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

Multiple Translational Containment Part I: An Approximate Algorithm

Karen Daniels * Victor J. Milenkovic†

SEPTEMBER 1995

Abstract

In Part I we present an algorithm for finding a solution to the two-dimensional translational *approximate multiple containment* problem: find translations for k polygons which place them inside a polygonal container so that no point of any polygon is more than 2ϵ inside of the boundary of any other polygon. The polygons and container may be nonconvex. The value of ϵ is an input to the algorithm. In industrial applications, the containment solution acts as a guide to a machine cutting out polygonal shapes from a sheet of material. If one chooses ϵ to be a fraction of the cutter's accuracy, then the solution to the approximate containment problem is sufficient for industrial purposes.

Given a containment problem, we characterize its solution and create a collection of containment subproblems from this characterization. We solve each subproblem by first restricting certain two-dimensional configuration spaces until a steady state is reached, and then testing for a solution inside the configuration spaces. If necessary, we subdivide the configuration spaces to generate new subproblems. The running time of our algorithm is $O(\left(\frac{1}{\epsilon}\right)^k \log\left(\frac{1}{\epsilon}\right) k^6 s \log s)$, where s is the largest number of vertices of any polygon generated by a restriction operation. In the worst case s can be exponential in the size of the input, but, in practice, it is usually not more than quadratic.

Part II, *Exact Algorithms*, presents algorithms for solving the translational *exact* multiple containment problem, for which no overlap between polygons is allowed.

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

*Harvard University, Division of Applied Sciences. Email address: daniels@das.harvard.edu. 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. Email address: vjm@cs.miami.edu. 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

This research 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. Fabric considerations, such as grain and stripe constraints, often limit rotations of the pieces to multiples of 90 degrees.

Researchers at Harvard's Center for Textile and Apparel Research have been working on marker-making problems since 1990 [22, 23]. We have focused on domains which possess a large/small piece dichotomy, such as pants and jeans. A pants marker typically contains from 100 to 200 pieces; roughly 25% of them can be classified as "large". Our overall strategy is to first identify and pack all the large pieces [23]. Regions of unused cloth between the large pieces form a collection of polygonal "containers" into which the smaller pieces, called trim, can then be placed (see the small example in Figure 1).

If we consider the collection of containers as a single, multiple-component container, the *trim placement problem* can be viewed as a single *containment problem*: determine whether or not there exist non-overlapping positions for the pieces inside the container. If we iterate over the discrete set of allowable orientations, we need only solve *translational containment* problems. However, translational containment is NP-complete, as we show later, and so solving containment for 100 or more pieces is not practical.

Instead of solving a single, large containment problem, our approach is to partition the problem into many, small containment problems in order to transform the trim placement problem into an "easier" NP-hard matching problem. We determine, for each container, groups of trim pieces which can fit into that container. Although translational containment is NP-complete, the number of polygons in a group is small for our application. Typically, the number is under ten, and in about 50% of our examples it is three or less. Selecting the collection of groups, one per container, which uses the most trim pieces is a purely combinatorial matching problem. This matching problem can be solved by heuristics, mixed integer programming, or approximation algorithms. An approximation algorithm for the matching problem is one which is guaranteed to find $(1-\rho)$ times the optimal number of matches of pieces to containers, where ρ is a fraction between zero and one.

Deciding which groups of trim pieces can fit into each container requires solving many containment problems. We therefore need a fast containment algorithm. We only seek a single "yes-no" answer to each problem; we do not require the set of all possible solutions. Often the pieces will not fit into the container. Thus, it is essential that our containment algorithm run quickly even if the answer is "no".

Trim placement is just one of several types of layout problems which can be solved using containment for small groups of pieces. We briefly mention one more here; this and others are described in detail in [6]. The class of *minimal enclosure* problems involves finding the smallest enclosure of a particular type for a given collection of items. If the enclosure can be specified using one parameter, then binary search on the value of the parameter is an effective way to find the minimal enclosure. Each solution to containment establishes a new upper or lower bound on the parameter, and one can also use translational *compaction* [15, 24, 18, 17] to diminish each upper bound and thus speed up the search. Examples of minimal enclosure problems include finding the minimal square enclosure, and the *strip packing* problem of finding the minimal length of a strip. The former problem is useful in clustering algorithms for layout, and the latter problem relates to the on-line problem of incremental marker-making - that is, adding one garment at a time to an existing marker.

The two parts of this paper present algorithms for translational containment. In Part I we relax the containment problem by allowing small overlaps between polygons. We therefore solve the two-dimensional translational *approximate multiple containment* problem: find translations for k polygons which place them inside a polygonal container so that no point of any polygon is more than 2ϵ inside of the boundary of any other polygon. The parameter ϵ is specified as input to the program. We refer to the solution of such a problem as an ϵ-*approximate* solution. The running time of this numerically approximate algorithm[1] is a function of $\frac{1}{\epsilon}$.

An approximate algorithm is acceptable for manufacturing applications involving cutting machinery that

[1]Note that this is different from an *approximation algorithm*. As discussed above, an approximation algorithm for matching, for example, is guaranteed to find $(1-\rho)$ times the optimal number of matches of pieces to containers. In an *approximate* algorithm, however, the solution generated by the algorithm is accurate to within a tolerance ϵ. For example, a containment algorithm which produces an ϵ-approximate solution is an approximate algorithm because the solution may contain small overlaps.

has a cutting tolerance. For example, if we set ϵ to be one quarter of the machine tolerance in clothing manufacture, then the overlaps will be insignificant because they will be smaller than the cutting machine tolerance. In fact, CAD systems in the apparel industry routinely round polygon coordinates to the nearest 0.01 inch before storing them in data files. Since cutter accuracies are no better than 0.05 inch, this rounding is insignificant.

The remainder of this section is structured as follows. Section 1.1 introduces notation. Section 1.2 suggests a way to capture the practical running time of a containment algorithm. Section 1.3 discusses related work. Section 1.4 gives an overview of the remainder of the paper.

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.

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

The *Minkowski sum* [26, 12, 28, 29] of two point-sets (of $\mathbf{R}^2$ in the case of this paper) is defined as

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

For a point-set A, let $\overline{A}$ denote the set complement of A and define $-A = \{-a \mid a \in A\}$. For a vector t, define $A + t = \{a + t \mid a \in A\}$ and $A - t = \{a - t \mid a \in A\}$. Note that $A + t = A \oplus \{t\}$.

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.

Definition 1.1 *A* **configuration** *of $\mathcal{P}$ is a k-tuple $\langle t_1, t_2, t_3, \ldots, t_k \rangle$ where t_i, $1 \leq i \leq k$, is a translation (x_i, y_i) of P_i. A* **valid configuration** *of $\mathcal{P}$ inside C is a configuration of $\mathcal{P}$ which satisfies*

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

A valid configuration is an exact solution to a translational containment problem. It corresponds to a point in $2k$-dimensional space. The set of points corresponding to all valid configurations for a given $\mathcal{P}$ within C is the $2k$-dimensional free configuration space for $\mathcal{P}$ with respect to C. Each V_i is the two-dimensional free configuration space for the problem of placing P_i in C. Similarly, each U_{ij} is the two-dimensional free configuration space for placing P_j with respect to P_i. Throughout the paper, we use the term "configuration space" to denote a two-dimensional configuration space.

For notational convenience, we sometimes denote by P_0 the complement of the container region C so that containment is equivalent to the placement of $k+1$ polygons $P_0, P_1, P_2, \ldots, P_k$ in non-overlapping positions. In this case, we define $t_0 = (0,0)$ because the container is fixed. For $1 \leq i \leq k$, we define

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

For $0 \leq i < j \leq k$, we define $U_{ji} = -U_{ij}$. Finally, for $0 \leq i \leq k$, we define $U_{ii} = \{(0,0)\}$ because a polygon cannot be displaced with respect to itself. We redefine $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$, and thus the list $\mathcal{U}$ then contains all the V_i polygons (in the form U_{0i}), as well as the U_{ij} polygons defined earlier.

We use a shorthand notation to describe various types of translational containment problems. The shorthand is based on the following definition.

Definition 1.2 *The* **kNN problem** *is that of finding a valid configuration of k nonconvex polygons in a nonconvex container. Variations replace N with either C (convex), R (rectangle), or P (parallelogram).*

We refer to the entire collection of two-dimensional translational multiple containment problems as *translational multiple containment* problems. Each problem has an approximate version, which is to find an ϵ-*approximate* configuration.

Definition 1.3 *An* ϵ**-approximate** *configuration of* $\mathcal{P}$ *within* C *is a configuration such that* $t_i \in V_i$ *for* $1 \leq i \leq k$ *and the distance from* $t_j - t_i$ *to* U_{ij} *is at most* 2ϵ *for* $1 \leq i < j \leq k$.

We define $m = \max_{1 \leq i \leq k} m_i$. In general, we express the running time of a containment algorithm in terms of m, n, and k. For our applications: $4 \leq m \leq 100$, $100 \leq n \leq 300$, and $1 \leq k \leq 10$.

1.2 Size Analysis

Containment algorithms tend to rely on consecutive polygon operations such as unions, intersections, and Minkowski sums. The worst-case running time of these algorithms therefore depends on the worst-case running time of these operations. In practice, the worst case is usually a gross over-estimate. In theory, for nonconvex polygons A and B with $|A|$ and $|B|$ vertices, $|A \cup B|$, $|A \cap B| \in \mathrm{O}(|A| \cdot |B|)$ and $|A \oplus B| \in \Theta(|A|^2|B|^2)$ [14, 4], but, in practice, often $|A \cup B|$, $|A \cap B| \approx |A| + |B|$ and $|A \oplus B| \approx |A||B|$. On the one hand, new vertices are created by edge-edge intersections, but, on the other hand, many vertices are discarded when the boundary of the union or intersection is computed. Har-Peled *et al.* [13] have recently shown that each connected component of $\overline{A \oplus B}$ has $\Theta(|A||B|\alpha(\min(|A|, |B|)))$ vertices, where $\alpha()$ is the inverse of Ackermann's function. For our applications, we find that $\overline{A \oplus B}$ tends to have few components.

We introduce the concept of *size analysis* in order to capture the practical running time of a containment algorithm. We observe that containment algorithms, in general, seem to involve operations that *restrict* the configuration spaces - replace them by subsets of themselves. Different algorithms might use different classes of restriction operations, but let us consider the class of all restriction operations used by all containment algorithms. One way of measuring the complexity of a set of input polygons is to consider the maximum complexity of any configuration space resulting from *any* sequence of restriction operations[2]. We call this maximum complexity s, the "size" of the problem. Expressing the running time of a containment algorithm in this way is called s-*analysis*, or *size analysis*.

For some classes of operations, s is bounded. For instance, s is bounded if the class includes only intersection with horizontal or vertical half-planes [6]. We conjecture that for the class of operations used by all current containment algorithms, s is also bounded.

This paper presents a *valid restriction* operation which replaces some of the configuration spaces by subsets of themselves while not eliminating any valid configurations (see Definition 4.3). To this we add intersection with a half-plane bounded by a horizontal or vertical line[3]. For this class of operations, we find that s ranges from $\mathrm{O}(n)$ to $\mathrm{O}(mn)$ in our applications.

1.3 Related Work

We know of at least one industrial solution to translational containment, and it is used in an interactive setting. That solution appears to be based on inner and outer approximations of polygonal boundaries, and the running time increases with the tightness of the fit. Published algorithms for translational containment for small k have asymptotic running times which 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. This is not surprising, since kNN is NP-complete, as we show in Section 2.2. Our proof that kNN is in NP is modeled after the proof of Milenkovic *et al.* [22] that the marker-making problem, in the absence of rotations, is in NP. That proof, in turn, relies on characterizing a solution in terms of the number of degrees of freedom and the associated number of contacts between polygons.

Characterizing a containment solution in terms of contacts and degrees of freedom is a popular method for developing a containment algorithm (see Chazelle's work on rotational containment for a single polygon [5] and Avnaim's 3NN algorithm [2]). Unfortunately, existing characterization-based algorithms iterate over vertices and/or edges of the configuration space boundaries. This causes the algorithms to perform poorly

[2]The sequence of operations may be infinite.

[3]We use this operation to subdivide the configuration space (see Section 6).

on infeasible containment problems. As an example, characterizing a containment solution as in the proof that kNN is in NP yields a "naive" kNN algorithm which runs in $O((mn)^{2k} \log n)$ time, assuming "constant" k and $n >> m$ (as is typical in our applications). The algorithm iterates over all choices of $2k$ contacting vertex/edge pairs among the polygons and container. The naive algorithm has running time in $\Omega((mn)^{2k})$ because it must iterate over all possible sets of contacts whenever the answer is "no". We describe this naive algorithm in more detail in Section 2.3.

Fortune [10] gives a solution to 1CN by computing the Minkowski sum using a generalized Voronoi diagram. Avnaim and Boissonnat [3, 2] use the Minkowski sum and convex decomposition to solve 1NN and 2NN, and Avnaim [2] gives a characterization-based (and, hence, iteration-based) algorithm for 3NN. Devillers [9] gives faster algorithms for 2CN and 3CN. These running times are summarized in the following table.

k	kCN		kNN	
1	$O(mn \log mn)$	[10]	$O(m^2n^2 \log mn)$	[3, 2]
2	$O(m^2n^2 \log m)$	[9]	$O(m^4n^4 \log mn)$	[3, 2]
3	$O(m^3n^3 \log m)$	[9]	$O(m^{14}n^6 \log mn)$	[2]

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]. Their 3NP algorithm is based on polygon unions, intersections, and Minkowski sum operations. Even though the asymptotic running time of the 3NP algorithm contains a polynomial whose degree is high compared with the others, in practice the 3NP algorithm is faster than Avnaim's 3NN algorithm, which is, in turn, faster than the naive algorithm. Using the s-analysis technique of Section 1.2, the 3NP algorithm has running time in $O(s \log s)$. Avnaim's 3NN algorithm iterates over all choices of edges[4] from V_1, V_2, V_3, U_{12} and U_{13}. (Avnaim uses a sweep-line algorithm to avoid iterating over the edges of U_{23}.) Its worst-case running time is therefore proportional to the product of the sizes of these sets. To determine that a configuration is impossible, the algorithm *must* iterate over all these choices. Its $O(m^{14}n^6 \log mn)$ running time is worse than the $O((mn)^6 \log n)$ running time for the naive algorithm for $k = 3$. However, the 3NN algorithm has running time in $O(s^5 \log s)$. The naive algorithm has running time in $O(s^{4k} \log s)$; for $k = 3$ this is in $O(s^{12} \log s)$. With this analysis, Avnaim's 3NN algorithm is faster than the naive algorithm.

Unfortunately, Avnaim shows that there is no formula for a solution to 3NN purely based on polygon union, intersection, and Minkowski sums [2]. Some sort of iteration over the edges of the configuration space polygons is required and therefore the running time is in $\Omega(s^2)$ in some "practical" sense. Note that such a lower bound cannot be proved rigorously because one could always create an algorithm that deliberately created a huge maximum polygon and thus had running time in $O(s)$, where s is the number of vertices of the maximum polygon.

Milenkovic, *et al.* [8] offer three approaches to translational multiple containment. The first method operates on convex polygons. Their algorithm for 2CN runs in $O(mn \log n)$ time. Their algorithm for 3CN has running time in $O(m^3 n \log mn)$, an improvement of a factor of $O(n^2)$ over previous algorithms, in particular [9]. The 3CN algorithm iterates over combinations of orientations for the lines which separate the convex polygons. The second method uses a MIP (mixed integer programming) model for kNN. This approach is described in detail in [15]. It builds constraints based on convex covers of $\mathcal{V}$ and $\mathcal{U}$, then looks for a feasible solution. It takes one or two minutes on a typical workstation for two or three polygons, but is slow for four or more polygons (it can easily take more than an hour to detect infeasibility in a 4CN problem). The third method is an approximate kNN algorithm, which is an earlier version of the algorithm presented in this paper. Milenkovic and Daniels [7] describe the 2CN algorithm of [8], give a simpler 3CN algorithm with the same running time as [8], provide a newer version of the approximate algorithm, and introduce an exact kNN algorithm. The exact 2CN, 3CN, and kNN algorithms also appear in Part II of this paper.

1.4 Overview

In Part I we characterize a solution to kNN and, from this characterization, create a collection of containment subproblems. Our algorithm solves each subproblem by applying restrictions to the lists $\mathcal{V}$ and $\mathcal{U}$ until a

[4] See Section 2.1 for a description of their 3NN characterization which allows this iteration.

steady state is reached, attempting to find a valid configuration using $\mathcal{V}$ and $\mathcal{U}$, and then subdividing the configuration space if a valid configuration is not yet found. The running time of our approximate algorithm is $\mathrm{O}((\frac{1}{\epsilon})^k \log(\frac{1}{\epsilon}) k^6 s \log s)$ (see Section 7.2).

In a sense, our algorithm substitutes subdivision for the iteration used in Avnaim's exact 3NN algorithm. In practice, much iteration is required in an exact algorithm before hitting the right combination of edges. If the answer is "no", one must check *all* combinations of edges. In our algorithm, restriction alone can often detect infeasibility without requiring the algorithm to use subdivision (see Section 8). While algorithms using restriction and subdivision often perform better in practice than those using only iteration, the latter usually have a better theoretical running time bound. Even when subdivision is unnecessary, the running times of our algorithm are in $\mathrm{O}(m^{28}n^{24}\log mn)$ for 3CN and in $\mathrm{O}(m^{60}n^{40}\log mn)$ for 3NN. These seem worse than the iteration-based methods but, in terms of s-analysis, the running times are both in $\mathrm{O}(s\log s)$ as opposed to $\mathrm{O}(s^5\log s)$ for the iteration-based algorithms. The tests we have run indicate that the running times of our algorithm are much faster than the best times we would estimate for iteration-based exact solutions to 3NN.

In Part II, *Exact Algorithms*, Milenkovic improves the exact algorithms of [8] for 2CN and 3CN and presents an exact algorithm for kCN which has running time $\mathrm{O}(m^{2k}n^k\log n)$ or, if $n < m^{k+1}$, $\mathrm{O}((mn)^{k+1})$. He also gives an exact algorithm for kNN with running time in

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

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

Part I of the paper is structured as follows. In Section 2 we use the notion of contacts to characterize a solution to kNN, establish that kNN is NP-complete, and describe a naive kNN algorithm. The remainder of Part I is devoted to our algorithm for finding an approximate solution to the NP-complete kNN problem. The high-level description of our algorithm in Section 3 draws its structure from our characterization of kNN because the characterization allows us to decompose kNN into a set of subproblems. The algorithm solves each subproblem by first restricting $\mathcal{V}$ and $\mathcal{U}$ until a steady state is reached, testing for a solution, and subdividing the configuration space if a solution is not found. Section 4 presents our restriction method. Section 5 discusses methods for testing for a solution to a subproblem. Section 6 describes our subdivision method. Section 7 establishes the correctness and running time of our algorithm. Section 8 discusses our implementation of the approximate algorithm. It presents containment examples and compares the performance of this algorithm with Milenkovic's exact 3CN and kNN algorithms, which appear in Part II of this paper. It also reveals that regularized set operations are insufficient for implementing our algorithm. Section 9 gives conclusions and outlines future work.

2 Characterization and NP-Completeness

In Section 2.1 we first characterize a solution to kNN using observations about contacts and degrees of freedom. The discussion of contacts is then used in Section 2.2 to establish that kNN is in NP. This, together with an argument that translational containment is NP-hard, shows that translational containment is NP-complete. The characterization suggests a naive algorithm for translational containment, which we describe in Section 2.3.

2.1 Characterizing a kNN Solution

Here we take advantage of the fact that the goal of our containment problem is to find a single valid configuration as opposed to finding the entire set of valid configurations. We characterize a containment solution by showing that if a valid configuration τ exists for a given containment problem, then a valid configuration τ' must exist, where τ' can be reached from τ by a non-overlapping motion of the pieces, and τ' possesses characteristics which facilitate the search for a solution. In particular, τ' is a vertex of a component of the $2k$-dimensional free configuration space, and this component contains τ. In this case we can solve the containment problem by finding τ'. This type of approach has been used for containment by Chazelle to solve a rotational version of 1NN [5], by Avnaim to solve 3NN [2], and by Milenkovic to solve 3CN [8, 7].

Our characterization of a solution to kNN stems from a small number of simple observations about contacts and degrees of freedom. First, we observe that because kNN allows translation in x and y, each polygon has two degrees of freedom. Second, the collection of k polygons has $2k$ degrees of freedom.

There are two types of contacts. The first type involves a vertex and an edge. A vertex-edge contact is a *single contact* because it implies a single linear constraint. The second type involves two vertices. A vertex-vertex contact is a type of *double contact* because it implies two linear constraints. A linear constraint reduces the number of degrees of freedom by one if and only if it is linearly independent of the existing system of constraints.

In terms of $\mathcal{V}$ and $\mathcal{U}$, a single contact of P_i with C means t_i is on the boundary of V_i, and a double contact means t_i is at a vertex of V_i. Similarly, a single contact of P_i with P_j means $t_j - t_i$ is on the boundary of U_{ij} and a double contact means $t_j - t_i$ is at a vertex of U_{ij}.

To simplify the characterization, we define $\mathcal{P} = \langle P_0, P_1, P_2, \ldots, P_k \rangle$, where $P_0 = \overline{C}$. The list $\mathcal{P}$ completely defines an instance of a containment problem.

Theorem 2.1 *If $\mathcal{P}$ has a valid configuration, then $\mathcal{P}$ has a valid configuration with the following characteristics:*

- $2k$ *linearly independent contacts;*
- 2 *linearly independent contacts per polygon* P_i, $0 \leq i \leq k$;

Proof: Assume $\mathcal{P}$ has a valid configuration. We first show that there is a valid configuration with $2k$ linearly independent contacts by arguing as in [22]. If there are fewer than $2k$ linearly independent contacts, then there must be at least one degree of freedom. There must therefore be some simultaneous motion of the polygons P_i, $1 \leq i \leq k$, for which at least some of the polygons move along lines while all polygons retain their contacts. If some of the polygons are moving along lines, then eventually some new contact must form. One of three events must occur: 1) two polygons collide, neither of which is P_0, or 2) two polygons collide, one of which is P_0, or 3) a new contact occurs between two already contacting polygons. The occurrence of one of these events is guaranteed by the fact that the complement of P_0 is bounded (recall that $\overline{P_0}$ is the container). If two polygons P_i and P_j collide, then a new vertex-edge contact is formed. If P_i and P_j already have a vertex-edge contact, then that vertex may slide to the end of the edge and form a vertex-vertex contact. We stop the motion when a new contact occurs. The new contact must be linearly independent of the existing contacts, because it is possible for the system of linear equations corresponding to the existing contacts to be satisfied whether or not a constraint for the new contact is satisfied. Repeating this process forms a set of $2k$ linearly independent contacts.

To establish the second part of the claim, we observe that if a valid configuration for $\mathcal{P}$ exists and a polygon has fewer than two linearly independent contacts, then we can translate that polygon until it loses both degrees of freedom. "Translating" P_0 is equivalent to fixing P_0 and translating the remaining P_i by the same amount in the opposite direction. This implies a valid configuration in which each polygon has at least two linearly independent contacts. ∎

A containment problem can be converted to a set of independent containment subproblems using Theorem 2.1. A collection of independent subproblems can be solved in parallel if sufficient processors are available. In the special case where some of the P_i are identical, equality of some of the U_{ij}s reduces the number of subproblems.

The cost-effectiveness of iterating over a particular set of subproblems should be carefully evaluated when designing a containment algorithm. Let "contacting pair" refer to a pair of polygons which is in contact. The characterization tells us which contacting pairs can occur. The number of possible contacting pairs is purely a function of k, not m or n. Testing the characterization requires solving a subproblem for each possible contacting pair. If the pair is in single contact, we solve a *boundary-restricted* subproblem by replacing U_{ij} by its boundary. If the pair is in double contact, we solve a *vertex-restricted* subproblem by replacing U_{ij} by its vertices. These replacements are restrictions because they replace U_{ij} by a subset of itself, but they are not *valid* restrictions because they do not necessarily preserve all the valid configurations (see Definition 4.3). Since k is a "constant", the characterization does not increase the asymptotic running time in terms of m or n. Nevertheless, the number of ways to choose contacting pairs is exponential in k, roughly k^{4k}.

For $k = 3$ the implication of the first part of the characterization ($\geq 2k$ contacts) is particularly nice. If there exists a solution to 3NN, then there exists a solution in which a set of $2k = 6$ determining contacts contains either a double contact or all single contacts. Testing for a double contact solution requires solving a vertex-restricted subproblem for each U_{ij}. For a solution where six pairs of polygons are in contact, each (P_i,P_j) pair must be in contact. This implies a subproblem in which every polygon in $\mathcal{U}$ is boundary-restricted. Avnaim [2] takes advantage of this to eliminate a factor of m^4 from the running time of his 3NN algorithm.

Unfortunately, for $k > 3$ we must choose $2k$ out of $\binom{k+1}{2}$ possible contacting pairs. The number of ways to choose $2k$ pairs is exponential in k. Both a generalization of Avnaim's 3NN algorithm and the naive algorithm must iterate over all ways of choosing $2k$ single contacts. Using the second part of the characterization (≥ 2 contacts per polygon) also generates an exponential number of subproblems if we apply it to every polygon. Applying it to only a single polygon P_i requires that we check two cases:

1. a double contact with at least one P_j, $0 \leq j \leq k$, $i \neq j$, or
2. a single contact with at least one P_j, $0 \leq j \leq k$, $i \neq j$ and a single contact with at least one $P_{j'}$, $0 \leq j' \leq k$, $i \neq j \neq j'$.

Testing the first case requires k vertex-restricted subproblems (one for each U_{ij}, $0 \leq j \leq k$, $i \neq j$). Each subproblem is vertex-restricted for U_{ij}. The second case requires $k(k-1)/2$ subproblems. Each subproblem is boundary-restricted for two U_{ij}s. This approach therefore requires solving $O(k^2)$ subproblems. In Section 3.1 we show how we partially apply the characterization in our algorithm to create k subproblems.

2.2 NP-Completeness of Translational Multiple Containment

The main result of this section is that kNN is NP-complete. Before proving this, we establish two lemmas. The first shows that kRR, a containment problem for rectangular items in a rectangular container, is NP-hard.

Lemma 2.2 *The kRR problem is NP-hard.*

Proof: Li [15] examines the two-dimensional coordinated motion planning problem of *strip compaction* in which the motion of the polygons is restricted to a strip of fixed width and arbitrary length, and the goal is to minimize the length of the strip. He shows this problem is NP-hard when all the polygons are rectangles and only translations are allowed. He reduces the NP-complete PARTITION problem to strip compaction. We use a similar reduction here to establish that kRR is NP-hard. In an instance of PARTITION [11], we are given a finite set A and a size $s(a) \in Z^+$ for each $a \in A$. We ask: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A \setminus A'} s(a)$? We reduce the NP-complete PARTITION problem to kRR. For each $a \in A$ we construct a rectangle of unit height and width $s(a)$. We also construct a container of height 2 and width $(\sum_{a \in A} s(a))/2$. PARTITION has a solution if and only if there is a valid configuration of the rectangular items inside the rectangular container. This establishes that kRR is NP-hard. ∎

Lemma 2.3 *Two-dimensional translational containment is in NP.*

Proof: Milenkovic, *et al.* [22] show that the marker-making problem, in the absence of rotations, is in NP. The main idea of their argument applies to translational containment. They show that, if a solution exists, then there exists a (possibly different) solution which is uniquely determined by its set of vertex-edge contacts. For containment, Theorem 2.1 guarantees that if a solution exists, then a solution exists which has $2k$ linearly independent contacts. Therefore, we can (nondeterministically) select the correct $2k$ contacts, solve the set of linear equations, and verify the solution. This is a polynomial-time process because the linear system has a polynomial number of equations, and a polynomial number of linear equations can be solved in time which is polynomial in the number of equations. ∎

Theorem 2.4 *Two-dimensional translational containment is NP-complete.*

Proof: Lemma 2.2 shows that kRR is NP-hard. Since all the translational containment problems are harder than kRR, this proves that translational containment is NP-hard. Lemma 2.3 shows that translational containment is in NP, which completes the proof. ∎

2.3 Naive Algorithm

The "naive" kNN algorithm runs in $O((mn)^{2k} \log n)$ time, assuming "constant" k and $n >> m$ (as is typical in our applications), by iterating over all choices of $2k$ contacting pairs among the polygons and container. It relies on the result in Theorem 2.1 that, if a valid configuration exists, then one exists which has $2k$ linearly independent contacts. There are $O((kmn + k^2m^2)^{2k}) = O((mn)^{2k})$ possible choices of $2k$ vertex-edge contacts. For each choice we can check whether the $2k$ contacts are linearly independent and then solve a set of $2k$ linear equations in $O(k^3)$ time (the cost of Gaussian elimination) to obtain t_i, $1 \leq i \leq k$. This configuration can be checked for overlap in $O(\log n)$ time, assuming the following preprocessing. Precompute $\mathcal{V}$ and $\mathcal{U}$. The number of vertices in each of these polygons has an upper bound of $O(m^2n^2)$. Preprocess the polygons in $\mathcal{V}$ and $\mathcal{U}$ so that a point-in-polygon test can be performed in $O(\log mn)$ time. To check a layout, perform a point-in-polygon test for each t_i in V_i and each $t_j - t_i$ in U_{ij}. This costs $O(k^2 \log mn)$ time, which is $O(\log n)$ time for constant k and $n >> m$. The naive algorithm has running time in $\Omega((mn)^{2k})$ because it must iterate over all possible sets of contacts whenever the answer is "no".

3 High-Level Description

Here we describe our algorithm for finding an approximate solution to kNN based on the characterization of Section 2.1. The algorithm creates a collection of k containment subproblems using the characterization. Recall the definitions of $\mathcal{V}$ and $\mathcal{U}$ and their initial values (Section 1.1). We define (in Section 4) and apply certain *valid restrictions* to $\mathcal{V}$ and $\mathcal{U}$, replacing each member by a subset while not eliminating any valid configurations. The overall structure of the algorithm for solving each subproblem is to apply restrictions until either infeasibility is detected or no more significant "shrinking" is possible. If no exact solution to the subproblem is found in these restricted sets, the algorithm *branches* by subdividing one of the V_i and recursing on each half while the other V_j, $1 \leq j \leq k$, $j \neq i$, remain the same. If the algorithm subdivides the configuration spaces down to tolerance without detecting infeasibility, it returns an ϵ-approximate solution: t_i lies within V_i and $t_j - t_i$ lies within 2ϵ of U_{ij}. The running time of our algorithm is $O(\left(\frac{1}{\epsilon}\right)^k \log\left(\frac{1}{\epsilon}\right) k^6 s \log s)$ (see Section 7.2).

3.1 Applying the Characterization

Our philosophy is to avoid iteration as much as possible because iteration contributes to the worst-case cost of the algorithm, particularly in terms of s-analysis. For instance, Avnaim and Boissonnat's algorithm for 3NP [3, 2] has running time $O(s \log s)$, and Avnaim's algorithm for 3NN [2] has running time $O(s^5 \log s)$. The algorithm for 3NP is faster because it avoids iterating over the edges and vertices of configuration space polygons. Like the algorithm for 3NP, we avoid this type of iteration as much as possible. We do this by using valid restriction and subdivision.

There is another type of iteration one must consider. This iteration is based on the characterization of Theorem 2.1. To apply the entire characterization, one needs to choose contacting pairs in a number of ways which is exponential in k, roughly k^{4k}. If one were to generalize Avnaim's algorithm for 3NN [2] to an algorithm for kNN, the factor of k^{4k} would not be a significant part of the cost compared to the factor dependent on m and n (roughly $m^{2k^2}(mn)^{2k}$). For our approach, however, a factor of k^{4k} is significant.

It is possible to use valid restriction and subdivision and avoid iterating over *any* contacting pairs. In fact, this is what we initially implemented. However, we feel that considering contacting pairs is potentially very useful. For instance, if we invoke a condition weaker than the second part of Theorem 2.1 to assume that a particular P_i is in contact with the boundary of the container, then we can replace V_i with its boundary. Note that this boundary restriction covers the cases where the boundary of the container has either a single or a double contact with some P_i.

This partial characterization implies a significant restriction. However, the disadvantage is that we must iterate over all k choices of i by solving k subproblems[5]. Our current conjecture is that iterating over up to k choices of (partial) characterization will allow us to avoid subdivision often enough to offset the extra

[5]However, as noted in Section 2.1, these subproblems can be solved in parallel. In our current implementation we solve them sequentially.

factor of k in cost. We present data in Section 8.3 to support this conjecture in the case of $k = 4$ for the sequential version of the algorithm.

The boundary restriction does not add an extra factor of k in cost for containment problems in which all the items are identical. In this case, all k subproblems are identical, and so only *one* subproblem must be solved.

Both our algorithm and Avnaim's 3NN algorithm use a characterization to create subproblems. An important distinction, however, is that Avnaim solves each subproblem by iterating over the vertices or edges of the configuration space polygons, while we solve each subproblem using valid restriction and subdivision, two processes which do not iterate over the vertices or edges of the configuration space polygons.

3.2 The Algorithm

In the algorithm, $\mathcal{P}$, $\mathcal{U}$, and $\mathcal{V}$ are set initially to the values given in Section 1.1. In the pseudo-code below we denote V_i by U_{0i} and redefine $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$; hence we need only discuss $\mathcal{U}$. We present the algorithm as a decision procedure. If a solution is found the algorithm returns YES (and saves the valid configuration); otherwise it returns NO. The high-level routine, APPROX-CONTAIN(), solves k boundary-restricted containment subproblems by calling SOLVE-SUBPROBLEM() for each of them. The list $\mathcal{U}^i$ represents a copy of $\mathcal{U}$ for the ith processor, and U_{0i}^i is processor i's configuration space for P_i with respect to the container. If any of the subproblems finds a solution, APPROX-CONTAIN() returns YES. Note that in order to perform a boundary restriction on a U_{0i}, we need only intersect it with the boundary of U_{0i}^{orig}, where U_{0i}^{orig} is U_{0i} at the start of the algorithm.

Before calling SOLVE-SUBPROBLEM(), the algorithm performs some initial tests on $\mathcal{U}$. First, it restricts the configuration spaces by calling SSGR(), which performs a valid restriction called Steady-State Geometric Restriction. SSGR() is described in Section 4. If some U_{ij} becomes empty as a result of this restriction, APPROX-CONTAIN() returns NO because the containment problem is infeasible. If no infeasibility is detected, a second test is performed. This test is a call to TOO-SMALL(). TOO-SMALL() checks if $\mathcal{U}$ satisfies the termination condition for the approximate algorithm; i.e. all of the U_{0i} (V_i) polygons are too small. If the condition is satisfied, BUILD-CONFIGURATION() constructs an ϵ-approximate configuration, APPROX-CONTAIN() returns YES and the algorithm terminates. The ϵ used in TOO-SMALL() is the same ϵ used within SSGR().

SOLVE-SUBPROBLEM() performs three operations on $\mathcal{U}$ for a subproblem. First, it restricts the configuration spaces by calling SSGR(). If no infeasibility is detected in the subproblem by SSGR(), GREEDY-EVALUATE() checks for a valid configuration within $\mathcal{U}$. If a valid configuration is found by GREEDY-EVALUATE(), SOLVE-SUBPROBLEM() returns YES. GREEDY-EVALUATE() is described in Section 5. If it finds a valid configuration, it returns YES and the algorithm terminates. GREEDY-EVALUATE() includes a call to the tolerance test TOO-SMALL(); thus, if $\mathcal{U}$ for the subproblem satisfies the termination condition, BUILD-CONFIGURATION() constructs an ϵ-approximate configuration, YES is returned, and the algorithm terminates. If GREEDY-EVALUATE() does not return YES, SIZE-SUBDIVIDE(), described in Section 6, subdivides $\mathcal{U}$ into $\mathcal{U}'$ and $\mathcal{U}''$ and then SOLVE-SUBPROBLEM() recurses on $\mathcal{U}'$ and $\mathcal{U}''$.

APPROX-CONTAIN($\mathcal{U}$, k)
    $\mathcal{U}^{\mathrm{orig}} \leftarrow \mathcal{U}$
    $\mathcal{U} \leftarrow$ SSGR($\mathcal{U}$, k)
    if ($\exists U_{ij}$ in $\mathcal{U}, U_{ij} = \emptyset$) **return** NO
    if (TOO-SMALL($\mathcal{U}$, k)) **return** BUILD-CONFIGURATION($\mathcal{U}$, k)

    /* Search for a boundary-restricted solution */
    for ($i = 1$ **to** k) **do in parallel**
        $\mathcal{U}^i \leftarrow \mathcal{U}$
        $U_{0i}^i \leftarrow U_{0i} \cap$ BOUNDARY-OF(U_{0i}^{orig})
        if (SOLVE-SUBPROBLEM($\mathcal{U}^i$, k) = YES) **return** YES
    return NO

SOLVE-SUBPROBLEM($\mathcal{U}$, k)
    $\mathcal{U} \leftarrow$ SSGR($\mathcal{U}$, k)

if ($\exists U_{ij}$ in $\mathcal{U}, U_{ij} = \emptyset$) **return** NO
if (GREEDY-EVALUATE($\mathcal{U}$, k) = YES) **return** YES
$\mathcal{U}', \mathcal{U}'' \leftarrow$ SIZE-SUBDIVIDE($\mathcal{U}$, k)
if (SOLVE-SUBPROBLEM($\mathcal{U}'$, k) = YES) **return** YES
if (SOLVE-SUBPROBLEM($\mathcal{U}''$, k) = YES) **return** YES
return NO

TOO-SMALL($\mathcal{U}$, k)
$V \leftarrow$ member of $\langle U_{0i}, 1 \leq i \leq k \rangle$ with maximum bounding box width or height
$\mathcal{B} \leftarrow$ bounding box of V
if ($height(\mathcal{B}) < \frac{\epsilon}{\sqrt{2}}$ **and** $width(\mathcal{B}) < \frac{\epsilon}{\sqrt{2}}$) **return** YES
return NO

BUILD-CONFIGURATION($\mathcal{U}$, k)
$t_0 = (0, 0)$
for ($i = 1$ **to** k) select $t_i \in U_{0i}$
$\tau \leftarrow \langle t_0, t_1, t_2 \ldots, t_k \rangle$
return YES

The following three parts of this algorithm are described in more detail in the next three sections: 1) SSGR(), 2) GREEDY-EVALUATE(), and 3) SIZE-SUBDIVIDE(). The first is described in Section 4, the second in Section 5, and the third in Section 6.

4 Steady-State Geometric Restriction

In this section we develop our Steady-State Geometric Restriction, SSGR(). In Section 4.1 we first formally define *valid restriction* and then present a valid restriction which operates on $\mathcal{V}$ and $\mathcal{U}$. For notational convenience, we denote V_i by U_{0i} and redefine $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$; hence we need only discuss $\mathcal{U}$. We explain in Section 4.2 that the restriction operation is a generalization of Avnaim and Boissonnat's formulas for solving 2NN and 3NP [3]. The steady-state restriction process applies the restriction operation until no more significant shrinking occurs. In Section 4.3 we give pseudo-code for SSGR() and prove a result about the power of our steady-state restriction process.

4.1 A Valid Restriction

In [7, 8] we present restrictions which operate directly on $\mathcal{U}$. The restrictions use the Minkowski sum and a *closed* intersection operator (see Section 8.5). Each member of $\mathcal{U}$ is replaced using operations that include other members of $\mathcal{U}$ as input. Definition 4.3 below gives a formal definition of *restriction*.

Part of the containment algorithm involves subdivision of the configuration spaces. Therefore a particular list $\mathcal{U}$ of configuration spaces may not cover all possible valid configurations. We must therefore consider those configurations that correspond to a particular list.

Definition 4.1 *Given a list $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$ of configuration space polygons, a configuration $\langle t_0 = (0,0), t_1, t_2, t_3, \ldots, t_k \rangle$ is* **valid with respect to** *$\mathcal{U}$ if $t_j - t_i \in U_{ij}$, for $0 \leq i < j \leq k$.*

Definition 4.2 *Given a list $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$ of configuration space polygons, for $0 \leq i < j \leq k$, define U^*_{ij} to be the set of vectors $t_j - t_i$ such that t_i and t_j are the ith and jth members of a configuration $\langle t_0 = (0,0), t_1, t_2, t_3, \ldots, t_k \rangle$ which is valid with respect to $\mathcal{U}$.*

In other words, U^*_{ij} is the projection of the set of all $2k$-dimensional valid configurations into U_{ij}. Clearly $U^*_{ij} \subseteq U_{ij}$.

Definition 4.3 *The list $\mathcal{U}' = \langle U'_{ij}, 0 \leq i < j \leq k \rangle$ is a* **restriction** *of $\mathcal{U} = \langle U_{ij}, 0 \leq i < j \leq k \rangle$ if $U'_{ij} \subseteq U_{ij}$ for $0 \leq i < j \leq k$. It is a* **valid restriction** *if it also satisfies $U^*_{ij} \subseteq U'_{ij}$ for $0 \leq i < j \leq k$.*

Note that in the definition, U^*_{ij} is defined with respect to the list $\mathcal{U}$, not the list $\mathcal{U}'$. Otherwise, every restriction would be valid. Since a valid restriction does not "throw away" any valid configurations, we think of it as an operation *on* $\mathcal{U}$ that replaces some or all of the U_{ij} in $\mathcal{U}$ by subsets of themselves.

Theorem 4.1 below encompasses all three restrictions from [7, 8]. Figure 2 illustrates the theorem.

Theorem 4.1 $U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj}),\quad 0 \leq h, i, j \leq k, h \neq i \neq j$, *is a valid restriction.*

Proof: If $\langle t_0 = (0,0), t_1, t_2, t_3, \ldots, t_k \rangle$ is valid with respect to $\mathcal{U}$, then $t_j - t_i \in U_{ij}$ for $0 \leq i, j \leq k$. It follows that,

$$t_j - t_i = t_h - t_i + t_j - t_h \in U_{ih} \oplus U_{hj}.$$

Therefore $t_j - t_i \in U_{ij} \cap (U_{ih} \oplus U_{hj})$. Since this is true for any valid configuration, the restriction given in the theorem must be valid. ■

The nature of the restriction in Theorem 4.1 differs from the boundary and vertex restrictions of Section 2.1 not only because it is valid, but also because we restrict a polygon of $\mathcal{U}$ using operations that include other polygons of $\mathcal{U}$ as input.

4.2 Avnaim and Boissonnat's Solutions to 2NN and 3NP

We show that the restriction of Theorem 4.1 can be viewed as a generalization of Avnaim and Boissonnat's [3] solutions to 2NN and 3NP.

Avnaim and Boissonnat show for 2NN that the set of all the valid relative positions of P_j with respect to P_i is given (in our notation) by: $U^*_{ij} = U_{ij} \cap (U_{i0} \oplus U_{0j})$. This is an exact solution. Each element of the set U^*_{ij} corresponds to a valid configuration. We observe that, for kNN, each valid displacement for two polygons must also be a solution for 2NN and therefore, for kNN, $U^*_{ij} \subseteq U_{ij} \cap (U_{i0} \oplus U_{0j})$, which is a special case of Theorem 4.1.

Avnaim and Boissonnat observe that, for 3NP, a valid translation t_1 exists for P_1 only if:

$$t_1 \in U_{01} \cap (U_{02} - (t_2 - t_1)) \cap (U_{03} - (t_3 - t_1)) \tag{2}$$

where $(t_2 - t_1) \in U_{12}$ and $(t_3 - t_1) \in U_{13}$. Thus, $U_{01} \leftarrow U_{01} \cap (U_{02} \oplus U_{21}) \cap (U_{03} \oplus U_{31})$ is a valid restriction for 3NP.

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. Suppose we have three parallelograms which are the same affine transformation of (different but axis-parallel) rectangles. If the three parallelograms intersect pairwise, then all three have a common point. This again is a special case of Theorem 4.1: $U^*_{ij} \subseteq U_{ih} \oplus U_{hj}$ for $h \neq i \neq j$.

In our notation, Avnaim and Boissonnat's solution to 3NP applies $U_{ij} \leftarrow U_{ij} \cap (U_{i0} \oplus U_{0j})$, $1 \leq i < j \leq 3$, followed by $U_{ij} \leftarrow U_{ih} \oplus U_{hj}$ for $1 \leq h \neq i \neq j \leq 3$. Each element of U_{ij} then corresponds to a solution to 3NP.

4.3 Steady-State Restriction

One would hope that repeated applications of the restriction in Theorem 4.1 would yield all the U^*_{ij} polygons in the limit. Actually, this is *always* true only for $k = 2$. For $k = 3$, one can construct counter-examples, but only with difficulty. For $k > 3$, it is easy to construct counter-examples.

We iteratively apply the valid restriction of Theorem 4.1 until no more significant shrinking occurs. We denote the resulting restriction by SSGR() (steady-state geometric restriction). In the pseudo-code below, $\mathcal{U}^{\emptyset}$ is a list containing $k(k+1)/2$ copies of the empty set.

```
SSGR(𝒰, k)
    do
        for (i = 0 to k)
            for (j = i + 1 to k)
                for (h = 0 to k)
```

if $(h \neq i \neq j)$
$U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj})$
if $(\exists U_{ij}$ in $\mathcal{U}, U_{ij} = \emptyset)$ **return** $\mathcal{U}^{\emptyset}$
$S \leftarrow$ maximum fractional area shrinkage in $\mathcal{U}$
$d \leftarrow$ maximum diameter of a polygon in $\mathcal{U}$
while $(S > \alpha$ **and** $d > \epsilon)$
return $\mathcal{U}$

The parameter ϵ is specified by the user. It is used to help judge when to terminate the restriction process. The parameter α is *not* specified by the user. It is the relative area shrinkage, $0 \leq \alpha \leq 1$. For $\alpha = 0$, the program achieves a "true" steady state (if the maximum polygon diameter is greater than ϵ and the maximum area is greater than ϵ^2) in which additional restrictions cannot shrink the polygons further. However, this can require an infinite number of iterations. We use $\alpha = 0.1$ and $\epsilon = 0.5$ in our implementation; these are satisfactory for our applications. Using nonzero values of α and ϵ gives us a bound on the number of restriction iterations (see Section 7.2).

The steady-state restriction process is quite powerful. We know that $U^*_{ij} \subseteq U_{ih} \oplus U_{hj}$ for $h \neq i \neq j$. 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 the notion of a *list restriction.* We then show in Theorem 4.2 that SSGR() satisfies all list restrictions when $\alpha = 0$.

Definition 4.4 *Given a list of indices* $\pi = i, \pi_1, \pi_2, \pi_3, \ldots \pi_{q-1}, \pi_q, j$, *the* **list restriction** *of* U_{ij} *corresponding to* π *is the restriction:*

$$U_{ij} \leftarrow U_{ij} \cap (U_{i\pi_1} \oplus U_{\pi_1\pi_2} \oplus U_{\pi_2\pi_3} \oplus \cdots \oplus U_{\pi_{q-1}\pi_q} \oplus U_{\pi_q j}).$$

Theorem 4.2 *If SSGR() reaches a steady state for $\alpha = 0$, then all possible list restrictions are satisfied.*

Proof: The proof is by induction on the length of a list, which is the number of indices in the list. The base case, for a path of length two, is true because SSGR() satisfies all list restrictions of length two. The induction hypothesis is that the list restriction for every list of length $q+1$ is satisfied. Consider the list $\pi = i, \pi_1, \pi_2, \ldots, \pi_a, \pi_b, \pi_c, \ldots, \pi_q, j$ of length $q+2$. We show that the list restriction for π is satisfied. First, by the inductive hypothesis, the list restriction for the path $\pi' = i, \pi_1, \pi_2, \ldots, \pi_a, \pi_c, \ldots, \pi_q, j$ of length $q+1$ is satisfied, so

$$U_{ij} = U_{ij} \cap (U_{i\pi_1} \oplus \cdots \oplus U_{\pi_a\pi_c} \oplus \cdots \oplus U_{\pi_q j}).$$

Now, because a steady state has been achieved, applying SSGR() does not change $U_{\pi_a\pi_c}$. This means

$$U_{\pi_a\pi_c} = U_{\pi_a\pi_c} \cap (U_{\pi_a\pi_b} \oplus U_{\pi_b\pi_c})$$

and hence $U_{\pi_a\pi_c} = U_{\pi_a\pi_b} \oplus U_{\pi_b\pi_c}$. Substitution yields

$$U_{ij} = U_{ij} \cap (U_{i\pi_1} \oplus \cdots \oplus U_{\pi_a\pi_b} \oplus U_{\pi_b\pi_c} \oplus \cdots \oplus U_{\pi_q j})$$

which is the list restriction for π. Thus, the list restriction for π is satisfied. ■

5 Greedy Evaluation

Given restricted $\mathcal{V}$ and $\mathcal{U}$ we want to check if there is a valid configuration associated with them. To answer this we can, of course, invoke either a MIP [8] or an exact algorithm (as in Part II) for kNN at this point, but that is costly. For the purposes of the algorithm given in Section 3.2, all we require is a fast test which constructs a configuration and tests to see if it satisfies the conditions for a valid configuration: 1) $t_i \in V_i$, $1 \leq i \leq k$ and 2) $t_j - t_i \in U_{ij}$, $1 \leq i < j \leq k$. If the test succeeds, we have a solution. If it fails, there may still be a solution, but additional subdivision is required to find it. Here, again, we define $U_{0i} = V_i$, $1 \leq i \leq k$.

5.1 A Greedy Algorithm

The method we present here is a greedy strategy which performs a "shrinking" operation on each U_{ij}. The operation selects a point u_{ij} in U_{ij}, replaces U_{ij} by $\{u_{ij}\}$, and then propagates the effects of this "shrinking" via SSGR(). If any U_{ij} becomes empty during this process, then the chosen combination of single U_{ij} points cannot be part of a valid solution. If, on the other hand, we succeed in shrinking each U_{ij} to a point and then applying SSGR() without any U_{ij} becoming empty, then a valid configuration exists (see the proof of Theorem 5.1). Section 5.2 suggests heuristics for selecting a shrinking point for each U_{ij}.

GREEDY-EVALUATE($\mathcal{U}$, k)
    if (TOO-SMALL($\mathcal{U}$, k)) **return** BUILD-CONFIGURATION($\mathcal{U}$, k)
    while ($\forall U_{ij}$ in $\mathcal{U}, U_{ij} \neq \emptyset$ and $\exists U_{ij}$ in $\mathcal{U}$, U_{ij} is not a single point)
        select smallest U_{ij} in $\mathcal{U}$ which is not a point
        select a point $u_{ij} \in U_{ij}$
        $U_{ij} \leftarrow \{u_{ij}\}$
        $\mathcal{U} \leftarrow$ SSGR($\mathcal{U}$, k)
    if ($\forall U_{ij}$ in $\mathcal{U}, U_{ij} \neq \emptyset$) **return** BUILD-CONFIGURATION($\mathcal{U}$, k)
    return MAYBE

Theorem 5.1 *If each U_{ij} has been shrunken to a point and a subsequent application of* SSGR() *does not cause any U_{ij} to be empty, then* GREEDY-EVALUATE() *yields a valid configuration for $\mathcal{P}$.*

Proof: By definition, a valid configuration for $\mathcal{P}$ is a k-tuple of translations $\langle t_0, t_1, t_2, \ldots, t_k \rangle$ such that $t_j - t_i \in U_{ij}$, $0 \leq i < j \leq k$. Since $U_{0j} \neq \emptyset$, $1 \leq j \leq k$ and consists of a single point, let t_j be that point. Since $t_0 = \{0, 0\}$, we have $U_{0j} = \{t_j - t_0\}$. Now, for U_{ij}, $1 \leq i \neq j \leq k$, SSGR($\mathcal{U}$, k) $\neq \emptyset \rightarrow U_{ij} \subseteq U_{ih} \oplus U_{hj}$. In particular, for $h = 0$, $u_{ij} = u_{0j} - u_{0i}$, where $u_{ij} \in U_{ij}$, $u_{0j} \in U_{0j}$, and $u_{0i} \in U_{0i}$. Since $u_{0j} = t_j - t_0$ and $u_{0i} = t_i - t_0$, $u_{0j} - u_{0i} = t_j - t_i$. Thus, the points of $\mathcal{U}$ constitute a valid configuration, and further restrictions cannot alter that fact. ■

5.2 Heuristics for Selecting a Shrinking Point

We present several heuristics for choosing a shrinking point $u_{ij} \in U_{ij}$. In Section 8.2 we compare the effectiveness of these methods in practice. The most naive heuristic is to select a point at random. Our other heuristics are based on the following claim, which gives a necessary but insufficient condition for the existence of a valid configuration.

Claim 5.2 *If a configuration $\langle t_1, t_2, t_3 \ldots, t_k \rangle$ for $\mathcal{P}$ is valid, then:*

$$U_{hi} \cap \bigcap_{j \neq i} (U_{hj} + (t_i - t_j)) \neq \emptyset.$$

Proof: For fixed h, for each i, $U_{hi} + t_h - t_i$ contains the origin. The claim follows by translating each $U_{hj} + t_h - t_j$, $i \neq j$, by $t_i - t_h$. ■

Claim 5.2 means that a solution exists only if there are choices for the u_{ij} which cause the translated U_{ij}s to intersect. SSGR() enforces *pairwise* intersection of the translated U_{ij}s. However, it does not guarantee the multiple intersection in Claim 5.2. This is illustrated by Figure 3. Figure 3a shows a containment problem for three convex items. Figure 3b depicts a zoomed-in screen dump of the three U_{0i} polygons for this example. U_{01} is not translated, U_{02} is translated by a vector in U_{21}, and U_{03} is translated by a vector in U_{31}. In this case, a solution is not found because the polygons do not have a common intersection.

Avnaim and Boissonnat's 3NP algorithm works because whenever the container is a parallelogram the U_{0i}s are parallelograms with sides parallel to the container. Avnaim and Boissonnat prove that for parallelograms with parallel corresponding sides, pairwise intersection implies multiple intersection. This version of Helly's Theorem holds even if $k > 3$.

Note: One might be tempted to conclude from this that our algorithm provides an exact polynomial-time solution for kNP. However, this is highly unlikely to be true because kNP is NP-complete (see Section 2.2).

The key to this paradox is that Claim 5.2 is necessary but *not* sufficient. For 3NP, we can select u_{12} and know that u_{23} and u_{31} exist and satisfy Claim 5.2. Unfortunately, for $k > 3$, we cannot necessarily obtain a self-consistent collection of u_{ij}s for which $u_{ij} = u_{ih} + u_{hj}$ for all $h \neq i \neq j$.

5.2.1 Convexity of Union

Theorem 5.3 gives sufficient conditions for a list of k closed polygons (convex or nonconvex) to have a common intersection[6].

Theorem 5.3 *Given a list of closed polygons $\mathcal{Q} = \langle Q_1, Q_2, \ldots, Q_k \rangle$, if $Q_i \cup Q_j$ is convex, $i \neq j$, $1 \leq i < j \leq k$, then*

$$\bigcap_{i=1}^{k} Q_i \neq \emptyset.$$

Proof: If Q_k is nonconvex, let q be a point which is on the boundary of Q_k but is not on the boundary of the convex hull of Q_k, which we denote by $\mathrm{CH}(Q_k)$. We claim $q \in Q_i$, $1 \leq i \leq k$. Since Q_k is a closed polygon it contains its boundary; thus $q \in Q_k$. Now, let $R = \mathrm{CH}(Q_k) \setminus Q_k$. Because $Q_i \cup Q_k$ is convex, $\mathrm{CH}(Q_k) \subseteq (Q_i \cup Q_k)$. This implies that $R \subseteq Q_i$. Since Q_i is a closed polygon it contains the boundary of any subset of itself. Therefore Q_i contains the boundary of R, which contains q. This establishes the claim that $q \in Q_i$, $1 \leq i \leq k$.

Now suppose Q_k is convex. We prove this case by induction. The base case for $k = 1$ is trivially true. The induction hypothesis is that our claim is true for $k - 1$. Now we consider Q_k. We claim that $Q_i \cup Q_k$ is convex implies that $Q_i \cap Q_k \neq \emptyset$ for $1 \leq i \leq k$. This is true because of the more general fact that if the union of two closed sets is connected, then they have a point in common. To establish this fact, we argue as follows. Let $Q_i \cup Q_k$ be connected, $q_i \in Q_i$ and $q_k \in Q_k$. If either $q_i \in Q_k$ or $q_k \in Q_i$, we are done. Otherwise, since $Q_i \cup Q_k$ is connected, there exists a path $\gamma(t)$, $0 \leq t \leq 1$, where $\gamma(0) = q_i$ and $\gamma(1) = q_k$ and either $\gamma(t) \in Q_i$ or $\gamma(t) \in Q_k$ for $0 < t < 1$. Consider $t^* = \sup\{t \mid \gamma(t) \in Q_i\}$. The point $\gamma(t^*)$ is a limit point of $Q_i \cap \gamma(t)$ as t increases from 0. Let $t^{**} = \inf\{t \mid \gamma(t) \in Q_k\}$. We claim $t^{**} \leq t^*$. For, if $t^* < t^{**}$ then there exists t' such that $t^* < t' < t^{**}$ and $\gamma(t')$ does not belong to either Q_i or Q_k. Therefore $\gamma(t^*) \in Q_k$. This establishes $Q_i \cap Q_k \neq \emptyset$ for $1 \leq i \leq k$.

Now we want to apply the induction hypothesis to the list of $k - 1$ nonempty polygons: $\langle (Q_1 \cap Q_k), \ldots, (Q_{k-1} \cap Q_k) \rangle$ to establish that $\bigcap_{i=1}^{k} Q_i \neq \emptyset$. We claim we can do this because $(Q_i \cap Q_k) \cup (Q_j \cap Q_k)$ is convex $1 \leq i < j < k$. This is true because

$$(Q_i \cap Q_k) \cup (Q_j \cap Q_k) = Q_k \cap (Q_i \cup Q_j).$$

This is the intersection of two convex sets and is therefore itself convex. This completes the induction and establishes the theorem. ■

Corollary 5.4 *Suppose $\mathcal{Q}$ is a list of polygons satisfying the conditions of Theorem 5.3. Let Q_0 be convex and intersect all the other polygons: $Q_0 \cap Q_i \neq \emptyset$, $1 \leq i \leq k$. It follows that,*

$$\bigcap_{i=0}^{k} Q_i \neq \emptyset.$$

Proof: In the proof of Theorem 5.3 for the case where one of the polygons is convex, we only used the property that $Q_i \cap Q_k \neq \emptyset$. That property is satisfied here because of the conditions of the theorem. ■

Note that we have an exact solution to 3NN if U_{01}, $U_{02} + u_{21}$, and $U_{03} + u_{31}$ satisfy the conditions of Theorem 5.3 or Corollary 5.4. However, these results cannot give an exact solution to kNN (see the note on page 13). The results are still useful because they suggest a heuristic we can use in our kNN algorithm for selecting u_{ij}. Theorem 5.3 suggests we choose u_{ij} which "maximizes the convexity" of $U_{0i} \cup (U_{0j} + u_{ji})$. There are many possible ways to measure the amount of convexity of a polygon. We choose to minimize the area of $\mathrm{CH}(U_{0i} \cup (U_{0j} + u_{ji})) \setminus (U_{0i} \cup (U_{0j} + u_{ji}))$. The problem of finding a translation which minimizes this area is an open one (see Section 9.2). In our implementation we find an approximate answer (see Section 8.2).

[6]The generalization from 3 to k polygons was suggested by Dan Roth.

5.2.2 Maximizing Intersection Area

In GREEDY-EVALUATE(), shrinking a particular U_{ij} to a single point $\{u_{ij}\}$ fixes the translation of P_j with respect to P_i. Subsequent applications of SSGR() can therefore cause the associated polygons in $\mathcal{U}$ to shrink. A containment solution can only be reached, however, if each polygon in $\mathcal{U}$ can be shrunk to a point without causing any polygon in $\mathcal{U}$ to vanish.

Suppose we choose the shrinking point for a U_{ij} to be the point which shrinks U_{0i} and U_{0j} the least. We might then have a better chance of preventing the polygons in $\mathcal{U}$ from vanishing. This suggests the following heuristic for selecting the shrinking point: choose u_{ij} to be the point in U_{ij} such that the area of $U_{0i} \cap (U_{0j} + u_{ji})$ is maximized. If U_{0i} and U_{j0} are simple polygons with a and b vertices, respectively, then the optimal u_{ij} can be found exactly in polynomial time using a brute-force configuration space method, by solving $O(a^2b^2)$ two-dimensional convex optimization problems [27].

Solutions to some closely related problems often provide large intersection area in practice. One possibility is to align the centroids of the polygons. Another is to maximize the Hausdorff distance between the polygons. An inexpensive approximation to this is given in [1]. That result states that if you superimpose the lower left corner of the bounding boxes of two polygons, the Hausdorff distance between them is bounded by $(1+\sqrt{2})\delta$, where δ is the minimum Hausdorff distance possible for the two polygons. Of course, there is nothing special about the lower left corner. The proof works for any corresponding pair of bounding box corners.

In practice, we use a method which (approximately) maximizes the area of $U_{0i} \cap (U_{0j} + u_{ji})$. We overlay a grid on each component of U_{ij} and, at each grid point u_{ij}, we measure the area of $U_{0i} \cap (U_{0j} + u_{ji})$. Section 8.2 describes this in more detail.

6 Size-Based Subdivision

There are many possible goals for a subdivision method [6]. One goal is to reduce the "size" of a U_{ij}, where size can be interpreted as the number of vertices, the number of reflex vertices, the number of components, the area, or the maximum (x or y) dimension of the bounding box, for example. We call such a method a *size-based* subdivision method.

We present here one size-based method which reduces the maximum (x or y) dimension of the bounding boxes of the U_{ij}s. Again, we define $U_{0i} = V_i$, $1 \le i \le k$. Our method operates on V in $\langle U_{0i}, 1 \le i \le k\rangle$ which has the maximum bounding box dimension. It cuts V with either a vertical or horizontal line, depending on the aspect ratio of V.

```
SIZE-SUBDIVIDE(𝒰, k)
    V ← member of ⟨U_0i, 1 ≤ i ≤ k⟩ with maximum bounding box width or height
    ℬ ← bounding box of V
    if (width(ℬ) > height(ℬ))
        ℬ' ← left half of ℬ; ℬ'' ← right half of ℬ
    else
        ℬ' ← top half of ℬ; ℬ'' ← bottom half of ℬ
    𝒰' ← 𝒰 with V replaced by V ∩ ℬ'
    𝒰'' ← 𝒰 with V replaced by V ∩ ℬ''
    return 𝒰', 𝒰''
```

7 Analysis

7.1 Correctness

Claim 7.1 *If a valid configuration exists, then the algorithm will find an ϵ-approximate configuration.*

Proof: The algorithm converges because each invocation of SIZE-SUBDIVIDE() reduces the maximum height or width of the bounding box of U_{0i}, $1 \le i \le k$, by a factor of two, stopping when the height and width reach the tolerance $\frac{\epsilon}{\sqrt{2}}$.

If a solution exists, then our characterization guarantees that one of the boundary-restricted subproblems has a solution. We claim that our algorithm does not eliminate any solutions to these subproblems. This is because Theorem 4.1 guarantees that SSGR() is a valid restriction.

Now we argue that any configuration which is found is an ϵ-approximate configuration. A configuration can only be generated by BUILD-CONFIGURATION(). There are two cases to consider. In the first case, BUILD-CONFIGURATION() is called by GREEDY-EVALUATE() because each U_{ij} has been shrunken to a point and a subsequent application of SSGR() does not cause any U_{ij} to be empty. In this case, Theorem 5.1 guarantees that the configuration contains no overlap; hence it is a valid configuration. Clearly, a valid configuration is also an ϵ-approximate configuration.

In the second case, BUILD-CONFIGURATION() is called because the test of TOO-SMALL() has been passed; i.e. the height and width of the bounding box of U_{0i}, $1 \leq i \leq k$, are both at most $\frac{\epsilon}{\sqrt{2}}$. In this case, the diameter of each U_{0i} is less than ϵ. We argue that no point of any polygon is more than 2ϵ inside of any other polygon's boundary. None of the polygons overlap the container, because each t_i chosen in BUILD-CONFIGURATION() is in U_{0i}. Thus, we need only bound the overlap between a pair of polygons P_i and P_j, $1 \leq i \neq j \leq k$.

Any $u_{ij} \in U_{ij}$, having survived SSGR(), can be expressed as $t_j - t_i$, where $t_i \in U_{0i}$ and $t_j \in U_{0j}$. Polygons $P_i + t_i$ and $P_j + t_j$ do not overlap. Now let v_i and v_j be the points of U_{0i} and U_{0j} chosen in BUILD-CONFIGURATION(). Let $D(p,q)$ denote the distance between points p and q. Since the diameter d of any U_{0i} is less than ϵ, we have $D(v_i, t_i) < \epsilon$ and $D(v_j, t_j) < \epsilon$. Translating P_i from t_i to v_i can therefore cause a point of P_i to be at most a distance of ϵ inside P_j. Translating P_j from t_j to v_j can at most double this overlap. Therefore, the maximum overlap between two polygons in the configuration $\langle t_1, t_2, t_3, \ldots, t_k \rangle$ is at most 2ϵ. ■

7.2 Running Time

Here we calculate the asymptotic running time of our algorithm. We assume:

- 2ϵ is the maximum overlap between two polygons permitted by the user,
- α is the fractional shrinkage in area required by each iteration of SSGR(),
- s is as defined in Section 1.2. We let the class of restriction operations consist of $U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj})$ together with intersection with a half-plane determined by a horizontal or vertical line.

Theorem 7.2 *The running time of our algorithm is in* $\mathrm{O}((\frac{1}{\epsilon})^k \log(\frac{1}{\epsilon}) k^6 s \log s)$.

Proof: Let S be the number of subdivisions and R be the running time of SSGR(). The running time is dominated by Sk^3R. To find S, we first assume that the maximum height or width of the bounding box of a U_{0i}, $1 \leq i \leq k$, is q. The height of the subdivision tree is therefore $\mathrm{O}(k(\log_2 q - log_2 \epsilon))$. The k term corresponds to the total number of polygons that are subdivided in the worst case. The number of nodes in the subdivision tree is therefore

$$S = \mathrm{O}(2^{k \log_2(\frac{q}{\epsilon})}) = \mathrm{O}(2^{\log_2((\frac{q}{\epsilon})^k)}) = \mathrm{O}\left(\left(\frac{q}{\epsilon}\right)^k\right) = \mathrm{O}\left(\left(\frac{1}{\epsilon}\right)^k\right).$$

To find R, let r be the number of iterations of the **while** loop in SSGR(). We have:

$$r = \mathrm{O}(\log_{1-\alpha}\left(\frac{1}{\epsilon}\right)) = \mathrm{O}(\log\left(\frac{1}{\epsilon}\right)).$$

We perform the operation $U_{ij} \leftarrow U_{ij} \cap (U_{ih} \oplus U_{hj})$ inside of three nested loops. Using s-analysis, each operation takes $\mathrm{O}(s \log s)$ time. Each loop executes k times. Therefore, $R = rk^3(s \log s) = \log\left(\frac{1}{\epsilon}\right) k^3 (s \log s)$.

The total running time is therefore:

$$Sk^3R = \mathrm{O}(\left(\frac{1}{\epsilon}\right)^k \log\left(\frac{1}{\epsilon}\right) k^6 s \log s).$$

■

8 Implementation and Experiments

This section describes our implementation of the approximate algorithm. We set α equal to 0.1 and ϵ to 0.5. Below, we focus on several important aspects of our implementation. Again, we define $U_{0i} = V_i$, $1 \leq i \leq k$. Section 8.1 gives running times for a variety of containment examples. Section 8.2 compares three heuristics for selecting a shrinking point within GREEDY-EVALUATE(). Section 8.3 evaluates the effectiveness of applying the partial characterization. Section 8.4 discusses the effectiveness of our sized-based subdivision method. Section 8.5 discusses the need for *closed* polygon operations. Section 8.6 compares running times for this algorithm with Milenkovic's exact 3CN algorithm from Part II of this paper. Section 8.7 compares running times for our approximate algorithm with Milenkovic's exact kNN algorithm from Part II.

C++ code for the approximate kNN algorithm presented in this paper and for the exact 2CN, 3CN and kNN algorithms of Part II of this paper has been licensed to a CAD firm[7] in the apparel manufacturing industry.

8.1 Performance

Tables 1 through 3 contain some examples of running times for our algorithm for a variety of problems from apparel manufacturing ($3 \leq k \leq 10$). Typical values of m and n are in the ranges $4 \leq m \leq 100$ and $100 \leq n \leq 300$. All running times given in this section are to the nearest second on a 28 MHz SPARCstation[8], unless otherwise noted. The setup time to calculate V_i polygons is not included in the running time; it is typically less than one second. The time to calculate U_{ij} polygons is included in the running time. Table entries for infeasible problems are identified by "*". All infeasible examples are ones for which each pair of polygons fits into the container, and the area of the container is larger than the total area of the polygons. Our restriction process enables the algorithm to often detect infeasibility very quickly.

Experiment#	1	2	3	4	5	6	7	8	9	10
Type	3CC	3CC*	3CN	3CN	3CN	3CN*	3CN*	3NC	3NN	3NN
Time	1	0	1	2	4	0	27	1	5	6

Table 1: Approximate algorithm: running time for $k = 3$. Time is in seconds on a 28 MHz SPARCstation. Infeasibility is denoted by "*".

Experiment#	1	2	3	4	5	6	7	8	9	10
Type	4CN	4CN	4CN	4CN	4CN	4CN*	4NN	4NN	4NN	4NN*
Time	5	19	30	43	192	3	90	111	111	10

Table 2: Approximate algorithm: running time for $k = 4$. Time is in seconds on a 28 MHz SPARCstation. Infeasibility is denoted by "*".

Experiment#	1	2	3	4	5	6	7
Type	5CC	5NN	5NN*	6NN	6NN	6NN*	10CC
Time	126	60	26	93	300	49	600

Table 3: Approximate algorithm: running time for $k > 5$. Time is in seconds on a 28 MHz SPARCstation. Infeasibility is denoted by "*".

[7]The licensing agreement is with Microdynamics, Inc. and its parent company, Gerber Garment Technology, Inc.

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

Figure 4 through Figure 7 show configurations achieved for four examples using our algorithm. The tightness of the layouts is typical. Figure 4 and Figure 5 are examples of type 3NN. Figure 6 is an example of type 4NN. Figure 7 is an example of type 5NN.

Table 4 below presents running times, in seconds, for the approximate algorithm on a different, small collection of examples for $5 \leq k \leq 7$. These tests are for a 50MHz SPARCstation. For each example, we also give the number of calls to GREEDY-EVALUATE(). Judging the approximate algorithm purely by its running time can be deceptive. Although the algorithm takes anywhere from 3 to 45 minutes in these examples, it is only calling GREEDY-EVALUATE() once or twice. Most of the running time is consumed by SSGR(), whose speed depends on the speed of our underlying polygon set operations. Our experiments on apparel data suggest that our current implementation of set operations allows the approximate algorithm to be practical for up to four items.

Experiment#	1	2	3	4	5	6
Type	5NN	5NN	5NN	6CN	6CN	7CC
Time	428	189	317	2540	2238	823
Calls	1	1	1	2	1	1

Table 4: Approximate algorithm: running time, in seconds, and number of calls to GREEDY-EVALUATE().

8.2 Comparing Heuristics for Selecting a Shrinking Point

We experimented with three different heuristics for selecting a shrinking point $u_{ij} \in U_{ij}$:

1. a random point;
2. a point which (approximately) maximizes the area of $U_{0i} \cap (U_{0j} + u_{ji})$;
3. a point which (approximately) "maximizes the convexity" of $U_{0i} \cup (U_{0j} + u_{ji})$.

Heuristics 2 and 3 use a two-dimensional grid to find an approximate maximum. For Heuristic 2, we overlay a grid on each component of U_{ij} and, at each grid point u_{ij}, measure the area of $U_{0i} \cap (U_{0j} + u_{ji})$. We choose a grid resolution for each component which gives ten grid points along the longest (x or y) dimension of the component. For each component, we find the average area and the maximum area. From the component with the highest average value, we select the point yielding the maximum value.

For Heuristic 3, we again overlay a grid on each component of U_{ij}. In this case, however, we measure the area of $\mathrm{CH}(U_{0i} \cup (U_{0j} + u_{ji})) \setminus (U_{0i} \cup (U_{0j} + u_{ji}))$. For each component, we find the average area and the minimum area. From the component with the lowest average value, we select the point yielding the minimum value.

The goal of our experiments was to see which of the three heuristics was able to solve the most containment examples without resorting to subdivision. We performed several experiments. Our early experiment with examples of type 3NN, using geometric restriction but not a steady-state loop, suggested that choices of u_{ij} leading to valid configurations often have regions of large intersection area between the translated U_{0i}s. For that version of the algorithm we tested 30 examples of types 4CN and 4NN, using Heuristic 2. A solution was found without subdividing in 66% of the 30 examples.

Once we had the steady-state restriction process in place, we ran another experiment using 30 test examples for $4 \leq k \leq 6$. The results appear in Table 5.

Table 5 shows that maximizing the area of intersection (Heuristic 2) performs best, solving 60% of the problems without subdivision. Maximizing convexity (Heuristic 3) solved 36% of the problems, and randomization (Heuristic 1) only 20%. Although the intersection method usually outperforms the convexity method, there are cases in which convexity is able to find a solution without subdivision when intersection cannot. For example, Figure 8a shows three U_{0i} polygons which do not have a triple intersection when U_{ij} is chosen by the intersection method. When the convexity method is used for the same example, Figure 8b shows that a triple intersection results.

Experiment#	1	2	3	4	5	6	7	Totals
Type	4CC	4CN	4NN	5CC	5CN	5NN	6NN	
Heuristic 1: Random	2/6	0/1	4/13	0/2	0/2	0/5	0/1	6/30
Heuristic 2: Intersection	6/6	1/1	5/13	2/2	1/2	2/5	1/1	18/30
Heuristic 3: Convexity	2/6	0/1	5/13	0/2	1/2	3/5	0/1	11/30

Table 5: Comparison of 3 heuristics for selecting a shrinking point: 1) random selection, 2) maximizing the intersection of translated U_{0i}'s, and 3) "maximizing the convexity" of the union of translated U_{0i}'s. For each heuristic, the number of examples solved by the approximation algorithm without subdivision is shown.

The grid analysis method for finding an approximate maximum for intersection appears to be effective. However, many set operations are performed in order to choose a single u_{ij} shrinking point. We would like a method which is as effective but is faster (see Section 9.2). Our initial experiments with inexpensive methods for encouraging large intersection area (such as aligning centroids and bounding box corners (see Section 5.2.2) suggest that these faster methods are not as effective as grid analysis.

8.3 Effectiveness of Partial Characterization

Here we evaluate the effectiveness of applying a partial containment characterization. That is, we compare the effect of using k subproblems versus not using subproblems. In our experiment, we maximize the intersection area of translated U_{ij}s (Heuristic 2 from Section 8.6). Our examples use $k = 4$. We compare the *sequential* cost, in terms of the number of calls to GREEDY-EVALUATE(), for infeasible as well as feasible examples. For 46 examples, the subproblem approach calls GREEDY-EVALUATE() a total of 162 times, whereas the non-subproblem approach calls GREEDY-EVALUATE() 345 times. These results suggest that, for $k = 4$, using the partial characterization decreases the sequential running time of the algorithm.

When all the items are identical, the k subproblems generated by the characterization are all identical, and so we need only solve one of them. Our experiments include 21 inputs for which all items are identical. The total number of calls to GREEDY-EVALUATE() in the subproblem case is 115, whereas the number of calls in the non-subproblem case is 174. The remaining 25 problems are for non-identical items. For the non-identical item problems, the total number of calls in the subproblem case is 47, whereas the number in the non-subproblem case is 171. Thus, for identical as well as non-identical item problems, the characterization produces fewer calls to GREEDY-EVALUATE() for $k = 4$ in our experiments.

Assessing the cost-effectiveness of applying the partial characterization to the sequential algorithm for $k > 4$ is a subject for future work. For problems with all identical items, only one subproblem must be examined. With an efficient implementation of closed operations on line segments and points, the use of the partial characterization should decrease the average running time of the algorithm for every value of k. For problems in which items are not all identical, it is not clear for $k > 4$ whether restricting one U_{ij} to its boundary in each subproblem will reduce the number of calls to GREEDY-EVALUATE() enough so that the total number of calls to GREEDY-EVALUATE() over all k subproblems will be smaller than the number of calls to GREEDY-EVALUATE() in the non-subproblem case. This will depend on how well SSGR() restricts the U_{ij}s in the non-subproblem case as k increases; if SSGR() is "weaker" for large k then the subproblem approach might be preferable. Daniels [6] examines some conditions which influence the effectiveness of SSGR(). Although it is possible to construct problems for which SSGR() greatly restricts the U_{ij}s for large k, we have found, for our apparel manufacturing data, that such problems rarely occur in practice.

Assessing the cost-effectiveness of the partial characterization for the *parallel* version of the approximate algorithm is also a subject for future work. One way in which course-grained parallelism can be applied to our algorithm is to let $\mathcal{U}$ represent the current state, or *hypothesis*, for the containment algorithm. A queue of hypotheses can be maintained. Each hypothesis represents an independent containment subproblem which is created via subdivision. Whenever a processor is available, it operates on the next hypothesis in the queue. The subproblems generated by the characterization are also independent from each other, and so the hypotheses they generate are independent and can be added to the queue. Note that applying the characterization creates subproblems which are *all* available at the start of the algorithm and therefore the characterization can make better initial use of the parallel processors, instead of leaving some idle.

8.4 Subdivision

The size-based method of Section 6 which employs vertical and horizontal cuts has been implemented. For our applications, CAD vendors round polygon coordinates to integers, where one unit is 0.01 inch, so an ϵ of 0.5 units is sufficiently small. It is rare for the algorithm to subdivide down to tolerance. Often, no subdivision is needed to find a valid configuration (see Section 8.2). Subdivision is rarely required for infeasible situations, because our restriction process often detects infeasibility early in the algorithm.

8.5 Closed Operators

Here we discuss an important design issue related to the type of set operations. Consider a general polygon, i.e. a collection of polygonal components, where each component is connected.

Definition 8.1 *A general polygon P is* **regularized** *if it is equal to the closure of its interior.*

Figure 9a depicts a regularization of the non-regularized general polygon of Figure 9b. Regularized polygons are not sufficient for our application. Using only regularized set operations can cause the algorithm to incorrectly conclude during a restriction that no solution exists. This can occur if the intersection of two polygons is either a line segment or a point. In fact, this situation can occur often for tight fits.

A non-regularized polygon does allow isolated points and line segments (see Figure 9b). Unfortunately, it also allows isolated points or line segments to be *removed* from its interior. Thus, a non-regularized polygon need not be *closed*, i.e. equal to its closure - its boundary plus its set of limit points. Since the configuration space we operate on is closed, we must deal with closed sets. Union and intersection operations on closed sets are defined in the normal way, because the union and intersection of two closed sets is closed. However, the complement of a closed set is open, so we take the closure of the complement of a closed polygon.

Correct implementation of the algorithm requires the use of robust, closed set operations. The set operations must be implemented robustly in order to minimize error accumulation. It is not practical to use "exact" arithmetic because outputs repeatedly become inputs (in SSGR(), for example). This means that the bit-complexity of exact coordinates would grow without bound. For our first implementation of SSGR(), we used robust intersection, union, and complement operators which handled only regularized polygons. These operators are based on geometric rounding [16, 25, 19, 20, 21]. In order to create a set of closed operators, we extended our general polygon representation to include lists of line segments and points as well as lists of polygons. We implemented naive, non-robust union, intersection, and Minkowski sum operators for cases involving line segments or points. However, our underlying polygon operators were still regularized. We are currently redesigning these operators so we will have a complete set of robust operators for closed polygons.

8.6 Comparison with Exact 3CN Algorithm

Table 6 compares running time for Milenkovic's exact 3CN algorithm from Part II of this paper with our approximate algorithm on examples of type 3CN. Infeasible examples are identified by "*". The exact algorithm tests separating line orientations between pairs of convex polygons. Because both algorithms are so fast, timing comparisons are based on 100 calls to each. For feasible examples, the exact algorithm performs about seven times faster than the approximate algorithm, on average. For infeasible examples, the exact algorithm is only about twice as fast, on average. This is no accident. When the exact algorithm was first designed, the approximate algorithm's powerful restriction process gave it a considerable advantage over the exact algorithm for infeasible examples. The exact algorithm was redesigned with the goal of detecting infeasibility quickly in mind. The major part of the redesign was a multiple resolution method for evaluating sets of separating lines.

8.7 Comparison with Exact kNN Algorithm

Table 7 compares running times for our approximate algorithm with Milenkovic's exact kNN algorithm from Part II. Infeasible examples are denoted by "*". Running times are expressed in seconds on a 50 MHz

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

SPARCstation. In most of the feasible examples the exact kNN algorithm finds a solution faster than the approximate algorithm. However, the approximate algorithm is faster at detecting infeasibility, which is important in practice.

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

As discussed in Section 8.1, the speed of the approximate algorithm is currently limited by the speed of SSGR(). Even in cases in which the approximate algorithm appears to be much slower than the exact algorithm, the number of calls to GREEDY-EVALUATE() is small. The approximate and exact algorithms have basically the same evaluate/subdivide structure, and the number of evaluations performed by the approximate algorithm is usually much smaller than the number of evaluations for the exact algorithm. For example, in one 4NN feasible problem, the approximate algorithm takes 202 seconds and performs only one evaluation, whereas the exact algorithm takes only 16 seconds but performs 19 evaluations. Increasing the speed of SSGR() should make the approximate algorithm practical for larger numbers of items. The exact algorithm is limited by the large number of evaluations it requires. For feasible problems, it is practical for up to four items[9] and can be practical for as many as seven items if both the items and container are convex. It is impractical for four or more items if the problem is infeasible.

9 Conclusion and Future Work

9.1 Conclusion

This paper has presented an approximate algorithm for kNN. Our experiments on data from the apparel industry suggest that this algorithm is currently practical for up to four polygons. As mentioned in Section 8, this software has been licensed to a CAD firm in apparel manufacturing. Our containment characterization allows us to solve containment by solving a set of k boundary-restricted subproblems. These problems can be solved in parallel. Even solving them sequentially can be beneficial, as we show in the case of $k = 4$. Our steady-state restriction process is a powerful technique for detecting infeasibility, which is important in practice. Our greedy evaluation technique often finds feasible solutions without resorting to subdivision. Most of the running time of our algorithm is used by our steady-state restriction process. The speed of the restrictions depends on the speed of our underlying polygon set operations. Improvements in the speed of our polygon set operations should directly improve the speed of the algorithm.

We have compared the performance of our approximate algorithm to Milenkovic's exact 3CN and exact kNN algorithms, which appear in Part II of this paper, *Exact Algorithms*. Initially, the success of the

[9]Our experiments use data from apparel manufacturing and $4 \leq m \leq 100$ and $100 \leq n \leq 300$.

restriction process of the approximate algorithm led to a redesign of the 3CN algorithm so that it could perform well on infeasible problems. As a result, the 3CN algorithm is faster on examples of type 3CN than the approximate algorithm. The exact kNN algorithm, which uses linear programming for its evaluation step, is limited by the large number of evaluations it requires. It typically performs many more evaluations than the approximate algorithm. The exact kNN algorithm is currently practical for up to four items if the problem is feasible. It is often faster than the approximate algorithm on feasible examples, in spite of the larger number of evaluations, because its linear programming evaluation is much faster than a call to our restriction-based GREEDY-EVALUATE().

The trim placement problem which motivated our containment research frequently generates containment problems for which $k \leq 3$, and many of these are infeasible. The approximate algorithm, the exact kNN algorithm, and Milenkovic's exact algorithms for convex polygons all perform well for $k \leq 3$ for feasible examples. The approximate algorithm and Milenkovic's exact algorithms for convex polygons perform well for $k \leq 3$ for infeasible examples. The approximate algorithm and Milenkovic's 2CN and 3CN algorithms for convex polygons are currently practical choices for trim placement.

9.2 Future Work

There are several directions for future work in containment. One direction is to increase the dimensionality of the containment problems we address. For instance, many important three-dimensional packing and layout problems could be solved using three-dimensional containment algorithms. Another way to increase the dimensionality of the containment problem is to allow rotation. Rotational containment adds an extra degree of freedom to each item by allowing it to rotate continuously within a range of angles. Many layout problems allow continuous rotation of items, particularly problems related to sheet metal work.

A second direction for future work involves improving our algorithms for translational containment so that larger numbers of items may be placed in a practical amount of time. One way to obtain a speed-up is to apply course-grained parallelism. Other ways to increase the speed of the algorithm include approximating the polygons and developing faster polygon set operations. We will continue to work on a fast, robust, set of closed polygon operators. These are crucial for fast, steady-state, geometric restriction.

We plan to evaluate the cost-effectiveness of using a partial characterization to produce k subproblems for $k > 4$. Our current experiments suggest that it is beneficial for $k = 4$, even when the subproblems are solved sequentially. We also wish to investigate the cost-effectiveness of the partial characterization within a parallel version of the algorithm.

We plan to develop more restrictions for translational containment. We will continue to explore the question of which restriction operations allow s, the maximum number of vertices in a restricted polygon, to remain finite even when an infinite number of operations is applied.

The following problems are associated with our greedy evaluation method:

- Given polygons Q_i and Q_j, is there a translation t for which $(Q_i + t) \cup Q_j$ is convex?
- Find t which minimizes the area of $\mathrm{CH}(Q_j \cup (Q_i + t)) \setminus (Q_j \cup (Q_i + t))$.
- Given a list of polygons $\mathcal{Q} = \langle Q_1, Q_2, \ldots, Q_k \rangle$, how fast can we find a translation t_i for each Q_i in $\mathcal{Q}$ such that the area of $\cap_i (Q_i + t_i)$ is maximized?

The approximate algorithm presented here and the exact kNN algorithm of Part II have complementary strengths. We are working on a hybrid containment algorithm which combines these strengths.

References

[1] H. Alt, B. Behrends, and J. Blömer. Approximate matching of polygonal shapes. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry*, pages 186–193, 1991.

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

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

[4] B. Baker, S. Fortune, and S. Mahaney. Polygon Containment under Translation. *Journal of Algorithms*, 7:532–548, 1986.

[5] B. Chazelle. The Polygon Containment Problem. In Preparata, editor, *Advances in Computing Research, Volume 1: Computational Geometry*, pages 1–33. JAI Press, Inc., Greenwich, Connecticut, 1983.

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

[13] A. Har-Peled, T. Chan, B. Aronov, D. Halperin, and J. Snoeyink. The Complexity of a Single Face of a Minkowski Sum. In *Proceedings of the 7th Canadian Conference on Computational Geometry*, Laval University, Quebec, 1995.

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

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

[16] 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.

[17] Z. Li and V. J. Milenkovic. A Compaction Algorithm for Non-Convex Polygons and Its Application. In *Proceedings of the 9th Annual ACM Symposium on Computational Geometry*, pages 153–162, May 1993.

[18] Z. Li and V. J. Milenkovic. The Complexity of the Compaction Problem. In *Proceedings of the 5th Canadian Conference on Computational Geometry, A. Lubiw and J. Urrutia, Editors*, pages 473–478, August 1993.

[19] V. J. Milenkovic. Applications of geometric rounding to polygon and polyhedron modeling. In *ARO/MSI Computational Geometry Workshop*, 1993.

[20] V. J. Milenkovic. Robust polygon modelling. *Computer-Aided Design*, 25(9):546–566, September 1993.

[21] V. J. Milenkovic. Practical Methods for Set Operations on Polygons using Exact Arithmetic. In *Proceedings of the 7th Canadian Conference on Computational Geometry*, pages 55–60, August 1995.

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

[23] 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.

[24] V. J. Milenkovic and Z. Li. A Compaction Algorithm for Nonconvex Polygons and Its Application. *European Journal of Operations Research, to appear.*

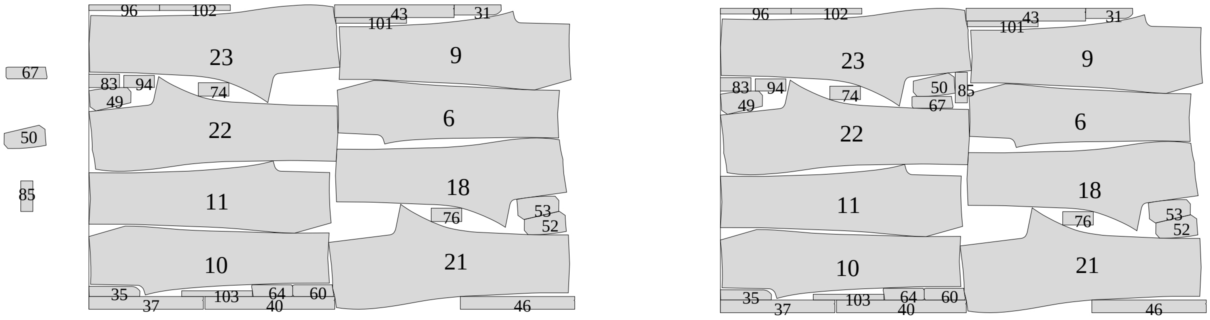
[25] V. J. Milenkovic and L. R. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proceedings of the Sixth Annual Symposium on Computational Geometry*, pages 244–252, 1989.

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

[27] J. Mitchell. Comment at MSI Workshop, October 1994.

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

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



(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.

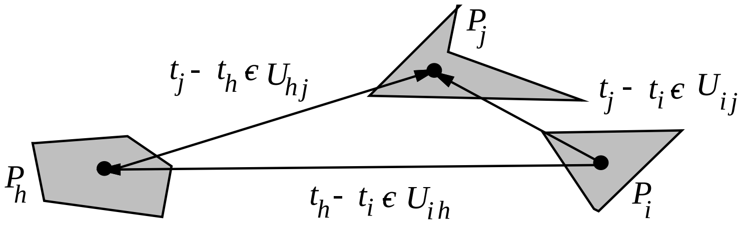
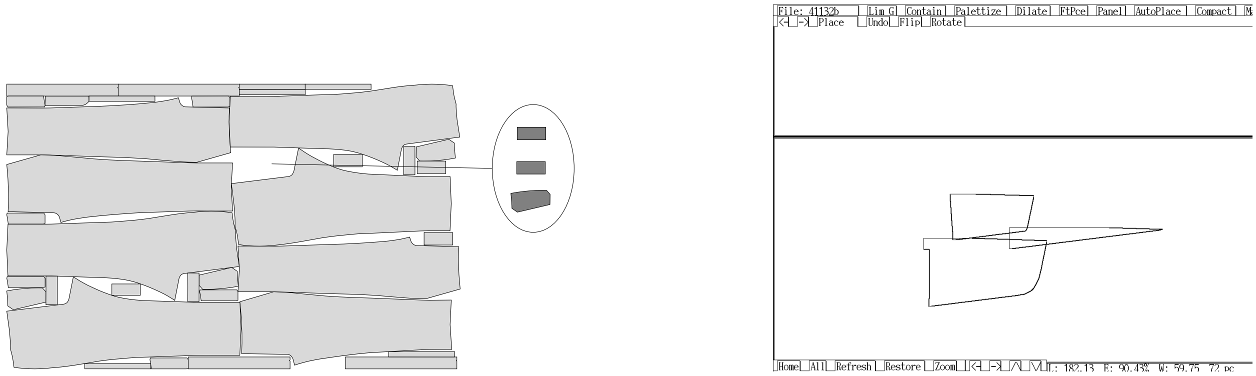


Figure 2: A valid restriction.



(a) (b)

Figure 3: (a) 3CN example which does not yield a common intersection of translated U_{0i}s. (b) A zoomed-in screen dump of the three translated U_{0i}s for this example. No triple intersection of translated U_{0i}s.

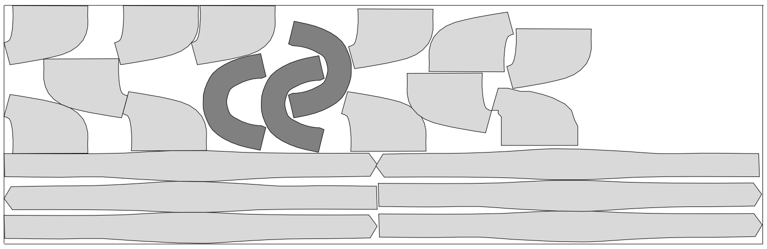


Figure 4: Example of type 3NN: time = 6 seconds.

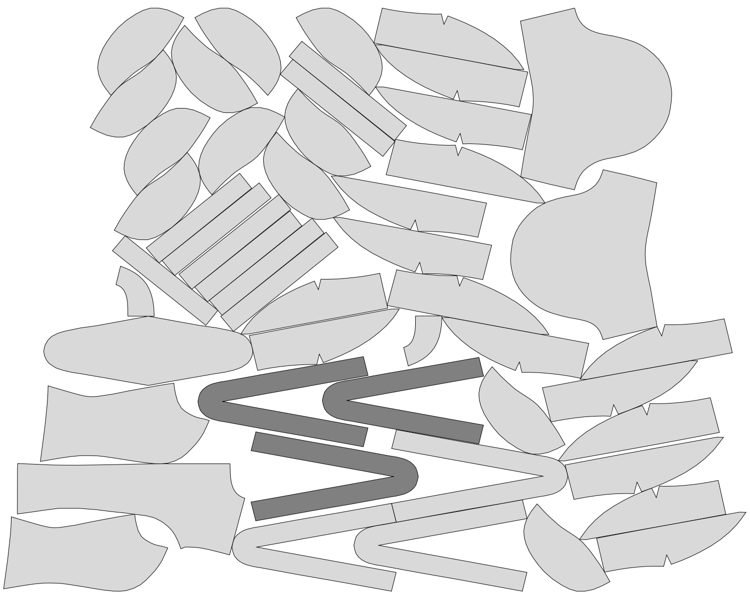


Figure 5: Example of type 3NN: time = 5 seconds.

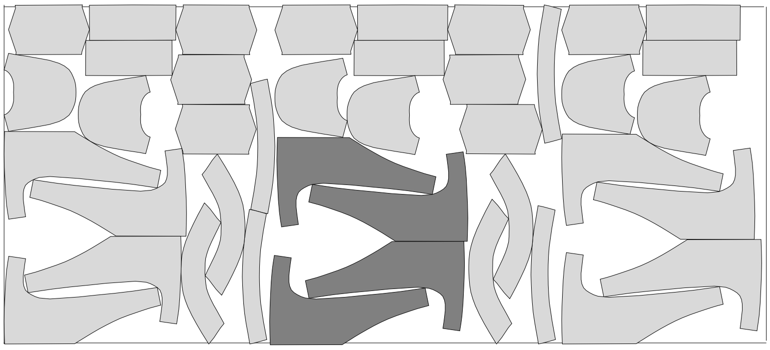


Figure 6: Example of type 4NN: time = 90 seconds.

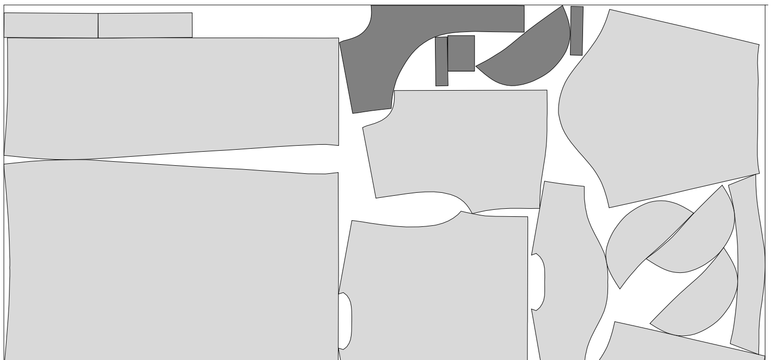


Figure 7: Example of type 5NN: time = 60 seconds.

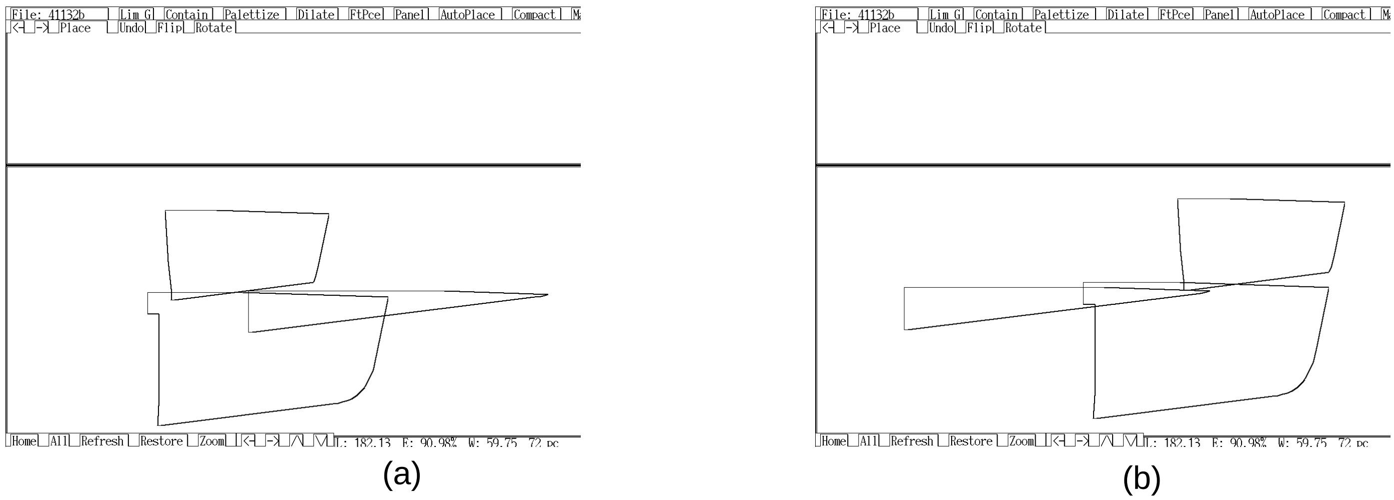


Figure 8: (a) No common intersection of translated U_{0i}s after maximizing intersection. (b) Common intersection after "maximizing convexity" of the union.

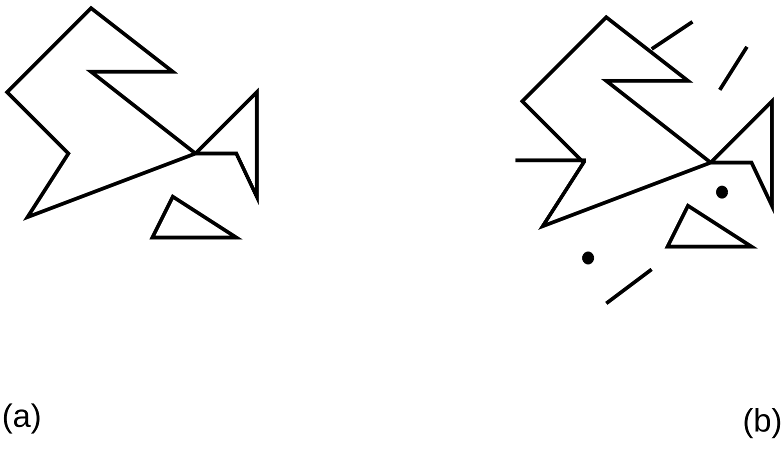


Figure 9: Regularization example. (a) Regularized, general polygon. (b) Nonregularized, closed, general polygon.