spoil the theoretical running time bound.

The function SUBDIVIDE selects a U_{ij}, $0 \leq i < j \leq k$, in $\mathcal{U}$ which is not y-monotone. If $i = 0$, then U_{ij} is V_j. If V_j has multiple components, it separates them into two groups. If V_j has a single component, it makes a vertical cut as described in Lemma 5.5. If $i \neq 0$, then SUBDIVIDE separates U_{ij} into U_{ij}^- and U_{ij}^+ (see Section 5.2.2). In either case, SUBDIVIDE has found two subsets U'_{ij} and U''_{ij} which cover U_{ij}. It makes two copies $\mathcal{U}'$ and $\mathcal{U}''$ of $\mathcal{U}$. In $\mathcal{U}'$ it replaces U_{ij} by U'_{ij}. In $\mathcal{U}''$ it replaces U_{ij} by U''_{ij}.

Note: choosing U_{ij}^- in $\mathcal{U}'$ implies the pairwise ordering that P_i is to the left of P_j. SUBDIVIDE takes the transitive closure of all the pairwise orderings implied by $\mathcal{U}'$. If this implies any other pairwise ordering, for instance that P_g is to the right of P_h, SUBDIVIDE makes the appropriate substitution for U_{gh} in $\mathcal{U}'$. In this instance it would replace it by U_{gh}^+. SUBDIVIDE does the same for $\mathcal{U}''$.

5.4.3 Analysis

Theorem 5.6 KCN *is correct and has running time in* $\mathrm{O}((mn)^{k+1})$.

Proof: SUBDIVIDE always chooses $U'_{ij}, U''_{ij} \subset U_{ij}$ such that $U_{ij} = U'_{ij} \cup U''_{ij}$. Therefore there is a solution in $\mathcal{U}$ if and only if there is a solution in $\mathcal{U}'$ or $\mathcal{U}''$. Assuming the call to Y-MONOTONE-KCN in EVALUATE works correctly, then KCN is correct.

Subdividing the U_{ij} polygons, $1 \le i < j \le k$, ultimately selects one of the $k!$ ways of ordering $P_1, P_2, \ldots, P_k$. The order of subdivision does not really matter, but we can assume that SUBDIVIDE subdivides the U_{ij} polygons first. Thus, KCN ultimately calls itself $k!$ times with the original V_i polygons, $1 \le i \le k$, and one of the $k!$ ways of substituting U_{ij}^- or U_{ij}^+ for each U_{ij} polygon. We claim that the cost of each of these calls is $\mathrm{O}((mn)^{k+1})$.

After subdividing all the V_i polygons, KCN ultimately calls Y-MONOTONE-KCN with every way of replacing each V_i with a y-monotone subset V_i^{mono}. The cost of each call to Y-MONOTONE-KCN is mn times the number of ways of selecting k cutting lines from all the V_i and U_{ij} polygons (see the proof of Lemma 5.3). To help the counting argument, assume the first cutting is selected from V_1^{mono} *or* some U_{ij}, $1 \le i < j \le k$. Assume the second cutting line is selected from V_2^{mono} *or* some U_{ij}, and so forth. Since $|U_{ij}| \le 2m$ and the U_{ij} polygons have $k(k-1)m$ edges in total, the total number of ways of selecting cutting lines is bounded by

$$\prod_{i=1}^{k}(|V_i^{\mathrm{mono}}| + k(k-1)m).$$

We can sum this over all ways of selecting y-monotone subsets. The distributive law implies that this sum equals

$$\prod_{i=1}^{k}\sum(|V_i^{\mathrm{mono}}| + k(k-1)m), \tag{16}$$

where each sum is carried out over all the y-monotone subsets $V_i^{\mathrm{mono}} \subseteq V_i$. We claim that

$$\sum(|V_i^{\mathrm{mono}}| + k(k-1)m)$$

is $\mathrm{O}(mn)$. Lemma 5.5 implies that the first terms sum up to no more than $|V_i| + 2n$: the number of edges in V_i plus the number of cut edges. The second term sums up to no more than $k(k+1)mn$ because there are at most n components.

Thus the total number of ways of selecting cutting lines in Equation 16 is $\mathrm{O}((mn)^k)$. Each call to Y-MONOTONE-KCN has an extra factor of mn. Therefore the total cost of all the calls to Y-MONOTONE-KCN is $\mathrm{O}((mn)^{k+1})$. This establishes the theorem. ■

6 Results and Conclusion

6.1 Results

6.1.1 2CN and 3CN

We have implemented versions of the 2CN and 3CN algorithms of Sections 2 and 3, and they run very quickly. Once V_1 and V_2 have been computed, the 2CN algorithm is so very fast that it is not worth comparing it to the more general approximate kNN algorithm (Part I [3]) or exact kCN algorithm (Section 5) applied to the case $k = 2$. Table 1, taken from Part I, compares the performance of the exact 3CN algorithm of Section 3 to that of the approximate kNN algorithm presented in Part I run on the same inputs. For feasible examples (Experiments 1-5), the exact 3CN algorithm performs about seven times faster than the approximate algorithm, on average. For infeasible examples (Experiments 6-10), the exact algorithm is only about twice as fast, on average. To achieve this speed on infeasible inputs, the algorithm must use multiple resolution to find a good orientation triple (Section 3.5). Without multiple resolution, the exact 3CN algorithm is *slower* than the approximate algorithm on infeasible inputs.

Experiment#	1	2	3	4	5	6	7	8	9	10
Type	3CN	3CN	3CN	3CN	3CN	3CN*	3CN*	3CN*	3CN*	3CN*
Approx. kNN	79	107	171	230	55	34	68	35	27	38
Exact 3CN	11	14	26	22	10	15	21	16	15	30

Table 1: Approximate kNN vs. exact 3CN running time (100 calls each). Time is in seconds on a 28 MHz SPARCstation. Infeasibility is denoted by "*".

6.1.2 kNN

Our initial experiments showed that the exact kNN algorithm of Section 4 is often slower than the approximate kNN algorithm of Part I [3]. The exact kNN algorithm "approximates" each U_{ij} configuration space region (or a subset of it) by a convex region C_{ij}.[4] The initial configuration space regions U_{ij}, $0 \leq i < j \leq k$, are highly nonconvex. In fact if $i, j > 0$, then U_{ij} is unbounded. The initial value of C_{ij} (Equations 12 and 13) covers a large subset of $\overline{U_{ij}}$. Much subdivision is required before the C_{ij} regions are exact subsets (or at least close approximations to subsets) of the U_{ij} regions. To make the initial U_{ij} regions more manageable, we apply the *geometric restriction* of Part I,

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

until we reach a steady state. (Recall from Section 1.1 that $U_{ij} = -U_{ji}$.) As shown in Part I, this restriction does not throw away any solutions. The approximate algorithm also applies this restriction as part of its normal operation. For all the experiments comparing the approximate kNN to the exact kNN algorithm, we apply this steady state geometric restriction to the initial hypothesis. The time to perform this initial restriction is included in the running times of both algorithms.

Table 2, also taken from Part I, compares the approximate kNN algorithm of Part I to the (modified) exact kNN algorithm of Section 4. These algorithms have comparable running times, even though they achieve it

Experiment#	1	2	3	4	5	6	7	8	9	10
Type	4NN	4NN	4NN	4NN	4NN	4NN*	4NN*	5CC	6CC	7CC
Approx. kNN	80	319	155	35	202	47	138	126	75	823
Exact kNN	24	89	19	43	16	152	3037	6	53	162

Table 2: Approximate vs. exact kNN running time. Time is in seconds on a 50 MHz SPARCstation. Infeasibility is denoted by "*".

in different ways. The approximate algorithm examines fewer nodes of the subdivision tree than the exact algorithm. The exact algorithm examines more nodes but it requires much less time per node. The exact algorithm is almost always faster on feasible inputs, and the approximate algorithm is faster on infeasible inputs.

6.1.3 kCN

The previous section shows that the linear programming based techniques of the exact kNN algorithm are valuable in practice. The kCN algorithm of Section 5 shows that linear programming based techniques are also theoretically interesting. Using these techniques, we achieve a $\mathrm{O}((mn)^{k+1})$ time bound, which is much faster than the naive running time of $\mathrm{O}(m^{2k}n^k \log n)$ for reasonable values of k, m, and n ($n < m^{k-1}$).

[4] Ironically, the *approximate* kNN algorithm acts on U_{ij} directly, yet the *exact* kNN algorithm acts on C_{ij}, an approximation to U_{ij}.

Unfortunately, we had to make several "practically damaging" technical changes to the kNN algorithm to give it a good worst-case time bound on kCN inputs. These technical changes make it much more likely that the kCN algorithm will *achieve* its worst case running time. Therefore, this particular algorithm is not likely to be good in practice. The most "damaging" technical change is to use rectangles instead of convex C_{ij} regions. As the previous section explained, when C_{ij} is a poor approximation to U_{ij}, more subdivision is required. Clearly, the best rectangular approximations is never better than the best convex approximation. The kCN algorithm solves linear programs with fewer variables and constraints than those of the kNN algorithm. Therefore, evaluation is much cheaper for the kCN algorithm. Nevertheless, we believe that the kCN algorithm will have to subdivide so much more and run evaluation so many more times that it will be slower than the kNN algorithm.

We have not implemented the kCN algorithm. At present, we believe that the most practical approach to solving kCN for $k > 3$ is to run the kNN algorithm. Even though this algorithm is not specifically designed for convex inputs, it still benefits from their simpler combinatorial structure. Experiments 8-10 in Table 2 appear to support this view: the exact kNN algorithm solves 6CC as fast as it solves 4NN. We must caution the reader that running times vary too dramatically among different inputs to truly extract a trend.

6.2 Conclusion

The solutions to 2CN and 3CN have worst case running times better than the naive algorithm and better than previous results in this area. In addition, these algorithms are designed to run much faster in the case of no solution, which is the worst case for all previous algorithms. We find that "no" is the most common answer in practice, and so this is an important property.

Assuming $k > 3$ and $n < m^{k-1}$, the kCN algorithm has a better running time bound than the naive kCN algorithm that iterates over all possible contacts for k convex polygons and a nonconvex container (see Lemma 5.1). The time bound for the kNN algorithm is worse than the naive running time (Section 1.3) by a factor of $\log n$. However, the naive algorithm frequently attains its worst case, particularly when the answer is "no". The non-naive kNN algorithm repeatedly computes "lower bounds" by solving linear programs for convex approximations to the U_{ij} regions. This technique enables it to quickly discard regions that have no solution and focus only on regions with a possible solution.

Implementations of the 2CN, 3CN, and kNN algorithms appear to be quite practical. We recently licensed this software to Gerber Garment Technologies, which is the largest supplier of CAD/CAM software to the textile industry in the United States. They plan to add our software as an option to their system.

The complementary nature of the approximate kNN algorithm of Part I [3] and the exact kNN algorithm of Section 4 naturally leads to the idea of combining them into a hybrid algorithm. This is the direction of our current research.

References

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

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

[3] 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.*

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] Z. Li and V. Milenkovic. Compaction and separation algorithms for non-convex polygons and their applications. Technical Report TR-13-93, Center for Research in Computing Technology, 1993.

[11] Z. Li and V. Milenkovic. Compaction and separation algorithms for non-convex polygons and their applications. *European Journal of Operations Research*, 1995.

[12] V. J. Milenkovic, K. Daniels, and Z. Li. Placement and Compaction of Nonconvex Polygons for Clothing Manufacture. In *Proceedings of the 4th Canadian Conference on Computational Geometry*, pages 236–243, 1992.

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

[14] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

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

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

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

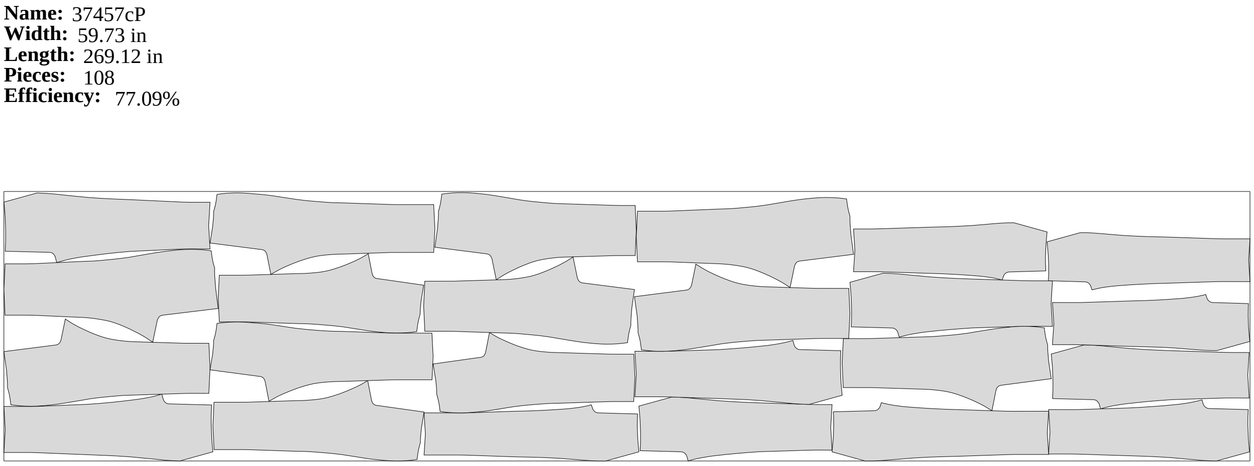

Figure 1: Automatically generated placement of panels.

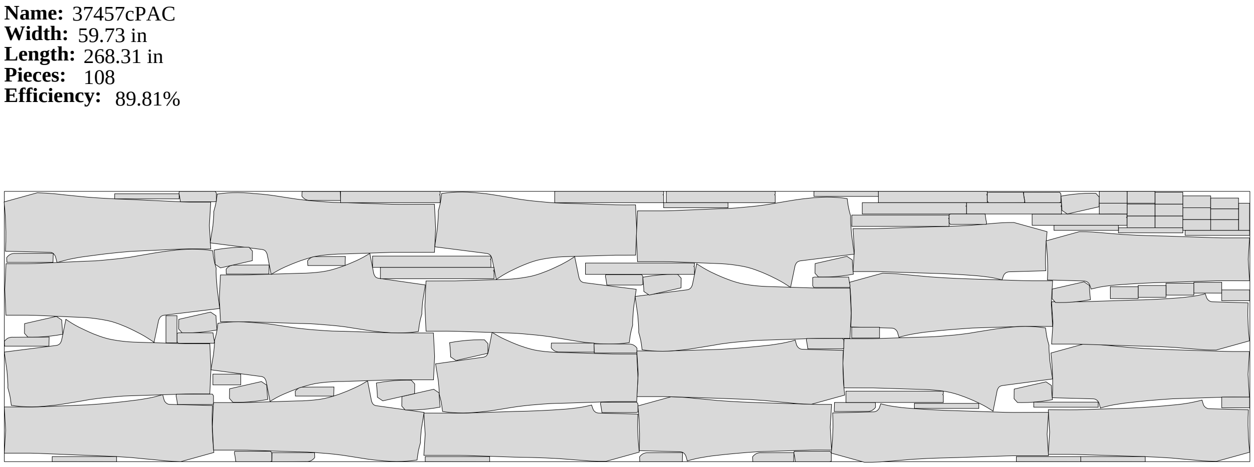

Figure 2: Layout with trim placed in gaps between panels.

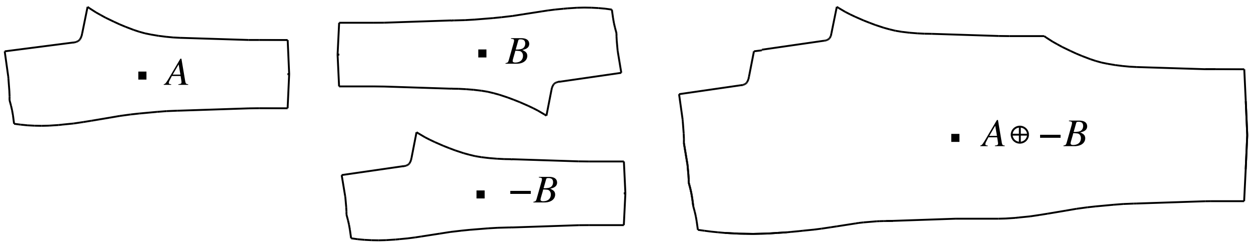

Figure 3: A, B, $-B$, and $A \oplus -B$.

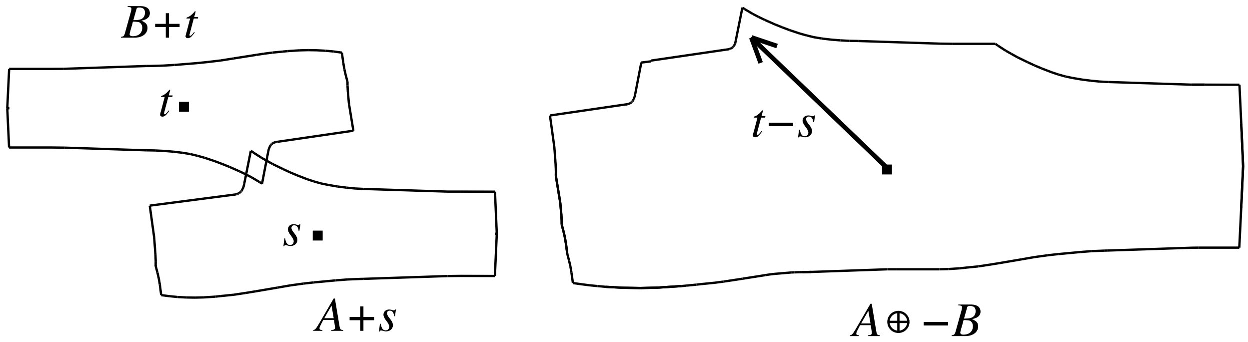

Figure 4: Illustration of Lemma 1.1. Black square indicates local origin of polygon.

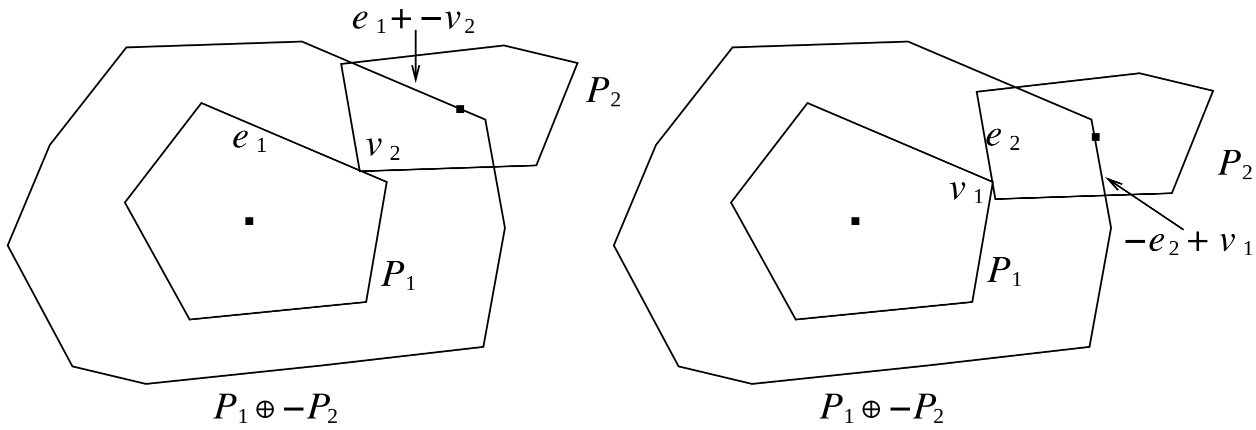

Figure 5: Minkowski Sum of Convex Polygons

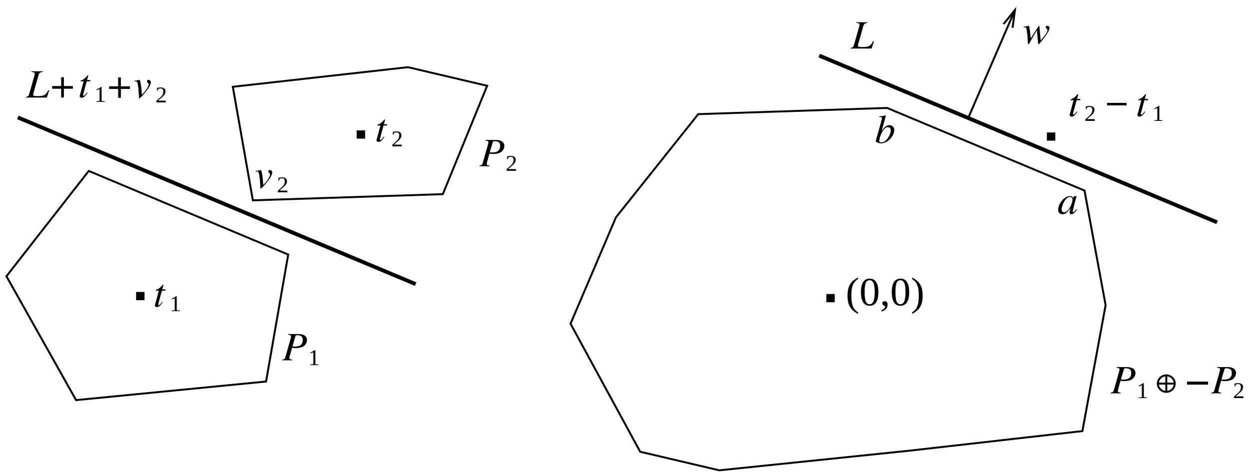

Figure 6: Separating Line for Nonoverlapping Convex Polygons

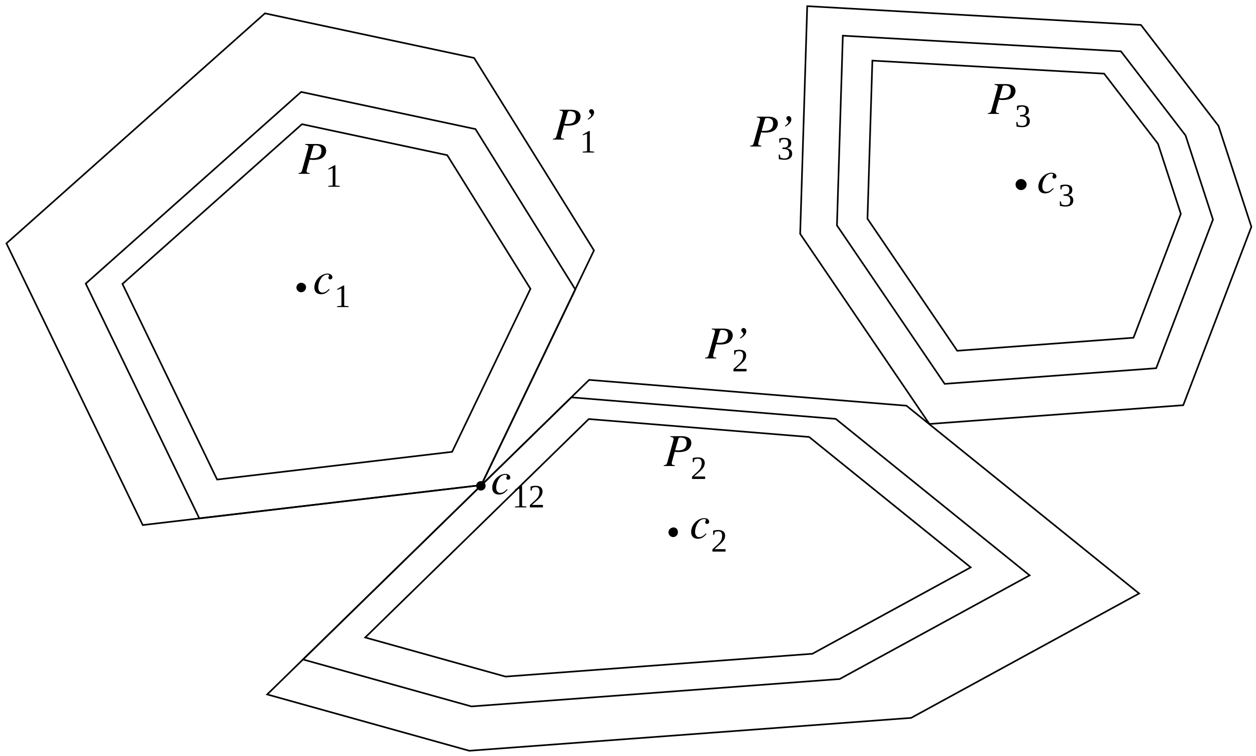

Figure 7: Two dilations result in two contacts.

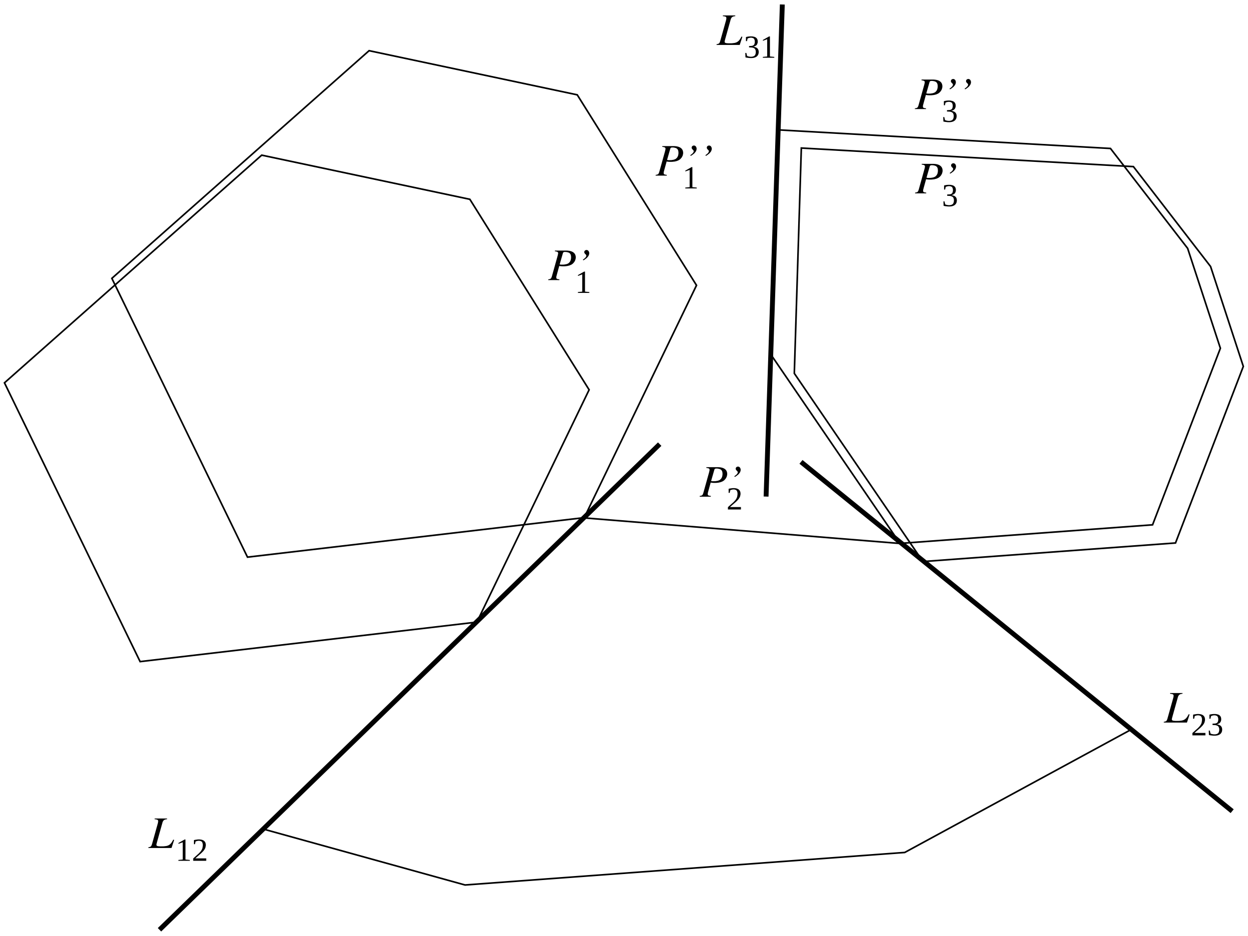

Figure 8: Slide P_1' and P_3' towards each other until each reaches the end of the edge with which it contacts P_2'.

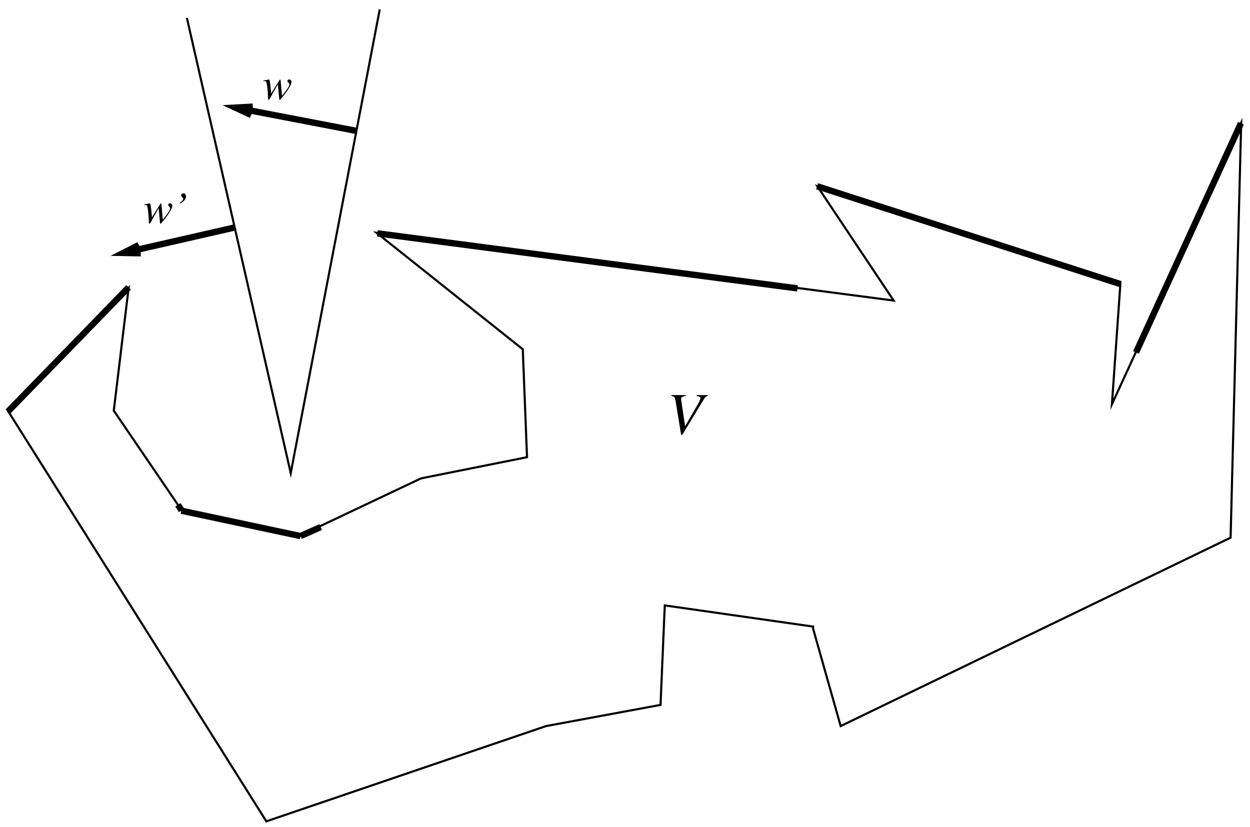

Figure 9: The thick line segments are the angle-visible envelope of V with respect to w and w'.

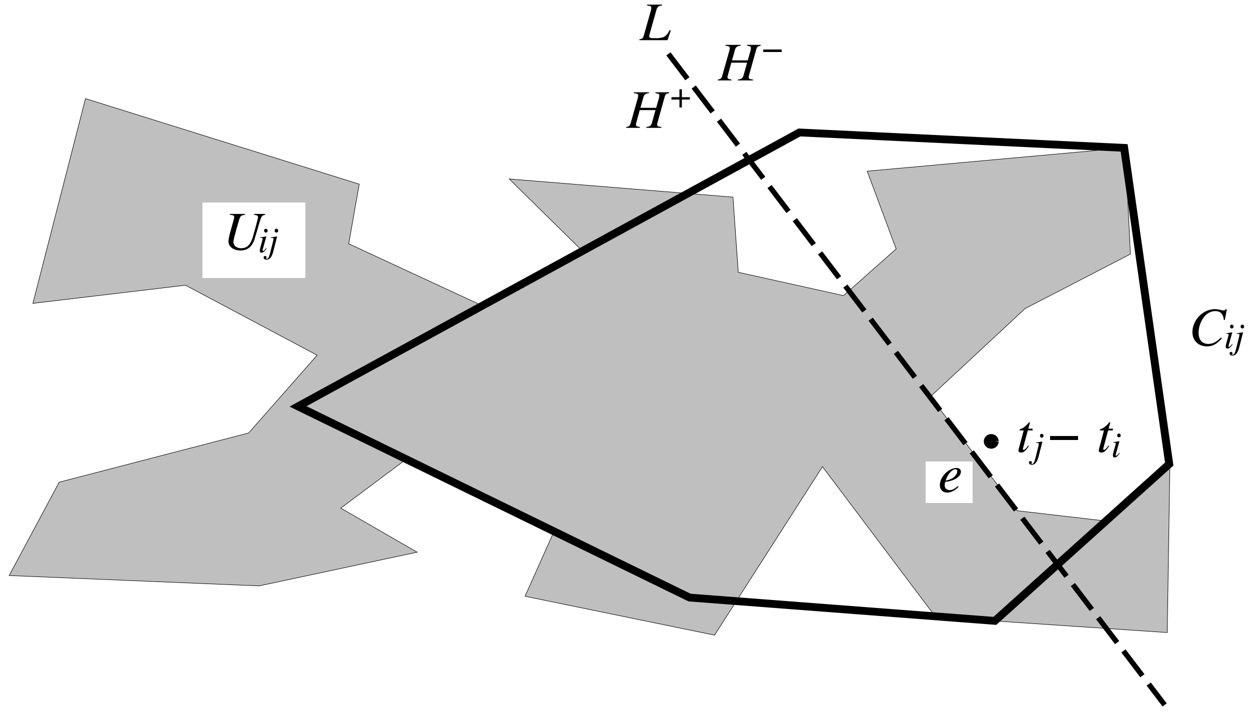

Figure 10: Choosing a cutting line L.

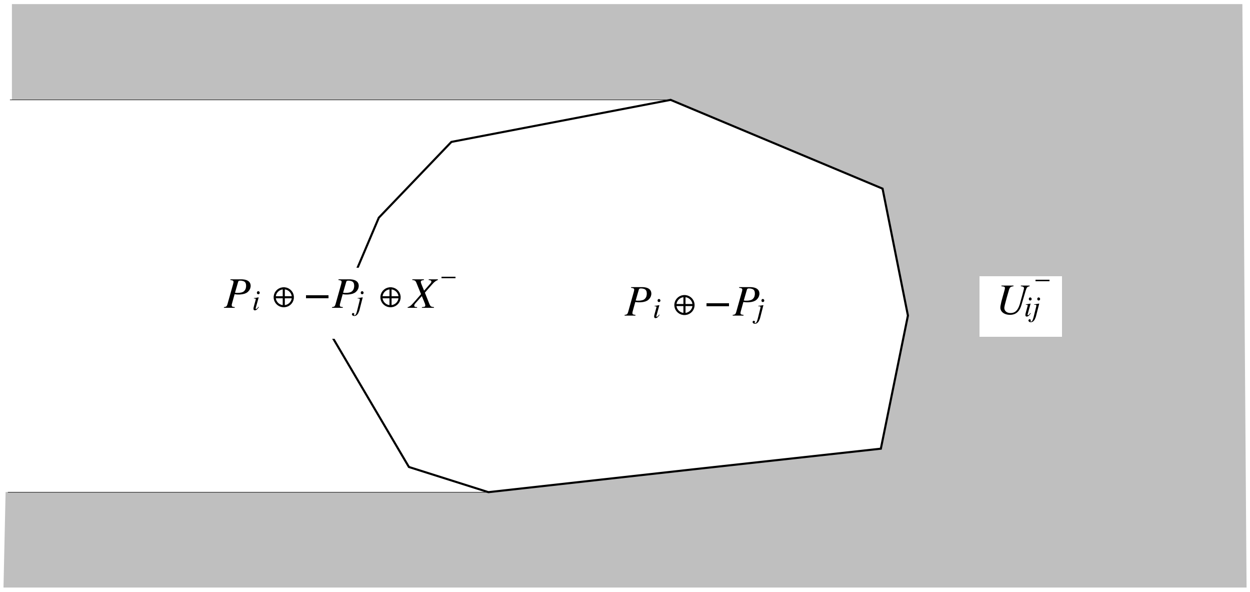

Figure 11: $P_j \oplus -P_i$, $P_j \oplus -P_i \oplus X^-$, and U_{ij}^-.

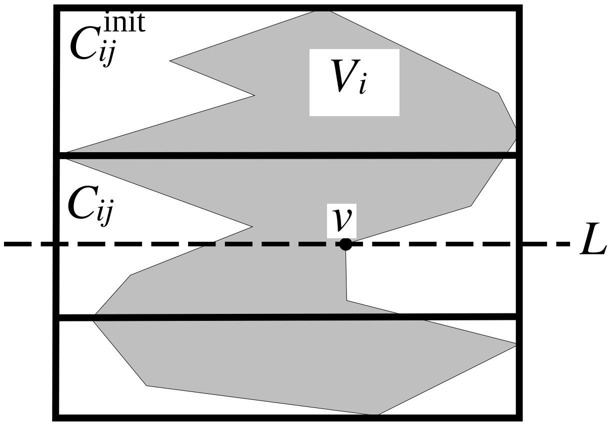

Figure 12: Choosing a cutting line L.

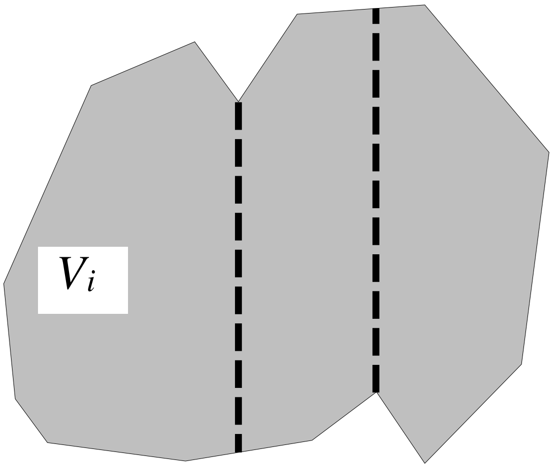

Figure 13: Cutting V_i into y-monotonic subsets.