Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA),

San Francisco, CA, January 22–24, 1995, pp. 205–214.

Multiple Translational Containment: Approximate and Exact Algorithms

Karen Daniels *   Victor J. Milenkovic†

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. We also give an *approximate* algorithm: given any ϵ, it finds a set of translations such that no point of any polygon is more than 2ϵ inside the boundary of any other polygon or outside the container.

The term kCN denotes a containment problem in which the k polygons are convex and the container is nonconvex, and kNN denotes nonconvex polygons and container. The polygons have up to m vertices, and the container has n vertices, where $n > m$ (typically).

We give exact algorithms for the following: 2CN in $O(mn \log n)$ time, 3CN in $O(m^3 n \log n)$ time, and kNN in $O((mn)^{2k+1} \mathrm{LP}(2k, 2kmn + k^2m^2))$ time, where $\mathrm{LP}(a, b)$ is the time to solve a linear program with a variables and b constraints. We present an approximate algorithm for kNN whose running time is $O(\left(\frac{1}{\epsilon}\right)^k \log\left(\frac{1}{\epsilon}\right) k^5 s \log s)$, where s is the largest number of vertices of any polygon that can generated by applying a certain set of operations to the input. We have no polynomial bound on s, but, in practice, it is usually not more than quadratic in n.

1 Introduction

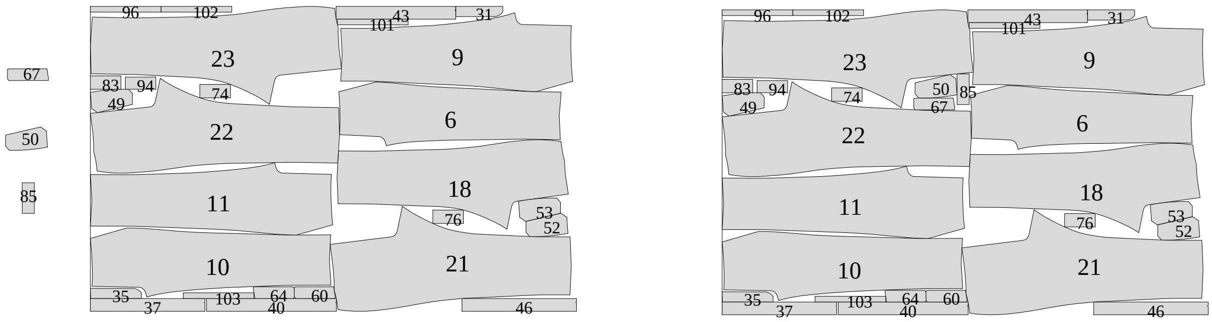
Our research into *translational containment* problems is motivated by an NP-hard layout problem in the apparel industry, the marker-making problem: lay out two-dimensional polygonal apparel pattern pieces on cloth inside a rectangular sheet of stock material of fixed width and minimum length. One promising approach is to first identify and pack all the large pieces, perhaps using a heuristic [10]. Regions of unused cloth between the large pieces form a collection of (usually nonconvex) polygonal "containers" into which the smaller pieces (trim) can then be placed (see Figure 1). To solve this (still NP-hard) *trim placement problem* we transform it into an "easier" NP-hard matching or set cover problem by determining, for each container, all the groups of trim pieces which can fit into that container. Choosing a collection of groups that covers the set of trim pieces is a purely combinatorial problem. Deciding which groups of trim pieces can fit into each container requires solving many containment problems. We only seek a single "yes-no" answer to each containment problem. We do not require the set of all possible solutions, but we do need a fast solution in both the "yes" and "no" cases.

Typically, the number of polygons in a group is under ten for our application, and for about half of the containers it is three or less. For some domains (pants, in particular), the trim pieces are convex. Therefore, we created fast exact algorithms for placing two or three convex polygons. Our algorithm for two convex polygons is based on the use of *separating line orientations* as a means of characterizing the solution. The solution for three polygons also uses a separating line orientation characterization plus a simple and robust "carousel" algorithm.

We present both exact and approximate algorithms for placing any number of nonconvex polygons. The exact algorithm uses the concept of subdivision trees and linear programming. The approximate algorithm finds an ϵ-*approximate* solution in which no point of any polygon is more than 2ϵ inside the boundary of any other polygon or outside the container. The value of ϵ is an input to the algorithm. We characterize a solution to translational containment and create a collection of containment subproblems from this characterization. The approximate algorithm solves each subproblem by restricting the configuration space until a steady state is reached. If necessary, it subdivides the configuration space to generate new subproblems. An approximate algorithm is acceptable for manufacturing applications involving cutting machinery that has a cutting tolerance. The cutting tolerance in clothing manufacture is 0.06 inch, and therefore solutions with $\epsilon = 0.01$ are perfectly acceptable.

*Harvard University, Division of Applied Sciences. This research was funded by the Textile/Clothing Technology Corporation from funds awarded to them by the Alfred P. Sloan Foundation.

†University of Miami, Department of Math and Computer Science. 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.



(a) Before placing pieces 50, 67, and 85 in gap between panels 22, 23, 9, 6.

(b) Pieces successfully placed.

Figure 1: Multiple Containment Example from the Apparel Industry

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 from P_i to P_j such that they do not overlap each other ($U_{ij} = -U_{ji}$).

The *Minkowski sum* [7] 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 $A-t = \{a-t \,|\, a \in A\}$. Note that $A + t = A \oplus \{t\}$.

Expressed in terms of set operations and Minkowski sums,

$$\begin{aligned} V_i &= \overline{\overline{C} \oplus -P_i},\ 1 \leq i \leq k, \\ U_{ij} &= \overline{P_i \oplus -P_j},\ 1 \leq i < j \leq k. \end{aligned}$$

Let $\mathcal{P}$, $\mathcal{V}$, and $\mathcal{U}$ denote the set of all P_i, V_i, and U_{ij}, respectively.

A *configuration* of $\mathcal{P}$ is a set of translations $\{t_i \,|\, 1 \leq i \leq k\}$. A *valid configuration* of $\mathcal{P}$ inside C is a configuration such that $t_i \in V_i$, $1 \leq i \leq k$, and $t_j - t_i \in U_{ij}$, $1 \leq i < j \leq k$. A valid configuration is an exact solution to a translational multiple containment problem.

Each V_i is the two-dimensional configuration space for the problem of placing P_i in C. Similarly, each U_{ij} is the two-dimensional configuration space for placing P_j with respect to P_i. Throughout the paper we use the term "configuration space" to denote the two-dimensional configuration spaces.

We define the kNN *problem* to be that of finding a valid configuration of k nonconvex polygons in a nonconvex container. Similarly, the kCN *problem* is that of finding a valid configuration of k convex polygons in a nonconvex container. Finally, (r,k)CN denotes the problem of determining which groups of k polygons out of a set of r convex polygons will fit into a nonconvex container. We refer to the entire collection of such problems as *translational multiple containment* problems. Each problem has an approximate version, which is to find an ϵ-approximate solution to the problem. We define $m = \max_{1 \leq i \leq k} m_i$. We assume that m is always less than n. In general, we express the running time of a containment algorithm in terms of m, n, and k. For our applications: $4 \leq m \leq 100$, $100 \leq n \leq 300$, $1 \leq k \leq 10$, and $50 \leq r \leq 400$.

§ 4.2 presents a set of geometrical *restriction* operations on $\mathcal{V}$ and $\mathcal{U}$. To these we add intersection with a half-plane bounded by a horizontal or vertical line.[1] These restrictions replace some set in $\mathcal{V}$ or $\mathcal{U}$ by a subset of itself. We define s to be the maximum number of vertices of any polygon that one can generate by applying *any* sequence of restriction operations to $\mathcal{V}$ and $\mathcal{U}$. All known containment algorithms appear to use at least some of these operations, and therefore s is a good measure of the complexity of a containment *problem.* Unfortunately, a (polygonal) subset of a polygonal region does not necessarily have fewer vertices, and it is an open question whether s is always finite. Nevertheless, restriction generally appears to reduce the number

[1] We need this operation to subdivide the configuration space (§ 4.1). One might also include bounding lines generated by extending an edge of a polygon (§ 3.2).

of vertices. We find that s ranges from n to mn in practice. Expressing the running time in terms of s is what we call *s-analysis* or *size analysis* of the algorithm. The use of s-analysis gives a measure of the practical running time of a containment algorithm.

1.2 Related Work. We know of at least one industrial solution to translational multiple containment, but it is used in an interactive setting where speed is less critical than in our application. That solution appears to be based on inner and outer approximations, and the running time increases with the tightness of the fit.[2] Published algorithms for translational multiple containment for small k are not fast enough for our purposes because they tend to depend on a high power of n. For arbitrary k and nonconvex polygons and container, the only known algorithms have running time which is exponential in k. In fact, kNN is NP-complete.[3]

The "naive" kNN algorithm runs in $O((mn)^{2k} \log n)$ time, assuming "constant" k and $n > m$ (as is typical), by iterating over all choices of $2k$ contacting pairs among the polygons and container.

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

Milenkovic, *et al.* [3] offer three approaches to translational multiple containment. The first method provides fast algorithms for two or three convex polygons. The 2CN algorithm presented here is essentially the same, but the 3CN algorithm is a much simpler algorithm with the same order running time. The second method of [3] uses a MIP (mixed integer programming) model for k nonconvex polygons and a nonconvex container (see [8] for full details). The MIP method takes a minute or two on a typical workstation for two or three polygons, but is slow for four or more polygons. The third method of [3] is an earlier version of the approximate algorithm presented here.

[2] Companies closely guard their products, and there is no way we can actually compare our algorithms to others short of buying very expensive systems.

[3] In the full version we prove that translational multiple containment is NP-complete by reduction from PARTITION [6].

1.3 Overview. In § 2.1 we present an exact algorithm for 2CN based on separating line orientations. In § 2.2 we give an exact algorithm for 3CN. This also uses a separating line orientation characterization plus a simple and robust "carousel" algorithm. In § 3 we describe an exact kNN algorithm which uses subdivision trees and linear programming. Our approximate algorithm for kNN is given in § 4. This is based on restriction and subdivision of the configuration space. Running times of our algorithms are given below. The expression $\mathrm{LP}(a, b)$ denotes the time to solve a linear program with a variables and b constraints. Recall from § 1.1 that s is an upper bound on the number of vertices of any polygon generated by any sequence of restrictions, and that $s \approx mn$ in practice.

Exact 2CN	$O(mn \log n)$ or $O(s \log s)$
Exact $(r, 2)$CN	$O((rmn + r^2m) \log n)$
Exact 3CN	$O(m^3n \log n)$ or $O(m^2s \log s)$
Exact kNN	$O((mn)^{2k+1}\mathrm{LP}(2k, 2kmn + k^2m^2))$
Approx. kNN	$O(\left(\frac{1}{\epsilon}\right)^k \log\left(\frac{1}{\epsilon}\right) k^5 s \log s)$

In § 5 we discuss our implementation of these algorithms.

2 Exact Algorithms for Small k

2.1 2CN Containment. Previous algorithms for 2CN containment have had running times essentially equal to the product of the times for solving two 1CN problems, in particular, $O(m^2n^2 \log m)$ or $O(s^2 \log s)$. This section establishes the somewhat surprising fact that 2CN containment requires no more time than 1CN asymptotically. Furthermore, solving $(r, 2)$CN requires $O(rmn \log n)$ (or $O(rs \log s)$) "preprocessing" time plus $O(r^2m \log n)$ time. In other words, the time is the same as r 1CN problems plus r^2 queries, each taking $m \log n$ time.

2.1.1 Separating Lines. The idea behind a fast solution of 2CN containment is that of a *separating line* for two nonoverlapping convex polygons: specifically, a line which has (the interiors of) the polygons on opposite sides. We state without proof that one can generate such a line by extending one of the edges of one of the two polygons. This fact implies that one need only consider $O(m)$ possible orientations for the separating line. These are also the orientations of the edges of the Minkowski sum $P_1 \oplus -P_2 = \overline{U_{12}}$.

We can also greatly limit the set of translations for P_1 to the vertices of the convex hull of V_1. To see this, suppose P_1 and P_2 have a valid placement inside a container C. Let L be a separating line, and let w be a vector normal to L pointing into the half-plane containing P_1. Choose $v_1 \in V_1$ which is extreme in the w direction. Vertex v_1 is at least as far from L as any other placement of P_1, and therefore we can safely place P_1 at v_1.

2.1.2 Computing a Solution. In order to compute v_1, we only need to know the normal vector w for a valid separating line L. This observation leads to the following algorithm for 2CN containment.

2CN-QUERY($\mathcal{P}$, $\mathcal{V}$, $\mathcal{U}$)
 for each vector w
 (inwardly) normal to an edge ab of U_{12}
 (outwardly normal to $P_1 \oplus -P_2$)
  1. $t_1 \leftarrow v_1 \in V_1$ that minimizes $v_1 \cdot w$
  2. $t_2 \leftarrow v_2 \in V_2$ that maximizes $v_2 \cdot w$
  3. **if** $(t_2 - t_1) \cdot w \geq a \cdot w$, **then**
   return t_1, t_2
 return NO-SOLUTION-EXISTS

If one precomputes the convex hulls $\mathcal{C}(V_1)$ and $\mathcal{C}(V_2)$, then one can perform Steps 1 and 2 by binary search on the normals. In this case, the input arguments must include $\mathcal{C}(V_1)$ and $\mathcal{C}(V_2)$. This leads to the following result:

LEMMA 2.1. *The 2CN containment algorithm uses time in* $O(m \log n)$ *plus the amount of time required to compute the convex hulls of* V_1 *and* V_2*, which is in* $O(mn \log n)$.

COROLLARY 2.1. *For* $r \geq n$ *(which is often true for our applications),* $\binom{r}{2}$ *applications of* 2CN-QUERY *solve the* $(r, 2)$*CN problem using time in* $O(r^2 m \log n)$.

The total query time dominates the time ($O(rmn \log n)$) to compute $V_1, V_2, \ldots, V_r$ and their convex hulls. For our applications, $m \approx 100$, $n \approx 300$, and $r \approx 400$, so this is very practical. It is also numerically stable. Finally, using the multiple resolution techniques of § 2.2.4, one can hope to reduce the total query time to something like $O(r^2 \log m \log n)$.

2.2 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 this case, 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 separating line orientation triples. For each triple, it checks if there is a configuration that has separating lines with these orientations.

Naively, one must consider $\Omega(m^3)$ orientation triples to solve 3CN, but § 2.2.1 shows that we need only consider $O(m^2)$ triples. Unlike the 2CN algorithm, the 3CN algorithm cannot move the solution to the vertices of the valid regions V_i (actually, the vertices of their convex hulls). However, it is possible to move the solution to the boundaries of the V_i. In fact, § 2.2.2 shows that one need only consider a special subset of the boundary, called the *angle-visible envelope*. § 2.2.3 gives a "carousel" algorithm that searches the envelopes for a solution in $O(|V_1|+|V_2|+|V_3|)$ time. Finally, § 2.2.4 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.

2.2.1 Orientation Triples.

LEMMA 2.2. *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 in 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. (Summary.) Polygons P_1, P_2, P_3 can be scaled (twice) until they have two contacts and then translated until the separating lines satisfy the lemma. Separating lines for the transformed polygons are also separating lines for the original polygons. The complexity bounds are consequences/modifications of the $O(m)$ bound on the number of vertices of $P_i \oplus -P_j$.

2.2.2 Angle-Visible Envelopes. Suppose we have a configuration t_1, t_2, t_3 of P_1, P_2, and P_3 inside a container C. Let L_{ij} be a line that separates P_i and P_j and let w_{ij} be the normal to the line pointing into the half-plane that contains P_j ($w_{ij} = -w_{ji}$). Define a partial order on the points of V_1: $v < v'$ if,

$$v \neq v' \text{ and } v \cdot w_{21} \leq v' \cdot w_{21} \text{ and } v \cdot w_{31} \leq v' \cdot w_{31}.$$

If $v_1 \in V_1$ satisfies $t_1 < v_1$, then P_1 can move from t_1 to v_1 without moving closer to either separating line L_{12} or L_{31}. The *angle-visible* envelope of V_1 with respect to w_{31} and w_{21} is the set of maxima under the partial order. The envelope is clearly a subset of the boundary of V_1, and any point on the envelope of V_1 can "see to infinity" along any direction in the (concave) angle defined by w_{21} and w_{31} (hence the name). Clearly, we can replace V_1 by its angle-visible envelope without eliminating a valid placement for P_1, P_2, and P_3, if one exists. The angle-visible envelopes for V_2 and V_3 are defined analogously. We state without proof the following lemma (see Figure 2):

LEMMA 2.3. *The angle-visible envelope of polygon* V *with respect to* w *and* w' *is monotonic with respect to* w *and* w'*. The (topological closure of the) envelope consists of* $O(|V|)$ *points and (closed) line segments.*

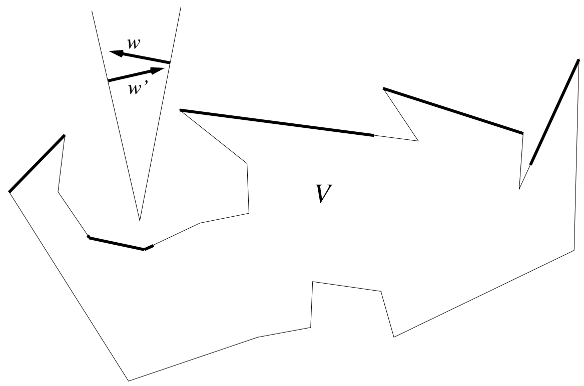


Figure 2: The thick line segments are the angle-visible envelope of V with respect to w and w'.

Using standard techniques, the envelope can be computed in time $\Theta(|V|)$ *for a simple polygon and in time* $\Theta(|V|\log|V|)$ *for a general polygon.*

Note that as t moves along the envelope, $t \cdot w$ increases and $t \cdot w'$ decreases, or *vice versa.*

2.2.3 The Carousel Algorithm. Once we have computed the angle-visible envelopes for a particular orientation triple, the algorithm for 3CN is conceptually very simple. Each polygon P_i is placed at its "first" envelope point: for P_1, that is the point t_1 which minimizes $t_1 \cdot w_{31}$ and maximizes $t_1 \cdot w_{21}$. 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 "carousel" algorithm.

The algorithm first makes one complete turn of the carousel (three "bumps"). After that, the algorithm continues to "bump" polygons forward, and it returns "yes" if ever any polygon is already on the correct side of the separating line (it does not need to be "bumped"). The algorithm returns "no" if a polygon is "bumped" past the last point of its envelope. Let $a_{ij}b_{ij}$ be the edge of the Minkowski sum $P_i \oplus -P_j$ with normal w_{ij} ($j = i + 1 \bmod 3$). The rule for bumping is as follows. Let t_i, t_j be the current positions of the polygons P_i and P_j on their respective envelopes. Polygon P_i "bumps" P_j forward to the position t'_j on its envelope that satisfies $t'_j \cdot w_{ij} \geq (t_i + a_{ij}) \cdot w_{ij}$ and maximizes $t'_j \cdot w_{kj}$ ($k = j + 1 \bmod 3$).

Because the envelope of V_j is monotonic with respect to w_{ij} and w_{kj}, "bumping" moves t_j in one direction only along its envelope. Unfortunately, bumping does not necessarily move t_j to a different edge on the envelope. The configuration t_1, t_2, t_3 might become "stuck" on three envelope edges. If this happens, then the algorithm advances t_1, t_2, t_3 to the endpoints of these respective edges and then runs the carousel two turns "backwards": P_1 bumps P_3, P_3 bumps P_2, P_2 bumps P_1, and so forth. If this finds a solution (one polygon does not need to be bumped after one full turn), the algorithm returns "yes". Otherwise, the algorithm freezes the reverse carousel just before it moves all three t_i. In particular, it stops when there is exactly one t_i still at an endpoint. The algorithm reverses direction again, and this t_i is guaranteed to be bumped to the next line segment of its envelope upon the next full turn of the forward carousel.

THEOREM 2.1. *For a given triple* w_{12}, w_{23}, w_{31} *of separating line orientations, the carousel algorithm correctly finds a solution or indicates none exists using running time in* $O(|V_1|+|V_2|+|V_3|)$. *Iterating over* $O(m^2)$ *triples takes time in* $O(m^3 n)$ *for simple* V_i *and* $O(m^3 n \log n)$ *for general polygonal* V_i.

Proof. Some technical lemmas are required to show that the carousel never "bumps" a polygon past a position on its envelope corresponding to a valid solution. The "backwards" carousel works because if a solution exists on three envelope segments, then there exists a solution for which one of the t_i is at an endpoint of its segment. The running time follows because only a constant number of carousel turns moves at least one t_i to the next line segment of its envelope. Under s-analysis, this running time bound is $O(m^2 s \log s)$ (see § 1.1).

2.2.4 Practical Improvements to Running Time. The full paper gives several methods to improve the practical running time of the algorithm for 3CN. The most effective of these involves using lower resolution versions of the Minkowski sums $P_i \oplus -P_j$: in particular, inscribed polygons. This reduces the number of separating line orientations. It is simple to increase the resolution of "good" separating lines by adding a vertex to the inscribed polygon. In particular, if edge ab of the inscribed polygon generates the separating line orientation, then one adds a new vertex to the inscribed polygon on the part of $P_i \oplus -P_j$ between a and b. 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 instead of adding one.

3 An Exact Algorithm for kNN

The multiple resolution heuristic for handling the separating lines in the case of 3CN (§ 2.2.4) provides the inspiration for the kNN algorithm. One can implement this algorithm using either multiple solutions of linear programs or a single solution to a mixed integer program.

3.1 Summary of the kNN Algorithm.

The condition,

$$t_i \in V_i,\ 1 \le i \le k,\ \text{and}\ t_j - t_i \in U_{ij},\ 1 \le i < j \le k,$$

is not solvable by linear programming because the V_i and the U_{ij} are usually not convex. However, we can "approximate" each by its convex hull. If the resulting linear program is infeasible, then there is no solution to the original problem. If the program is feasible, then an arbitrary solution may or may not be a solution to the original kNN problem. If it is not, then some t_i lies outside its V_i (but inside its convex hull) or some $t_j - t_i$ lies outside its U_{ij}. Let us say that it is V_i that causes the failure. We divide V_i into two polygons V_i' and V_i'' of "greater convexity". The algorithm recurses on two subhypotheses: one in which V_i' replaces V_i and the other in which V_i'' replaces V_i.

3.2 Analysis.

There are many ways to subdivide the V_i polygons and the U_{ij} polygons into convex subsets. One method involves extending an edge e of V_i (or U_{ij}) into a line L_e and cutting the polygon by that line.

THEOREM 3.1. *If subdivision is implemented using cutting lines of the form L_e described above, the kNN algorithm runs in $O((mn)^{2k+1} LP(2k, 2kmn + k^2m^2))$ time, where $LP(a, b)$ is the time to solve a linear program with a variables and b constraints.*

Proof. The basic observation is that there are only $O(mn)$ such L_e instead of $O(|V_1| + |V_2| + |V_3| + \cdots + |V_k|) = O(m^2n^2)$. This follows because each edge arises from a contact of some vertex of P_i with some edge of the container C or *vice versa*. Each L_e corresponds to a cutting plane π_e of E^{2k}, the full $2k$-dimensional configuration space. Standard techniques ensure that the cutting planes are in general position. For each feasible hypothesis H, solving the linear program (LP) generates a feasible configuration $t \in E^{2k}$ which is an intersection of $2k$ cutting planes. If t is an invalid (overlapping) configuration, the algorithm selects a cutting plane π_e to cut H into H' and H''. Vertex v is feasible for exactly of one of these, w.l.o.g. H''. Vertex v is not feasible for H' nor for any of its subhypotheses, and so the algorithm must solve a new LP to evaluate the feasibility of H'. We "charge" v for one LP call every time we split a hypothesis (H, H'', ...) which has v as its feasible vertex. Total number of LP calls is therefore number of different v times number of cutting planes: $O((mn)^{2k})$ times $O(mn)$ equals $O((mn)^{2k+1})$. Each call to the LP has $2k$ variables, and the number of constraints equals the number of cutting lines which equals the number of possible vertex-edge contacts: $2kmn$ (P_i contacting C) plus $k(k-1)m^2$ (P_i contacting P_j). Incidentally, this analysis assumes that the initial constraint set is empty, and does not contain the lines bounding the convex hulls of the V_i and U_{ij}. This assumption does not significantly affect the analysis since it is possible to show that the total number of edges of these convex hulls is bounded by $4kmn + 2k^2m^2$.

No matter which subdivision method we use, it is possible to set this problem up as a mixed integer program. Basically, each 0-1 variable corresponds to a choice of side for each cutting plane.

4 Approximate Algorithm for kNN

4.1 High-Level Description.

The algorithm creates a collection of containment subproblems using the following characterization.

LEMMA 4.1. *If an instance of kNN has a solution, then it has a solution with the following two characteristics: 1) $\ge 2k$ contacts, and 2) ≥ 2 contacts per polygon, where a polygon is either C or P_i, $1 \le i \le k$.*

The number of ways of assigning the $2k$ contacts among the $k(k+1)/2$ pairs of polygons is exponential in k. By only applying part of the second characteristic to C, we consider only $2k$ subproblems. Our empirical results indicate this choice represents a good balance between the amount of restriction and the number of subproblems. Specifically, we solve k subproblems in which one of the V_i is replaced by its set of vertices, and then we solve k subproblems in which one of the V_i is replaced by its boundary. The lemma implies that if a solution exists, then one of the subproblems has a solution. In a natural fashion, one can apply course-grain parallelism to solving the k subproblems, and we plan to attempt this on our multiprocessor workstation.

Although we restrict V_i to its vertices or boundary, we do *not* iterate over the vertices or edges of V_i, unlike Avnaim's 3NN algorithm and the naive algorithm. Instead, we solve each subproblem using a steady-state restriction process combined with subdivision. We define (§ 4.2) and apply certain *restrictions* to $\mathcal{V}$ or $\mathcal{U}$, replacing each element by a subset. The overall structure of the algorithm for solving each subproblem is to apply restrictions until no more significant "shrinking" is possible. If no solution to the subproblem is found in these restricted sets, the algorithm selects the largest $V_i \in \mathcal{V}$ and subdivides it into V_i' and V_i''. It then *branches* by replacing $V_i \in \mathcal{V}$ by either V_i' or V_i'' and recursing on the resulting sets $\mathcal{V}'$ and $\mathcal{V}''$. Our subdivision method cuts the selected V_i with either a vertical or horizontal line, depending on its aspect ratio. As shown in § 4.4, the running time of our algorithm is $O((\frac{1}{\epsilon})^k \log(\frac{1}{\epsilon}) k^5 s \log s)$ (see § 1.1 for a definition of s).

In the pseudo-code below, all functions use call-by-value except RESTRICT-SS, which uses call-by-reference. APPROX-SOL calls TEST-FOR-SOLUTION. If the call to TEST-FOR-SOLUTION does not find an exact solution, APPROX-SOL constructs an ϵ-approximate solution by selecting a point from each V_i. RESTRICT-SS and TEST-FOR-SOLUTION are described in more detail in the next two sections.

```
SOLVE-SUBPROBLEM(P, V, U)
    status ← RESTRICT-SS(V, U)
    if status = V_OR_U_NULL then
        return FAILURE
    if status = ALL_TOO_SMALL then
        return APPROX-SOL(P, V, U)
    status ← TEST-FOR-SOLUTION(P, V, U)
    if status = EXACT-SUCCESS then
        return EXACT-SUCCESS
    return SUBDIVIDE(P, V, U)
```

4.2 Restrictions. Theorem 4.1 below presents three types of restrictions which operate on $\mathcal{V}$ and $\mathcal{U}$ (see full version for proof). After each restriction, V_i remains a superset of the set of valid translations of P_i that allows the other $k-1$ polygons to be placed inside C without overlapping (each other and the container). Similarly, U_{ij} remains a superset of the set of valid displacement of P_i and P_j that allows the other $k-2$ pieces to be placed without overlapping each other.

THEOREM 4.1. *The following are valid restrictions:*

1. *RESTRICT-1:* $U_{ij} \leftarrow U_{ij} \cap (-V_i \oplus V_j)$ *for* $i \neq j$;
2. *RESTRICT-2:* $V_i \leftarrow V_i \cap \bigcap_{j \neq i} V_j \oplus U_{ji}$;
3. *RESTRICT-3:* $U_{ij} \leftarrow U_{ij} \cap \bigcap_{h \neq i,j} U_{ih} \oplus U_{hj}$.

These restrictions are generalizations of Avnaim and Boissonnat's [2] solutions to 2NN and 3NP. Let V_i^* be the set of all points in V_i corresponding to a valid configuration. Similarly, let U_{ij}^* be the set of all points in U_{ij} corresponding to a valid configuration. In our notation, Avnaim and Boissonnat show that, for 2NN, the set of all the valid relative positions of P_j with respect to P_i is given by: $U_{ij}^* = U_{ij} \cap (-V_i \oplus V_j)$. This yields an exact solution to 2NN. We observe that, for kNN, each valid displacement for two polygons must also be a solution for 2NN. The first application of RESTRICT-1 enforces this condition.

RESTRICT-2 can be viewed as a generalization of Avnaim and Boissonnat's observation (for 3NP) that a valid translation t_1 exists for P_1 only if:

$$t_1 = V_1 \cap (V_2 - (t_2 - t_1)) \cap (V_3 - (t_3 - t_1)) \neq \emptyset,$$

where $(t_2 - t_1) \in U_{12}$ and $(t_3 - t_1) \in U_{13}$.

For their solution to 3NP, Avnaim and Boissonnat show that any vector in $U_{12} \cap (U_{13} \oplus U_{32})$ is guaranteed to have an associated displacement for P_2 with respect to P_1 corresponding to a solution to 3NP. The proof relies on their application of Helly's Theorem: if three parallelograms which are the same affine transformation of three (different but axis-parallel) rectangles intersect pairwise, then all three have a common point. We observe that, in the general case, $U_{ij}^* \subseteq U_{ih} \oplus U_{hj}$ for $h \neq i \neq j$. This motivates RESTRICT-3.

RESTRICT-SS accepts $\mathcal{V}$ and $\mathcal{U}$ as arguments. It calls RESTRICT-1, RESTRICT-2, and RESTRICT-3 in a loop until either 1) some polygon of $\mathcal{V}$ or $\mathcal{U}$ becomes null (in this case it returns V_OR_U_NULL), or 2) all polygons of $\mathcal{V}$ and $\mathcal{U}$ have diameter less than ϵ (in this case it returns ALL_TOO_SMALL), or 3) the fractional area shrinkage of all polygons of $\mathcal{V}$ and $\mathcal{U}$ is less than α. The parameter α, $0 \leq \alpha \leq 1$, is part of the algorithm, not part of the input. For $\alpha = 0$, the program achieves a "true" steady state in which additional restrictions cannot shrink the polygons further. However, this can require an infinite number of iterations. We use $\alpha = 0.1$ in our implementation. This is satisfactory in practice, and it gives us a bound of the number of restriction iterations.

The steady-state restriction process is quite powerful. For example, we noted above that $U_{ij}^* \subseteq U_{ih} \oplus U_{hj}$ for $h \neq i \neq j$. This is enforced by RESTRICT-3. However, it is also true that: $U_{ij}^* \subseteq U_{ih} \oplus U_{hq} \oplus U_{qj}$ for $q, h \neq i \neq j$. In order to generalize this result, we define below the notion of a *path restriction*. Theorem 4.2 below asserts that RESTRICT-SS satisfies all path restrictions. In the theorem (proved by induction on path length in the full version), our use of the term "steady state" refers to the "true" steady state described above.

DEFINITION 4.1. *Given the complete graph on k nodes, two nodes i and j, and a path $\pi = i, \pi_1, \pi_2, \ldots \pi_q, j$ in the graph from i to j such that $i \neq \pi_1 \neq \cdots \neq \pi_q \neq j$, the* **path restriction** *of U_{ij} corresponding to π is the restriction:* $U_{ij} \leftarrow U_{ij} \cap (U_{i\pi_1} \oplus U_{\pi_1\pi_2} \oplus \cdots \oplus U_{\pi_q j})$.

THEOREM 4.2. *If multiple applications of RESTRICT-3 within RESTRICT-SS reach a steady state, then all possible path restrictions are satisfied.*

4.3 Testing for Solution. For each U_{ij}, in turn, TEST-FOR-SOLUTION selects (using a heuristic described below) $u_{ij} \in U_{ij}$ and replaces U_{ij} by the set $\{u_{ij}\}$. After each replacement, it applies RESTRICT-SS to restrict the configuration sets. If any configuration set restricts to the empty set, then TEST-FOR-SOLUTION fails: a solution might exist, but it does not find it. If TEST-FOR-SOLUTION successfully replaces every U_{ij} by a single point, then it can find a valid solution: it selects any $t_1 \in V_1$ and sets $t_j = t_1 + u_{1j}$ for $j = 2, 3, \ldots, k$. Appropriate selection of $u_{ij} \in U_{ij}$ appears to increase the chance that TEST-FOR-SOLUTION succeeds. If TEST-FOR-SOLUTION succeeds, the overall algorithm avoids subdivision. We developed two heuristics for selecting $u_{ij} \in U_{ij}$. In § 5.4 we compare their effectiveness in practice.

Both heuristics are based on choosing u_{ij} such that RESTRICT-2 (§ 4.2) does not cause V_i to vanish. Note that if $U_{ji} = \{u_{ji}\}$, then $V_j \oplus U_{ji} = V_j + u_{ji}$. The first heuristic is based on making each pairwise intersection $V_i \cap (V_j \oplus U_{ji})$ as large as possible. Thus, we choose $u_{ji} \in U_{ji}$ that maximizes the area of $V_i \cap (V_j + u_{ji})$. We conjecture that an exact calculation of this u_{ji} requires at least $\Omega(|V_i|^2|V_j|^2)$ time. There is a variety of ways to approximate the maximum area of intersection of two polygons (see full version for details). In practice, we use sampling.

The second heuristic selects u_{ji} to maximize the *convexity* of the *union* $V_i \cup (V_j + u_{ji})$. It is based on the following theorem (proved in the full paper).[4]

THEOREM 4.3. *Given a set of polygons* $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_k\}$, *if* $Q_i \cup Q_j$ *is convex,* $i \neq j$, $1 \leq i < j \leq k$, *then* $\bigcap_{i=1}^{k} Q_i \neq \emptyset$.

4.4 Analysis. In the full paper we prove that, if a valid configuration exists, then the algorithm will find an ϵ-approximate solution. To calculate the asymptotic running time of our algorithm we assume: 1) s is an upper bound on number of vertices of any polygon generated by the algorithm (see § 1.1), 2) 2ϵ is the maximum overlap between two polygons. In the full version we prove the following theorem.

THEOREM 4.4. *The running time of our algorithm is in* $O((\frac{1}{\epsilon})^k \log(\frac{1}{\epsilon}) k^5 s \log s)$.

5 Implementation and Results

5.1 Exact Algorithms for Small k. We have implemented versions of the 2CN and 3CN algorithms, and they run very quickly. Use of multiple resolution to find a good orientation triple speeds up the algorithm by one to two orders of magnitude. Typical running times (with this speedup) are in the range of 0.2 second on a 50 MHz SPARCstation[5] for $m = 50$. 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. In § 5.4 we compare the performance of our 3CN algorithm to our approximate kNN algorithm on 3CN examples.

5.2 Exact Algorithm for kNN. Some preliminary results for the subdivision-based kNN algorithm are shown in Table 1. The value of n is estimated because we compute each V_i directly from the union of Minkowski sums with the neighboring pieces instead of explicitly computing a container polygon. This algorithm subdivides a polygon by extending an edge of the most concave angle of the polygon. It uses repeated solutions of linear programs (not mixed integer programming).

Type	Time	Type	Time
3NN	5	3NN*	9
3NN	5	3NN*	5
4NN	21	4NN*	57
4NN	107	4NN*	175
4CN	13	4CN*	5
4NN	30	4NN*	323

Table 1: Running times for the exact kNN algorithm in seconds: $60 \leq m \leq 90$, $n \approx 100$. (* denotes infeasible)

The asymptotic running time bound for the kNN algorithm is worse than the naive running time (§ 1.2). However, the naive algorithm frequently attains its worst case, particularly when the answer is "no". The kNN algorithm repeatedly computes "lower bounds" by solving linear programs for convex approximations to the V_i and the U_{ij} polygons. This technique enables it to quickly discard regions that have no solution and focus only on regions with a possible solution. In § 5.4 we compare the performance of this algorithm to our approximate kNN algorithm.

5.3 Approximate Algorithm for kNN.

5.3.1 Performance. We ran the approximate algorithm on a collection of 62 examples, with k ranging

[4]Dan Roth originally conjectured this result for $k = 3$.

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

from 3 to 10. Representative running times are given in Table 2. All running times are to the nearest second on a 28 MHz SPARCstation and do not include the time to calculate $\mathcal{V}$. Infeasible examples (containment problems without any solution) are identified by *; we selected ones for which each *pair* of polygons fits into the container and for which the area of the container is larger than the total area of the polygons. Each feasible solution for $k = 3$ was found in an average of 2 seconds.[6] Infeasible situations for $k = 3$ were detected in an average of 4 seconds each. Figure 3 shows[7] configurations achieved for a 3NN apparel example using our algorithm. The tightness of the layouts is typical.

Type	Time	Type	Time	Type	Time
3CN	2	3CN	2	3CN	3
3CN	2	3CN	2	3CN*	27
3CN	2	3NC	1	3NN	5
3NN	5	4CN	26	4CN	30
4CN*	3	4CN	5	4CN	19
4CN	192	4CN	43	4NN	111
4NN	111	4NN	90	4NN*	10
5CC	126	5NN	60	5NN*	26
6NN	93	6NN	300	6NN*	49
7CC	823	10CC	600		

Table 2: Running times for the approximate kNN algorithm in seconds. (* denotes infeasible)

5.3.2 Regularization.

Polygon modeling algorithms typically produce *regularized* unions and intersections which replace each point-set by the closure of its interior. This is not sufficient for our application; a restricted V_i or U_{ij} can contain isolated points or line segments as well as polygonal components with non-zero area. We do not want to throw away any potential solutions, and so we use *closed* polygon operations which replace each point-set by its closure. These are harder to implement, but necessary.

The set operations must also be implemented robustly in order to minimize error accumulation during the steady-state restriction process. Our algorithms for polygonal union and intersection are based on [9].

5.3.3 Comparing Heuristics for Selecting u_{ij}.

We experimented with three different heuristics for selecting a shrinking point $u_{ij} \in U_{ij}$: 1) a random point, 2) the point which (approximately) maximizes the area of $V_i \cap (V_j + u_{ji})$, and 3) the point which (approximately) maximizes the convexity of the union of the V_i polygons. Heuristics (2) and (3) use samples from a 10x10 grid to find an approximate maximum. Our experiments show that maximizing the area of intersection (heuristic (2)) performs best, solving 60% of 30 test problems *without* subdivision. Maximizing convexity (heuristic (3)) solved 36% of the problems without subdivision, and random selection (heuristic (1)) only 20%.

5.4 Comparisons.

Table 3 compares running times for our 3CN exact algorithm with our approximate kNN algorithm on 3CN examples. For feasible configurations, the exact algorithm performs about seven times faster than the approximate algorithm, on average. This is because the 3CN algorithm takes advantage of the convexity of the polygons, whereas the 3NN can handle arbitrary nonconvex polygons.

Type	Appr.	Ex.	Type	Appr.	Ex.
3CN	79	11	3CN*	34	15
3CN	107	14	3CN*	68	21
3CN	171	26	3CN*	35	16
3CN	230	22	3CN*	27	15
3CN	55	10	3CN*	38	30

Table 3: Approximate vs. exact 3CN running times in seconds (100 calls each)(* denotes infeasible)

Type	Appr.	Ex.	Type	Appr.	Ex.
3NN	3	3	3NN	3	7
3NN	19	22	3NN*	2	4
3NN	2	6	3NN*	3	5
3NN	2	2	3NN*	4	29
3NN	3	5	3NN*	1	2
3NN	1	3	3NN*	2	5
3NN	1	4	4NN	207	53
3NN	5	4	4NN	319	89
3NN	1	1	4NN*	423	266
3NN	5	6	4NN*	47	152
3NN	3	4	4NN*	138	3037

Table 4: Approximate vs. exact kNN running times in seconds. (* denotes infeasible)

Table 4 compares running times for the approximate kNN algorithm with the exact kNN algorithm. For 3NN examples, the approximate algorithm finds valid configurations slightly faster than the exact algorithm. For 4NN examples, the exact algorithm can

[6] The typical values of m and n were $m = 100$ and $n = 100$.

[7] The pieces to be placed are above the marker in the "palette", whose scale is different from that of the marker.

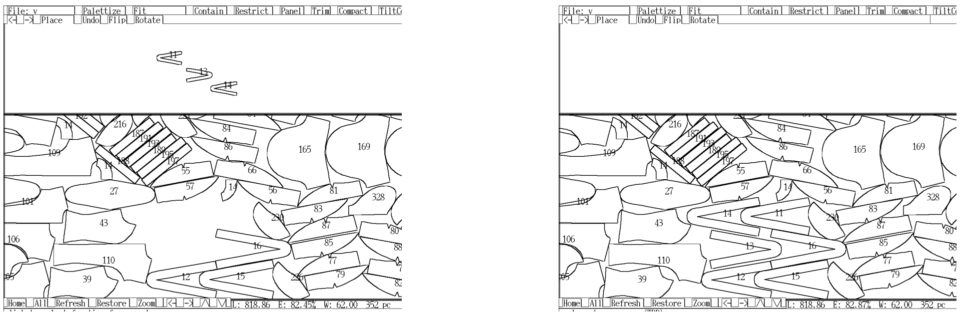


Figure 3: 3NN Example (Time = 5 seconds)

often find valid configurations several times faster than the approximate algorithm. The difference is primarily caused by the time required to execute the steady-state restrictions within the approximate algorithm. Recall that restrictions are used at each subdivision level in the approximate algorithm. The time required to perform restrictions is well spent if the problem is infeasible; in one 4NN* example, the approximate algorithm is more than 20 times faster than the exact algorithm.

6 Conclusion

Our 2CN and 3CN algorithms are fast enough to solve our $(r, 2)$CN and $(r, 3)$CN containment problems for apparel marker making. There is probably no faster 2CN algorithm in practice, but it may be possible to solve 3CN a factor of m faster. For modest values of k ($k < 10$), our kNN algorithms run fast enough for many practical applications. According to our tests, our exact and approximate algorithms are fastest on different types of input. This result suggests that a hybrid containment algorithm has the best chance for successfully handling "large" values of k ($10 \leq k \leq 20$). We are currently experimenting with a hybrid algorithm which combines restrictions, multiple resolutions, and linear programming. It may also be possible to generalize the carousel algorithm to kNN or develop other methods. Since kNN containment is such a useful operation in practice and since there are many approaches to solving it, it is a fruitful area for further 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, 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.

[4] O. Devillers. Simultaneous containment of several polygons: Analysis of the contact configurations. Technical Report 1179, INRIA, 1990.

[5] 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.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[7] 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.

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

[9] V. J. Milenkovic. Robust Polygon Modeling. *Special Issue of Computer-Aided Design on Uncertainties in Geometric Computations*, 25(9):546–566, September 1993.

[10] 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.