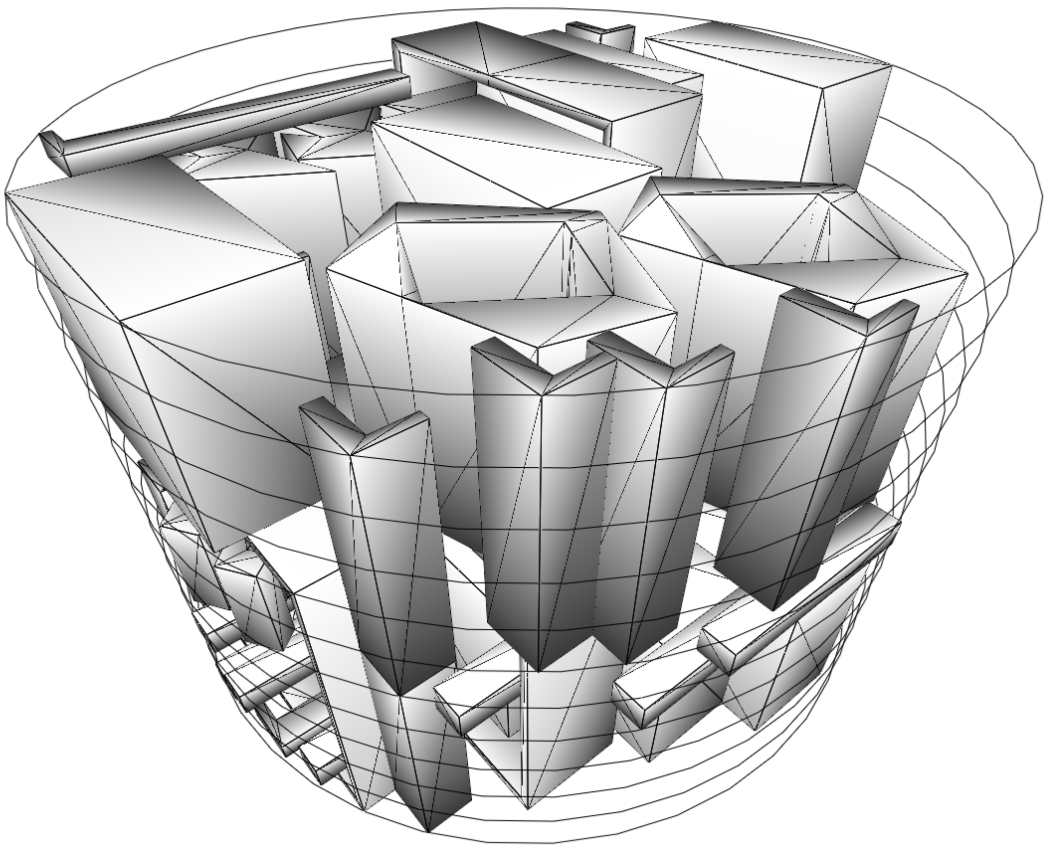
Fast Neighborhood Search for the Nesting Problem[1]



Benny Kjær Nielsen and Allan Odgaard
{benny, duff}@diku.dk

February 14, 2003

[1]Technical Report no. 03/02, DIKU, University of Copenhagen, DK-2100 Copenhagen Ø, Denmark.

Contents

Chapter 1

Introduction

The main subject of this thesis is the so-called *nesting* problem, which (in short) is the problem of packing arbitrary two-dimensional shapes within the boundaries of some container. The objective can vary e.g. minimizing the size of a rectangular container or maximizing the number of shapes in the container, but the core problem is to pack the shapes tightly without any overlap. An example of a tight packing is shown in Figure 1.1.

A substantial amount of literature has been written about this problem, where most of it has been written within the past decade. A survey of the existing literature is given in Chapter 2 along a detailed description of different problem types and various geometric approaches to these problems. At the end of the chapter a three-dimensional variant of the problem is described and the more limited amount of literature in this area is also discussed.

The rest of the thesis is focused on solving the nesting problem with a meta heuristic method called *Guided Local Search* (GLS). It is a continuation of the work presented in a written report (in Danish) by Jens Egeblad and ourselves [28], which again is a continuation of the work presented in an article by Færø et al. [32]. Throughout this thesis we will often refer to the work done by Jens Egeblad and ourselves (Egeblad et al. [28]).

Chapter 3 presents GLS and some other issues regarding the basics of our solution method. Most importantly the neighborhood for the local search is presented — it is a fast search of this neighborhood which is the major strength of our approach to the nesting problem.

Færø et al. successfully applied the GLS meta heuristic to the rectangular bin packing problem in both two and three dimensions. The speed of their approach was especially due to a simple and very fast translation algorithm which could find the optimal (minimum overlap) axis-aligned placement of a given box. Egeblad et al. [28] realized that a similar algorithm was possible for translating arbitrary polygons. However, they did not prove the correctness of the algorithm.

A major part of this thesis (Chapter 4) is dedicated to such a proof. The proof is stated in general terms and translation and polygons are introduced at a very late stage. By doing this it is simultaneously shown that the algorithm could be useful for other transformations or other shapes. At the end of the chapter some additional work is done to adapt the general scheme to an algorithm for rotation of polygons.

Although not explicitly proven it is easy to see that the fast translation and rotation algorithms are also possible for three dimensions. A description of how to do this in practice for translation is given separately in Chapter 6.

The two-dimensional nesting problem appears in a range of different industries and quite

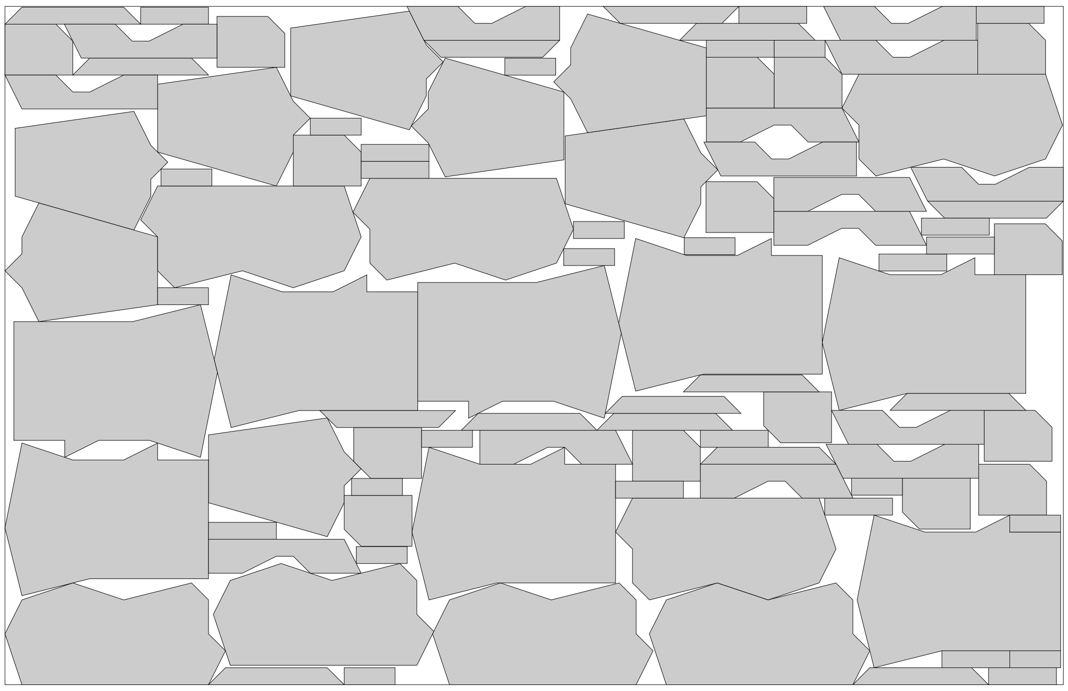


Figure 1.1: A tight packing using 86.5% of the available area (generated by 2DNest developed for this thesis).

often it is not stated in its pure form, but includes a series of extra constraints. Chapter 5 discusses how a wide range of such constraints can be handled by our solution method quite easily. A subset of these constraints are also handled by our implementation which is described in more detail in Chapter 7. This chapter also includes a description of our 3D nesting implementation. Experiments with the implementations are described in Chapter 8 and most of these are focused on the optimization of various parameters influencing the efficiency of the implementation. The best parameters are then used to perform a series of benchmarks to evaluate the performance in relation to existing published results.

The quality of solutions for two-dimensional nesting are in general better than those reported by Egeblad et al. which again is not matched by any benchmarks in the academic literature and the amount of supported constraints in our implementation also exceeds most of what we have seen in competing methods. Comparisons are also done with a commercial solver which is considerably faster than ours, but we are close in quality. Only few results exist for three-dimensional nesting which can be used for comparisons and it was no problem to outperform the results found in the literature.

It should be noted that parts of Chapter 4 are inspired by a draft proof by Jens Egeblad for the special case of translation of polygons. This especially concerns the formulation of Theorem 2 and the formulation and proof of Theorem 3. His work has been a great help.

Chapter 2

The Nesting Problem

2.1 Problem definitions

The term *nesting* has been used to describe a wide variety of two-dimensional cutting and packing problems. They all involve a non-overlapping placement of a set of irregular two-dimensional shapes within some region of two-dimensional space, but the objective can vary. Most problems can be categorized as follows (also see the illustration in Figure 2.1):

- *Decision problem.* Decide whether a set of shapes fit within a given region.
- *Knapsack problem.* Given a set of shapes and a region, find a placement of a subset of shapes that maximizes the utilization (area covered) of the region.
- *Bin packing problem.* Given a set of shapes and a set of regions, minimize the number of regions needed to place all shapes.
- *Strip packing problem.* Given a set of shapes and a width W, minimize the length of a rectangular region with width W such that all shapes are contained in the region.

An interesting problem type, which is not included in the above list, is a variant of the strip packing problem that deals with repeated patterns i.e. the packing layout is going to be reused on some material repeatedly in 1 or 2 directions. With one direction of repetition this problem can be interpreted as a nesting problem on the outside of a cylinder, where the objective is to minimize the radius. We will get back to this variation in Chapter 5. We will denote this problem the *repeated pattern problem.*

As noted the region used for strip packing is rectangular and a typical example would be a cloth-strip in the textile industry. All other regions can have any shape. It could be animal hides in the leather industry, rectangular plates in the metal industry or tree boards in the furniture industry.

Note that all of the problems have one-dimensional counterparts, but the one-dimensional decision and strip packing problems are trivial to solve.

In this thesis the focus is on the decision problem, but this is not a major limitation. Heuristic solution methods for bin/knapsack/strip packing can easily be devised when given a (heuristic) solution method for the decision problem, e.g. by fixing the strip length or the number of bins and then solve the decision problem and if it is a success then the strip can be shortened or the number of bins decreased and vice versa if it fails. When to do what is not

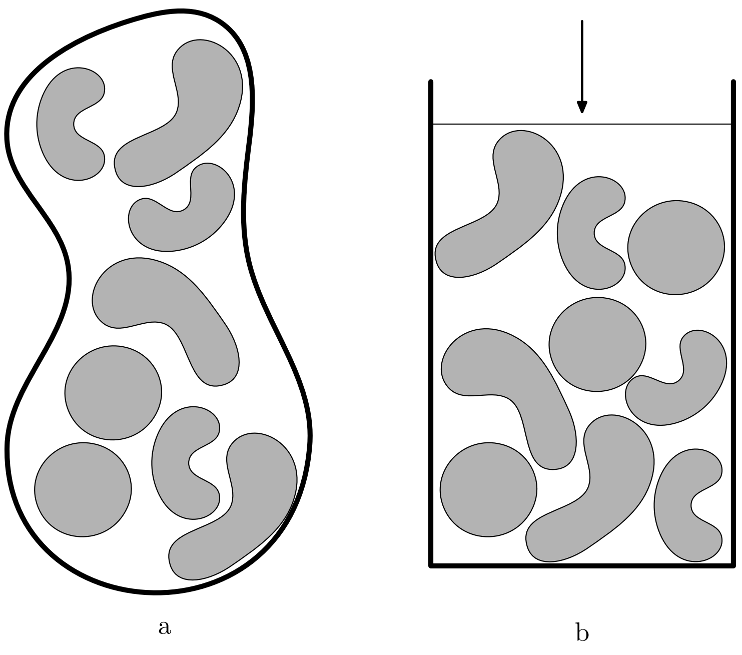


Figure 2.1: a) Most variants of the nesting problem is the problem of packing shapes within some region(s) without overlap (decision, knapsack and bin packing). b) The strip packing variant asks for a minimization of the length of a rectangular region.

a trivial problem and we will briefly get back to this in Section 3.4. It is important to note that the various packing problems can also be solved in more direct ways, which could be more efficient. Nevertheless, our focus is mainly on the decision problem.

In industrial settings a multitude of additional constraints are very often necessary, e.g. the shapes or regions can have different quality zones or even holes (animal hides). Historically, the clothing industry has had special attention. But even though this industry introduces a multitude of possible extra constraints to the problem, these constraints are often not included in the published solution methods. An exception is Lengauer et al. [35, 37] who describe and handle a long range of the possible additional constraints posed in the leather and the textile industry. But most of the existing literature does not handle any additional constraints. We will not consider them any further in this chapter and instead a more detailed discussion of various constraints (in relation to our solution method) is postponed to Chapter 5.

As indicated above the nesting problem occurs in a number of industries and it seems to have gotten just as many names. In the clothing industry it is usually called *marker making*, while the metal industry prefers to call it *blank nesting* or simply nesting. There is no consensus in the existing literature either. Some call the shapes irregular, others call them non-convex. In a theoretical context the problem is most often called the *two-dimensional irregular cutting stock problem*. This is quite verbose and therefore it is quite natural that shorter variants have been preferred such as *polygon placement*, *polygon containment* and (irregular) nesting. These names also indicate that the irregular shapes are almost always restricted to being polygons. A quick look at the reference list in this thesis emphasizes the diversity in the naming of this problem.

In relation to Dyckhoff's [25] typology we are dealing with problems of types 2/V/OID/M — many small differently shaped figures packed in one or several identical/different large shape(s) in 2 dimensions.

Our general choice of wording follows below. None of the words imply any restrictions on the shapes involved.

- *Nesting.* A short name for the problem, which is often used in the existing literature.
- *Stencil*[1]. A domain specific name for the pieces/shapes/polygons to be packed.
- *Material.* A general word for the packing region which can be used to describe garments, metal plates, wood, glass and more.
- *Placement.* The positioning of a set of stencils on the material. A *legal placement* is a placement without any overlapping stencils and all stencils placed within the limits of the material.

Not surprisingly, the nesting problem is $\mathcal{NP}$-hard. Most existing articles state this fact, but only few have a reference. Some refer to Fowler et al. [31] who specify a very constrained variant, BOX-PACK, as the following problem: Determine whether a given set of identical boxes (integer squares) can be placed (without rotation) at integer coordinates inside a region in the plane (not necessarily connected) without overlapping. They prove that BOX-PACK is $\mathcal{NP}$-complete by making a polynomial-time reduction of 3-SAT which is the problem of determining whether a Boolean formula in conjunctive normal form with 3 literals per clause is satisfiable.

Note that the region defined in the BOX-PACK problem is not required to be connected. This means that the region could be a set of regions thus the problem also covers the bin packing problem. Also note that the 1-dimensional variant of BOX-PACK is a trivial problem to solve. This is not true for less constrained variants of the nesting problem.

The majority of articles related to nesting only handle rectangular materials and this is a constraint which is not included in the BOX-PACK problem. If it was then the problem (packing identical squares in a rectangle) would no longer be $\mathcal{NP}$-hard. It would have a trivial solution. Some authors seem to have missed this point (e.g. [35, 36]).

To remedy this situation we define a new problem, BOX-PACK-2: Determine whether a given set of integer rectangles (not necessarily identical) can be placed (without rotation) inside a rectangular region without overlapping. Inspired by the proof in [49]:

Theorem 1. *BOX-PACK-2 is $\mathcal{NP}$-complete.*

Proof. We are going to make a polynomial-time reduction of 1-dimensional bin packing [33].

The bin packing problem can be stated as follows: Given a finite set of integers X, a bin capacity B and an integer K, determine whether we can partition X into disjoint sets $X_1, ..., X_K$ such that the sum of integers in each X_i is less than or equal to B.

Now replace each integer $x \in X$ with a rectangle of height 1 and width x and use a given algorithm to solve BOX-PACK-2 to place these rectangles in a rectangle of width B and height

[1]Stencil, n. [Probably from OF. estincelle spangle, spark, F. ['e]tincelle spark, L. scintilla. See Scintillate, and cf. Tinsel.] A thin plate of metal, leather, or other material, used in painting, marking, etc. The pattern is cut out of the plate, which is then laid flat on the surface to be marked, and the color brushed over it. Called also stencil plate. Source: Webster's Revised Unabridged Dictionary, 1996, 1998 MICRA, Inc.

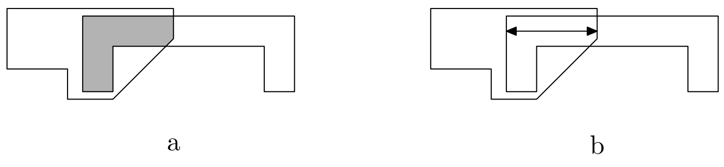


Figure 2.2: The degree of overlap can be measured in various ways. Here are two examples: a) The precise area of the overlap. b) The horizontal intersection depth.

K. By post processing this solution (e.g. sliding the rectangles down and to the left) we will get a solution to the bin packing problem by using the width of the rectangles in each row as the integers in each bin.

Clearly, BOX-PACK-2 is also in $\mathcal{NP}$ and thereby $\mathcal{NP}$-complete. $\square$

2.2 Geometric aspects

Before we proceed with a description and discussion of some of the existing solution methods, we will describe some of the different approaches to the geometric aspects of the nesting problem. In this section only polygons are considered since most solution methods can only handle these.

The basic requirement is to produce solutions with no overlap between polygons. This means that either the polygons must be placed without creating overlap or it should be possible to detect and eventually remove any overlap occurring in the solution process.

If overlap is allowed as part of the solution process then there is a lot of diversity in the geometric approaches to the problem. Given two polygons P and Q, one or more of the following problems need to be handled.

- Do P and Q intersect?
- If P and Q intersect, how much do they intersect?
- If P and Q do not intersect, how far are they apart?

The question of the size of an overlap is handled in very different ways. The most natural way is to measure the exact area of the overlap (see Figure 2.2a). This can be an expensive calculation and thus quite a few alternatives have been tried. Oliveira and Ferreira [50] used the area of the smallest rectangle containing the intersection, but this has probably not saved much time since the hard part is to find the intersection points and *not* to do the calculation of the area. It has also been suggested to use the *intersection depth* (see Figure 2.2b), which makes sense since the depth of the intersection also expresses how much one of the polygons must be moved to avoid the overlap. Dobkin et al. [21] describe an algorithm to get the minimum intersection depth i.e. they also find the direction that results in the smallest possible intersection depth. Unfortunately they require one of the polygons to be convex.

The distance between two polygons, which are *not* intersecting, can be useful if they are to be moved closer together. But most algorithms are more focused on the relations between polygons and empty areas. Dickinson and Knopf [19] introduce a *moment based metric* for both 2D and 3D. This metric is based on evaluating the compactness of the remaining free

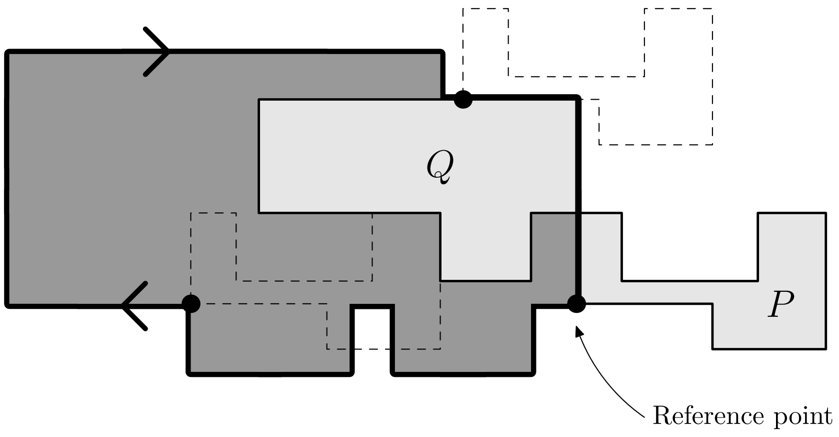


Figure 2.3: Example of the *No-Fit-Polygon* (thick border) of stencil P in relation to stencil Q. The reference point of P is not allowed inside the NFP if overlap is to be avoided.

space in an unfinished placement. They use this to implement a sequential packing algorithm in 3D [18].

Some theoretical work has also been done by Stoyan et al. [54] on defining a so-called Φ-function. Such a function can differentiate between three states of polygon interference; intersection, disjunction and touching. But again, they only handle convex polygons.

Solution methods which do not involve any overlapping polygons at any time in the solution process almost always use the concept of the *No-Fit-Polygon* (NFP). It is often claimed to have been introduced by Art [4], which is not entirely correct. The NFP is a polygon which describes the legal/illegal placements of one polygon in relation to another polygon. Art introduced and used the *envelope*, which is a polygon that describes the legal/illegal placements of a polygon in relation to the packing done so far. The calculations are basically the same and they are strongly connected with the so-called Minkowski sum.

Given two polygons P and Q the construction of the NFP of P in relation to Q can be found in the following way: Choose a reference point for P. Slide P around Q as close as possible without intersecting. The trace of the reference point is the contour of the NFP. An example can be seen in Figure 2.3. To determine whether P and Q intersect it is only necessary to determine whether the reference point of P is inside or outside their NFP. But the real benefit of the NFP is when the polygons are to be placed closely together. This can be done by placing the reference point of P at one of the edges of the NFP. If P and Q have s and t edges, respectively, then the number of edges in their NFP will be in the order of $O(s^2t^2)$ [5].

The NFP has one major weakness. It has to be calculated for all pairs of polygons. If the polygons are not allowed to be rotated this is not a big problem since it can be done in a preprocessing step in a reasonable time given that the number n of differently shaped polygons is not too large (requiring n^2 NFPs). But if a set of rotation angles and/or flipping are allowed then even more NFPs have to be calculated e.g. 4 rotation angles and flipping would require the calculation of $(4 \cdot 2 \cdot n)^2$ NFPs. It is still a quadratic expression, but even a small number of polygons would require a large number of NFPs e.g. 4 polygons would require 1024 NFPs. If free rotation was needed then an approximate solution method using a large number of rotation angles would not be a viable approach. Nevertheless NFPs are still a powerful tool for restricted problems.

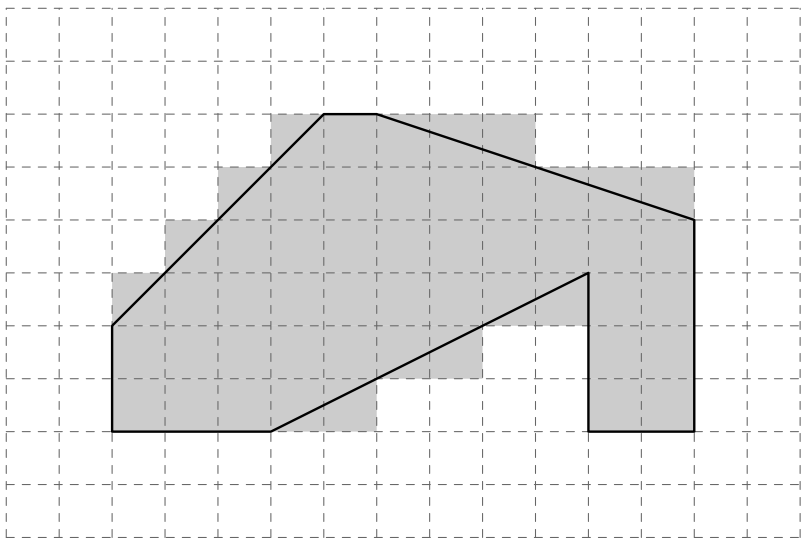


Figure 2.4: The raster model requires all stencils to be defined by a set of grid squares. The drawing above is an example of a polygon and its equivalent in a raster model.

A fundamentally different solution to the geometric problems is what some authors call *the raster model* [45, 50, 35]. This is a discrete model of the polygons (which then do not have to be polygons), created by introducing a grid of some size to represent the material i.e. each polygon covers some set of raster squares. The polygons can then be represented by matrices. An example of a polygon and its raster model equivalent is given in Figure 2.4.

Translations in the raster model are very simple while rotations are quite difficult. An overlap calculation takes at least linear time in the number of raster lines involved and this also clearly shows the weakness of this approach. A low granularity of the raster model provides fast calculations at the expense of little precision. A high granularity will result in very slow calculations. Comparisons with the polygonal model was done by Heckmann and Lengauer [35] and they concluded that the polygonal model was the better choice for their purposes.

With the exception of the calculation of the intersection area of overlapping polygons, none of the methods described above handle rotation efficiently. This is also reflected in the published articles which rarely handle more than steps of 180° or 90° rotation. Free rotation is usually handled in a brute-force discrete manner i.e. by calculating overlap for a large set of rotation angles and then select a minimum.

A more sophisticated approach for rotation has been published by Milenkovic [47]. He uses mathematical programming in a *branch-and-bound* context to solve very small *rotational containment problems* to near-optimality[2]. Very small is in the order of 2-3 polygons. Li and Milenkovic [43] have used mathematical programming to make precise local adjustments of a solution, so-called compaction and separation. This could be useful in combination with other nesting techniques.

[2]Given a set of polygons and a container, find rotations and translations that place the polygons in the container without any overlaps.

2.3 Existing solution methods

There exists a substantial amount of literature about two-dimensional cutting and packing problems, but most of it is focused on the rectangular variant. A recent survey has been written by Lodi et al. [44] about these restricted problems and this includes references to other surveys. In the following we are going to focus on articles about irregular nesting problems.

Most articles about the nesting problem has been written within the past decade. The reason for this is unlikely to be a lack of interest since the industrial uses are numerous as indicated in the section above. It is more likely that the necessary computing power for the needed amount of geometric computations simply was not available until the beginning of the 90's. Far more work has been done on the geometrically much easier problem of packing rectangles (or 3D boxes).

The strip packing problem is the problem most often handled. In the following it is implicitly assumed that the articles handle this problem, but most of the methods described could easily be adapted to some of the other problem types. When other problems than strip packing are handled it will be explicitly noted.

Lower bounds have not received much attention either[3] — with the exception of recent work by Heckmann and Lengauer [36]. Unfortunately their method is not applicable for more than about 12 stencils. By selecting a subset of stencils it can also produce lower bounds for larger sets of stencils, but the quality of the bound will not be good unless there is only a small number of large stencils in the original set.

Milenkovic [48] describes an algorithm which can solve the *densest translation lattice packing* for a very small number of stencils — not more than 4. This is the problem which we have denoted the repeated pattern problem (in two dimensions).

Most articles are focused on heuristic methods. Basically, they can be divided into two groups. Those only considering legal placements in the solution process and those allowing overlap to occur during the solution process.

Several surveys have been written. Dowsland and Dowsland [22] focus explicitly on irregular nesting problems, while other surveys [26, 38] focus on a wider range of cutting and packing problems. The latter of these is strictly focused on the use of meta-heuristic methods. A detailed discussion of meta-heuristic algorithms applied to irregular nesting problems can be found in the introductory sections of Bennell and Dowsland [6]. Finally, a very extensive list of references can be found in Heistermann and Lengauer [37], although most of them are about very restricted problems — e.g. rectangular shapes.

In the following paragraphs, we will discuss some of the heuristic methods described in the existing literature. This is far from a complete survey of applied methods, but it does include most of the interesting (published) methods.

The methods can be divided into two basically different groups.

- Legal placement methods

  These methods never violate the overlap constraint. An immediate consequence is that placement of a stencil must always be done in an empty part of the material.

  Some of the earliest algorithms doing this only place each stencil once. According to some measures describing the stencils and the placement done so far, the next best stencil is

[3]A very simple lower bound can be based on the total area of all stencils, but it will usually be far from the optimal solution.

chosen and placed. This is a fast method, but the quality of the solution is limited since no backtracking is done.

Most methods for strip packing follow the basic steps below:

1. Determine a sequence of stencils. This can be done randomly or by sorting the stencils according to some measure e.g. the area or the degree of convexity.
2. Place the stencils with some first/best fit algorithm. Typically a stencil is placed at the contour of the stencils already placed (using an NFP). Some algorithms also allow hole-filling i.e. placing a stencil in an empty area between already placed stencils.
3. Evaluate the length of the solution. Exit with this solution or change the sequence of stencils e.g. randomly or by using some meta-heuristic method and repeat step 2.

Unfortunately the second step is very expensive and these algorithms can easily end up spending time on making almost identical placements.

Legal placement methods not doing a sequential placement do exist. These methods typically construct a legal initial solution and then introduce some set of moves (e.g. swapping two stencils) that can be controlled by a meta heuristic method to e.g. minimize the length of a strip. Examples are Burke and Kendall [12, 10, 11] (simulated annealing, ant algorithms and evolutionary algorithms) and Blazewicz et al. [8] (*Tabu Search*). The latter is also interesting because it allows a set of rotation angles.

- Relaxed placement methods

  The obvious alternative is to allow overlaps to occur as part of the solution process. The objective is then to minimize the amount of overlap. A legal placement has been found when the overlap reaches 0.

  In this context it is very easy to construct an initial placement. It can simply be a random placement of all of the stencils, although it might be better to start with a better placement than that.

  Searching for a minimum overlap can be done by iteratively improving the placement i.e. decrease the total overlap. This is typically done by moving/rotating stencils. The neighborhood of solutions can be defined in various ways, but all existing solution methods have restricted the translational moves to some set of horizontal and vertical translations.

Clearly the relaxed placement methods has to handle a larger search space than the legal placement methods, but it is also clear that the search space of the relaxed methods do not exclude any optimal solutions. This is not true for most of the legal placement methods and it is also one of their weaknesses. On the other hand they can often produce a good legal solution very quickly.

Both legal and relaxed placement methods sometimes try to pair stencils that fit well together (using their NFP). They do this by minimizing the waste according to some measure e.g. the area of any holes produced in the pairing. So far the results of this approach have not been convincing, but it could probably be used to speed up calculations for other solution methods provided that stencils are paired and divided dynamically when needed.

2.4 Legal placement methods

The first legal placement method we describe is also the oldest reference we have been able to find which handles nesting of irregular shapes. It is described by R. C. Art [4] in 1966. His motivation is a shortage of qualified human marker makers in the textile industry and he describes the manual solution techniques which are either done with full-sized cardboard stencils or with plastic miniatures 1/5 in size. He also notes that approximations of the stencils only need to be within the precision performed by the seamstresses.

Art introduces and argues for the following ideas; disregard small stencils since they will probably fit in the holes of the nesting solution of the large stencils (he would do this manually), use the convex counterparts of all stencils to speed up execution time, use the envelope (described earlier) to place stencils, do bottom-left packing, use meta-stencils (combinations of stencils) and split them up when necessary. He used all but the last idea and implemented the algorithm on an *IBM 7094 Data Processing System.* This machine was running at a whopping 0.5 MHz and the average execution time for packing 15 stencils was about 1 minute. He concludes that the results are not competitive with human marker makers, but this is also still a challenge more than 30 years later.

Art also mentions the advantages of free rotation both in sheet metal work and marker making (small adjusting rotations), but notes that his algorithm cannot handle this and that it is a “problem of a higher order”.

10 years later Adamowicz and Albano [1] presents an algorithm that can also handle rotation. Their algorithm works in two stages. In the first stage the stencils are rotated and clustered (using NFPs) to obtain a set of bounding box rectangles with as little waste as possible and in the second stage these rectangles are packed using a specialized packing algorithm. The idea of clustering resembles Arts suggestion of meta-stencils, but Adamowicz and Albano are the first to describe an algorithm which utilizes this idea. Note that the same approach has been used by a commercial solver by Boeing (see Section 2.6). Albano [2] continued the work in an article about a computer aided layout system which uses the above algorithm as an initial solution and then allows an operator to interactively manipulate the solution using a set of commands.

A few years later Albano and Sappupo [3] abandons the idea of using rectangular packing algorithms. They present a new algorithm which resembles Arts algorithm, but they add the ability to backtrack and the ability to handle non-convex polygons (using NFPs). They only allow a set of rotation angles and the examples only allow 180° rotation. The idea of clustering stencils is also abandoned.

Blazewicz et al. [8] (1993) describe one of the few legal placement methods which includes the use of a meta heuristic method. After an initial placement has been found the algorithm moves, rotates and swaps stencils to find better placements. This work is further refined by Blazewicz and Walkowiak [9].

A specialized algorithm for the leather manufacturing industry is described by Heistermann and Lengauer [37]. This is the only published work allowing irregular material (animal hides) and *quality zones.* The latter is a subdivision of material and/or stencils into areas of quality and quality requirements. The method used is a fast heuristic greedy algorithm which tries to find the best fit at a subpart of the contour i.e. no backtracking is done. To speed up calculations approximations of the stencils are created and the stencils are grouped into topology classes determined by the behavior of their inner angles. It is also attempted to place hard-to-place parts first. The algorithm has been in industrial use since 1992 and the implementation constitutes 115000 lines of code (see Section 2.6).

In 1998 Dowsland et al. [23] introduce a new idea, *jostling for position*. They use a standard bottom left placement algorithm with hole filling for their first placement. Then they sort the stencils in decreasing order using their right-most x-coordinates in the placement. A new placement is then made using a right-sided variant of the placement algorithm. This process is repeated for a fixed number of iterations (this is the jostling). In addition to NFPs, all polygons are split into *x-convex* subparts to speed up calculations. Any horizontal line through an x-convex polygon crosses at most two edges. Dowsland et al. emphasize that their algorithm is intended for situations where computation time is strictly limited, but exceeds the time needed for a single pass placement. A test instance with 43 stencils takes around 30 seconds for 19 jostles on a Pentium 166 MHz.

Time is not the main consideration in an algorithm described by Oliveira et al. [51] (2000). Their placement algorithm TOPOS (a Portuguese acronym) is actually 126 different algorithms. The basic algorithm is a leftmost placement without hole filling. The sequence of stencils is either based on some initial sorting scheme (length, area, convexity, ...) or is determined iteratively by which stencil would result in the best partial solution in the placement according to some measure of best fit. Five test instances are used in the experiments on a Pentium Pro 200 MHz. Only the best results are presented, but average execution times are also given. The test instance from the jostle algorithm is also tested (it was actually created by Oliveira et al.). The length is about 4% worse and it has taken 34.6 seconds to find. But this is not quite true since this is only the time used by 1 out of 126 algorithms. Using the average execution times the real amount of time used is more like 45 minutes. Even though Oliveira et al. has done a lot of work to be able to evaluate the efficiency of different strategies, the small amount of test instances limit the conclusions to be made. In their own words: "the geometric properties of the pieces to place [...] have a crucial influence on the results obtained by different variants of the algorithm".

Recently Gomes and Oliveira [34] continued their work. Now the focus is on changing the order in the sequence of stencils used for the placement algorithm. The placement algorithm is a greedy bottom-left heuristic with hole filling. The sequence of stencils is changed by 2-exchanges i.e. by swapping two stencils in the sequence used for the placement algorithm. Different neighborhoods are defined that limit the number of possible 2-exchanges and different search strategies are also defined (*first better*, *best* and *random better*). They are all local search variants and include no hill climbing. Again all combinations are attempted on 5 test instances and they obtain most of the best published solutions, but the computation times are now even worse than before. The above mentioned data instance takes more than 10 minutes for the best solution produced (Pentium III 450 MHz). Other instances take hours for the best solution and days if all search variants (63) are included in the computation time.

The above article is one of the few addressing the problem of limited freedom of rotation when using NFPs, but they simply claim: "Fortunately, in real cases the admissible orientations are limited to a few possibilities due to technological constraints (drawing patterns, material resistance, etc.)". This is obviously not true for a lot of real world problems e.g. in metal sheet cutting (although rotation can be limited in this industry as well). Even the textile industry allows a small (but continuous) amount of rotation as already noted by Art back in 1966.

In the same journal issue a much faster algorithm is presented by Dowsland et al. [24]. This is also a bottom-left placement heuristic with hole filling using NFPs. Experiments are done on a 200 MHz Pentium and solutions are on average found in about 1 minute depending on various criteria. The quality of the solutions can be quite good, but they cannot compete with the best results of the method applied by Gomes and Oliveira. The speed of their algorithm makes it very interesting for the purpose of generating a good initial solution for some of the relaxed placement methods — including the solution method presented in this thesis.

2.5 Relaxed placement methods

Methods allowing overlap as part of the solution process have a much shorter history than the legal placement methods, but they do make up for this in numbers. During the 90's quite a few meta heuristic methods have been applied to the nesting problem, but it has not been a continuous development — even test instances are rarely reused. This means that a lot of different ideas have been tried, but very few of them have been compared.

The most popular meta heuristic method applied to the nesting problem is the same as in most areas of optimization, *Simulated Annealing* (SA). One of the first articles on the subject by Lutfiyya et al. [45] (1992) is mainly focused on describing and fine tuning the SA techniques. Among other things their cost function maximizes edgewise adjacency between polygons, which is an interesting approach. It also minimizes the distance to the origin to keep the stencils closely together. Overlap is determined by using a raster model and to remedy the inherent problems of this approach they suggest for future research to increase the granularity as part of the annealing process. The neighborhood is searched by randomly displacing a polygon, interchanging two polygons or rotating a polygon (within a limited set of rotation angles).

In the same year Jain et al. [40] also use SA, but their approach is quite different since it is focused on a special problem variation briefly mentioned in the beginning of this chapter as the repeated pattern problem. They only nest a few polygons, three or fewer, and the nesting is supposed to be repeated on a continuous strip with fixed width. Changing the *repeat distance* is part of the neighborhood which also includes translation and rotation. Jain et al. refers to blank nesting in the metal industry as the origin of the problem, but it is also relevant for other industries, e.g. the textile industry [48].

The raster model technique returns in Oliveira et al. [50]. They present two variations of SA, one using a raster model and one using the polygons as given. Both with neighborhoods which only allow translation. In the polygonal variant they claim that the rectangular enclosure of the intersection of two polygons can be "fairly fast" computed, but they do not describe how. It cannot be much faster than calculating the exact overlap since the hard part is to find the intersection — not to calculate its area.

A much more ambitious application of SA is described by Heckmann and Lengauer [35]. They focus on the textile manufacturing industry, but their method can clearly handle problems from other industries too. The neighborhood consists of translation, rotation and exchange. A wide range of constraints are described and handled and the implementation has clearly been fine-tuned e.g. by using approximated polygons in the early stages to save computation time. The annealing is used in 4 different stages. The first stage is a rough placement, the second stage eliminates overlaps, the third stage is a fine placement with approximated stencils and the last stage is a fine placement with the original stencils. This algorithm has evolved into a commercial nesting library (see Section 2.6). Some experiments has also been done with other meta heuristic methods and *Threshold Accepting* is concluded to be faster, but it does not increase the yield. Whether it decreases the yield is not stated.

In the same year Theodoracatos and Grimsley [55] published their attempt at applying SA to the nesting problem. They try to pack both circles and polygons (separately). The number of polygons is very limited though, and they have some unsupported claims about their geometric methods. E.g. they claim that the fastest point inclusion test is obtained by testing against all the triangles in a triangulation of the polygons and intersection area is calculated by calculating intersection areas of all pairs of triangles.

Jakobs [41] tried to apply a genetic algorithm, but just like Adamowicz and Albano he actually packs bounding rectangles and then compact the solution in a post-processing step. No comparisons with existing methods are done, but his method is most likely not competitive.

Bennell and Dowsland [6] has a much more interesting approach using a tabu search variant called *Tabu Thresholding* (TT). This is one of the few nesting algorithms which uses intersection depth to measure overlap. They only use horizontal intersection depth and this introduces some problems since it does not always reflect the true overlap in an appropriate way. This is partly fixed by punishing moves that involve large stencils. Rotation is not allowed.

A couple of years later Bennell and Dowsland [7] continue their work. Now they try to combine their TT solution method and the ideas for compaction and separation by Li and Milenkovic [43] briefly described in Section 2.2. Their hybrid algorithm changes between two modes, using the TT algorithm to find locally optimal solutions and using the LP-based compaction routines to legalize these solutions if possible. They also use NFPs in new ways to speed up various calculations.

As mentioned in the introduction, unpublished work was done by Egeblad et al. [28] in 2001. This was a result of an 8 week project and it was the first attempt of applying *Guided Local Search* (GLS) to the nesting problem — based on a successful use of GLS by Færø et al. for the rectangular packing problem in two and three dimensions. A very fast neighborhood search for translation was presented and the results were very promising (better than those published at the time).

2.6 Commercial solvers

The commercial interest in the nesting problem is probably even greater than the academical interest. Their motivation is naturally that large sums can be saved if a bit of computing time can spare the use of expensive materials. A quick search of the Internet revealed the commercial applications presented in Table 2.1.

Although quite a few of the solvers are available as demo programs, it is very difficult to obtain information about the solution methods applied. Exceptions are the two first solvers in the list. 2NA is developed by Boeing and they describe their solution method on a homepage. It is based on combining stencils to form larger stencils with little waste. These are packed using methods applied to rectangular packing problems. It is fast, but it is probably not among the best regarding solution quality.

AutoNester is based on the work done by Heistermann and Lengauer [37] and Heckmann and Lengauer [35]. It is actually two different libraries called AutoNester-T (textile) and AutoNester-L (leather) using two very different solution methods. We believe the nesting algorithm in AutoNester-T using SA to be one of the best algorithms in use both regarding speed and quality (also see Section 8.1.8 for benchmarks).

Package name	Homepage
2NA	`http://www.boeing.com/phantom/2NA/`
AutoNester	`http://www.gmd.de/SCAI/opt/products/index.html`
NESTER	`http://www.nestersoftware.com/`
Nestlib	`http://www.geometricsoftware.com/geometry_ct_nestlib.htm`
OPTIMIZER	`http://www.samtecsoft.com/anymain.htm`
MOST 2D	`http://www.most2d.com/main.htm`
PLUS2D	`http://www.nirvanatec.com/nesting_software.html`
Pronest/Turbonest	`http://www.mtc-limited.com/products.html`
radpunch	`http://www.radan.com/radpunch.htm`
SigmaNEST	`http://www.sigmanest.com/`
SS-Nest/QuickNest	`http://www.striker-systems.com/ssnest/proddesc.htm`

Table 2.1: A list of commercial nesting solvers found on the Internet. Most of them are intended for metal blank nesting.

2.7 3D nesting

The nesting problem can be generalized to three dimensions, but the literature and commercial interest for this problem is not overwhelming. The exception is the simpler problem of packing boxes in a container. There are several reasons for this lack of interest. First of all, the problem is even harder than the two dimensional variant which is hard enough as it is. Secondly, the problem seems to have fewer practical applications.

Recently, new technologies have emerged which could benefit from solutions to a 3D generalization of the nesting problem. This is especially due to the concept of *rapid prototyping* which is an expression used about physical prototypes of 3D computer models needed in the early design/test phases of new products (anything from kitchen utensils to toys). It is also called *3D free-form cutting and packing.*

A survey of various rapid prototyping technologies is given by Yan and Gu [58]. One of these technologies, *selective laser sintering process*, is depicted in Figure 2.5. The idea is to build up the object(s) by adding one very thin layer at the time. This is done by rolling out a thin layer of powder and then sinter (heat) the areas/lines which should be solid by the use of a laser. The unsintered powder supports the objects build and therefore no pillars or bridges have to be made to account for gravitational effects. This procedure can be quite slow (hours) and since the time required for the laser is significantly less than the time required for preparing a layer of powder, it will be an advantage to pack as many objects as possible into one run.

A few attempts have been done to solve the 3D nesting problem and a short survey has been made by Osogami [52]. Most of the limited number of articles mentioned in this survey are also briefly described in the following paragraphs.

Ikonen et al. [39] (1997) is responsible for one of the earliest approaches to the nesting problem. They chose to use a genetic algorithm and although the title refers to "non-convex objects" then the actual implementation only handle solid blocks (or bounding boxes for more complex objects), but they claim the results to be promising. The examples are quite small and no benchmarks or data are available. They do handle rotation, but it is limited to 24

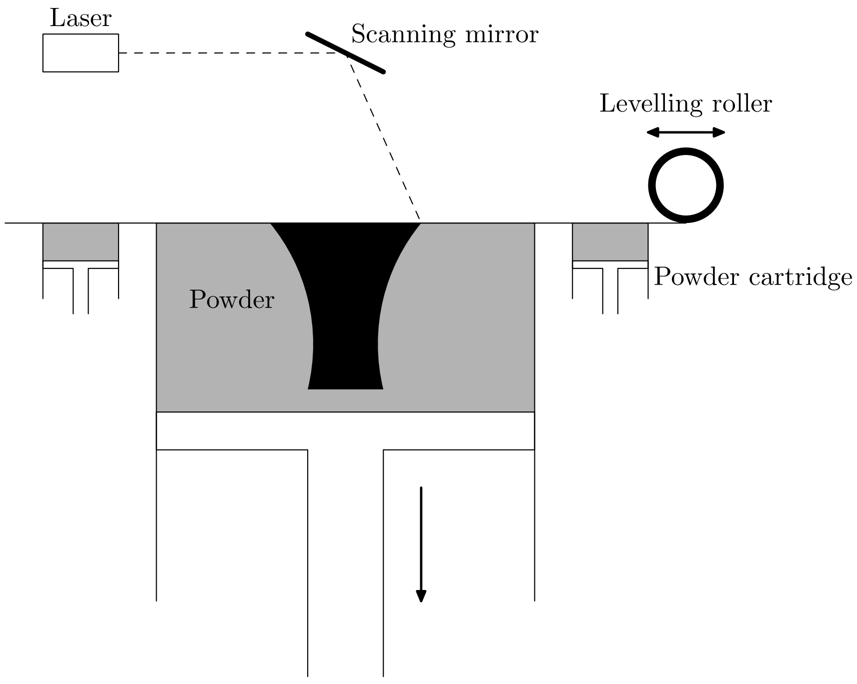


Figure 2.5: An example of a typical machine for rapid prototyping. The powder is added one layer at the time and the laser is used to sinter what should be solidified to produce the objects wanted.

orientations (45 degree increments). In their conclusions they state that it should be considered to use a hill-climbing method to get the genetic algorithm out of local minima.

The approach by Cagan et al. [13] is more convincing. They opt for simulated annealing as their meta heuristic approach and they allow both translation and rotation. They also handle various optimization objectives e.g. taking routing lengths into account. Intersection is allowed as part of the solution process and the calculation of intersection volumes is done by using octree decompositions which closely relates a hierarchical raster model for 2D nesting. The decompositions are done at different levels of resolution to speed up calculations. Each level uses 8 squares per square in the previous level. This means that level n uses 8^{n-1} squares. The level of precision follows the temperature in the annealing (low temperature requires high precision since only small moves are done). The experiments focused on packing are done with a container of fixed size and a number of objects, and the solutions involve overlap which is given in percent. They pack cubes and cog wheels clearly showing that the algorithm works as intended.

The SA approach was an example of a relaxed placement method applied to 3D nesting. Dickinson and Knopf [18, 17] use a legal placement method in their algorithm and as most 2D algorithms in this category it is a sequential placement algorithm. No backtracking is done and the sequence is determined by a measure of best fit. This measure is also usable for 2D nesting and was mentioned at the end of Section 2.2. It is a metric which evaluates the compactness of the remaining free space in an unfinished placement. The best free space is in the form of a sphere (a circle in 2D). Each stencil is packed at a local minimum with respect to this metric. How this minimum is found is not described in detail, but in later work [19] SA is used for this purpose. There are no restrictions on translation and rotation.

Other 2D calculation methods can also be generalized. This includes intersection depth [21] and NFPs. A list of results regarding the latter (also known as Minkowski sums) is presented by Asano et al. [5] (2002). These results (including intersection depth) only involve convex polyhedra.

Dickinson and Knopf [20] have also done a series of experiments with real world problems. This includes objects with an average of 17907 faces each. To be able to handle problems of this size they also introduce an approximation method which is described below. It is our impression that the solution method described by Dickinson and Knopf is the current state-of-the-art in published 3D nesting methods, but it also seems that currently a human would be more efficient at packing complicated 3D objects.

Obviously there is a lot of work to be done in the area of 3D nesting. One important area is approximations of objects. 3D objects are often very detailed with thousands of faces. Very few algorithms can handle this without somehow simplifying the objects. Cohen et al. [14] describe so-called simplification envelopes for this purpose and they seem to be doing really well. In one of their examples they reduce 165,936 triangles to 412 while keeping the object very close to the original. The approximation method used by Dickinson and Knopf [20] is more straightforward. Intuitively, plane grids are moved towards the object from each of the 6 axis-aligned directions (left, right, forwards, backwards, up, down). Whenever a grid square hits the object it stops. The precision of this approximation then depends on the size of the grid squares.

Another problem with 3D nesting is the fact that not all legal solutions (regarding intersection) are useful in practice. E.g. consider an object having an interior closed hole (like inside a ball). There is no point in packing anything inside this hole because it will not be possible to get it out. A simple solution is to ignore these holes, but the problem can be more subtle since unclosed holes or cavities in objects can be interlocked with other objects. Avoiding this when packing is a difficult constraint.

Most of this thesis is not directly referring to 3D nesting, but most of the solution methods applied to the nesting problem is also applicable in 3D. This includes the fast neighborhood search in Chapter 4. As a "proof of concept" Chapter 6 is dedicated to a description of the necessary steps to generalize the 2D nesting heuristic. An implementation has also been created (the title page is an image of a placement found by this implementation).

Chapter 3

Solution Methods

The survey of existing articles presented a wide range of solution methods applied to the nesting problem. Our solution method is in the category of relaxed placement methods i.e. we allow overlap as part of the solution process. The choice of meta heuristic, *Guided Local Search* (GLS), is based on previous work with promising results. Færø et al. [32] applied GLS to the rectangular 2D and 3D packing problems and their work was adapted to the nesting problem by Egeblad et al. [28].

In the following sections we are going to describe the basic concepts of a heuristic approach for solving different variants of the nesting problem. The solution process can be split into 4 different heuristics.

- Initial placement
  For some problems and/or problem instances it is necessary to have an initial solution e.g. to get a small initial strip length for the strip packing problem. In general, a good initial solution might help save computation time, but this can be a difficult trade off. It has to be faster than the main solution method itself (GLS).

- Local search
  By specifying a neighborhood of a given placement we can iteratively search the neighborhood to improve the placement (minimizing total overlap). The search strategy can vary, but there is no hill-climbing abilities in this heuristic.

- Guided Local Search
  Since a local search can quickly end up in local minima it is important to be able to manipulate it to leave local minima and then search other parts of the solution space. This is the purpose of GLS.

- Problem-specific heuristic
  Depending on the problem type solved, e.g. knapsack or strip packing, some kind of heuristic is needed to control the set of elements in the knapsack or the length of the strip.

Note that the problem-specific heuristic could be included in the GLS, but we have chosen to keep them apart to make the solution scheme more flexible. Other approaches could be more efficient. The GLS is focused on solving the decision problem i.e. can a given set of stencils fit inside a given piece of material.

Before describing the GLS we will discuss the neighborhood and the basic local search scheme. Short discussions of initial placements and problem-specific heuristics are postponed to the end of this chapter.

3.1 Local search

Assume we are given a set of n stencils $\mathcal{S} = \{S_1, \ldots, S_n\}$ and a region of space M which corresponds to the material on which to pack them. Although we are trying to solve a decision problem we are going to solve it as a minimization problem. Given a space of possible placements $\mathcal{P}$ we define the objective function,

$$g(p) = \sum_{i=1}^{n} \sum_{j=1}^{i-1} overlap_{ij}(p), \ p \in \mathcal{P},$$

where $overlap_{ij}(p)$ is some measure of the overlap between stencils S_i and S_j in the placement p. In other words the cost of a given placement is determined exclusively by the overlapping stencils. As long as all overlaps add some positive value then we know that a legal placement has been found when the cost $g(p)$ is 0 i.e. we are going to solve the optimization problem,

$$\min g(p), p \in \mathcal{P}.$$

Let us now define a placement a bit more formally. A given stencil S always has a position (s_x, s_y). Depending on the problem solved it can also be described by a degree of rotation s_θ and a state of *flipping*, $s_f \in \{\text{false}, \text{true}\}$, i.e. a placement of a stencil can be described by a tuple $(s_x, s_y, s_\theta, s_f) \in \mathbb{R} \times \mathbb{R} \times [0^\circ, 360^\circ[\times\{\text{false}, \text{true}\}$. Examples of various placements can be seen in Figure 3.1. The figure includes examples of flipping which we define to be the reflection of S about a line going through the center of its bounding box. This line can follow the rotation of the stencil to ensure that the order of rotation and flipping does not matter.

Flipping is possible in many nesting problems since the front and the back of the material is often the same (e.g. metal plates). The garments used in the textile industry are not the same on the front and the back, but sometimes flipping is still allowed. This can happen if the mirrored stencils are also needed to produce the clothes. In practice, the garment is then folded to produce both the stencils and the mirrored stencils while only having to cut once. This is illustrated in Figure 3.2. Clearly, only half of the stencils need to be involved in the nesting problem and individual stencils can be flipped without changing the end result.

Next, we are going to define the neighborhood of a given solution. The neighborhood is a finite set of *moves* which changes the current placement by changing the placement of one or more stencils. A local search scheme uses these moves to iteratively improve the placement. The size of the neighborhood of a given placement is an important consideration when designing a local search scheme. A local search in a large neighborhood will be slow, but it will be expected to find better solutions than a search in a small neighborhood. In our context it is natural to let the neighborhood include displacement, rotation and flipping of individual stencils. One could also consider stencil exchanges. Arbitrary displacements constitutes a very large neighborhood, which would take a long time to search exhaustively. Since any displacement can be expressed as a pair of horizontal and vertical translations then it is a good compromise to include these limited moves in the neighborhood.

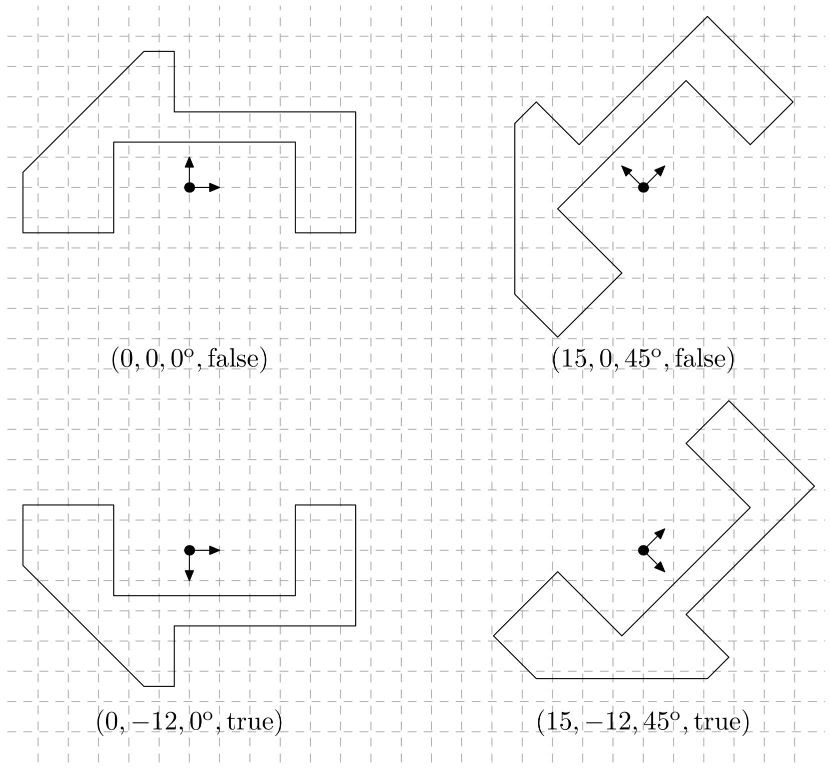

$(0, 0, 0°, \text{false})$
$(15, 0, 45°, \text{false})$
$(0, -12, 0°, \text{true})$
$(15, -12, 45°, \text{true})$

Figure 3.1: Examples of different placements of a polygon. In the upper left corner the polygon is at its origin, $(s_x, s_y, s_\theta, s_f) = (0, 0, 0°, \text{false})$.

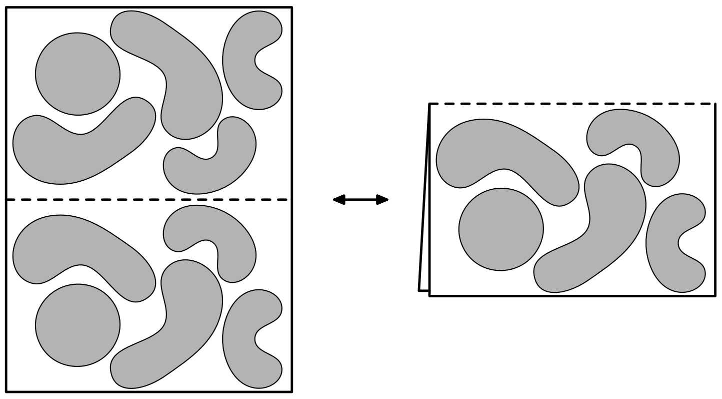


Figure 3.2: In the textile industry both the stencils and their mirrored counterparts are sometimes needed. This is often handled by folding the material and then only cut once to get both sets of parts. With respect to the nesting problem it means that only one set is involved in the nesting, where flipping of individual parts is then allowed.

We have not yet determined the domains of stencil positioning and rotation. Existing solution methods most often have a discrete set of legal positions/rotations to limit the size of the search space. Some even alter these sets as part of the solution process to obtain higher precision (making smaller intervals for a fixed number of positions/rotation angles).

Now, in a given placement the neighborhood of a stencil S will include the following moves (assume the current state of the stencil is $(s_x, s_y, s_\theta, s_f)$).

- Horizontal translation.
  A new placement (x, s_y, s_θ, s_f), where $x \in]-\infty, \infty[$.
- Vertical translation.
  A new placement (s_x, y, s_θ, s_f), where $y \in]-\infty, \infty[$.
- Rotation.
  A new placement (s_x, s_y, θ, s_f), where $\theta \in [0°, 360°[$.
- Flipping.
  A new placement with S flipped.

To obtain a legal placement S must also be within the bounds of the material M, but this is not necessarily a requirement during the solution process.

We can now define a neighborhood function, $N : \mathcal{P} \to 2^{\mathcal{P}}$, which given a placement returns all of the above moves for all stencils (the moves *cannot* be combined). Using this definition a placement p is a local minimum if

$$\forall x \in N(p) : g(p) <= g(x),$$

i.e. there is no move in the neighborhood of the current placement which can improve it.

Obviously, this neighborhood is infinite in size, but we will see later (Chapter 4) that there exists translation/rotation algorithms for polygons which are polynomial in time with respect to the number of edges involved. For translation the algorithm is guaranteed to find the position with the smallest overlap with other stencils.

As already noted a local search scheme uses the neighborhood of the current placement to iteratively improve it. The local search can be done with a variety of strategies. These include a *greedy strategy* choosing the best move in the entire neighborhood or a *first improvement strategy* which simply processes the neighborhood in some order following any improving moves found. Either way when no more improving moves are found the local search is in a local minimum.

3.2 Guided Local Search

The meta heuristic method GLS uses local search as a subroutine, but the local search is very often changed to what is known as a *Fast Local Search* (FLS). This name simply means that only a subset of the neighborhood is actually searched. We will get back to how this can be done for the nesting problem.

GLS was introduced by Voudouris and Tsang [56] and has been used successfully on a wide range of optimization problems including *constraint satisfaction* and the *traveling salesman problem*. It resembles Tabu Search (TS) since it includes a "memory" of past solution states, but this works in a less explicit way than TS. Instead of forbidding certain moves as done in TS (most often the reverse of recent moves) to get out of local minima, GLS punishes the bad characteristics of unwanted placements to avoid them in the future.

The characteristics of solutions are called *features* and in our setting we need to find the features that describe the good and bad properties of a given placement. A natural choice is to have a feature for each pair of polygons that expresses whether they overlap,

$$I_{ij}(p) = \begin{cases} 0 & \text{if } \mathit{overlap}_{ij}(p) = 0, \\ 1 & \text{otherwise,} \end{cases} \qquad i, j \in \{1, \ldots, n\},\ p \in \mathcal{P}.$$

GLS uses the set of features to express an *augmented* objective function,

$$h(p) = g(p) + \lambda \cdot \sum_{i=1}^{n} \sum_{j=1}^{i-1} p_{ij} I_{ij}(p),$$

where p_{ij} (initially 0) is the penalty count for an overlap between stencils S_i and S_j. In GLS this function replaces the objective function used in the local search.

GLS also specifies a function used for deciding which feature(s) to penalize in an illegal placement,

$$\mu_{ij}(p) = I_{ij}(p) \cdot \frac{c_{ij}(p)}{1 + p_{ij}},$$

where the *cost function* $c_{ij}(p)$ is some measure of the overlap between stencils S_i and S_j. The feature(s) with the largest value(s) of μ are the ones that should be penalized by incrementing their p_{ij} value. To ensure diversity in the features penalized, the above function also takes into account how many times a feature has already been penalized and this lowers its chance of being penalized again.

It is left to us to specify the cost function and the number of features to penalize. A natural choice for the cost function would be some measure of the area of the overlap. This could be the exact area, a rectangular approximation or other variants.

A simplified example of how GLS works is given in Algorithm 1.

Algorithm 1 could use a normal local search method, but the efficiency of GLS can be greatly improved by using fast local search (FLS). To do this we need to divide the neighborhood into a set of sub-neighborhoods, which can either be active or inactive indicating whether the local search should include them in the search. In our context the natural choice is to let the moves related to each stencil be a sub-neighborhood resulting in n sub-neighborhoods. It is the responsibility of the GLS algorithm to activate neighborhoods and it is the responsibility of the FLS to inactivate neighborhoods when they have been exhausted. Furthermore the fast local search can also activate neighborhoods depending on the strategy used.

A logical activation strategy for the GLS is to always activate the neighborhoods of stencils involved in the new penalty increments. Possible FLS strategies include the following:

- Fast FLS.
  Deactivate a sub-neighborhood when it has been searched and has not revealed any possible improvements.

- Reactivating FLS.
  The same as above, but whenever a stencil is moved, activate all polygons which overlaps the stencil before and/or after the move.

Algorithm 1 Guided Local Search

```
Input: Stencils $S_1, ..., S_n$.
Generate initial placement $p$.
for all $S_i, S_j, i > j$ do
    Set $p_{ij} = 0$.
end for
while $p$ contains overlap do
    Set $p = \text{LocalSearch}(p)$.
    for all $S_i, S_j, i > j$ do
        Compute $\mu_{ij}(p)$.
    end for
    for all $S_i, S_j, i > j$ such that $\mu_{ij}(p)$ is maximum do
        Set $p_{ij} = p_{ij} + 1$.
    end for
end while
Return $p$.
```

3.3 Initial solution

As noted earlier our focus is mainly on the decision problem, but most realistic problems are strip packing, bin packing or knapsack problems. A good initial solution can help the solution process, but experiments presented by Egeblad et al. [28] indicated that it is not essential for the solution quality. They found that a random initial placement with overlap worked just as well as more ambitious legal placement strategies for the strip packing problem. Nevertheless, a fast heuristic in the initial stages can be an advantage concerning computation time e.g. to shorten the initial length in strip packing.

We are not going to describe specialized initial solution heuristics for the various problem types. To get an initial number of bins and an initial strip-length one can simply adopt the approach by Egeblad et al. which is a bounding-box first-fit heuristic (see Figure 3.3). This heuristic sorts all stencils according to height and then places them sequentially which means that the largest stencils are gathered in one end of the strip or in the same bin. To avoid that this initial solution affects the GLS in an unfortunate way it could be wise to discard the solution and instead make a random placement only using the strip length or number of bins provided by the initial solution.

3.4 Packing strategies

The various problem types also require different packing strategies and the following is a very short discussion of how this can be done for each problem type. The main purpose of this discussion is to show that all of the problems can be solved with the use of a solver to the decision problem. No claims are made about the efficiency of these methods.

- Bin packing problem. This is straightforward. The initial solution provided a number of bins for which there exists a solution. Simply remove one bin and solve this problem as a decision problem. If a solution is found then another bin is removed, and so forth. Note that for a fixed number of bins the bin packing problem can be solved as a

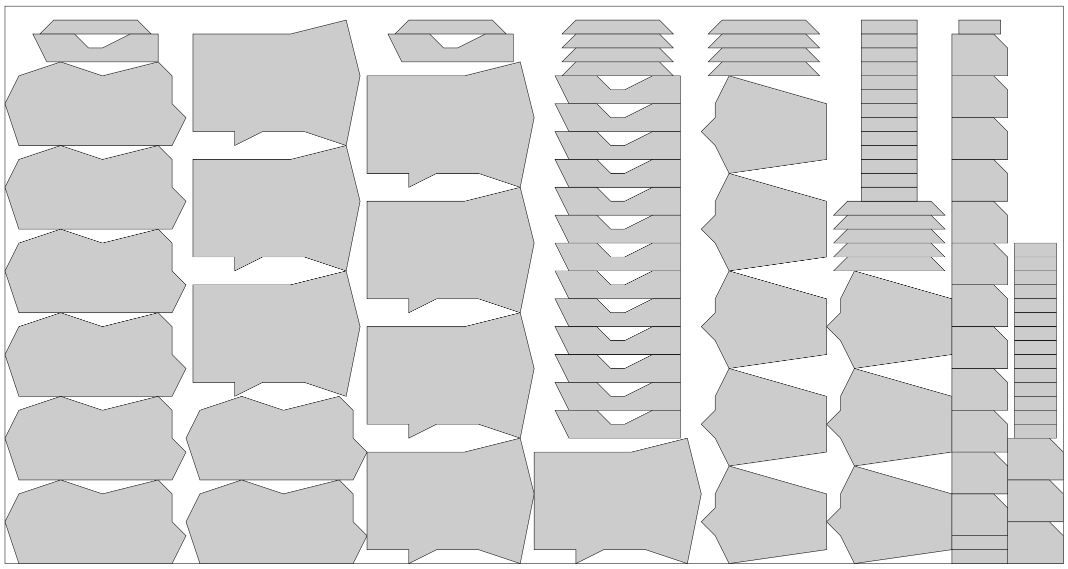


Figure 3.3: An example of a simple bounding box first fit placement.

decision problem, but it requires the shape of the material to be disconnected in a way such that translations can move stencils between bins. This is probably an inefficient approach and it would be better to keep the bins separately and instead introduce an extra neighborhood move to move stencils between bins.

Another aspect of bin packing is that it is more likely to be possible to find an optimal solution. When a legal placement is found and a bin is removed then it should be verified that the area of the stencils can fit inside the remaining bins. If not, an optimal solution has already been found. This can also be calculated as an initial lower bound.

- Strip packing problem. The strip packing problem (and the repeated pattern variant) is a bit more difficult to handle. Whenever a legal placement has been found the strip needs to be shortened. Two questions arises. How much shorter? What if a solution is not found (within some limit of work done)? Two examples of strategies could be.
  - Brute force strategy. Whenever a legal placement is found make the strip $k\%$ shorter, where k could be about 1 or less depending on the desired precision.
  - Adaptive strategy. Given a step value x whenever a legal placement is found make the strip x shorter. If a new legal placement has not been found within e.g. n^2 iterations of GLS, increase the length with $x/2$ and set $x = x/2$. If a solution is found very quickly one could also consider to increase the step value.
- Knapsack problem. Like the bin packing problem the knapsack problem is $\mathcal{NP}$-hard even when reduced to 1 dimension. Algorithm 2 is a simple strategy for finding placements for the knapsack problem using the decision problem. It is recursive and needs two arguments, a current placement (initially empty) and an offset in the list of stencils (initially 1).

Algorithm 2 Knapsack algorithm

```
Knapsack($P$, $i$)
for $j$=$i$ to $n$ do
    Add $S_j$ to the placement.
    if A legal placement with $S_j$ can be found within some time limit then
        Remember the placement if it is the best found so far.
        Knapsack($P$, $j + 1$)
    end if
    Remove $S_j$ from the placement.
end for
```

Chapter 4

Fast Neighborhood Search

4.1 Background

In the neighborhood for our local search we have both translation and rotation, but we have not specified how these searches can be done. Basically, there are two ways to do it. The straightforward way is to simply use an overlap algorithm to calculate the overlap at some predefined set of translation distances or rotation angles, e.g. $\{0, 1, 2, \ldots, 359\}$, and then choose the one resulting in the smallest overlap. There is one major problem with this approach. Overlap calculations are expensive and therefore one would like to do as few as possible, but precision requires a large predefined set. In the SA algorithm by Heckmann and Lengauer [35] this dilemma is solved by having a constant number of possible moves. The maximum possible distance moved is then decreased with time (and the set scaled accordingly) thereby keeping a constant computational overhead without sacrificing precision entirely.

An alternative is to find a precise minimum analytically. This might seem very difficult at first, but it turns out to be possible to do quite efficiently and it is the subject of this entire chapter.

Færø et al. [32] presented a fast horizontal/vertical translation method for boxes in the rectangular packing problem. The objective was to find the horizontal or vertical position of a given box that minimized its overlap with other boxes. They showed that it was sufficient to calculate the overlap at a linear number of breakpoints to achieve such a minimum.

Egeblad et al. [28] extended the methods used on the rectangular packing problem to make an efficient translation algorithm for the nesting problem. However, they did not prove the correctness of this algorithm.

In this chapter we will not only prove the correctness of the translation algorithm, but we will also generalize the basic principles of the algorithm. This will reveal a similar algorithm for rotation and it will also be an inspiration for three dimensional algorithms (see Chapter 6).

Although the practical results in this chapter concerns the translation and rotation of polygons, the theoretical results in the following section is of a more general nature. This is done partly because the results could be useful in other scenarios (e.g. the problem of packing circles) and partly because the theory would not be much simpler if restricted to rotation and translation of polygons.

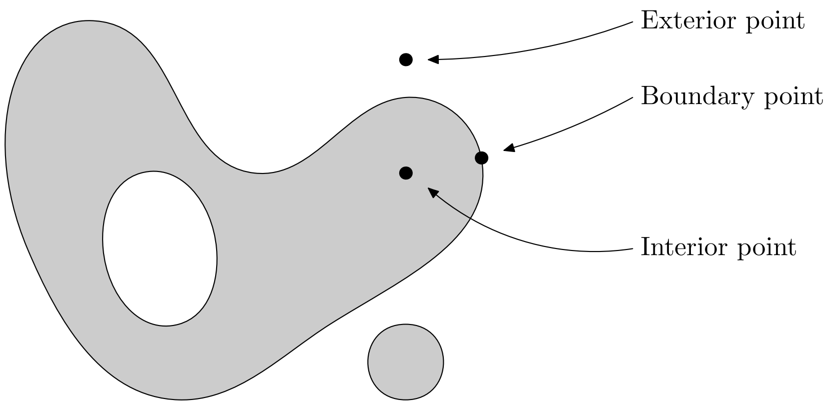


Figure 4.1: The above gray area is an example of a set of points in the plane. Examples of interior and boundary points are marked and the entire boundary is clearly marked as three closed curves.

4.2 A special formula for intersection area

At the end of this section a proof is given for a theorem that can be used to calculate intersection areas. The theorem is stated in a very general setting making it useful for the specification of some transformation algorithms in later sections (especially related to translation and rotation). To obtain this level of generality the following subsections are mostly dedicated to the precise definition of various key concepts.

4.2.1 Vector fields

First we have to recap some basic calculus regarding sets of points in the plane.

Definition 1. *Given a point p_0 and a radius $r > 0$, we define a* **disc** *of radius r as the set of points with distance less than r from p_0 and we denote it $N_r(p_0)$. More precisely,*

$$N_r(p_0) = \{p \in \mathbb{R}^2 : ||p - p_0|| < r\}.$$

Given a set of points S in the plane, we say that

- *p_0 is a* **boundary point** *if for any $r > 0$, $N_r(p_0)$ contains at least one point in S and at least one point outside S. The set of all boundary points is called* **the boundary** *of S.*
- *S is* **closed** *if the entire boundary of S belongs to S.*
- *an* **exterior point** *is a point not in S. The set of all such points is the* **exterior** *of S.*
- *an* **interior point** *is a point in S which is not a boundary point. The set of all such points is the* **interior** *of S.*

The practical results in this chapter involves translation and rotation, but at this point we introduce a more general concept useful for a wide range of transformation methods.

Definition 2. *Given two continuous functions $F_1(x, y)$ and $F_2(x, y)$, a* **(planar) vector field** *is defined by*

$$\mathbf{F}(x, y) = F_1(x, y)\mathbf{i} + F_2(x, y)\mathbf{j},$$

where $\mathbf{i}$ *and* $\mathbf{j}$ *are basis vectors.*

A **field line** *for a point p is defined to be the path given by following the vector field forwards from p. Since the vector field specifies both direction and velocity then this field line can be described by a parametric curve $r_p(t)$. We will say that the point $p_t = (x_t, y_t) = r_p(t)$ is the location of p at time t along its field line, in particular $p = p_0 = (x_0, y_0) = r_p(0)$.*

If the field line is a closed path ($\exists t' > 0 :\ p_{t'} = p_0$) then we define the field line to end at the end of the first round i.e. t is limited to $[0, t']$ where t' is the smallest value for which $p_{t'} = p_0$.

Note that a field line cannot cross itself since that would require the existence of a point where the vector field points in two different directions. The same argument shows that two different field lines cannot cross each other, but they can end at the same point.

Some examples of vector fields and field lines can be seen in Figure 4.2. In practice we are only going to use the straight and rotational vector fields, but the other vector fields help to emphasize the generality of the following results.

4.2.2 Segment and vector field regions

The next definition might seem a bit strange, but it provides us with a link between curve segments and areas which depends on the involved vector field.

Definition 3. *Given two curve segments a, b and a vector field, the* **segment region** $\mathbf{R}(a, b)$ *is defined as follows: All points p for which there exists a unique point p_a on a, a unique point p_b on b and time values t', t'' such that*

- $0 < t' < t''$,
- $r_{p_a}(t') = p$
- *and* $r_{p_a}(t'') = p_b$.

Less formally, there must exist a unique field line from a to b which goes through p.

Note that $\mathbf{R}(a, b)$ is not the same as $\mathbf{R}(b, a)$. In many vector fields at least one of these two regions will be empty unless a and b are crossing. Furthermore, if no field lines cross both curve segments then the set will definitely be empty. The uniqueness of the field line ensures that it also makes sense to move backwards on a field line in a segment region.

An advanced example of a segment region is given in Figure 4.3. Later we will see how these segment regions can be used to find intersection areas. At this point more definitions are needed.

Arbitrary vector fields are a bit too general for our uses. The purpose of the next definition is to enable us to focus on useful subparts of vector fields.

Definition 4. *A* **vector field region** *is defined by two straight line segments c_1 and c_2 to be the segment region $\mathbf{R}(c_1, c_2)$, where c_1 and c_2 are the shortest lines possible to create this region. Denote these lines the* **first cut line** *and the* **second cut line***.*

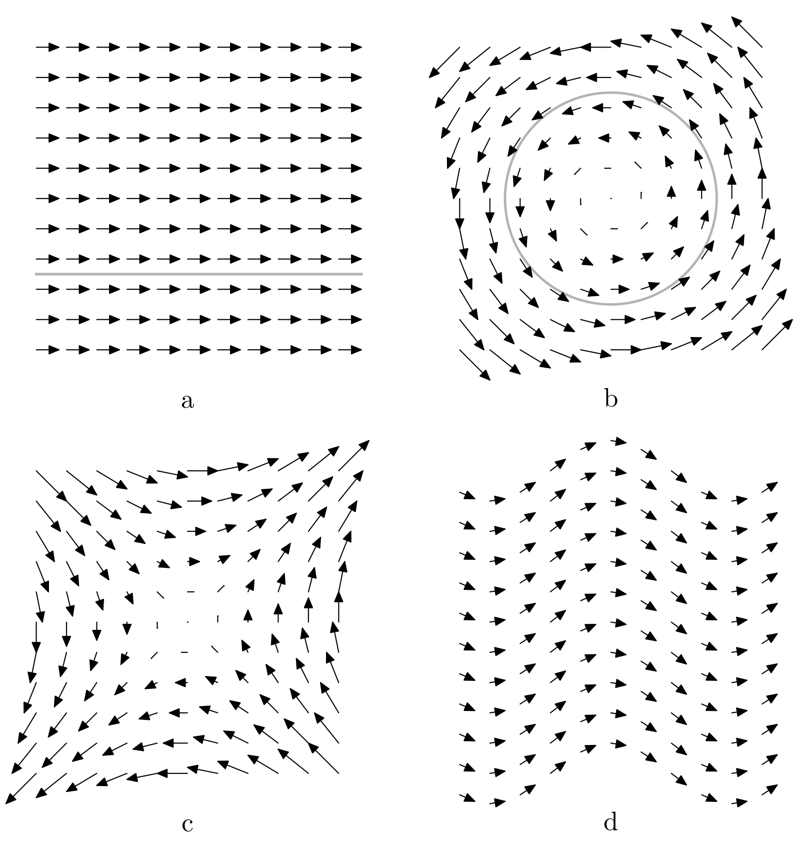


Figure 4.2: Vector field examples. The gray lines in the top row are examples of field lines. a) A very simple horizontal vector field, $\mathbf{F}(x, y) = (1, 0)$. b) A rotational vector field, $\mathbf{F}(x, y) = (-y, x)$. c) d) Advanced examples.

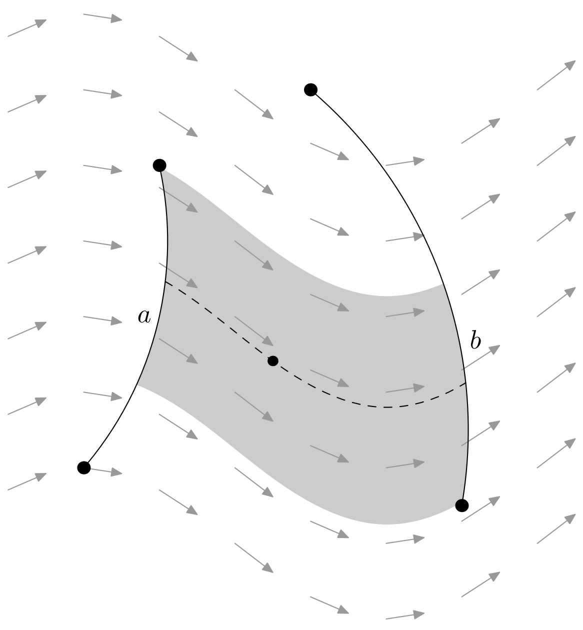


Figure 4.3: An advanced example of a segment region $\mathbf{R}(a, b)$. One of the points in the region is shown with its corresponding field line segment from a to b (the dashed curve segment). Note that $\mathbf{R}(b, a)$ is an empty region (assuming no field lines are closed paths).

Although it is essentially the same as the segment regions, it is more appropriate to differentiate between the two concepts. The limitation of using straight lines is not necessary, but we have no use of a more general definition. Note that a vector field region is a bounded area.

Whenever a set of points S is said to be in (or inside) a vector field region then it simply means that $S \subseteq \mathbf{R}(c_1, c_2)$. Examples of the most important vector field regions are given in Figure 4.4.

4.2.3 Signed boundaries and shapes

Next we need some definitions to be able to describe the kind of shapes and vector fields we are going to handle.

Definition 5. *Given a vector field, a set of points S and a boundary point p of S let $r(t)$ be the field line through p such that $r(0)$ is on the first cutline. Let t_p be the value for which $r(t_p) = p$. Now, p is said to be* **positive** *if for any disc $N_r(p), r > 0$ there exists t', t'' such that $t' < t_p < t''$, $r(t')$ is an exterior point of S and $r(t'')$ is an interior point. A boundary point is said to be* **negative** *if $t'' < t_p < t'$.*

A boundary segment of S is said to be positive/negative if all points on the segment are positive/negative. A boundary segment is said to be **neutral** *if some field line is overlapping the entire segment.*

In other words a positive boundary point is crossed by a field line that goes from the outside of the shape to the inside and vice versa for negative boundary points. Using Definition 5 we can now give a precise description of shapes.

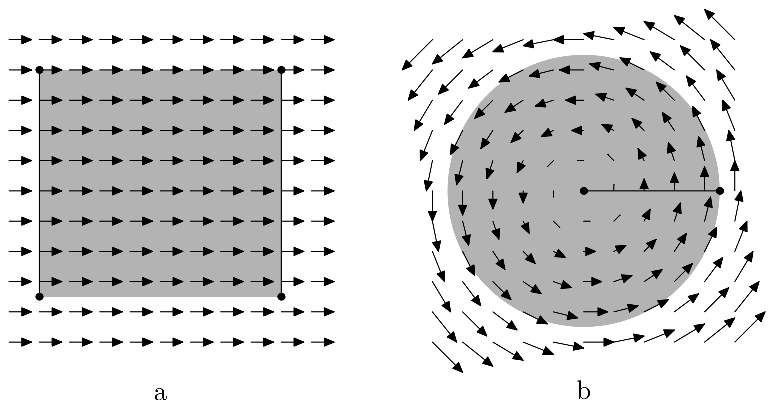


Figure 4.4: Examples of vector field regions. a) A horizontal vector field region with two finite cut lines. b) A rotational vector field region, where the two cut lines are identical.

Definition 6. *A* **shape** *S in a vector field is a set of points which adhere the following conditions: For the boundary B of S there must exist a division into a* finite *set of boundary segments $b_1, ..., b_n$, where*

- *each boundary segment can be categorized as being positive, negative or neutral,*
- *for any $i, j \in \{1, \ldots, n\}, i \neq j : b_i \cap b_j = \emptyset$ (no points are shared between segments)*
- *and $b_1 \cup \ldots \cup b_n = B$ (the segments cover the entire boundary).*

Furthermore, all boundary points must be in contact with the interior i.e. for any $p_0 \in S$ and $r > 0$, $N_r(p_0)$ contains at least one point which is in S, and which is not a boundary point.

An example of a shape is given in Figure 4.5. A shape does *not* have to be connected or without holes and it cannot contain isolated line segments or points. Note that any singular point in a boundary can be categorized as a neutral boundary segment, but a shape cannot have an infinite number of them since the total number of boundary segments is required to be finite.

Figure 4.5 includes an example of a field line. The following lemma concerns the behavior of such field lines.

Lemma 1. *Suppose S is a shape in a vector field region. For any field line in the region (from first to second cutline), which does not include any neutral boundary points from the shape, the following will be true:*

- *If the field line crosses the shape then it will happen in an alternating fashion, regarding positive and negative boundary segments, starting with a positive boundary segment and ending with a negative boundary segment. Whether the field line is inside the shape alternates in the same way starting outside the shape.*
- *The field line will cross an even number of boundary segments.*
- *If a point $p \in S$ is on the field line then moving forwards from p the field line will cross one more negative boundary segment than positive boundary segments. Moving backwards*

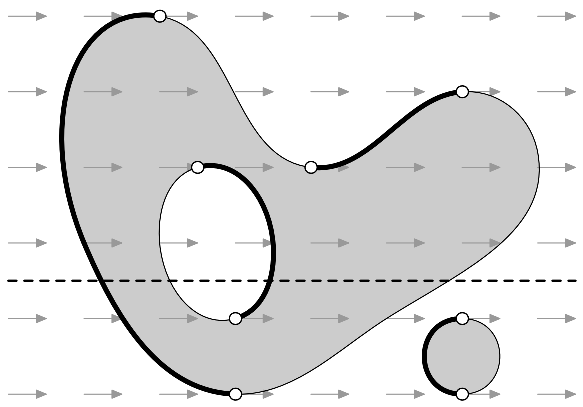


Figure 4.5: The above set of points constitutes a *shape* in a horizontal vector field. The bold boundary segments are positive boundary segments and the other boundary segments are negative with the exception of the endpoints which are neutral. The dashed line is an example of a field line. Note how it crosses the boundaries of the shape.

from p the field line will cross one more positive boundary segment than negative boundary segments.

- *If a point $p \notin S$ is on the field line then moving forwards or backwards from p the field line will cross an equal number of positive and negative boundary segments.*

Proof. We are only proving the first statement since the others follow easily.

Initially (at the first cut line) the field line is outside S. From the definition of positive and negative boundary segments it is easy to see that when moving forwards on the field line the first boundary segment must be positive. It also follows from the definitions that the field line then continues inside S and the next boundary segment must then be negative. This continues in an alternating fashion ending with a negative boundary segment — otherwise S would not be within the vector field region. □

4.2.4 The Intersection Theorem

Now, we are getting closer to the main theorem. The next definition specifies a relation between a point and a segment region which simply states whether the point is inside or outside the region.

Definition 7. *Given boundary segments a and b and a point p in a vector field region, we define the* **containment function** *as*

$$\mathbf{C}(a, b, p) = \begin{cases} 1 & \text{if } p \in \mathbf{R}(a, b), \\ 0 & \text{if } p \notin \mathbf{R}(a, b). \end{cases}$$

This can be generalized to sets of boundary segments A and B:

$$\mathbf{C}(A, B, p) = \sum_{a \in A,\ b \in B} \mathbf{C}(a, b, p).$$

Using the containment function and the following explicit division of boundary segments in positive and negative sets, we are now ready to present an essential theorem.

Definition 8. *Given a shape S, we denote a finite set of boundary segments (as required by the definition of a shape) S_b. We also denote the sets of positive and negative segments as $S_b^+ = \{b \in S_b \mid b \text{ is positive}\}$, $S_b^- = \{b \in S_b \mid b \text{ is negative}\}$.*

Theorem 2 (Containment Theorem). *Suppose we are given a pair of shapes S and T and a point p in a vector field region, and that the field line of the point p does not contain any neutral boundary points from the shapes, then the following is true:*

$$\begin{aligned} p \in S \cap T \quad &\Rightarrow \quad w(p) = 1 \\ p \notin S \cap T \quad &\Rightarrow \quad w(p) = 0, \end{aligned}$$

where

$$w(p) = \mathbf{C}(T_b^+, S_b^-, p) + \mathbf{C}(T_b^-, S_b^+, p) - \mathbf{C}(T_b^+, S_b^+, p) - \mathbf{C}(T_b^-, S_b^-, p) \tag{4.1}$$

Proof. In the following, whenever the terms forwards and backwards are used it means to start at p and then move forwards and backwards on the field line which goes through p.

From the definition of segment regions, it easily follows that the only pairs of boundary segments affecting $w(p)$ are those which intersect the field line going through p. Furthermore, the sum above is only affected by boundary segments from T found when moving backwards and boundary segments from S found when moving forwards.

Assume that s positive boundary segments from S are crossed when moving forwards and t negative boundary segments from T are crossed when moving backwards. Now, by using Lemma 1 we can quite easily do the proof by simply counting.

First, assume that $p \in S \cap T$. By definition $p \in S$ and $p \in T$. It then follows from Lemma 1 that $s+1$ negative and $t+1$ positive boundary segments are crossed when moving forwards and backwards, respectively. Inserting this in equation 4.1 and doing the math reveals

$$\begin{aligned} w(p) &= (t+1)(s+1) + ts - (t+1)s - t(s+1) \\ &= ts + t + s + 1 + ts - ts - s - ts - t \\ &= 1. \end{aligned}$$

Now assume that $p \notin P \cap Q$. There are three special cases, which can all be handled in a very similar way. In short,

$$\begin{aligned} p \notin S \wedge p \notin T : ts + ts - ts - ts &= 0, \\ p \in S \wedge p \notin T : t(s+1) - ts + ts - t(s+1) &= 0, \\ p \notin S \wedge p \in T : (t+1)s - ts + (t+1)s - ts &= 0. \end{aligned}$$

□

Definition 9. *Given boundary segments a and b, let the area of a region $\mathbf{R}(a, b)$ be defined as $\mathbf{A}(a, b)$. We extend this definition to two sets of boundary segments A and B as follows:*

$$\mathbf{A}(A, B) = \sum_{a \in A,\, b \in B} \mathbf{A}(a, b)$$

Standard calculus gives us that since $\mathbf{R}(a, b)$ is a bounded area then we have,

$$\mathbf{A}(a,b) = \int\int_{R(a,b)} 1dA = \int\int_{p\in\mathbb{R}^2} \mathbf{C}(a,b,p))dA.$$

We can now easily state and proof the main theorem.

Theorem 3 (Intersection Theorem). *In a vector field region the intersection area α of two shapes S and T can be calculated as:*

$$\alpha = \mathbf{A}(T_b^+, S_b^-) + \mathbf{A}(T_b^-, S_b^+) - \mathbf{A}(T_b^+, S_b^+) - \mathbf{A}(T_b^-, S_b^-)$$

Proof. Disregarding a finite number of field lines, we know that $w(p) = 1$ if $p \in S \cap T$ and $w(p) = 0$ otherwise. The finite number of field lines do not contribute with any area, so α can be calculated as:

$$\alpha = \int\int_{p\in\mathbb{R}^2} w(p)dA.$$

Using equation 4.1 and the fact that the integral of a sum is the same as the sum of integrals,

$$\begin{aligned}\int\int_{\mathbb{R}^2} w(p)dA &= \int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^+, S_b^-, p)dA + \int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^-, S_b^+, p)dA \\ &- \int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^+, S_b^+, p)dA - \int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^-, S_b^-, p)dA\end{aligned}$$

Let us only consider $\int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^+, S_b^-, p)dA$ which can be rewritten,

$$\begin{aligned}\int\int_{p\in\mathbb{R}^2} \mathbf{C}(T_b^+, S_b^-, p)dA &= \int\int_{p\in\mathbb{R}^2} \sum_{a\in T_b^+,\ b\in S_b^-} \mathbf{C}(a,b,p) \\ &= \sum_{a\in T_b^+,\ b\in S_b^-} \int\int_{p\in\mathbb{R}^2} \mathbf{C}(a,b,p) \\ &= \sum_{a\in T_b^+,\ b\in S_b^-} \mathbf{A}(a,b) \\ &= A(T_b^+, S_b^-).\end{aligned}$$

Rewriting the other three integrals we achieve the desired result.

□

The Intersection Theorem gives us a special way of calculating intersection areas, which we are going to see an example of in the next section. Note that it can also be used to calculate the area of a single shape S by letting the "shape" T be a half space placed appropriately such that S is entirely inside T. This brings the Intersection Theorem very close to known formulas for area computations.

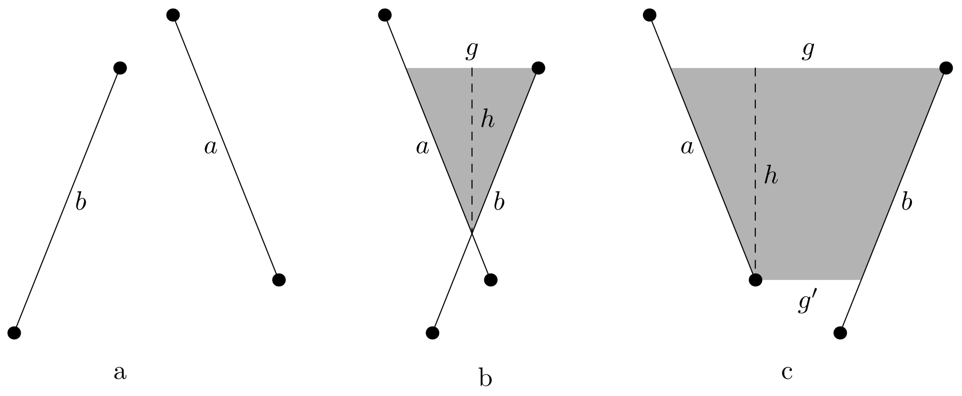


Figure 4.6: The area $\mathbf{A}(a, b)$ can be in three different states. a) An empty region. b) A triangular area, $\mathbf{A}(a, b) = \frac{hg}{2}$. c) A trapezoidal area, $\mathbf{A}(a, b) = \frac{h(g+g')}{2}$.

4.2.5 A simple example

It is time to make some practical use of the theoretical results. We are going to take a closer look at the horizontal vector field and to simplify calculations we are going to limit the shapes to be polygons (holes are allowed). An edge excluding endpoints from a polygon is always a strictly positive, negative or neutral boundary segment no matter where it is placed in the horizontal vector field and it is therefore natural to let the set of edges be the set of boundary segments (endpoints are neutral boundary segments). Note that this is most often not a minimal set.

The basic calculation needed to use Theorem 3 is the area between two line segments. Formally, we are given two line segments a and b and we are interested in finding $\mathbf{A}(a, b)$ i.e. the area of the region between a and b. The first observation to make is that if the vertical extents of the two line segments do not overlap then the region is always empty. If the vertical extents do overlap then the region can be in three basically different states. The two lines can intersect, a can be to the left of b or a can be to the right of b. Examples of all of these three cases are given in Figure 4.6.

Finding $\mathbf{A}(a, b)$ is easy when a is to the right of b since this makes the region empty. The other areas are not much harder. Intersecting segments reveal a triangular area and when a is to the left of b a trapezoidal area needs to be found.

We can now give an example of how to use Theorem 3. In Figure 4.7 all regions involved in the calculation of the intersection of two polygons is given. The example uses polygons and a simple horizontal vector field, but other shapes could easily be allowed as long as one is able to calculate the area of the regions. The same goes for the vector field.

Also note that if an edge from S does not have anything from T on its left side then it is not involved in any regions used for the intersection calculation. This can hardly be a surprise, but it is exactly this asymmetry of the Intersection Theorem which will be utilized in the next section.

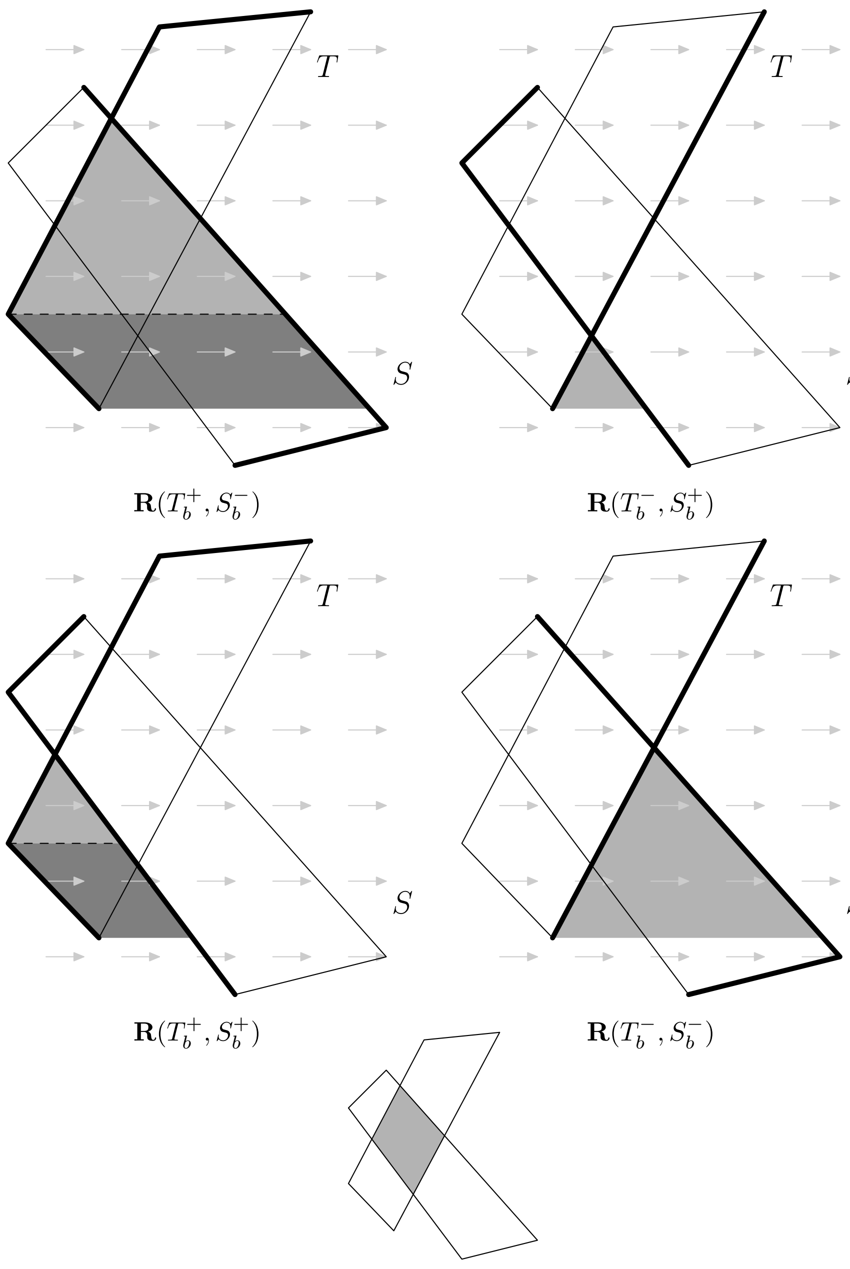

T
S
$\mathbf{R}(T_b^+, S_b^-)$
T
S
$\mathbf{R}(T_b^-, S_b^+)$
T
S
$\mathbf{R}(T_b^+, S_b^+)$
T
S
$\mathbf{R}(T_b^-, S_b^-)$

Figure 4.7: The above regions can be used to find the area of the intersection of the two polygons S and T. The bold edges are the edges involved in the specified sets. The shaded areas in the top row contribute positively and the shaded areas in the bottom row contribute negatively. Area $= \mathbf{A}(T_b^+, S_b^-) + \mathbf{A}(T_b^-, S_b^+) - \mathbf{A}(T_b^+, S_b^+) - \mathbf{A}(T_b^-, S_b^-)$.

4.3 Transformation algorithms

For some vector fields the calculation of intersection areas as given by the Intersection Theorem has a much more interesting property. To see this we first need yet another definition and a small lemma.

Definition 10. *Given a shape S in a vector field region and a time value t we define the transformation $S(t)$ to be the shape obtained by moving all points of S along their corresponding field lines with the given time value.*

This is not as cryptic as it might seem. In the horizontal vector field this is a simple horizontal translation and in the rotational vector field this is a rotation of S around the center of the vector field. Now we need a simple but essential lemma about the behavior of boundary segments under a transformation.

Lemma 2. *A boundary segment b in a shape S will not change type (positive, negative or neutral) when S is transformed by some value t.*

Proof. This follows directly from the definition of the sign of boundary points (Definition 5). Assume that b is a positive boundary segment and let p_0 be a point on b. We then know that for an arbitrarily small disc $N_r(p_0)$ there exists at least one exterior point p'_0 and at least one interior point p''_0 in the disc such that both points are on the field line and p'_0 is before p''_0.

When S is transformed the point p_0 will be moved to p_t. To show that p_t is still a positive boundary point we need to show that an arbitrarily small disc $N_r(p_t)$ contains at least one exterior point and at least one interior point such that both points are on the field line and the exterior point comes before the interior point. This is clearly obtained by using the transformation of the points from a disc sufficiently small at p_0 i.e. p'_t and p''_t □

The immediate consequence of this lemma is that the sets S_b^+ and S_b^- are unchanged when S is transformed. This means that when calculating intersection areas between a shape T and $S(t)$ for any value t then it is the same pairs of boundary segments that are involved in the calculation. This is an advantage if one wants to find intersection areas for a set of values of t.

Unfortunately there is a problem with transformations of S in most vector fields. The form and the size of the shape is not preserved i.e. the distance between any pair of points in the shape is not the same before and after a transformation. In the following we are going to focus on vector fields that do preserve the shape.

Definition 11. *A* **shape-preserving** *vector field is a vector field in which any transformation preserves the distance between any two points.*

Within the above type of vector fields, we can make a simple algorithm for two shapes S and T which finds a good transformation of S with respect to having small overlap with T (Algorithm 3).

A small example of this algorithm is given in Figure 4.8. The example shows two triangles S and T in a horizontal vector field. There are 8 breakpoints and the algorithm transforms S to each of these breakpoints to calculate the overlap with T. It turns out that a placement with no overlap can be found.

If we assume that it takes linear time to find the boundary segments and that it takes constant time to find the area of a region between two boundary segments then the running time of Algorithm 3 is as follows. Let n and m be the number of boundary segments in S and T,

Algorithm 3 Simple breakpoint method

```
Transform $S$ such that $S(0)$ is before $T$ regarding field lines.
Find sets of boundary segments $S_b$ and $T_b$.
Initialize an empty set of breakpoints $B$.
for all $a \in S_b$ do
  for all $b \in T_b$ do
    if $\mathbf{R}(a, b)$ is non-empty then
      Find timing value $t$ where $a$ and $b$ starts intersecting.
      Find timing value $t'$ where $a$ and $b$ stops intersecting.
      Save the timing values $t$ and $t'$ as breakpoints in $B$.
    end if
  end for
end for
Initialize the smallest overlap found, $min = \infty$.
for all $t \in B$ do
  if $S(t)$ is within the vector field region then
    Set $tmp$ = area of $S(t) \cap T$ (using the Intersection Theorem).
    if $tmp < min$ then
      Set $min = tmp$.
    end if
  end if
end for
Return $min$.
```

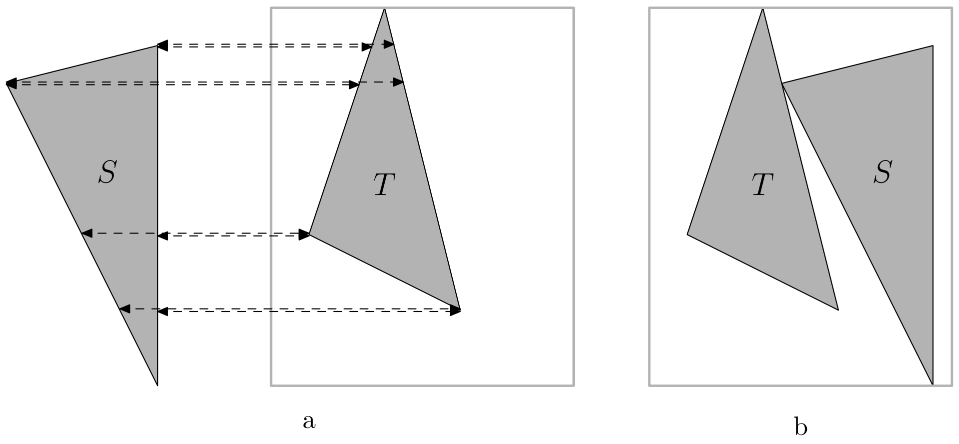


Figure 4.8: A small example of how the horizontal placement of S is found in relation to T within the boundaries of a vector field region (the box) using Algorithm 3. The algorithm will find 8 breakpoints (the lengths of the double arrows). Translating S according to each of these breakpoints reveals a non-overlapping solution within the boundaries of the box (the last breakpoint).

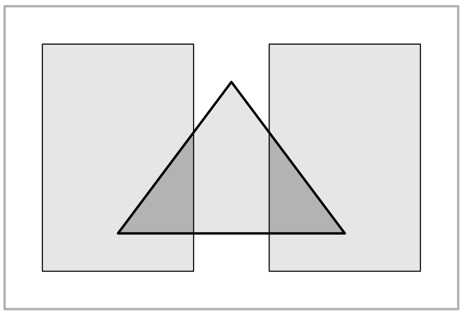


Figure 4.9: The horizontal position of the triangle above does not correspond to any breakpoint. The area of overlap is clearly increased if the triangle is shifted left or right. Therefore Algorithm 3 cannot find the optimal horizontal translation of the triangle (which is to leave it where it is).

respectively, then there are $O(nm)$ checks for breakpoints which produce at most $k = O(nm)$ breakpoints. For each of these breakpoints the calculation of overlap can be done in $O(k)$ time (only the segment pairs which also cause breakpoints need to be considered to contribute to the overlap). This reveals a total running time of $O(nm + k^2)$ or $O(n^2m^2)$ in the worst case.

Finding the overlap at a single breakpoint is worst case $O(nm)$ which is also a lower limit for the worst case of the problem of finding the intersection area of two polygons.

The algorithm does not necessarily find the best placement of S since this is sometimes between two breakpoints. An example of such a situation is given in Figure 4.9. But the algorithm still has a very nice property as the following lemma shows.

Lemma 3. *Suppose we are given a vector field region in which two shapes S and T are placed. If there exists a transformation of S, where S does not overlap T then either Algorithm 3 will find it or it is located where S touches a limit of the vector field region.*

Proof. Assume we are given a placement of S which does not overlap with T and which does not correspond to a breakpoint as found in Algorithm 3. Then it is clearly possible to either move S until it touches T, which corresponds to a breakpoint, or until it touches a limit of the vector field region. In both cases no overlap is involved and the algorithm would have tried the placement as one of the breakpoints. This contradicts the initial assumption. $\square$

The running time of Algorithm 3 is quite good when compared to a standard overlap algorithm which could also be used at each breakpoint. An efficient implementation might run run in $O((n + m) \log(n + m))$ on an average basis (worst case is still $O(nm)$) resulting in a running time of $O(k(n+m) \log(n+m))$ if k breakpoints are found. Assuming k is in the order of $O(n + m)$ this is a running time of $O((n^2 + m^2) \log(n + m))$. Algorithm 3 can handle it in $O(n^2 + m^2)$.

But in some cases we can make an even smarter algorithm. It depends on whether we can express how the area of a given segment region grows with respect to the transformation time t. In a horizontal vector field this reveals an algorithm which is faster than Algorithm 3 and which is guaranteed to find a minimum (Section 4.4). First we are going to present the algorithm in a more general setting as seen in Algorithm 4.

Algorithm 4 Iterative breakpoint method

```
Transform $S$ such that $S(0)$ is before $T$ regarding field lines.
Find sets of boundary segments $S_b$ and $T_b$.
Initialize an empty set of breakpoints $B$.
for all $a \in S_b$ do
  for all $b \in T_b$ do
    if $\mathbf{R}(a, b)$ is non-empty then
      Find timing value $t$ where $a$ and $b$ starts intersecting.
      Find timing value $t'$ where $a$ and $b$ stops intersecting.
      Mark the breakpoints as primary and secondary breakpoints respectively.
      Save the timing values $t$ and $t'$ as breakpoints in $B$.
    end if
  end for
end for
Sort the breakpoints in $B$ in ascending order.
Initialize the smallest overlap found, $min = \infty$.
Initialize an intersection area function, $Int(x) = 0$.
for all $t \in B$ do
  if $t$ is a primary breakpoint then
    A new segment region is "growing". Find a function $A(x)$ which reflects this.
  else if $t$ is a secondary breakpoint then
    An existing segment region is "changing". Find a function $A(x)$ which reflects this.
  end if
  Set $Int(x) = Int(x) + A(x)$.
  Set $tmp$ = the minimum of $Int(x)$ in $[t, t']$, where $t'$ is the next breakpoint.
  (disregarding special cases e.g. to ensure that $S$ is inside the vector field region)
  if $tmp < min$ then
    Set $min = tmp$.
  end if
end for
Return $min$.
```

There are three major stages in this algorithm: Collecting breakpoints, sorting breakpoints and iterating breakpoints. In the first stage the breakpoints are explicitly marked as either primary or secondary. This can be useful information in the third stage, where the essence of this algorithm lies. Unlike Algorithm 3 the breakpoints are sorted in the second stage. By doing this the third stage becomes surprisingly easy. It turns out that we no longer need to use the Intersection Theorem repeatedly. This is due to the fact that the involved pairs of boundary segments at time t_1 is a subset of the involved pairs of boundary segments at time t_2 if $t_2 > t_1$.

Note that for some boundary segments it might be convenient/necessary to introduce extra breakpoints between the primary and secondary breakpoints to be able to make more changes to the area function. But in the following we are going to assume that two are enough.

Algorithm 4 finds the transformation of S that results in the smallest possible intersection area with T. The efficiency of the algorithm depends on how well segment region functions can be added and how easy it is to find the minimum of the resulting function within a given interval. Detailed examples are presented in the following sections.

Now there is one final general observation to do.

Lemma 4. *There exists only two classes of two-dimensional shape-preserving vector fields (and regions): Rotational and translational vector fields.*

Proof. This follows directly from a known fact in Euclidean geometry, in which there only exists three kinds of shape-preserving transformations: Rotation, translation and reflection. We have already seen the vector fields related to rotation and translation. It is not possible to construct a vector field which matches reflection. This is intuitively clear since reflection requires one to lift the shape and turn it around which clearly requires the freedom of three dimensions to maintain the shape in the process. □

It is not unlikely that non-shape-preserving vector fields could be useful, but they are not considered any further in this thesis.

4.4 Translation of polygons

We have already seen how to calculate the intersection area of two polygons in a horizontal vector field. The only thing left to be able to use Algorithm 3 is to find the breakpoints and this is a simple matter. Therefore we already have an algorithm which is able to find non-overlapping solutions to the translation of a given polygon (if they exist).

To be able to use Algorithm 4 we need to do some additional calculations. Finding the breakpoints is no different than before and sorting them is no problem. The new part is that we need to find functions that tell us how the area of a region changes with translation. In Figure 4.10 the various stages of a segment region are illustrated. It also shows the placement of the segments at the primary and secondary breakpoints as they are described in the algorithm. Between the primary and the secondary breakpoint the segment region is a triangular area (Figure 4.10b). After the secondary breakpoint it changes to a trapezoidal area. We have seen these areas earlier, but this time we need to describe the areas with respect to the time of the translation.

The area of the trapezoid is very easy to describe. It is simply the area of the triangle in Figure 4.10c plus a simple linear expression based on the height of the trapezoid. Using the

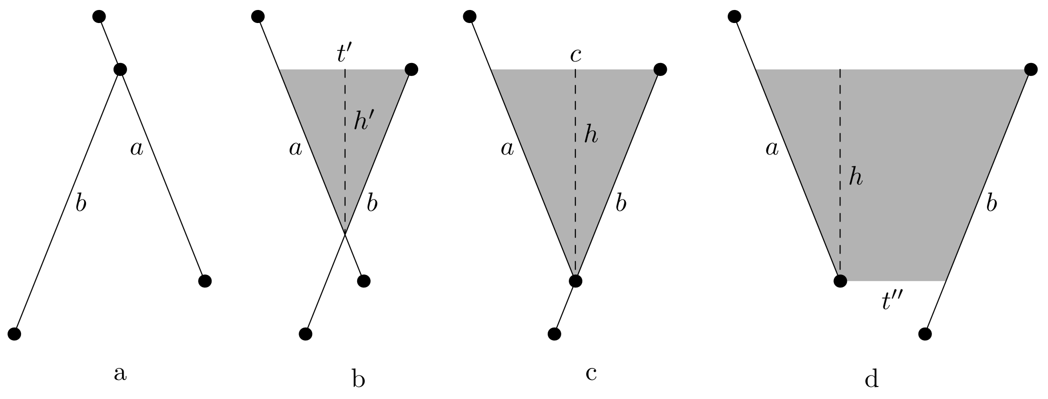


Figure 4.10: a) The segment region is empty until the primary breakpoint. b) The area is growing quadratically between the breakpoints. c) At the secondary breakpoint the area growth changes... d) to grow linearly.

variables shown in the figure,

$$A_s(t'') = ht'' + \frac{hc}{2},$$

where A_s is the area of the segment region after the secondary breakpoint.

The triangle is a bit harder. The important observation is that the height h' of a triangle must be $\frac{h}{c}t'$. Then it easily follows that

$$A_p(t') = \frac{h't'}{2} = \frac{h}{2c}(t')^2,$$

where A_p is the area of the segment region between the primary and the secondary breakpoint.

Now the above functions depend on translation values t' and t'' which are conveniently zero at the primary and secondary breakpoints respectively. To add such functions we need to formulate them using the same translation offset value t. If we let t_p, t_s be the translation distances to the involved primary and secondary breakpoints (the breakpoint time values in the algorithm) then we can do this by simply letting $t' = t - t_p$ and $t'' = t - t_s$ in the above formulas. This reveals the following two formulas for triangles and trapezoids respectively,

$$\begin{aligned} A_p(t) &= \frac{h}{2c}(t - t_p)^2 = \frac{h}{2c}(t^2 - 2t_pt + t_p^2) = \frac{h}{2c}t^2 - \frac{ht_p}{c}t + \frac{ht_p^2}{2c}, \\ A_s(t) &= h(t - t_s) + \frac{hc}{2} = ht - ht_s + \frac{hc}{2}. \end{aligned}$$

These functions can be added such that the result is never worse than a quadratic function for which we can easily find a minimum.

Now, imagine we are translating a polygon P over a polygon Q to find an optimal placement (minimum overlap). Finding the breakpoints is easy and can be done with a few comparisons and calculations for each pair of edges (one from P and one from Q). Assume there is n edges in P and m edges in Q. Finding the breakpoints then takes time $O(nm)$. In the worst case breakpoints are produces by all pairs of edges giving us $k = O(nm)$ breakpoints. Sorting the breakpoints takes time $O(k \log k)$. Constant work is done at each breakpoint (adding a function and finding a minimum) and therefore the total running time will be $O(nm + k \log k)$.

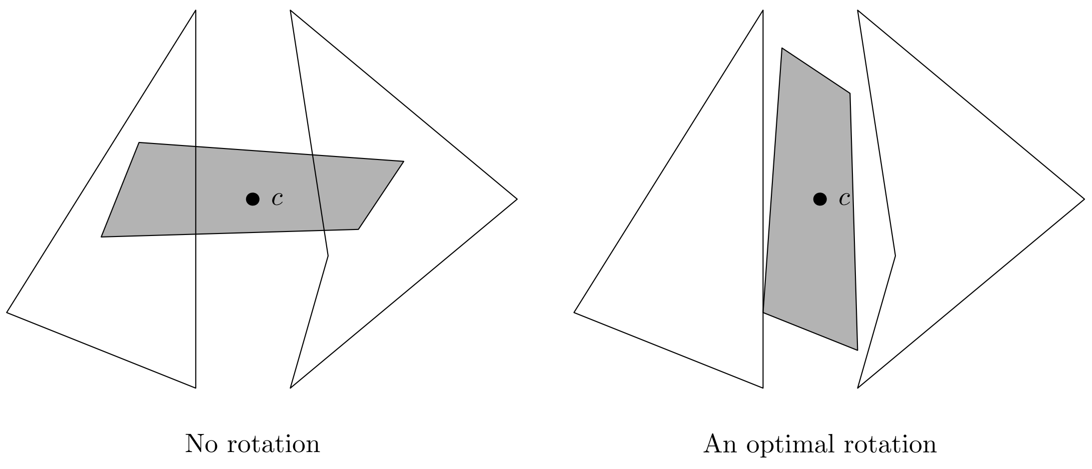


Figure 4.11: A simple example of a rotation yielding a placement with no overlap.

Although the above calculations are based on a horizontal translation, an algorithm for vertical translation easily follows. Using appropriate area calculations it is possible to do the translation in any direction. More generally it is only the area calculations that is the reason to restrict the shapes to be polygons. Algorithms could easily be made to translate other shapes e.g. circles, but it might not be possible to add the area functions in an efficient way thereby complicating the calculation of overlap at a breakpoint and especially the search for a minimum. This problem is also one of the main concerns in the following section.

4.5 Rotation of polygons

We have efficiently solved the problem of finding an optimal translation of a polygon. A very similar problem is to find the optimal rotation of a polygon i.e. how much is a polygon to be rotated to get the smallest possible overlap with all of the other polygons. A simple example is given in Figure 4.11.

This problem corresponds to the rotational vector field and is basically no different than translation apart from the area calculations. Unfortunately there are a couple of extra issues which need to be handled.

First of all the edges of a polygon cannot be directly adopted as boundary segments since some edges can have both positive and negative parts. These edges are quite easy to identify and they can always be split into two parts. This is illustrated in Figure 4.12. Note that neutral boundary segments are always singular points since a straight line can never overlap a field line.

Another problem is that the cut line(s) can rarely be placed without cutting some polygons (especially the one to be rotated). These shapes will then violate the condition of all shapes being inside the vector field region. It turns out that this problem is not easily handled. In the following, we simply assume that the cut line is placed as was shown earlier in Figure 4.4b and that no shapes crosses this line. In practice a simple solution is to cut all polygons into two parts at the cut line, but obviously this is not very efficient.

Polygons are normally given in Cartesian coordinates, but in the following context it will be much easier to work with polar coordinates. The center for these coordinates are natu-

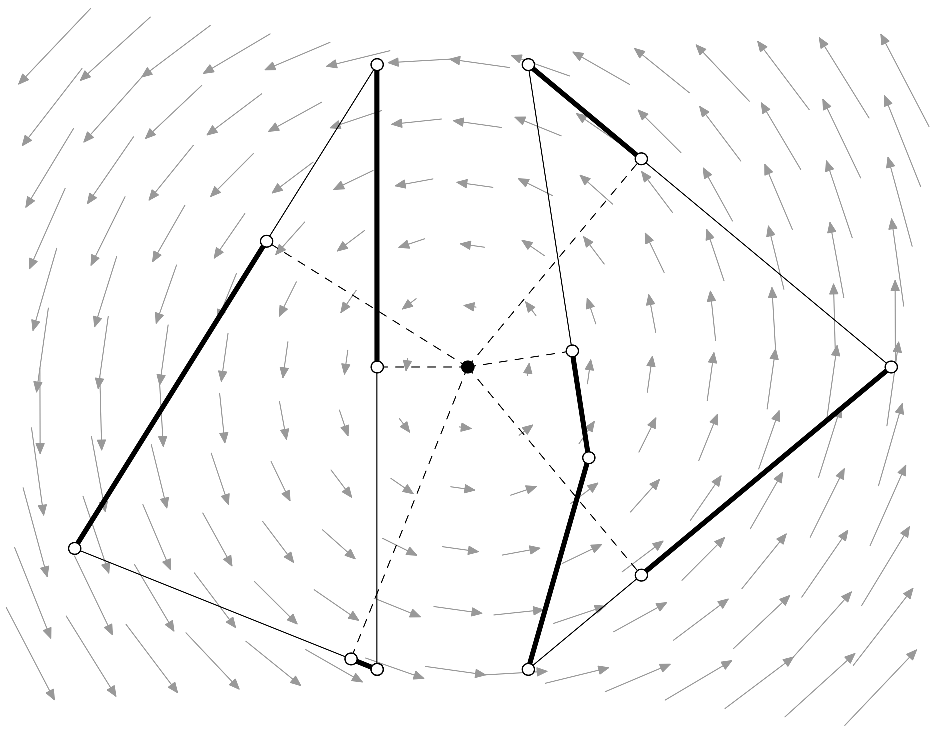


Figure 4.12: If an edge passes the center of rotation perpendicularly then it must be split into two parts. This is to ensure that all edges are positive, negative or neutral. The bold lines above are the positive boundary segments and the dashed lines show where it is necessary to split lines.

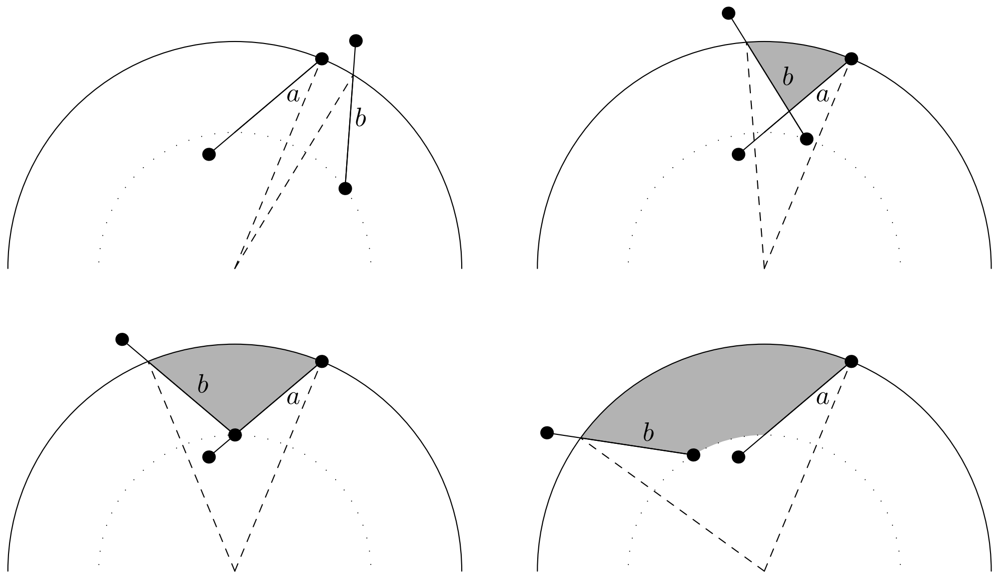


Figure 4.13: A segment region goes through 3 stages. 1. No overlap. 2. An almost triangular area (see the formula in the text). 3. A concentric trapezoid.

rally placed at the center of the rotation i.e. the center of the polygon to be rotated. These coordinates can be precalculated for the polygon to be rotated, but unfortunately they need to be calculated for all other involved polygons. Given Cartesian coordinates (x, y) the polar coordinates (r, θ) are given by,

$$r = \sqrt{x^2 + y^2}, \quad \tan\theta = \frac{y}{x}.$$

Just like translation a segment region $\mathbf{R}(a, b)$ between two boundary segments, a and b, goes through three different stages. These stages are illustrated in Figure 4.13 and 4.14. These figures also show that there are two basic variations of the area between two segments depending on how the segments meet.

An implementation of the simple breakpoint algorithm (Algorithm 3) is now straightforward. Finding the breakpoints is a question of calculating intersections between circles and segments and area calculations does not involve much more than an intersection calculation between boundary segments.

Now the big question is whether or not we can make an iterative algorithm as we could for the translational problem. To do this we first need to find functions expressing the area of the segment regions.

Area calculation is almost the same for the two basic variations. In both cases we need to calculate the area of a *circle segment*[1] and a triangle. This is illustrated in Figure 4.15 and 4.16. In the following we refer to the variables used in these illustrations. The easy part to calculate is the area of the circle segment, which is simply

$$A_s(\theta) = \frac{1}{2}r^2(\theta - \sin\theta).$$

[1]Given two points on a circle the circle segment is the area between the circle arc and the chord between the two points.

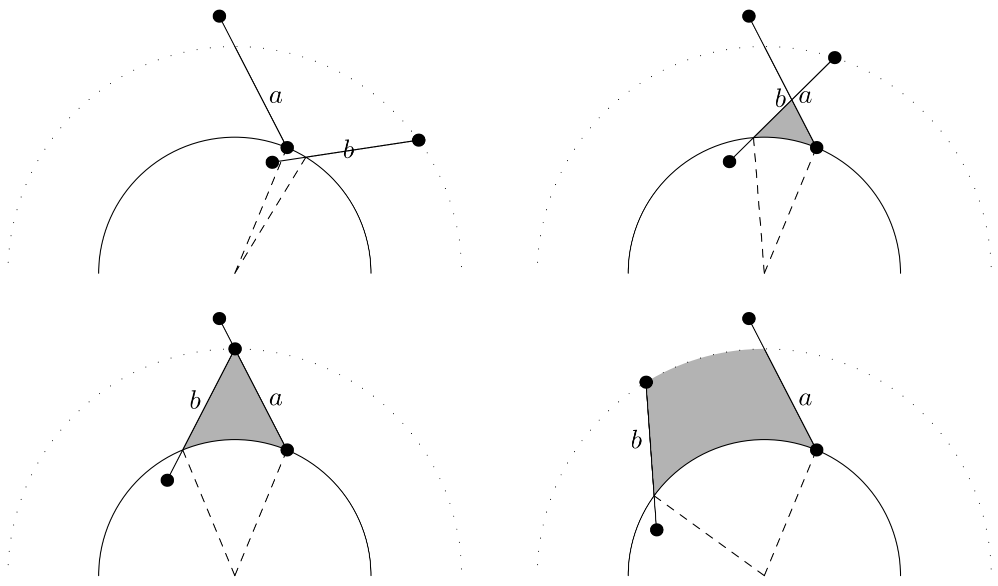


Figure 4.14: The area between line segments have two basic variations. In the above example the line segments meet first closest to the center. In Figure 4.13 it was the opposite. Note that the areas which need to be calculated have different shapes.

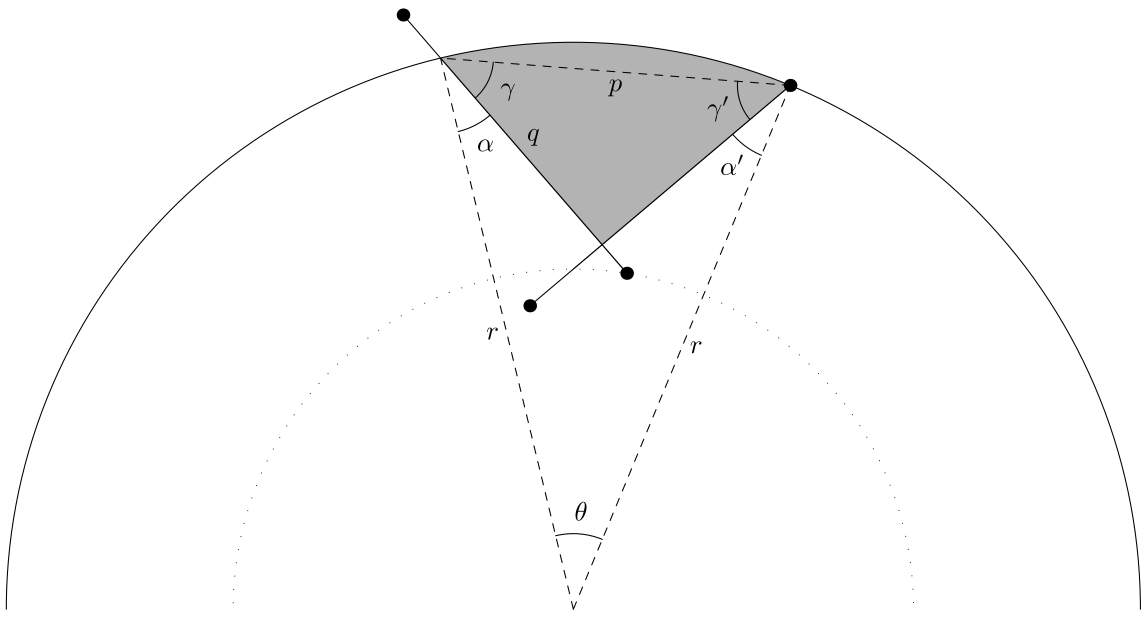


Figure 4.15: Variation 1 of the area between intersecting boundary segments.

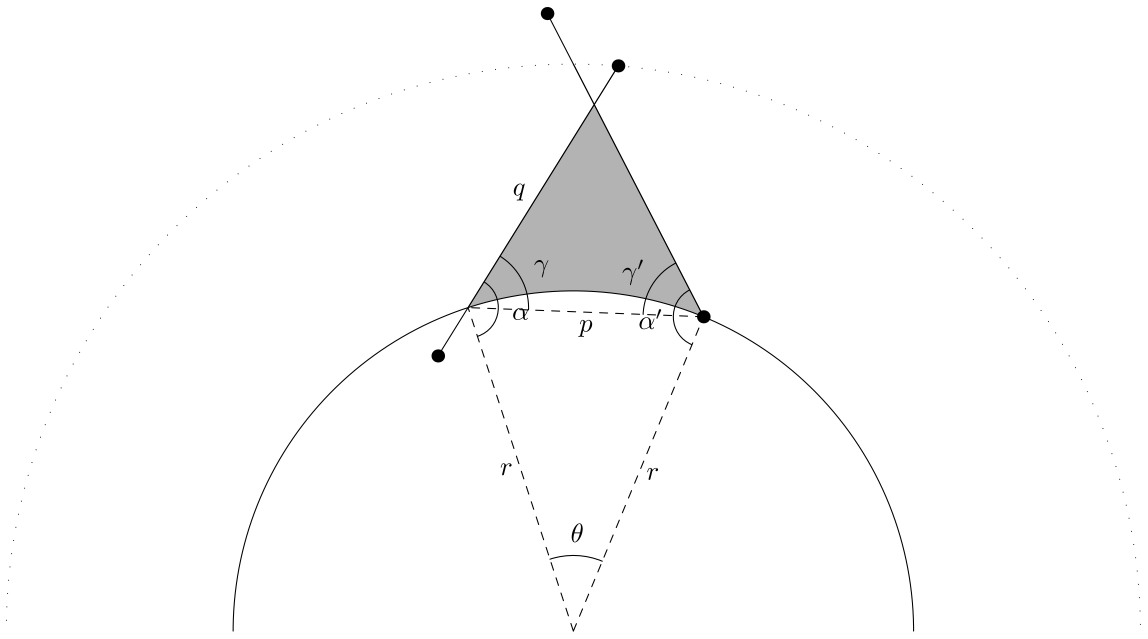


Figure 4.16: Variation 2 of the area between intersecting boundary segments.

The pq-triangles in the figures are a bit more difficult. Let $x = \frac{\pi-\theta}{2}$ which is the angle at the dashed corners on the figures.

Now we need γ and γ' shown on the figures and they can be calculated in the following way for both variations,

$$\begin{aligned}\gamma &= |x-\alpha|,\\ \gamma' &= |x-\alpha'|,\end{aligned}$$

Using the variables above and the sine relations the following expressions can be deduced,

$$\begin{aligned}p &= 2r\sin\frac{\theta}{2},\\ q &= p\frac{\sin\gamma'}{\sin(\gamma+\gamma')},\\ A_t(\theta) &= pq\sin\gamma.\end{aligned}$$

Now the area between boundary segments for variation 1 is $A_t(\theta)+A_s(\theta)$ and for variation 2 it is almost the same, $A_t(\theta)-A_s(\theta)$. When the segments no longer intersect then a constant can be calculated e.g. by using the above formulas and then a linear expression must be added. If θ is the rotation since the second breakpoint and r_1 and r_2 are the radii to the two breakpoints ($r_2 < r_1$) then it can be expressed as

$$A(\theta) = \frac{1}{2}(r_1^2 - r_2^2)\theta.$$

The functions needed to describe these areas vary a lot in complexity and this affects the running time of a rotational variant of Algorithm 4. The functions are not easily added with the exception of the linear expressions and finding the minimum between two breakpoints is far from trivial. We are not going to pursue this subject any further in this thesis.

Chapter 5

Miscellaneous Constraints

5.1 Overview

Until now we have shown how to handle a quite general nesting problem allowing free translation, rotation and flipping. As mentioned in Chapter 2 there can be a multitude of extra constraints for an industrial nesting problem depending on the industry in question. We believe that our solution method can handle many of these constraints without a lot of hassle. To substantiate this claim this chapter will present and discuss various constraints. For most of them this includes a description of how to incorporate them to be handled by our solution method. Some of them will also be part of our implementation.

Note that the majority of solution methods in the existing literature can only handle few of the following constraints. And more importantly, it is most often very unclear how they can be adapted to allow more constraints. This is especially true for the legal placement methods.

Also note that not all of the following are constraints in a strict sense. Some are better described as variants of the nesting problem which require special handling.

- Stencil shapes.
  The shapes of the stencils are never a problem. Any shape can be approximated with a polygon which we are able to handle. It can also have holes since this is clearly no problem for the translation and rotation algorithms.

  Note that it could be advantageous to preprocess holes in polygons filling them as densely as possible. This could be stated as a knapsack problem and would afterwards make the remaining problem easier.

- Fixed positions.
  Sometimes it is convenient to fix the position of a stencil. This is no problem for GLS since the offsets of the stencil can be set to the desired position and then never changed again while still including the stencils in any overlap calculations with other stencils.

- Material shape.
  In some cases the material needs to have an irregular shape. Combining the two first constraints this turns out to be an easy problem. Simply make a stencil corresponding to a hole and fix its position.

- Correspondence between stencils.
  In the textile industry the garment could have some kind of pattern. To allow this

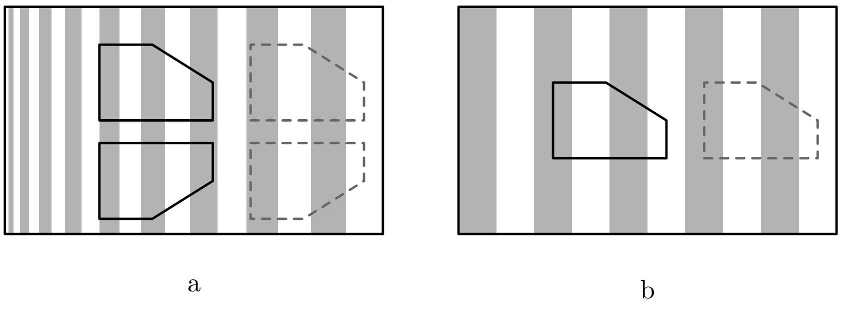


Figure 5.1: Examples of patterns and stencils requiring special attention due to pattern requirements. a) The two shapes can be freely translated vertically, but horizontally they have to follow each other if their patterns are supposed to be the same. b) If the shape needs to have the pattern in a certain way then it can only be translated to positions where the pattern is repeated.

pattern to continue across a seam it might be necessary to fix two or more stencils to always being moved the same amount horizontally and/or vertically. It might even be necessary to rotate them by the same amount. An example is given in Figure 5.1a.

For our solution method this is simply a question of involving more stencils in the translation/rotation algorithms.

- Pattern symmetries.
  In some cases, the placement options of stencils might be limited to some fixed set. The reason could (again) be patterns in a garment and the desire to get specific pattern symmetries or pattern continuations at seams.

  It is no problem for our translation algorithm to handle this. Just avoid finding the true minimum and instead find the minimum among the legal set of translations. This can e.g. be done by introducing special breakpoints corresponding to the distances for which overlap calculations must be done. Intervals could also be handled if necessary.

- Limited rotation angles.
  Some nesting problems might only allow a fixed set of rotation angles. This is most often the case in the textile industry since most kinds of fabric have a grain. Though often some level of "fine tuning" is allowed i.e. a rotation within e.g. $[-3°, 3°]$.

  No matter what sets or intervals of rotation angles are allowed it is no problem to handle in the rotation algorithm.

- Quality areas.
  In some cases, material and stencils can be divided into regions of different quality and quality requirements (respectively) with the purpose that a legal placement adheres all quality requirements. An example and a surprisingly easy way to handle this can be found below in Section 5.2.

- Margins between polygons.
  Some cutting technologies require a certain distance between all stencils. This seems to be a quite easy constraint since polygons can simply be enlarged by half the required

distance, but it turns out that there is a few pitfalls. This is described and handled in Section 5.3.

- Folding.
  Now, imagine a sheet of fabric which is folded once. Some symmetric stencils can be placed at the folding such that only half of it is visible on the upper side i.e. only half of it is involved in the nesting given that it is placed somewhere along the folding. In some cases this might allow better solutions to be found. It is also possible to allow non-symmetric stencils to be placed at the folding, but it will invalidate other areas of the fabric and it will complicate the cutting process.

  Our algorithm easily allows a stencil to only being allowed to be moved along the folding, but the difficulty lies in deciding when it is profitable to do so. We are not going to handle this constraint.

- Stencil variants.
  In the textile industry some stencils can be allowed to be divided into two (or more) parts since they can be stitched together at a later stage. Naturally they fit very well together so it might not be an advantage to split them since it also makes the problem harder (more stencils).

  This is also very close to the concept of meta stencils suggested by various authors [4, 1] i.e. the pairing of stencils to reduce the complexity of the problem. Dynamic pairing and splitting of stencils is probably an improvement to most solution methods, but the strategies for finding good pairings are not trivial. We are not going to discuss it any further in this thesis.

The final couple of items in this list is not really constraints. They are variations of the nesting problem, which are both interesting and very relevant in some industrial settings.

- Repeated pattern nesting.
  This variant of strip packing was mentioned in Chapter 2. We are now ready to discuss it in more detail in relation to our solution methods. Repeated pattern nesting allows stencils to wrap around the edges of the material. The reasoning for allowing this is that sometimes one needs to nest a few stencils which is then going to be cut repeatedly from a long strip of material. An example is given in Figure 5.2. Note that the height is fixed in the example. One could also allow the pattern to repeat both vertically and horizontally.

  It is surprisingly “easy” to alter our solution method to allow repeated patterns. Translation becomes similar to rotation in the general case since rotation already works with a repeated pattern. If the material is wrapped around a cylinder this becomes even more evident. Minor issues must also be handled e.g. to handle rotation, but there are no serious problems with the repeated pattern variation of strip packing. Not even if one wants to do it both horizontally and vertically.

- Minimizing cutting path length. When an efficient nesting has been found and it is ready for use, another interesting problem turns up. The stencils now need to be cut out of the material, which can happen in different ways depending on the material and the cutting technology. Metal can in some cases be punched out of the material, but often some kind

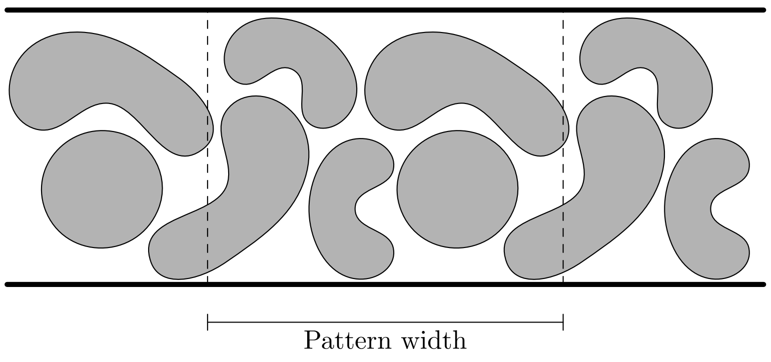


Figure 5.2: An example of repeated pattern nesting in 1 dimension. The pattern width is to be minimized, but unlike strip packing the stencils can wrap around the vertical limits.

of laser, water or plasma cutter is used. Other technologies apply in the textile industry, but most often there is a need for a cutting path. To save time it is important that the cutting path is as short as possible.

The length of the cutting path depends on the placement found in the nesting process. One of the original goals of this thesis was to examine whether the minimization of the cutting path length could be integrated with the nesting algorithm. This goal was discarded in favor of 3D nesting, but in Section 5.4 we describe an algorithm to simply find the shortest cutting path for a given placement.

5.2 Quality areas

The leather nesting industry poses one of the most interesting constraints. Imagine that we are going to make a leather sofa and that we are given an animal hide so that we can cut out the needed parts. The animal hide is of varying quality and we need to make sure that the good areas of the hide are used in the most demanding areas of the sofa.

Now assume that the animal hide can be divided into a number of areas which correspond to different quality levels $Q_1, ..., Q_n$. Assume that Q_1 is the best quality. If we divide the stencils into areas of minimum required quality levels then we can formulate the problem we need to solve: Pack the stencils within the material (the animal hide) so that no stencils overlap and so that the quality requirement Q_i of any part of any stencil is the same or worse than the quality of the material Q_j i.e. $i \geq j$. An example of a piece of material with different levels of quality is given in Figure 5.3.

Although this seems like a complicated problem, it can actually be solved with the help of the constraints already described by doing the following.

- Divide each stencil into a set of stencils corresponding to each quality area. Fix the interrelationship of these stencils i.e. when one is moved/rotated then the others follow (they should have a common center for rotation).
- Add a stencil with fixed position corresponding to each quality area of the material.
- Alter the transformation algorithms to only include segment pairs which violate the quality requirements.

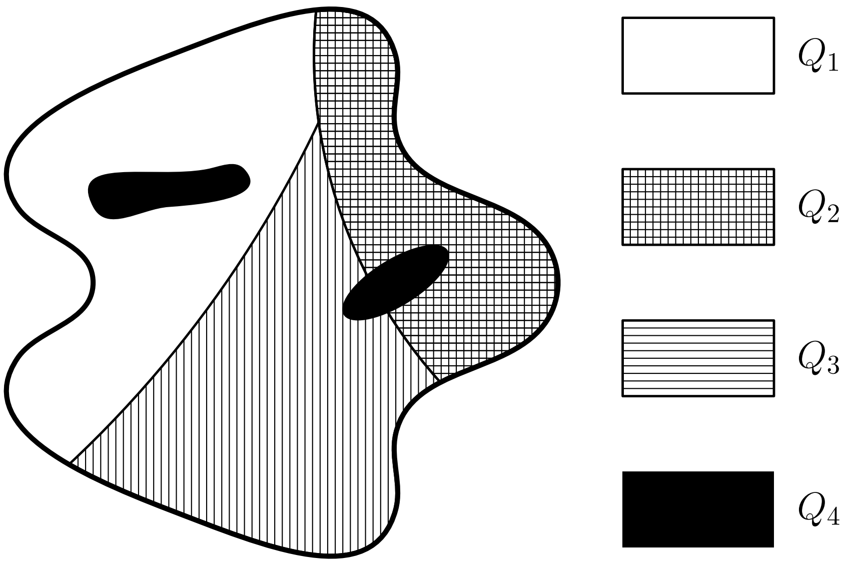


Figure 5.3: A piece of material, e.g. an animal hide, can have areas with different levels of quality. The stencils to be placed can have different quality requirements — even a single stencil can be divided into areas with specific quality requirements.

5.3 Margins between polygons

Some cutting technologies require a certain distance between the stencils as illustrated in Figure 5.4a. In the following we will assert that the required distance between any pair of polygons is the same. This assertion is essential for all results in this section.

5.3.1 A straightforward solution

The obvious way to get around this requirement is to enlarge all stencils by adding a margin which has half the width of the required distance. This is e.g. suggested by Heckmann and Lengauer [35] in an article focused on the textile industry, but they do not specify how to do this in practice. They simply state that one should calculate an enclosure where all edges have a distance of *at least* half of the required distance from the edges of the original stencil.

Their illustration of such an enclosure seems to indicate that they do this as illustrated in Figure 5.4a. Each edge in the original stencil is translated half the required distance between stencils and these edges are then stretched/shortened to make the enclosure. It is quite easy to calculate the corner points of this enclosure, but it is also obvious that this enclosure is not a perfect approach. This is emphasized in Figure 5.4b where two stencils are forced to be farther apart than necessary. The angle α determines how much farther. This is illustrated in Figure 5.5a where the distance x can be expressed as:

$$x = \frac{M}{\sin\left(\frac{\alpha}{2}\right)}, \qquad 0 < \alpha \leq \pi$$

$$x = \frac{M}{\sin\left(\frac{\alpha-\pi}{2}\right)}, \quad \pi \leq \alpha < 2\pi$$

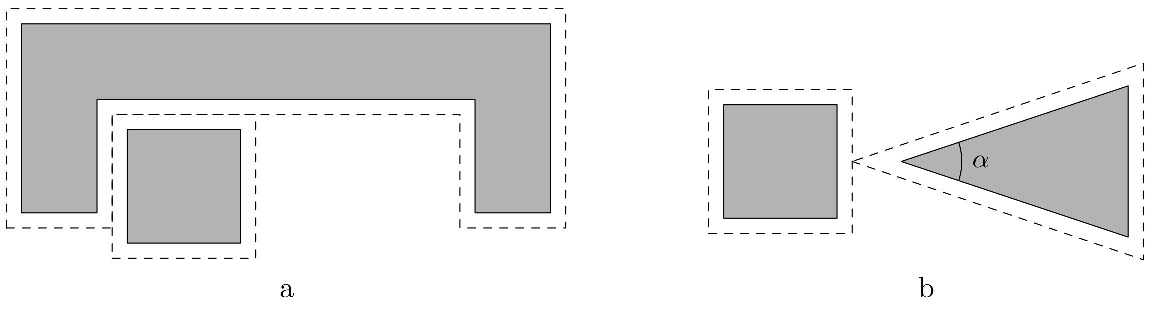


Figure 5.4: a) A simple enlargement of two polygons can guarantee some required minimal distance between them. b) A simple enlargement can force polygons to be farther apart than necessary

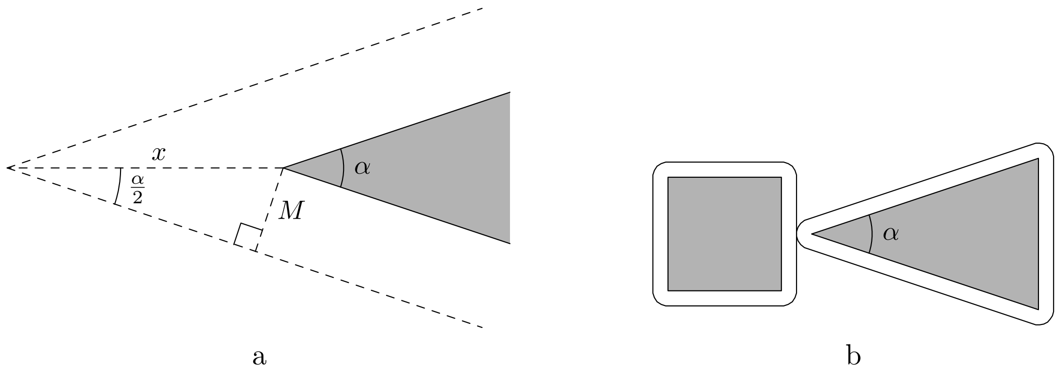


Figure 5.5: a) The distance x should be close to the distance M, but this is clearly not the case. b) The problem can be solved by rounding the corners

The value M is the width of the added margin to the stencil and optimally x should have the same value. Unfortunately $x \to \infty$ when $\alpha \to 0$. The same is true for $\alpha \to 2\pi$, but this is not the only problem with the latter formula since it is actually not correct. When $\alpha > \pi$ (concave corners) the length of the neighboring edges also need to be taken into account. At this point it only causes the enlargement to be bigger than necessary (not always), but later it even causes too small enlargements. We will deal with it then and show a different way of handling the concave corners.

5.3.2 Rounding the corners

In the following we are focusing on the case of $\alpha < \pi$. An optimal enclosure, i.e. an enclosure that does not remove any configurations from the set of legal solutions, must have rounded corners as illustrated in Figure 5.5b. The rounding of a corner is simply a circular arc with radius M. Unfortunately we only allow polygons and therefore we need to make some approximation. We have three objectives for this approximation.

- It must contain the optimal enclosure.
- It should be as small as possible.
- It should introduce as few new vertices as possible.

In other words, we need to maximize the effect of using extra vertices. If we only use 1 vertex at a given corner then its optimal position will be as already shown in Figure 5.5a. The angle at this new corner is identical to the angle α in the original stencil. When using more than 1 vertex then we would like the new angles to be as large as possible. It turns out that this can be done quite easily as illustrated in Figure 5.6.

The span of the round corner is $\alpha' = \pi - \alpha$ and given V extra vertices we get $x = \frac{\alpha'}{V}$. Now due to the right angles it can easily be shown that all the new angles are $\pi - x = \pi - \frac{\alpha'}{V}$. The distance from the original corner point to each of the new corner points are the same and using sine relations this can be expressed as $\frac{M}{\sin(\frac{\pi - x}{2})}$.

5.3.3 Self intersections

The above approach works fine when observing a single corner point, but problems arise when we look at the entire polygon. The first problem is quite obvious. Some polygons when enlarged will self intersect. This is a problem which can basically be solved in two ways. Either the self intersections are removed or else they are ignored. If they are ignored then all other algorithms will be required to work correctly with self intersecting polygons.

The second problem is a bit more subtle. When enlarging a polygon the rounded corner of a convex corner can interfere with a concave corner in such a way that the polygon is not only self intersecting — it also no longer contains the optimal enclosure. An example can be seen in Figure 5.7a. The problem is caused by the assumption that all of a rounded corner is part of an optimal enclosure. The easiest way to avoid this problem is to add two vertices for each concave corner as illustrated on Figure 5.7b. They are placed with a distance of M in perpendicular directions to the two neighboring edges.

Now an algorithm can round the convex corners freely, place two points for each concave corner and finally remove the self intersections. Note that this algorithm is stated without proof of correctness.

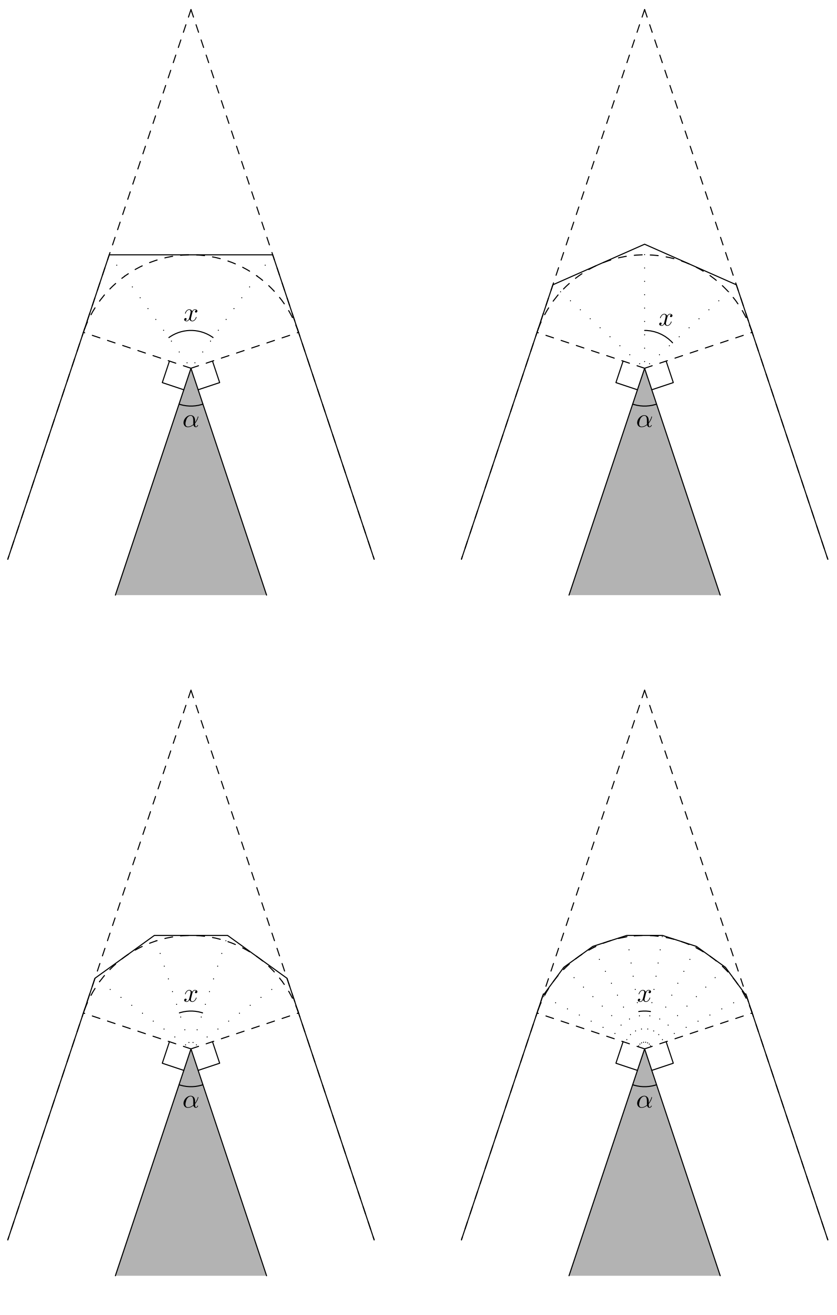


Figure 5.6: Examples of approximated rounded corners using 2, 3, 4 and 8 extra vertices. The dashed circular arc is the optimal enclosure.

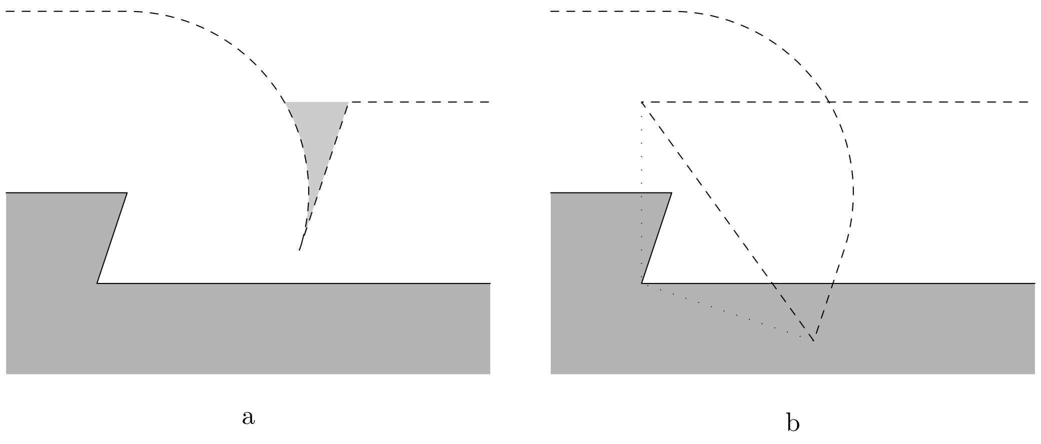


Figure 5.7: a) A naive rounding of corners can cause the resulting polygon to be illegal in the sense that the margin is no longer guaranteed. b) A subtle fix to this problem is the addition of two points for each concave corner (and a removal of the self intersection.

5.4 Minimizing cutting path length

When given a legal placement the problem of finding a short or even the shortest cutting path is a natural next step. In the following we will shortly describe how hard this problem is and how it can be solved.

Clearly all line segments in a given placement p must be cut. Now let G_p be the graph containing all of the lines which need to be cut as edges and all endpoints of lines as nodes. Edge weights are the Euclidean distances. Some line segments can overlap and thereby only need to be cut once. Assume that these have been combined in G_p such that all edges are unique. An example of this simple transformation is given in Figure 5.8a and b. Note that in practice precision problems can make it difficult to find overlapping lines.

Now, assume that this graph is connected which it certainly will be if it is a good solution to a nesting problem. A shortest cutting path might involve lines not present in this graph, but first we assume that the problem we need to solve is to find a path in this graph which visits all edges at least once. We would also like the path to end where it started although this might not be important in some cases. This problem is known as the *Chinese postman problem* (CPP) and it can be solved in polynomial time as shown by Edmonds and Johnson [27]. If all nodes have even degrees then it is even easier. This is one of the oldest problems in graph theory: Finding an Euler tour [30, 27], which is a very simple problem and a solution will only visit each edge exactly once. Therefore nothing would be gained by introducing new edges.

Most often all nodes do not have even degrees which means that a shorter path might be found by introducing new edges in the graph corresponding to moving the cutting head without cutting.

If we assume the graph is connected then the extra edges can be reduced to the edges that remains to make G_p a full graph. Any other edges would have to at least have one end on an existing edge creating a node of degree 3. The degree of any node must be even in a solution and therefore yet another edge must be connected to this node. Two non-cutting edges are now connected to this node which means that they could just as well be connected directly leaving the edge as it was.

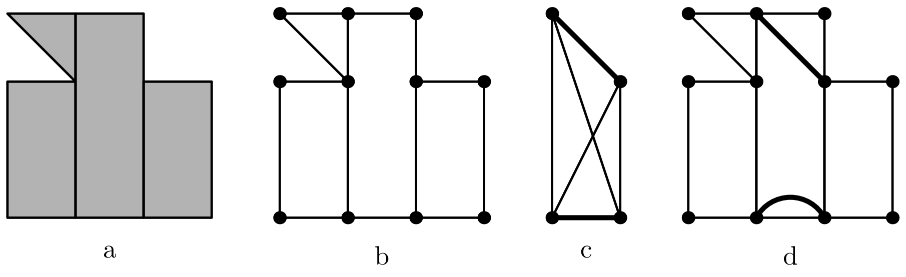


Figure 5.8: Various steps in the process of finding a shortest cutting path. a) A given nesting solution. b) A graph corresponding to the edges which need to be cut. c) The odd nodes of the graph with all vertices connected. The thick lines show a minimal 1-matching. d) The edges from the 1-matching is added to the original graph revealing a graph with even nodes only. A solution is now easily found.

The problem we end up with is a full graph where a cyclic path has to be found which must include a connected subset of the graph. This problem is a restricted version of the *rural (Chinese) postman problem* (RPP) and this is in general an $\mathcal{NP}$-hard problem [42], but if the subgraph G_p is connected then the problem essentially reduces to CPP [29] i.e. it can be solved in polynomial time using almost the same algorithm as for CPP [27] (actually the algorithm is even simpler because we are using a full graph with Euclidean distances). Four steps are necessary:

1. Generate a new full graph G_{odd} with the odd nodes from G_p (Figure 5.8c).
2. Find a minimum *1-matching* of G_{odd} (a 1-matching is a subset of edges of G_{odd} such that each node is connected to exactly one of these edges).
3. Add the edges from the 1-matching to the original graph G_p to make a new graph G'_p (Figure 5.8d).
4. Find an Euler tour in G'_p.

The hardest step is the second step, but according to Cook and Rohe [15] a 100000 node geometric instance can be solved in about 3 minutes on a 200 MHz Pentium-Pro.

The following observations have also been made regarding the cutting path. If a placement consists of highly irregular shapes which rarely share edges then the graph will have very few odd nodes making the problem very easy. If the graph is disconnected (e.g. because of some margin requirements for the solution) then not only is the problem $\mathcal{NP}$-hard, the additional edges are not necessarily between nodes either. Finally, in practice it might take time to switch the cutting machine on and off making it more expensive to add non-cutting lines. It might also take extra time to turn at corners.

Chapter 6

3D Nesting

6.1 A generalization

As noted earlier our fast translation/rotation methods are not restricted to two dimensions. In this Chapter we will describe this in more detail, but we will not generalize the proofs from Chapter 4.

Shapes in three dimensions can be defined just like we did for shapes in two dimensions i.e. a *3D-shape* is a volume in $\mathbb{R}^3$ with all boundary points in contact with the interior. It does not have to be a connected volume and it can contain holes (see a selection of objects in Figure 8.7). Boundary segments are replaced by *boundary surfaces*, segment regions (areas) are replaced by *surface regions* (volumes) and so forth. It is straightforward to design 3D-algorithms to translate in the x, y or z direction and to rotate around two axes which would be enough to generalize the two-dimensional neighborhood. The only problem is the calculation of the volumes between surface regions.

To keep things simple we are going to focus on translation and we are only going to handle polyhedra. These polyhedra must have convex faces (there is no problem in theory, but it will simplify an implementation). The boundary surfaces of a polyhedron is simply the faces of the polyhedron. Whether a boundary surface is positive or negative can be determined from the normal of the face. This naturally requires that the faces are all defined in the same direction related to the interior of the polyhedra.

Our only real problem is to calculate the region volume $\mathbf{R}(f, g)$ between two faces f and g when given a translation direction. This is the subject of the following two sections. The first section presents a simple approach (in theory) to the problem and in the second section this approach is refined to be more efficient.

6.2 The straightforward approach

Assume that translation is done along the direction of the x-axis. An illustration of a region volume is given in Figure 6.1. Note that the volume will not change if we simplify the faces to simply being the end faces of the region volume. This can easily be done by projecting the faces onto the yz-plane, find the intersection polygon and then project this back onto the faces. We can even take this one step further by triangulating the intersection polygon. This way we have reduced the problem to the calculation of the region volume between two triangles in 3D-space, and we know that the three pairs of endpoints will meet under translation. Sorted

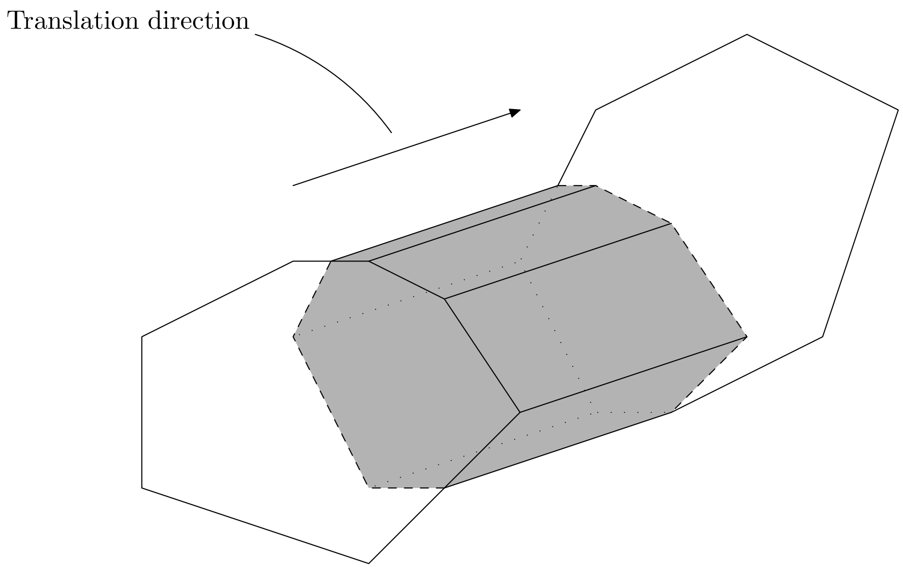


Figure 6.1: An illustration of the region volume between two faces. The faces are not necessarily parallel, but the sides of the region volume are parallel with the translation direction. The region volume would be more complicated if the two faces were crossing.

according to when they meet we will denote these the first, second and third breakpoint.

An illustration of the translation of two such triangles is given in Figure 6.2. Such a translation will almost always go through the following 4 phases.

1. No volume (Figure 6.2a).
2. After the first breakpoint the volume becomes a growing tetrahedron (Figure 6.2b).
3. The second breakpoint stops the tetrahedron (Figure 6.2c). The growing volume is now a bit harder to describe (Figure 6.2d). We will take care of it in a moment.
4. After the third breakpoint the volume is growing linearly. It can be calculated as a constant plus the area of the projected triangle multiplied with the translation distance since the corner points met.

We have ignored the special cases of pairs of corner points meeting at the same time. If the faces are parallel then we can simply skip to phase 4 and use a zero constant. If the two last pairs of corner points meet at the same time then we can simply skip phase 3. The question is, what to do if the first two pairs of corner points meet at the same time? The answer is to skip phase 1 and the reasoning is quite simple.

Figure 6.2c illustrates that it is possible to cut a triangle into two parts which are easier to handle than the original triangle. The upper triangle is still a growing tetrahedron, but the lower triangle is a bit different. It is a tetrahedron growing from an edge instead of a corner and it can be calculated as a constant minus the area of a shrinking tetrahedron.

The basic function needed is therefore the volume $V(x)$ of a growing tetrahedron (a shrinking tetrahedron then follows easily). This can be done in several different ways, but one of

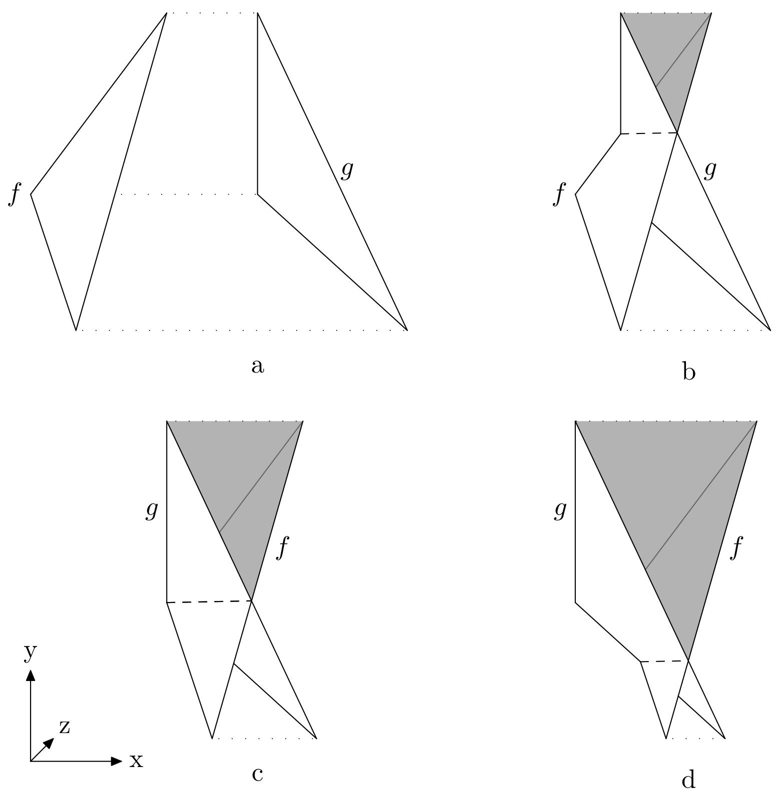


Figure 6.2: The x-translation of a triangle f through another triangle g, where the triangles have the same projection onto the yz-plane. The region volume $R(f, g)$ changes shape each time two corner points meet.

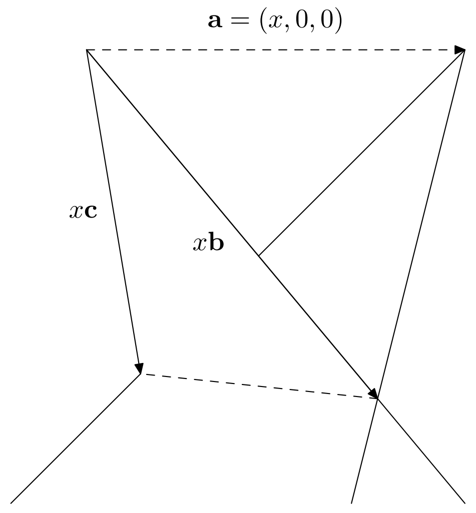


Figure 6.3: The volume of the above tetrahedron can be calculated from the three vectors **a**, **b** and **c**. In our case **b** and **c** are linearly dependent on x which is the length of **a** (and the translation distance since the tetrahedron started growing).

them is especially suited for our purpose. The following general formula can be used if given three vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ from one of the vertices of the tetrahedron,

$$V = \frac{1}{3!}|\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})|.$$

In our case one of the vectors is parallel to the x-axis corresponding to the translation direction. An example of three vectors is given in Figure 6.3.

Now, since all angles in the tetrahedron are constant when translating, then the length of all the vectors must be proportional. This is indicated in the drawing where x is the amount of translation. The vectors **b** and **c** do not change direction under translation, but are simply scaled by the value x. Using the formula above, we can derive the following formula for the change of volume when translating:

$$\begin{aligned} V(x) &= \frac{1}{3!}|\mathbf{a} \cdot (x\mathbf{b} \times x\mathbf{c})| \\ &= \frac{1}{3!}\left| x^3 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \cdot \left(\begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} \times \begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} \right) \right| \\ &= \frac{1}{6}\left|(b_y c_z - b_z c_y)x^3\right|. \end{aligned}$$

This function is not quite useful yet since it is based on the assumption that the translation is 0 when $x = 0$. We need to allow a translation offset t by replacing x with $x - t$.

$$V(x) = \frac{1}{6}\left|(b_y c_z - b_z c_y)(x^3 - 3tx^2 + 3t^2x - t^3)\right|.$$

Now it is a simple matter to use Algorithm 4 in Chapter 4 for translating 3D objects.

Now let us take a quick look at the running time. The volume function is a cubic equation for which addition and finding minimum are constant time operations. Assume we are given two polyhedra with n and m faces respectively (with a limited number of vertices each), then the running time of a 3D variant of Algorithm 4 would be $O(nm + k \log k)$ given that k intersecting pairs of faces are found. This is no different than the 2D variant for polygons. Naturally, the constants involved are quite a bit larger. Reducing the size of the constants considerably is the subject of the next section.

6.3 Reducing the number of breakpoints

The reader should now be convinced that translation in three dimensions will work, and so we go on to describe an alternative approach for the actual implementation — this method is without a triangulation step and thus will often result in fewer breakpoints. The faces are still required to be convex.

We first provide a view of how it works and afterwards we will derive the third degree polynomial for a growing pentahedron, as this turns out to be required in addition to the growing tetrahedron.

As with the previous method we only look at two faces, the face which is being translated, denoted "the moving face", and a face which our moving face will gradually pass through, denoted "the static face".

We start by finding the two end faces of the region volume as previously illustrated in Figure 6.1. As described earlier this can easily be done by projecting the faces onto the plane perpendicular to the translation axis, finding the intersection polygon, and then project this back onto the faces.

We define the 2D intersection polygon to be P. Furthermore we call the 3D projection of this polygon onto the moving face M_P and onto the static face for S_P.

The breakpoints can be found as the distance between equal (on the two axes not being the axis of translation) points in M_P and S_P. We denote these $b_0, b_1, \ldots, b_n$ enumerated according to their relative distance, i.e. b_0 is the first breakpoint and b_n is the last.

Since both M_P and S_P are end faces of the region volume then we can settle by looking at the intersection process of these two polygons (rather than the actual faces intersecting).

Two non-parallel planes will intersect each other in one unique line, see Figure 6.4. On the two planes we have placed M_P and S_P to show that the same is true for these two (convex) polygons, as long as we are in the interval where the two polygons are passing each other ($t \in [b_0, b_n]$).

If M_P and S_P are parallel then the interval of passing is zero ($b_0 = b_n$), and we simply skip the following and jump directly to the linearly growing volume.

As we move M_P along the translation axis, the line of intersection will also move (its length as a piecewise linear function). To simplify things we will start by looking at a polygon in 2D corresponding to the projection of M_P (or S_P) along the translation axis. At time b_0 this polygon is a single point (the point in P from which b_0 was calculated), see Figure 6.5a, at time b_n the polygon is P (Figure 6.5c) and at time b_i the polygon is given by all points in P for which the derived b_j is less than b_i (Figure 6.5b).

We focus on this line of intersection (which we refer to as $\mathcal{L}$) and note that it will go from breakpoint to breakpoint in ascending order, until it has left the polygon entirely. Remember that the polygon is convex i.e. $\mathcal{L}$ can intersect at most 2 breakpoints at the same time.

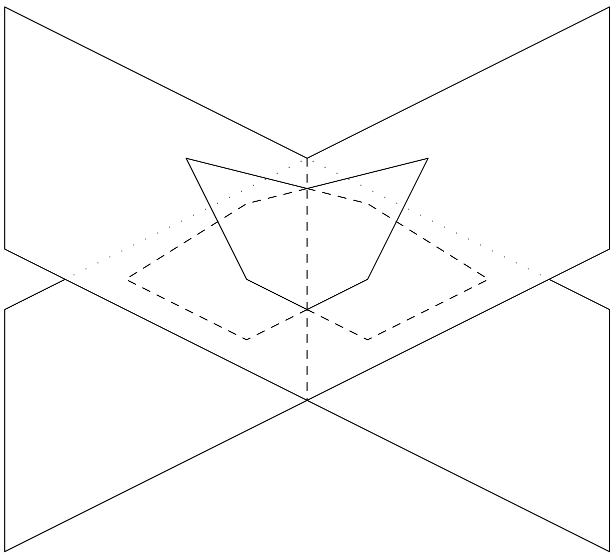


Figure 6.4: Just like two non-parallel planes have a single unique line in common so will two crossing faces have a unique line segment in common.

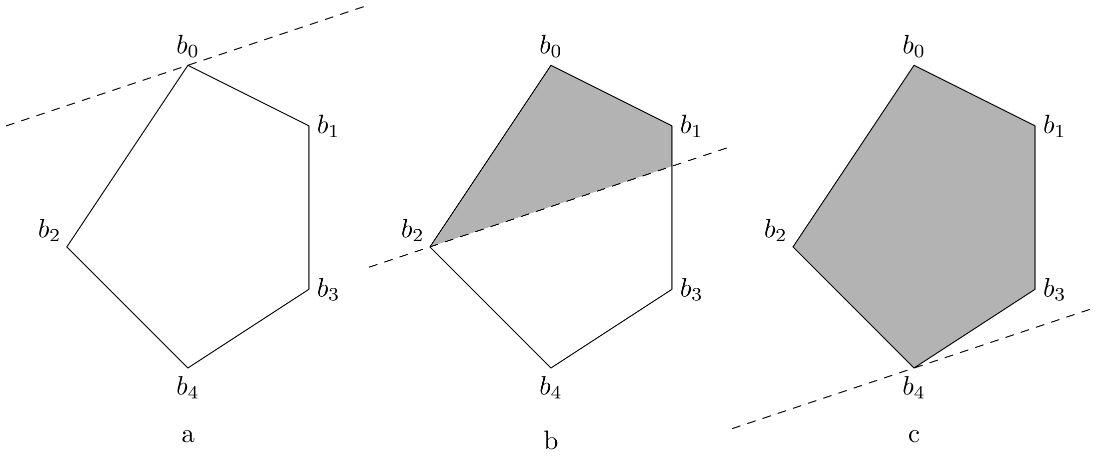


Figure 6.5: This is a 2D illustration of the process of two faces (with identical 2D projections) passing each other. The dashed line is the line of intersection from Figure 6.4. a) The first breakpoint when the two faces meet for the first time. b) At b_2 the projected polygon is given by all previous breakpoints and the intersection line. c) At b_4 the two faces have passed through each other entirely.

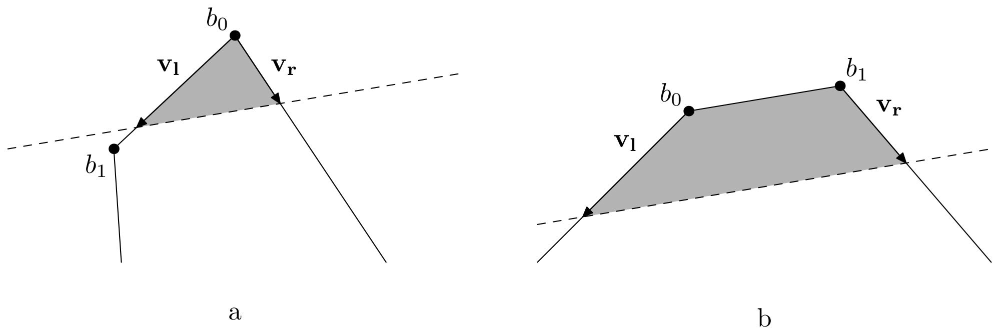


Figure 6.6: Two situations need to be handled at the initial breakpoint. a) The intersection line meets a single breakpoint b_0 and "grows" a triangle. b) The intersection line meets two breakpoints simultaneously and "grows" a quadrilateral.

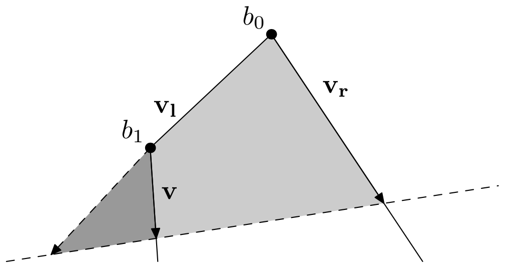


Figure 6.7: At each breakpoint after the initial breakpoint(s) the shape changes. This can be handled subtracting a new growing area (dark area) from the existing growing area (light and dark area).

Initially one of two situations can arise. Either we have a unique first breakpoint, or we have two. What happens in both situations is that the edges moving away from the breakpoint(s) (oriented toward the direction of $\mathcal{L}$) will start to build P. The two situations are illustrated in Figure 6.6 — we denote these two edges (vectors) $\mathbf{v_l}$ and $\mathbf{v_r}$.

Since P is convex then we know that all points passed by $\mathcal{L}$ would be contained in the triangle or quadrilateral spawned above (if $\mathbf{v_l}$ and $\mathbf{v_r}$ were scaled to intersect $\mathcal{L}$).

While $\mathbf{v_l}$ and $\mathbf{v_r}$ follows the exterior of P then we can easily calculate the correct area of the polygon we are building, and even create a closed formula expressing this area as a function of x. But at the next breakpoint this property no longer holds. What happens is that the edge in P which follows $\mathbf{v_l}$ (on the illustration, the following would also work if it had been $\mathbf{v_r}$) stops, and is replaced by a new one which we denote $\mathbf{v}$. If we continue to grow $\mathbf{v_l}$ then the area will be too large, and exactly how much is given by the triangle spawned by $\mathbf{v_l}$ and $\mathbf{v}$ (starting at the new breakpoint). To compensate we simply express this area as another function of x and subtract it from the first one (when we encounter this breakpoint). An illustration of this negation of area can be seen on Figure 6.7.

We iteratively repeat the previous step with $\mathbf{v}$ substituted for $\mathbf{v_l}$ (or $\mathbf{v_r}$) until we reach b_n, at which point we have covered the entire polygon.

The jump back to 3D can be done by realizing that what we express here is really just slices

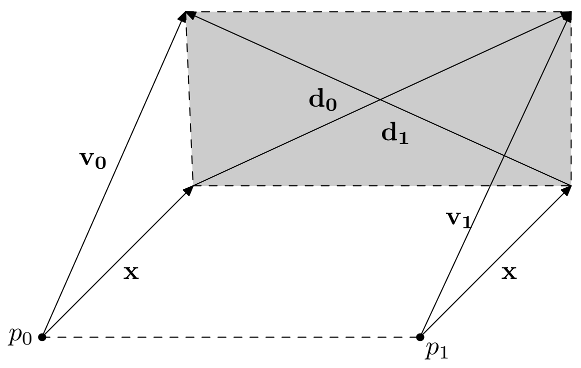


Figure 6.8: It is necessary to also be able to calculate the volume of a growing pentahedron as the one above. Note that $\mathbf{x}$ is a vector parallel to the x-axis. Its length is the translation distance.

added to the volume. By adding a vector parallel to the translation axis to each breakpoint (which also grows as a function of x), then instead of letting the triangle express the area for a given slice, then the tetrahedron will express the volume at that time.

We have already shown that the volume of a tetrahedron can be expressed as a third degree polynomial, but in the situation where we have two initial breakpoints with same offset, then we spawn a quadrilateral, and if we add vectors to the two breakpoints then we have 9 vectors, 6 vertices and 5 faces. This forms a pentahedron and knowing the rate of growth for the vectors we can arrive at a third degree polynomial.

First we remember that the area of a quadrilateral can be found as

$$A = \frac{1}{2}\mathbf{n} \cdot |\mathbf{d_0} \times \mathbf{d_1}|,$$

where $\mathbf{n}$ is the normal and $\mathbf{d_0}, \mathbf{d_1}$ are the two diagonals in the quadrilateral.

What we have is the two starting points of the pentahedron (p_0, p_1) and four vectors ($\mathbf{v_0}, \mathbf{v_1}$ and two times $\mathbf{x}$) starting at these points (two from each). The two points are static and the four vectors are all growing relatively to x. The $\mathbf{x}$ vector is parallel to the translation direction. If we set $\mathbf{x} = (1, 0, 0)$ then the two diagonals of the quadrilateral in the "back" of the pentahedron can be expressed with the translation distance x (see Figure 6.8)

$$\begin{aligned}
a &= p_0 + x\mathbf{v_0}, & b &= p_0 + x\mathbf{v_1} \\
c &= p_1 + x\mathbf{x}, & d &= p_1 + x\mathbf{x} \\
\mathbf{d_0} &= a - c, & \mathbf{d_1} &= b - d \\
&= (p_0 + x\mathbf{v_0}) - (p_1 + x\mathbf{x}), & &= (p_0 + x\mathbf{v_1}) - (p_1 + x\mathbf{x}) \\
&= p_0 - p_1 + x(\mathbf{v_0} - \mathbf{x}), & &= p_0 - p_1 + x(\mathbf{v_1} - \mathbf{x}) \\
&= \mathbf{k_0} + x(\mathbf{k_1}), & &= \mathbf{k_0} + x(\mathbf{k_2}).
\end{aligned}$$

So we get,

$$A(x) = \frac{1}{2}\mathbf{n} \cdot |(\mathbf{k_0} + x(\mathbf{k_1})) \times (\mathbf{k_0} + x(\mathbf{k_2}))|$$

Now $A(x)$ is not entirely what we need, because x is not necessarily the height of the volume. The height h of the volume is linearly dependent on x and can easily be found i.e. we know $h = sx$ for some constant s. Using this constant the following formula can be derived for the volume of the pentahedron,

$$V(x) \quad = \quad s \int_0^x \frac{1}{2}\mathbf{n} \cdot |(\mathbf{k_0} + x(\mathbf{k_1})) \times (\mathbf{k_0} + x(\mathbf{k_2}))| \, .$$

Finally there's just the question of what to do in the final breakpoint. Here the faces have fully passed each other, and thus the region volume will continue to grow following this formula $V(t_2) - V(t_1) = A(P)(t_2 - t_1)$ — what we do is simply to replace the previous formula with the one just given.

Chapter 7

Implementation Issues

7.1 General information

The implementation consists of two programs with the imaginative titles 2DNest and 3DNest — both around 3000 lines of code written in C++ making use of the standard template library (STL). They are for the most parts platform neutral, but for visualization we have used OpenGL and GLUT for 3DNest and the native GUI of Mac OS X for 2DNest, but they can both be used without their graphical interfaces. 2DNest is also able to output the solutions in postscript instead of on-screen.

The following sections describe various aspects of the implementations, starting with a short description of the available parameters for 2DNest. After that Sections 7.3 and 7.4 cover some key aspects of the implementation of GLS and the neighborhood search. Section 7.5 describes how to handle irregular materials and finally Section 7.6 describes a speed improving method attempted in the implementation. Section 7.7 is just a short note on how to best find the center of rotation for a polygon.

7.2 Input description

We have used XML as the input file format for 2DNest to specify a problem instance. This was chosen to provide flexibility throughout the life-time of the project and it has proven to be able to handle all of the changing requirements of our implementation. We have created tags in this format to represent arrays, associative arrays, floats, integers and strings. This allows us e.g. to have a polygon defined as an associative array with a key for the vertices (stored in an array) and other keys for other properties of the polygon. An example of an input file is given in Figure 7.1.

3DNest uses a de-facto input format from the rapid prototyping industry. The format is called STL which is an acronym for stereolithography. This is one of the earliest rapid prototyping techniques. The format is very simple and only contains a list of triangles and a face normal for each of them (one object per file).

7.3 Slab counting

In the penalty step of GLS we need to calculate a utility function for each pair of polygons, which is basically a scaling of the intersection area of the two polygons in question. This would

```
<dict>
        <key>Height</key>
        <float>30</float>

        <key>Width</key>
        <float>30</float>

        <key>Polygons</key>
        <array>
                <dict>
                        <key>Label</key>
                        <string>Triangles</string>

                        <key>Vertices</key>
                        <array>
                                <float>0.0</float> <float>0.0</float>
                                <float>5.0</float> <float>0.0</float>
                                <float>2.5</float> <float>2.5</float>
                        </array>

                        <key>RequiredQuality</key>
                        <integer>1</integer>

                        <key>Quantity</key>
                        <float>40</float>
                </dict>

                <dict>
                        <key>Label</key>
                        <string>Frame</string>

                        <key>OffsetX</key>
                        <float>5</float>

                        <key>OffsetY</key>
                        <float>5</float>

                        <key>FixedPosition</key>
                        <integer>1</integer>

                        <key>QualityLevel</key>
                        <integer>2</integer>

                        <key>Vertices</key>
                        <array>
                                <float> 0.0</float> <float> 0.0</float>
                                <float>20.0</float> <float> 0.0</float>
                                <float>20.0</float> <float>20.0</float>
                                <float> 0.0</float> <float>20.0</float>
                        </array>
                </dict>
       </array>
</dict>
```

Figure 7.1: An example of an input file for 2DNest.

amount to calling our intersection area function n^2 times. Not only is calculating the area of intersection for two polygons expensive, but it is also highly redundant, because most polygons should not overlap at all (remember that the penalty step is applied after running FLS). This is why it would be desirable to have knowledge about exactly which pairs overlap, and it is the topic of this section.

To keep things simple, the focus in this section will be on translation, but the discussion (and the algorithm) is also relevant for rotation.

There is at least two ways this knowledge can be stored/represented. Either as an $n \times n$ matrix of boolean variables (indexes into the matrix would be the two polygons) or as a set containing only the pairs that overlaps. The latter method will (except for worst-case scenario) be faster to traverse/exhaust in the penalty step, since the working set is assumed only to contain a few polygons. But the former has the clear advantage of allowing overlap-state changes in constant time, which is assumed to be the dominating operation for this cache.

When we step through the breakpoints as described in Algorithm 4 then we maintain a function that expresses the overlap of the translated polygon with all other polygons (for a given position). This function is the sum of several functions, each provided by one of the polygons being passed, and each function being the expression for the overlap of a single edge from that polygon.

Since the sum of all functions provided by one polygon evaluates to the overlap with only that polygon, we can keep this sum of functions separately for each polygon. Each time a new contribution is added, then we check if the function changes from or to zero, which should only happen when we enter or leave a polygon entirely.

This can be done without affecting the overall time-complexity of stepping through the breakpoints, since at each breakpoint it is only necessary to update one function (plus the sum of all functions) which can be done in constant time.

This approach will require that we introduce an epsilon, since the precision provided by floating points is not sufficient, and this may also open up the door to illegal solutions (although only with very small overlaps).

Alternatively we can look at the individual edges. We know if we sweep a line across a polygon then all positive edges will be "canceled" by negative edges. This follows from the alternating property in lemma 1 and it implies that it might be possible simply to count positive and negative edges as a part of Algorithm 4 and thereby be able to determine whether two polygons overlap. Unfortunately there is no 1:1 mapping of these edges, but if we focus on the sum of heights of the positive edges and the sum of heights of negative edges, then we can see that these must also be equal. More precisely, if we encounter a positive edge with a height of a, then we must also encounter negative edges that together span the same height. Should we meet further positive edges, before the first one is canceled, then these also need to be canceled.

A simple example of using the above property is shown in Figure 7.2. Only positive edges meeting negative edges and vice versa is counted in the example. This is sufficient to determine overlap. Note, that we do not give any formal proof of this algorithm.

This is better than having to test the intersection function for a zero result, but since edge heights are represented in floating points then we may still have a precision problem, which is why we wish to map the heights of edges to integers.

Given n edges we have $2n$ vertices (not necessarily unique) and thus there can be at most $2n$ different vertical positions, y_i, for these vertices. If we enumerate them in ascending order with respect to y_i, then we have our mapping. The height can now be obtained using the

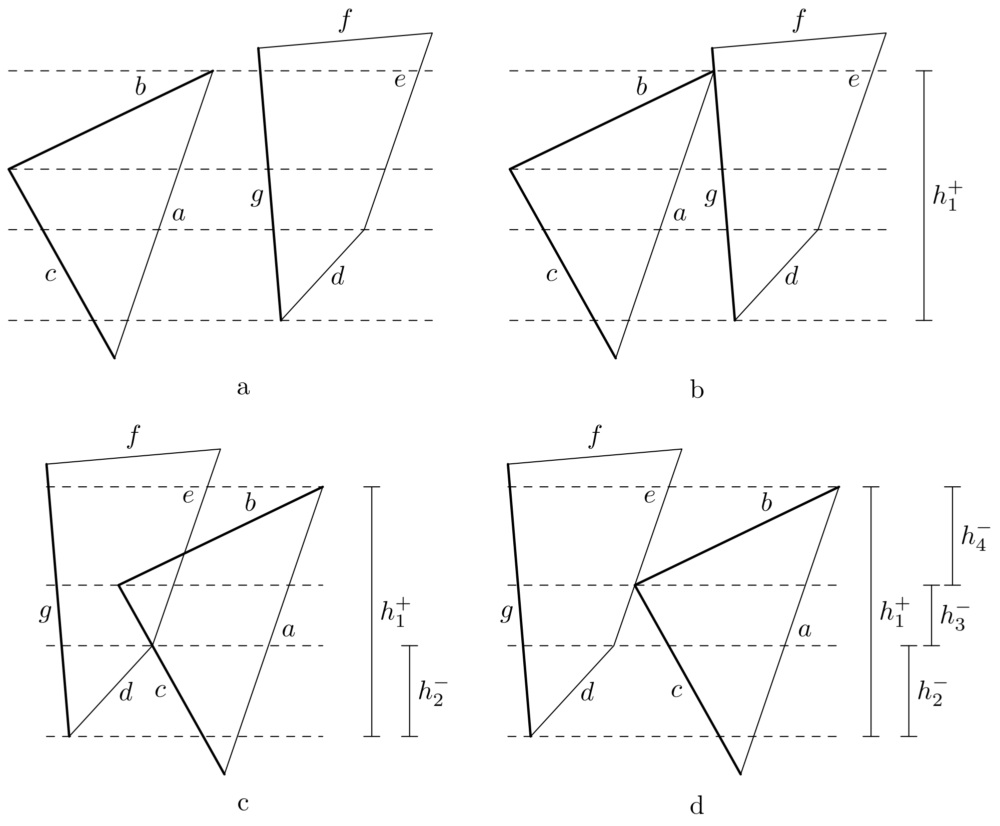


Figure 7.2: A simple example of using the height of slabs to determine overlap. a) Two polygons are given. Positive edges are bolded. b) The first breakpoint is when the negative edge a starts crossing the positive edge g. The height of their overlap is h_1^+ c) The second relevant breakpoint is when the positive edge c stops crossing edge d. The height of their overlap is h_2^-. d) The last relevant breakpoints is when the positive edges b and c stops crossing edge e. The heights of their overlaps are h_3^- and h_4^-. It is easy to see that $h_1^+ - (h_2^- + h_3^- + h_4^-) = 0$, which means that there is no longer any overlap.

mapped coordinates, and these heights will always be integers.

Since we split the plane into slabs (of different heights) then we refer to the height of an edge, in the mapped slab-coordinates, as slab count, since it is in fact just the number of slabs that the edge spans.

Algorithm 5 shows how to calculate the slab counts with respect to horizontal translation. Calculating the slab counts for vertical translation is only a matter of using horizontal positions instead of vertical positions.

Algorithm 5 Slab counting

```
Input: An array of breakpoints, $Breakpoints$, each having $MinY, MaxY$ values
corresponding to the vertical extent of overlap of the involved lines.
Initialize a sorted set, $YPositions$. Initially empty.
for $i = 1$ to $Breakpoints.size$ do
    $YPositions.add(Breakpoints[i].MinY)$.
    $YPositions.add(Breakpoints[i].MaxY)$.
end for
Initialize an empty mapping set, $YToSlab$.
for $i = 1$ to $YPositions.size$ do
    $YToSlab[YPositions[i]] = i$.
end for
for $i = 1$ to $Breakpoints.size$ do
    $minY = Breakpoints[i].MinY$.
    $maxY = Breakpoints[i].MaxY$.
    $Breakpoints[i].SlabCount = YToSlab[maxY] - YToSlab[minY]$.
end for
```

The algorithm has 3 loops running $O(k)$ iterations each (k being the number of breakpoints). If we assume that the mapping $YToSlab$ uses perfect hashing then access to members is constant and thus the two latter loops run in $O(k)$ time. The first loop needs to update a sorted set which would normally be $O(\log k)$ for each update (e.g. using red-black or AVL trees) — this makes the entire loop run in $O(k \log k)$ and thus it is the dominating factor of this algorithm.

Since the breakpoints have already gone through a sorting-process then this extra step do not worsen the overall time complexity. Also note that there will be a lot of duplicate positions so the expected running time of the above is probably closer to linear.

7.4 Incorporating penalties in translation

According to the description of how we have modified our objective function (see 3.2) then our translation algorithm needs to be modified to take penalties into consideration when reporting the optimal placement of a stencil. This is almost trivial since there is at least one breakpoint each time the overlap between two stencils change. Furthermore, there is at most one change in the slab-count at each breakpoint, so when changing the slab-count for a polygon, we check to see if it changes from zero to non-zero or the opposite, and if so, add/subtract the penalty to a variable expressing the accumulated penalties, which is then added to the price.

A non-trivial part of this scheme appears when we have several breakpoints (from different polygons) with the same offset. A possible situation is illustrated in Figure 7.3. Here we

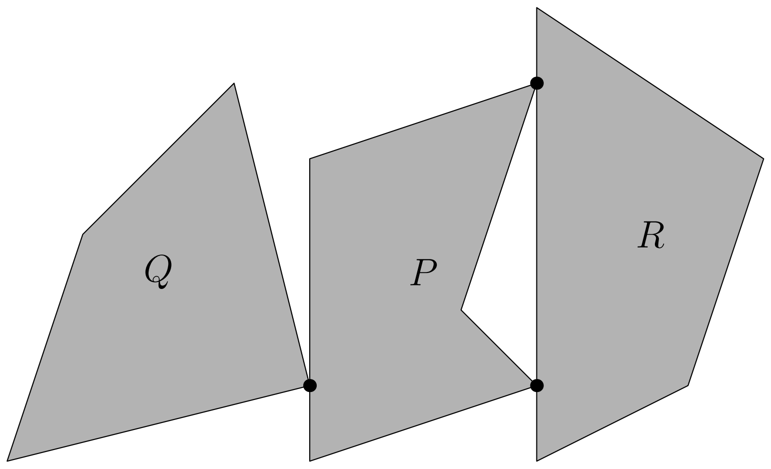


Figure 7.3: Polygon P is under translation and is now exactly between Q and R in a non-overlap situation. A series of breakpoints occur at the same time.

translate P and on the illustration the polygon is exactly between Q and R. There are several breakpoints with the same offset, some for the edges in Q and others for the edges in R. If we handle these breakpoints in a random order then a situation may arise where we first process those from R, which denote a beginning of an overlap, and thus an increase in penalty, and then those from Q, which subtract the penalty representing an overlap with Q. So at no time do we have a penalty of zero.

In the illustrated situation we can use the slab-count as a secondary sort criterion, but it will not solve all situations. Instead we use the polygon index as a secondary sort criterion and when we process breakpoints we loop over all those with the same offset (taking chunks of those from the same polygon at a time). We accumulate the slab-count for each and if it is negative we subtract it and adjust penalty accordingly. If it is positive then we store it for later processing. When we have gone through them all then we look at the price and update our minimum if it is smaller. Then we go on to adding all of the earlier stored slab-counts, which represent the polygons we have entered at this offset.

7.5 Handling non-rectangular material

It happens that we wish to solve the nesting problem for a material that is not restricted to being a rectangle. Since we support holes in our stencils and also fixed positions then this case can simply be handled by creating the exterior of our material as a hole in a stencil which we give a fixed position such that it overlaps the entire material. An example of this can be seen in Figure 7.4.

This technique can also be used in three dimensions, but in the experiments we use a dataset which must be packed in a cylinder and here it would take quite a lot of faces to make a good approximation. Since each added face will add to the running time of our implementation then we have chosen a different approach. Our translation algorithm already accepts a legal starting and ending point for the stencil (in this case polyhedron) to be translated, and will only place it between these two points. So if we know the first and last points which are inside

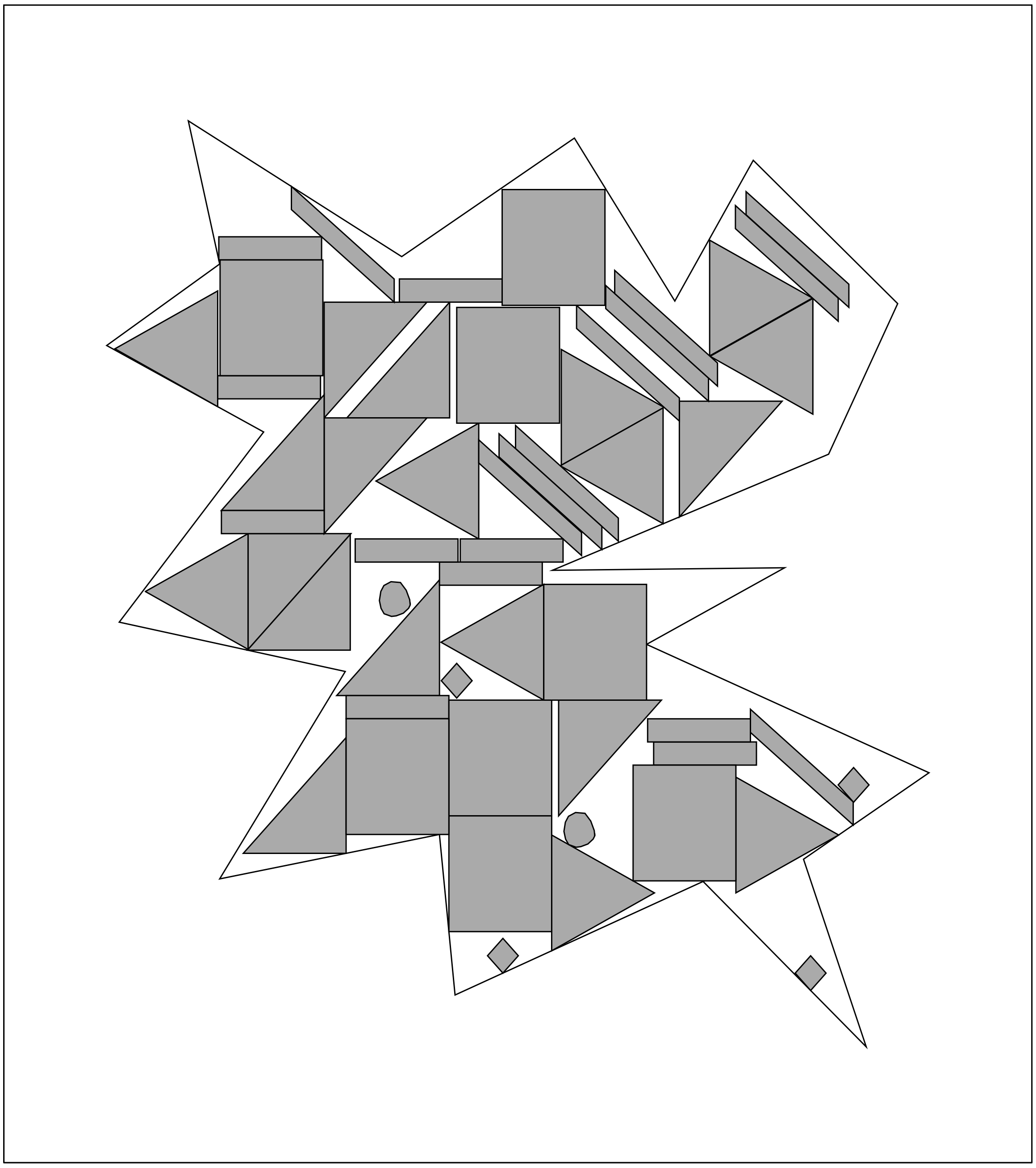


Figure 7.4: An example of nesting inside an irregular material. It should be noted that this is not a difficult nesting problem — it was found within 10 seconds using the implemented nesting solver.

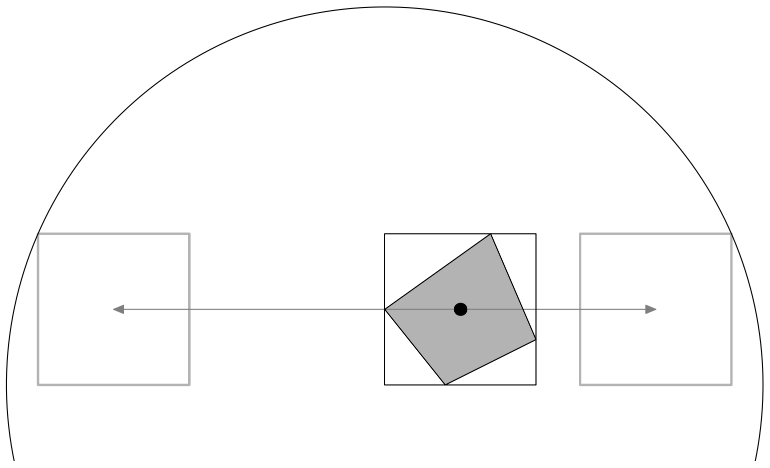


Figure 7.5: Keeping polyhedra within the boundaries of a cylinder can as a rough approximation be reduced to the problem of keeping a bounding box inside a circle.

the material, then we can make use of the above feature.

Assume that the height of the cylinder is along the z-axis. If we imagine we are translating on the x-axis then given the y position of our polyhedron we can find the first and last x position within the material by solving $x^2 = r^2 - y^2$ for each point in the polyhedron. In our implementation we have chosen an even simpler solution. The bounding box of polygons provides us with a rough approximation which will do for the data instances that we need to pack — an illustration of the approach can be seen in Figure 7.5.

7.6 Approximating stencils

The speed of algorithms working with polygons almost always depend on the number of vertices. It has been suggested [37] to make both exterior and interior approximations of polygons to help speed up any work done with the polygons. An exterior approximation is a polygon which contains the original polygon. This is very useful for the polygons to be packed since any legal placement with the approximated polygons will also be a legal placement with the original polygons. Interior approximations are useful for simplifying the polygon(s) describing the material. In the following we will focus on exterior approximations which in most cases are the most important since the material is very often rectangular. Note that if the material is interpreted as an infinite polygon with one or more holes then we can also use the exterior approximation algorithm for the material.

If the area of the original polygon is A and the area of the exterior approximation is A_e then we can measure the waste of an approximation as $\frac{A_e - A}{A_e}$. Now we can state the problem of finding a good exterior approximation of an arbitrary shape, not necessarily a polygon, in two different ways:

Problem 1: Using a fixed number of vertices, find an exterior approximation which wastes as little material as possible.

Problem 2: Given a maximum percentage of waste, find an exterior approximation using as few vertices as possible.

The nesting problem is about maximizing the use of material and therefore it is important to limit the amount of wasted material in the exterior approximations. In this regard, the second problem is a bit more adaptive than the first problem since some shapes will need more vertices in their approximations than others to keep the same level of waste. One could also imagine a combination of the two problems e.g. using a maximum waste per vertex.

It should be noted though that an optimal solution to the second problem is not necessarily the best possible approximation to use in the nesting problem. It does not guarantee that the shape of the exterior approximation is close to the original polygon. Thin "spikes" could be introduced in an approximation which would not waste much space, but they would make the nesting problem much harder.

Nevertheless we will focus on the second problem, but we will not find an optimal solution. We do not know the complexity of any of the stated problems, but we will present a heuristic solution method which satisfies our needs for exterior approximations. The heuristic only works for polygons.

The heuristic is greedy and it is based on a heuristic described by Heistermann and Lengauer [37]. First we need to define the concept of border triangles. These come in two types and two very simple examples are given in Figure 7.6a and 7.6b. The vertices of a polygon can be divided into two different groups depending on the angle they make with their neighboring vertices. An angle below 180 degrees indicates a concave vertex and an angle above 180° indicates a convex vertex.

Using these we can find two different types of border triangles. *Concave border triangles* are found by simply taking all concave vertices and use their neighboring vertices to make triangles. *Convex border triangles* are a bit more subtle. These require two neighboring convex vertices and the third point of the triangle is found by extending the edges as shown in 7.6b (it is not possible if the combined left turn of the points exceed 180°).

Now we can describe a very simple heuristic for approximating a polygon within a fixed percentage of waste. Note that Heistermann and Lengauer [37] only use concave border triangles in their approximation heuristic.

1. Find the border triangle of the current polygon with the smallest area.
2. If the waste of the polygon including this area is larger than the allowed amount of waste then stop the process.
3. Otherwise, change the polygon to include the found area. For concave border triangles this is done by removing a vertex from the polygon. For convex border triangles this means to remove two vertices and adding a new one. In both cases the resulting polygon has one vertex less.
4. Remove any self intersections caused by changing the polygon.
5. Now go back to step 1.

With the exception of the removal of self intersections the above algorithm is quite simple. As noted earlier though we have to look out for any spikes i.e. thin convex border triangles with a small area. An extreme example is given in Figure 7.6c. We need to have another criterion for choosing the "smallest" border triangle. In Figure 7.6d there is a less extreme example of convex border triangle. It is tempting to simply require that the angle α should be greater than some fixed value, but this does not really reflect what we want. It is a bit

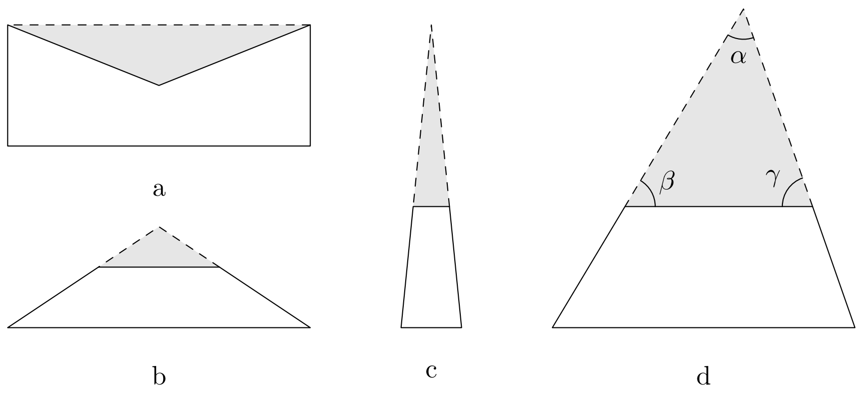


Figure 7.6: a) Three points making a right turn (counter clockwise) represent a concavity which can be removed by removing the middle point. b) Four points making two left turns (counter clockwise) which do not exceed 180° represent a convexity which can removed by adding a new point at the intersection of lines from the first and the last edge (and removing the two obsolete points). c) An example of a convexity with a triangle of little area but significant topological influence. It should be avoided to include such triangles in an approximation. d) A less extreme example of a triangle related to a convexity. It is not easy to determine which triangles to include.

too restrictive. A better alternative is to require that α is greater than at least one of the two other angles β or γ.

Now, let us take a look at the algorithm in action. In Figure 7.7 a highly irregular polygon is reduced to a simple quadrilateral. It is also shown that the result is not optimal regarding minimum waste using 4 vertices. Also note that if only concave border triangles had been removed then the resulting polygon would have had 6 vertices.

Now by varying the maximum percentage of wasted material the heuristic can provide us with different levels of approximated polygons. These can be used in other algorithms to speed up calculations when the original polygons are not strictly necessary.

In the above algorithm the waste is computed in relation to the polygon itself. This means that a border triangle of the same size on a small and on a large polygon will be evaluated differently. By considering all polygons at once when making the approximation this can be avoided and a better approximation might be found i.e. fewer vertices within the same bound. This is examined as a part of our experiments (Section 8.1.4).

7.7 Center of a polygon

For each polygon we need to determine a center of rotation. Intuitively, a really good choice would be the center of the *smallest enclosing circle* (SEC) of all the vertices of the polygon. This will ensure that the focus is on improving the solution by rotating the polygon and not by translating it (we have the translation part of the neighborhood for that purpose).

Finding the SEC (also known as the *minimal enclosing circle* or *minimum spanning circle*) can (a bit surprising) be done in linear time [46]. Formally the problem is: Given a set of points on the plane, find the smallest circle which encloses all of them.

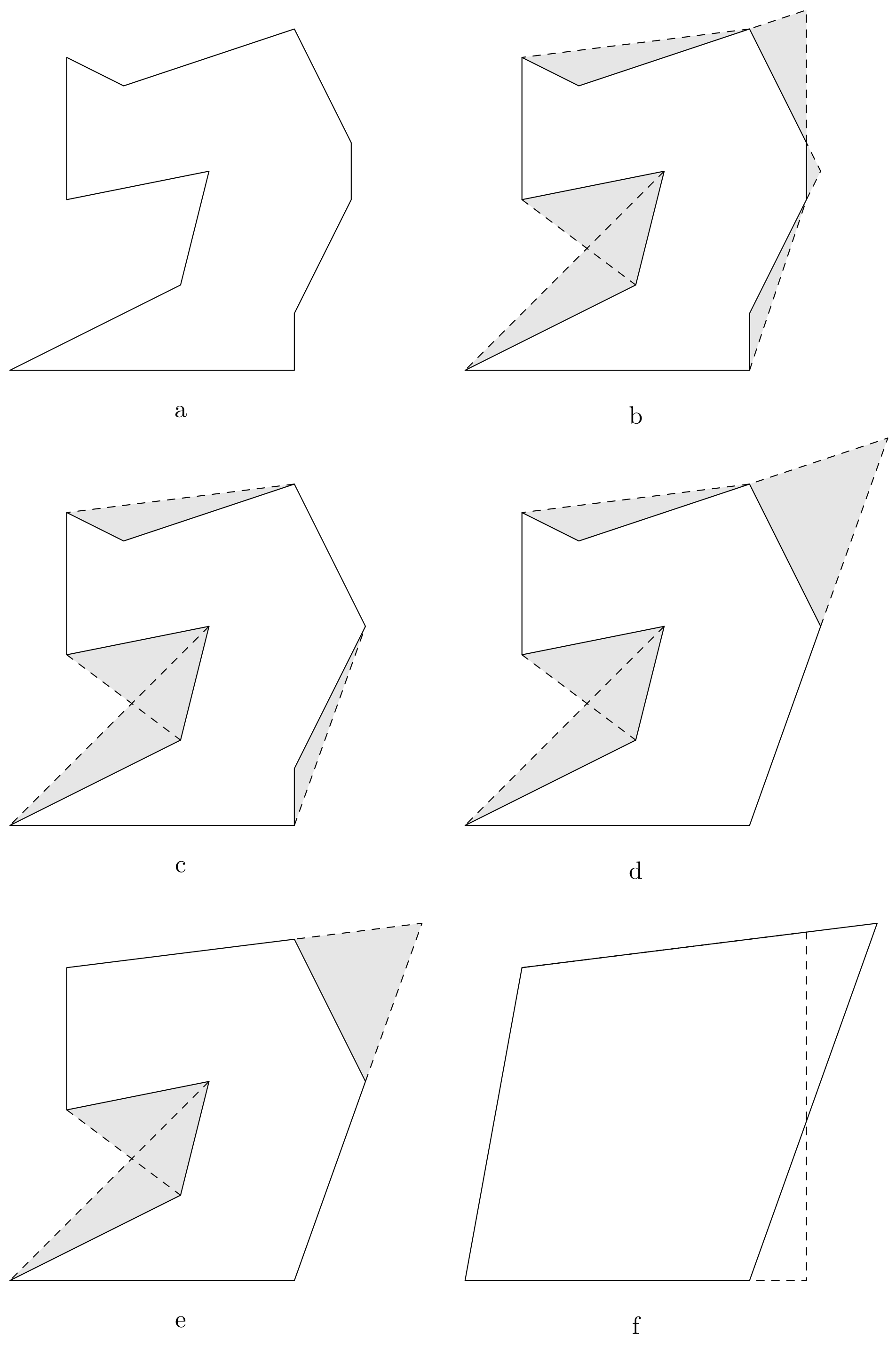


Figure 7.7: a) A highly irregular polygon with 11 vertices. b) The set of all border triangles which it is possible to remove. 2 convex and 5 concave border triangles. c-e) The triangle of smallest area is iteratively removed. The list of border triangles is updated at each step. f) Continuing as long as possible results in a polygon with just 4 vertices and naturally no concavities. The dashed line indicates a quadrilateral with smaller area not found by the algorithm.

The $O(n)$ algorithm is quite complicated, but luckily, we can find the SECs in a preprocessing step i.e. we do not need the algorithm to be linear in time. We can choose to implement a simple algorithm by Skyum [53] running in time $O(n \log n)$ or an even simpler algorithm by Welzl [57, 16] that runs in expected linear time. We have chosen the latter algorithm.

Note that when choosing the center of the enclosing circle as the center of rotation then the enclosing circle can also be used as a bounding circle. It will then be very simple to skip all polygons too far away to ever overlap the polygon to be rotated.

Chapter 8

Computational Experiments

8.1 2D experiments

8.1.1 Strategy

The following experiments should be viewed as complimentary to those which Egeblad et al. [28] conducted for their implementation, JAB. Although no source is shared between the implementations the solution methods are very similar and so some choices have been based on the experience gained from earlier experiments.

After a presentation of the data instances in the following section we will discuss and experiment with the lambda value, approximation schemes and strip length strategy. An example with quality requirements is also given and finally some benchmark comparisons are done with published results and with a commercial solver.

8.1.2 Data instances

The amount of publically available data instances is limited. Table 8.1 presents some statistics about the instances which we will use for our experiments and benchmarks. They vary between 43 and 99 stencils per set and the average number of vertices per stencil is between 6 and 9 with the exception of Swim which has 20 vertices per stencil. All of the sets can be seen in Figure 8.1 where it is also indicated that they all contain subsets of identical stencils. Note that we do not utilize this property in any way and therefore it does not make the problems easier for us.

Name	Stencils	Vertices (avg.)	Area (avg.)	Area (total)	Strip width
Artificial	99	7.19	9.6	947	27
Shapes0	43	9.09	37.1	1596	40
Shapes2	28	6.29	11.7	328	15
Shirts	99	6.05	22.1	2183	40
Swim	48	20.00	530078.4	25443764	5600
Trousers	64	6.06	269.0	17216	79

Table 8.1: Testdata

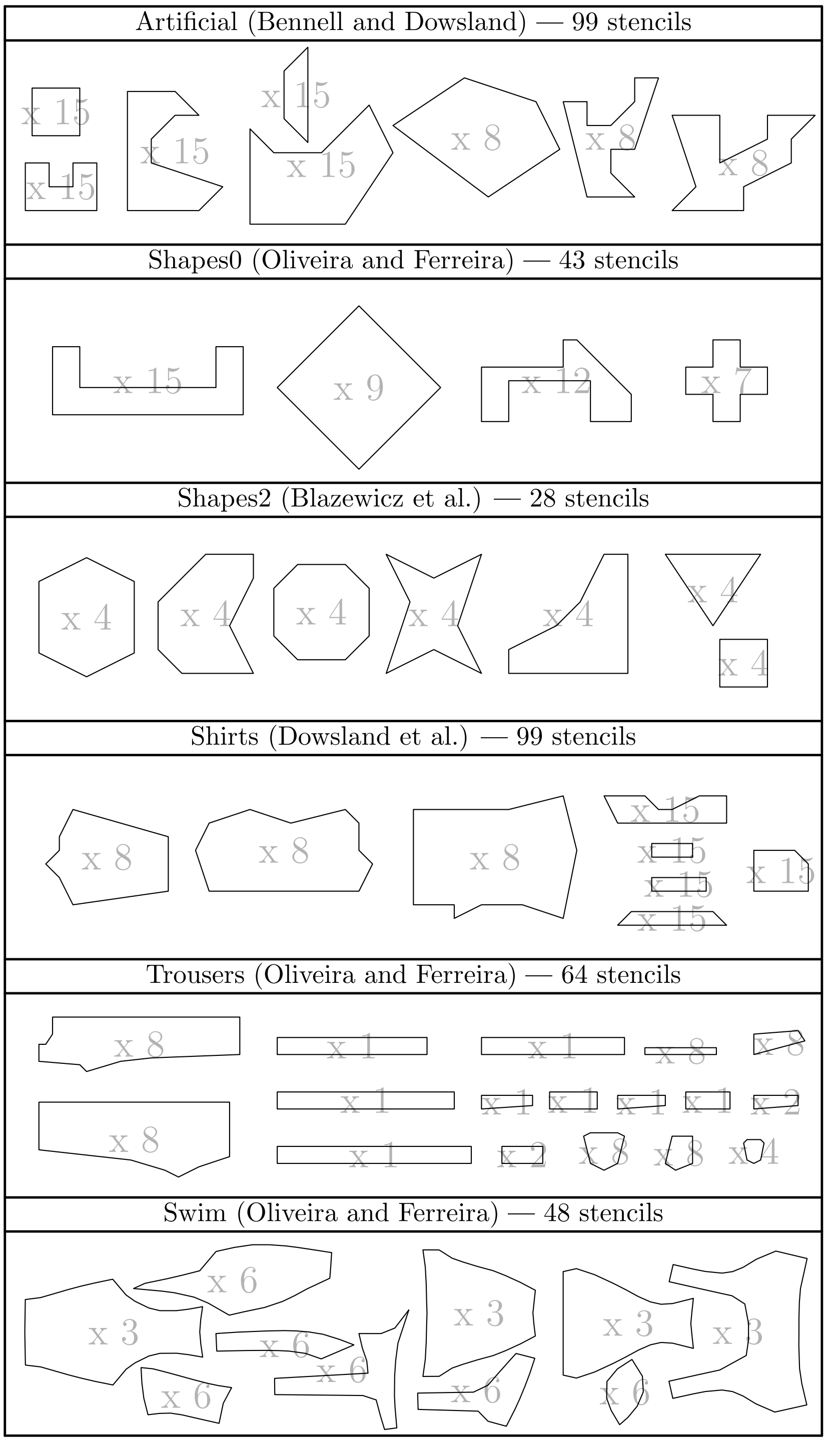

Artificial (Bennell and Dowsland) — 99 stencils
x 15
x 15
x 15
x 15
x 15
x 8
x 8
x 8
Shapes0 (Oliveira and Ferreira) — 43 stencils
x 15
x 9
x 12
x 7
Shapes2 (Blazewicz et al.) — 28 stencils
x 4
x 4
x 4
x 4
x 4
x 4
x 4
Shirts (Dowsland et al.) — 99 stencils
x 8
x 8
x 8
x 15
x 15
x 15
x 15
x 15
Trousers (Oliveira and Ferreira) — 64 stencils
x 8
x 1
x 1
x 8
x 8
x 1
x 1
x 1
x 1
x 1
x 2
x 8
x 1
x 2
x 8
x 8
x 4
Swim (Oliveira and Ferreira) — 48 stencils
x 6
x 3
x 3
x 3
x 3
x 6
x 6
x 6
x 6
x 6

Figure 8.1: The data instances we use for most of our experiments and benchmarks.

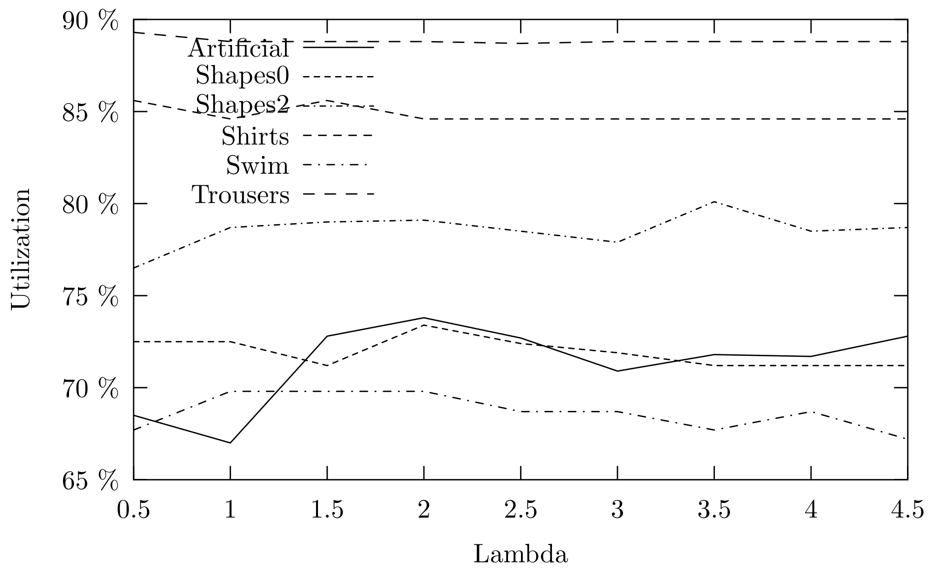


Figure 8.2: Solution quality for different values of lambda after 5 minutes.

8.1.3 Lambda

Experiments regarding the value of lambda and FLS strategy were conducted by Egeblad et al. We are only going to repeat the lambda experiment to make sure that this value is set appropriately. We know from their experiments that this value should be in the range of 1%-4% of the area of the largest stencil and we have tried values in this interval for all data instances. The results can be seen in Figure 8.2 and they are quite inconclusive. We settle for a value of 3%.

8.1.4 Approximations

From our previous work with GLS we have already experienced that Swim is a time-consuming dataset to pack, and for this reason we have implemented an approximation strategy to reduce the number of vertices in the stencils which in this case must be polygons. The first experiment we have done shows the number of vertices left in the dataset as a function of the maximum allowed waste and can be seen in Figure 8.3. To get an idea of how the dataset is affected we also present an example of exactly how Swim gets affected by an approximation allowing a waste of 5%, this can be seen in Figure 8.4.

To see if approximated vertices actually make a difference in the time used to find good solutions we have tried to solve Swim both with and without approximated vertices for an allowed waste of 2.5%, 4.0% and 5.0%. We have done this for all combinations of parameters taken by our approximation scheme, see Section 7.6 for details, and the results can be seen in Table 8.2.

The conclusion must be that approximation is an advantage for the Swim dataset. Exactly how much waste one should allow, and which approximation scheme should be used, are still open questions. Since a waste percentage of 2.5 has the best average in solution quality we have chosen to use this value, and furthermore to only use the concave approximation scheme and look at all stencils since that worked best for this percentage.

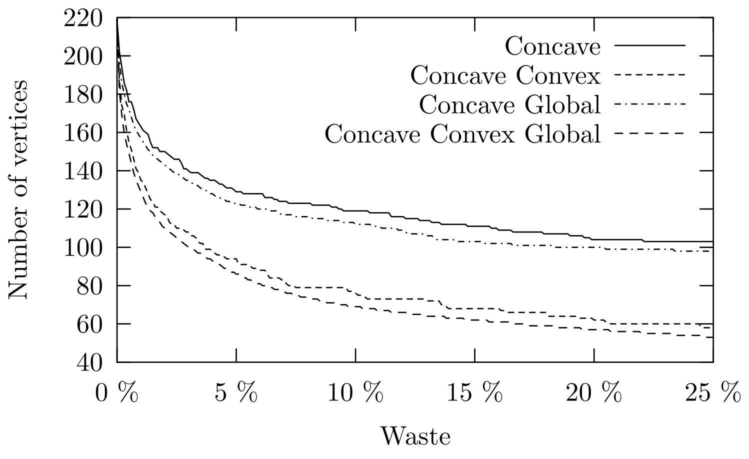


Figure 8.3: The number of vertices left in the Swim instance when different levels of waste are allowed.

Waste	0.0%	2.5%	4.0%	5.0%
Concave	66.7%	67.4%	66.6%	64.1%
Concave Convex	66.7%	67.7%	63.0%	64.2%
Concave Global	66.7%	67.9%	65.4%	67.1%
Concave Convex Global	66.7%	67.9%	65.4%	63.0%

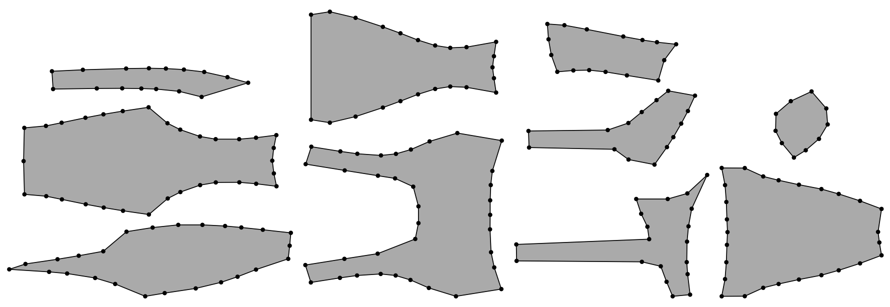
Table 8.2: The utilization after one hour for different approximation parameters and waste thresholds.

Another way we have tried to speed up our implementation is by reducing each translation to a subset of the material (relative to the position of each stencil). This is given as a percentage of the materials size. Since we have already established that Swim benefits from approximated vertices we will try this bounded translation both with and without approximated stencils, to see if a) bounded translation can compete with approximated vertices and b) to see if bounded translation can further improve approximated vertices. Our findings can be seen in Figure 8.5 and our conclusion is that, at least for Swim, bounded translation is not an advantage. We have skipped this idea for our further experiments.

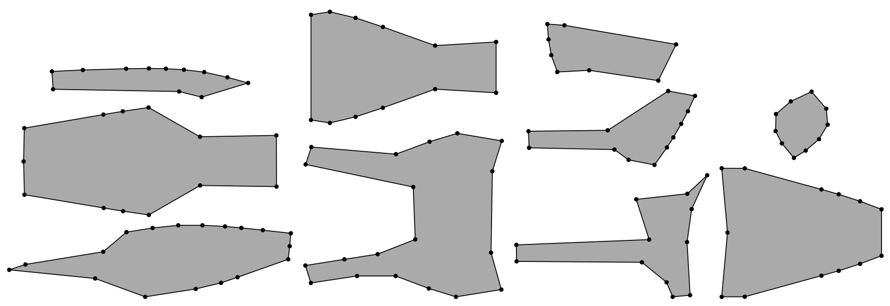
8.1.5 Strip length strategy

As described earlier we can solve the strip-length problem as a decision problem where we decrement the size of our material after each successful placement. The size of our decrementing step is subject for the following experiment. The first thing we notice is that our datasets vary significantly in size and thus a decrement step given in units make little sense. Instead we define it as a percentage of the initial length — we have chosen 1% because smaller improvements are probably of little interest (at least in the beginning). We could set it higher, but given enough time, our implementation should be able to reach the same solutions as can be found using a higher initial stepping factor.

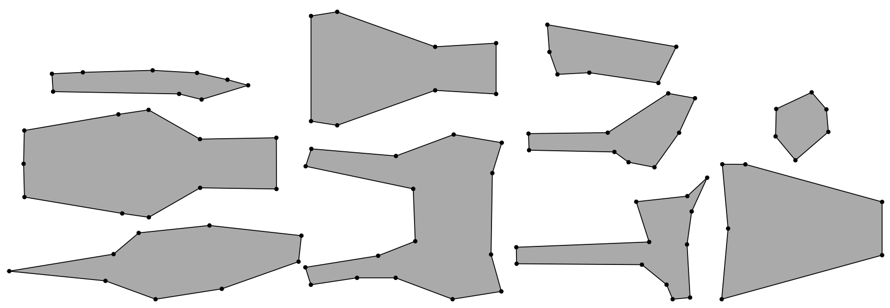
At some point we may have shrinked the length so much that 1% of the initial length is too



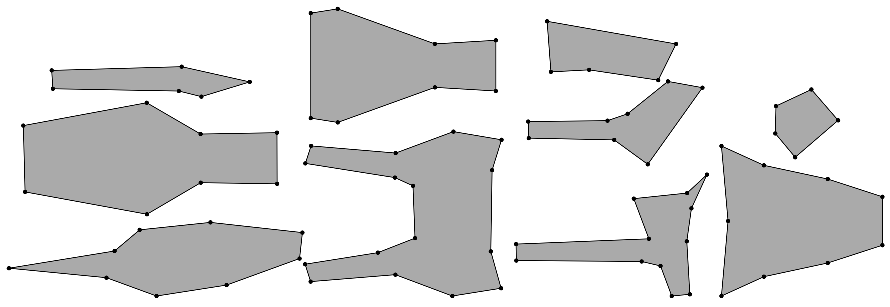
Original stencils (219 vertices).



Each stencil reduced with concave triangles (129 vertices).

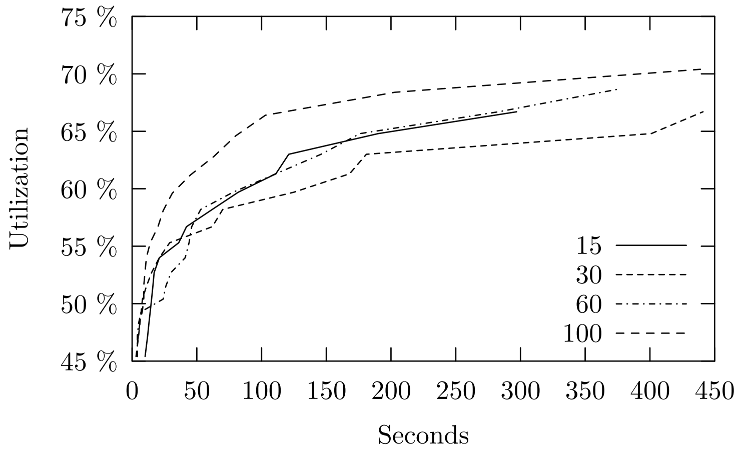


Each stencil reduced with concave and convex triangles (92 vertices).



Global approach looking at triangles in all stencils simultaneously (86 vertices).

Figure 8.4: The Swim data instance with various approximation schemes applied which all keep the total waste below 5%.



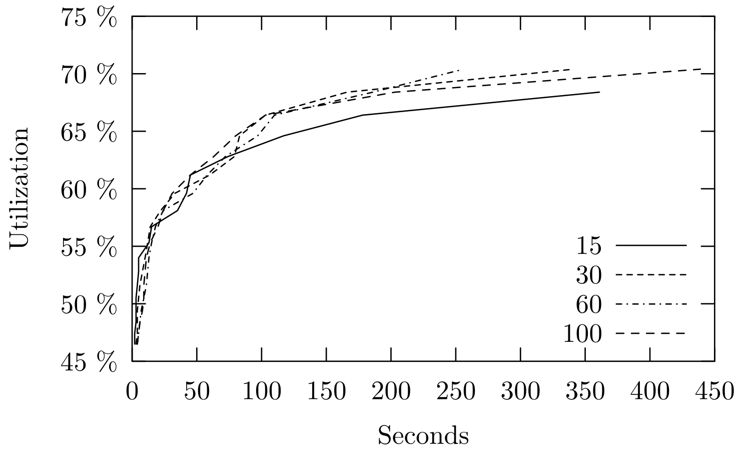


Figure 8.5: Experiments with bounded translation of stencils. The curves represent 15%, 30%, 60% and 100% translation relative to the maximum translation distance. Swim was used in both of the above experiements, but the latter also used approximated stencils.

much of a step to find the next legal placement — for this reason we should introduce a way to adapt our decremented size when such situations arise (and a way to find these situations). Though if we continue to make our decrementing step smaller then we may eventually end up with an infinitely small step.

All this has resulted in the following 3 experiments.

- Always decrement by 1%, which we refer to as "fixed".
- Decrement by 1% until $4n^2$ iterations have been executed without finding a legal placement. Make the decrement step 70% smaller and backtrack to last legal solution, which we refer to as "adaptive".
- Same as above but using a lower limit of 0.1% of the length as a minimum for the decrement step, referred to as "adaptive w/ LB".

The results can be seen in Table 8.3 where each strategy has been tried for all instances and the utilization obtained after one hours run is reported. A fourth strategy is also presented which is not directly related to the stepping strategy. Here we test a simple rounding strategy where we try to reduce the number of decimals, and thus the possible positions of stencils, by rounding the result from our translation to integer coordinates. It is presented in this table simply to make comparisons easier.

It is clear from this experiment that the adaptive approach with a lower bound is the best choice for stepping strategy. Interestingly the rounding-strategy seemed to pay off on the artificial datasets (with rotation), but did not improve anything for the examples from the textile industry (on the contrary), the reason for this is most likely that the artificial datasets have been defined by humans and so have integer sizes.

Strategy	Fixed	Adaptive	Adaptive w/ LB	Rounded
No rotation				
Artificial	69.9%	59.3%	70.8%	70.6%
Shapes0	65.0%	64.9%	66.2%	66.4%
Shapes2	78.9%	77.5%	80.8%	80.2%
Shirts	84.6%	84.3%	84.3%	84.3%
Swim	69.4%	69.7%	70.8%	70.2%
Trousers	85.4%	85.8%	85.6%	85.7%
180° rotation				
Artificial	73.8%	73.8%	74.4%	74.5%
Shapes0	72.3%	71.4%	72.5%	72.5%
Shapes2	80.1%	78.4%	81.1%	81.6%
Shirts	85.6%	86.4%	86.5%	86.2%
Swim	71.5%	72.9%	73.5%	72.9%
Trousers	88.8%	89.5%	89.8%	89.2%

Table 8.3: The utilization for 3 different stepping-strategies and an attempt of a rounding-strategy.

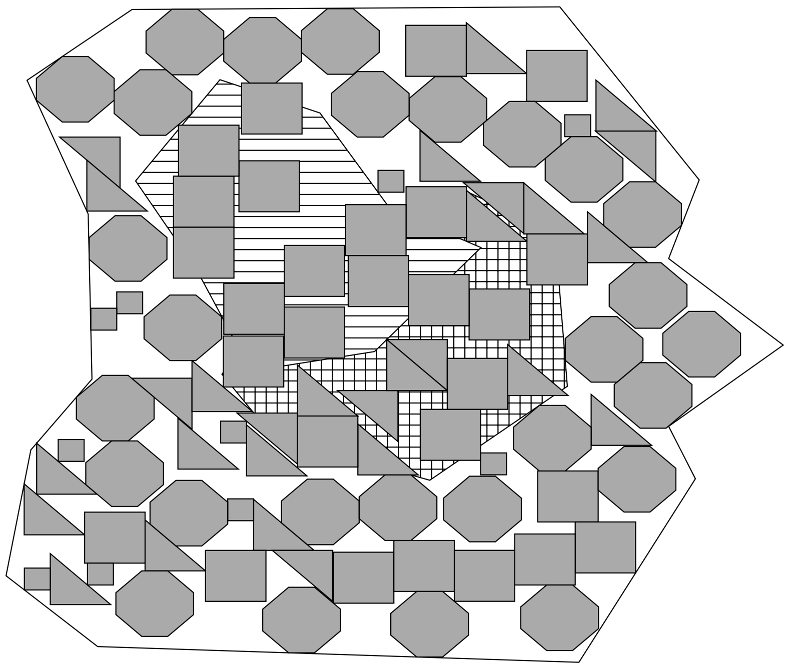


Figure 8.6: The large boxes require quality Q_2, the triangles require quality Q_1 and the rest require quality Q_0. The striped pattern is of quality Q_2 and the squared pattern is of quality Q_1. This legal placement was found within a few seconds by 2DNEST.

8.1.6 Quality requirements

2DNEST supports quality levels and requirements. As an example of this ability we created a simple (but ugly) animal hide data instance (guess an animal) with a few fixed areas with quality levels and a lot of small stencils with various quality requirements. This is illustrated in Figure 8.6.

8.1.7 Comparisons with published results

The data instances we are using have also frequently been used in existing literature. Some of the best results are gathered in Table 8.4 where they can be compared with results from 2DNEST. The computation times vary a lot, from minutes to hours, so the results should be interpreted with care. Furthermore, the very small number of instances makes it difficult to make any certain conclusions.

Nevertheless, Table 8.4 shows that we are doing quite well when it comes to solution quality. We believe that 2DNEST has not been optimized to its full potential and that improvements in solution strategies are still possible. Images of our placements for all 6 data instances can be found in Appendix A (with 180° rotation).

It should be stated that the results reported for JAB are produced much faster than for 2DNEST and it is our general impression that our old implementation was faster than 2DNEST

Name	2DNest	Jab	Topos	Jostling	2-exchange	Hybrid.
			No rotation			
Artificial	70.8%	-	-	-	-	(78.38%)[1]
Shapes0	66.2%	65.4%	59.8%	63.3%	61.4%	62.81%
Shapes2	80.8%	77.1%	-	-	-	76.58%
Shirts	84.3%	84.4%	-	-	-	81.18%
Swim	68.3%	65.8%	-	-	-	
Trousers	85.6%	84.4%	-	-	-	
			180° rotation			
Artificial	74.4%	-	-	-	-	-
Shapes0	72.5%	72.5%	65.4%	63.3%	67.6%	-
Shapes2	81.1%	77.1%	74.7%	-	79.1%	-
Shirts	86.5%	85.7%	81.3%	83.1%	85.5%	-
Swim	71.0%	67.2%	-	-	-	-
Trousers	89.8%	88.5%	82.8%	-	88.6%	-

Table 8.4: 2DNest (1 hour) is here compared with Jab by Egeblad et al. [28] (10 minutes), Topos by Oliveira et al. [51], *Jostling for position* by Dowsland et al. [23], 2-exchange by Gomes and Oliveira [34] and a hybridized tabu search with optimization techniques by Bennell and Dowsland [7]

— especially in the early steps. Although we have not pinpointed the exact problem then we present here some plausible explanations.

- The use of double precision arithmetic is on average slower, however, it also introduces situations where stencils are repeatedly moved only small fractions of units, each providing a very small improvement, and each opening up for another potential improvement on another translation axis — we did in fact experience such situations, but have tried to remedy them by setting an upper limit on the amount of allowed movements with the same stencil and as can be seen from Table 8.3 then we have also tried a limited rounding strategy. A better solution would be desirable.

- By using the described slab-counting we require that the solution is analytical legal, where our previous implementation allowed a very small overlap per stencil. This results in extra time spent each time a new legal solution has to be found.

- Our previous implementation used a better strategy to find an initial solution saving time in the initial stage.

- Our current implementation is created with flexibility and adaptability in mind, which cover using third party libraries like STL for data structures which although priced for its efficiency cannot compete with rolling your own solution.

[1]We were surprised by the result for Artificial and tried to examine it a bit closer. Correspondence with the authors did not clarify the issue, but their solution method might be especially suited for this instance. Note that our result is close to the result of the commercial solver presented in section 8.1.8.

8.1.8 Comparisons with a commercial solver

We have also obtained access to a commercial solver, AutoNester-T, which has been under development for about a decade. The results of this solver is very impressive especially regarding speed. The following is a citation from the homepage of AutoNester-T:

> "AutoNester-T combines many of the latest optimization techniques in order to achieve the best results in a very short time. We use local search algorithms based on new variants of Simulated Annealing, multiply iterated greedy strategies, paired with efficient heuristics, pattern-recognition techniques, and fast Minkowski sum calculators. In the local search algorithms, we use fully dynamic statistical cooling schedules and dynamic parameter choice.
>
> For the calculation of optimality proofs and efficiency guarantees on markers we use a combination of branch-and-bound algorithms and linear programming."

It must be emphasized that AutoNester-T is optimized for marker making and therefore the results of the artificial data instances (Artificial, Shapes0 and Shapes2) are not as good as for the others (Shirts, Swim, Trousers). The results of AutoNester-T at various time points and our results after an hour is given in Table 8.5. AutoNester-T was run on a 600 MHz processor. The 2DNest results were obtained on a 1.4GHz processor. Note that AutoNester-T quickly finds very good solutions, but it seems that it does often not find better solutions when given more time. We know from experience with 2DNest that it often continues to find better solutions when given more time.

Test instance	2DNest	0.2 min.	1 min.	10 min.
No rotation				
Artificial	70.8%	69.3%	70.1%	70.7%
Shapes0	66.2%	61.4%	62.7%	64.4%
Shapes2	80.8%	77.6%	78.8%	80.2%
Shirts	84.3%	82.2%	83.6%	84.4% (2.2 min.)
Swim	68.3%	69.4%	70.6%	71.4%
Trousers	85.6%	83.8%	84.3%	84.5%
180° rotation				
Artificial	74.4%	74.1%	75.9%	76.7% (2.8 min.)
Shapes0	72.5%	68.8%	71.3%	71.3% (0.5 min.)
Shapes2	81.1%	79.6%	81.1%	81.8% (2.9 min.)
Shirts	86.5%	85.5%	86.8%	87.4% (1.8 min.)
Swim	71.0%	74.4%	75.6%	75.6% (0.8 min.)
Trousers	89.8%	89.3%	89.8%	90.6%

Table 8.5: These are results from the commercial solver AutoNester-T (version 3.5 [build 3087]). Note how fast the solutions are generated. The results reported for 2DNest are with a 1 hour time limit.

[1] `http://www.gmd.de/SCAI/optimierung/products/autonester-t/index.html`

8.2 3D experiments

8.2.1 Data instances

The amount of publically available data instances for the two-dimensional nesting problem is limited and it does not get better by adding a dimension to the problem. It is quite easy to obtain 3D objects from other areas of research, but these will typically have more than 10000 faces each and would therefore require to be approximated as a first step. We have not implemented an approximation algorithm for 3D objects and therefore this is not an option.

In the literature only one set of simple data instances has been used. They were originally created by Ilkka Ikonen and later used by Dickinson and Knopf [18] to compare their solution method with Ikonen et al. [39]. There are eight objects available in the set and they are presented in Table 8.6. Some of them have holes, but they are generally quite simple. They can all be drawn in two dimensions and then just extended in the third dimension. They have no relation to real-world data instances. See them in Figure 8.7.

Name	# Faces	Volume	Bounding box
Block1	12	4.00	1.00 × 2.00 × 2.00
Part2	24	2.88	1.43 × 1.70 × 2.50
Part3	28	0.30	1.42 × 0.62 × 1.00
Part4	52	2.22	1.63 × 2.00 × 2.00
Part5	20	0.16	2.81 × 0.56 × 0.20
Part6	20	0.24	0.45 × 0.51 × 2.50
Stick2	12	0.18	2.00 × 0.30 × 0.30
Thin	48	1.25	1.00 × 3.00 × 3.50

Table 8.6: The Ikonen data set.

In the existing literature the above data set has been used with a cylindrical volume (which is often the case for rapid prototyping machines). We have chosen to follow this approach since it will make it easier to compare results. This introduces the problem of keeping the object within the limits of a cylinder. We handle this in a simple way using the bounding box of the objects. This means that a small part of the real solution space is not available (which is a handicap), but this simplification has no or only a minor effect on each of the objects in the above table.

8.2.2 Statistics

The number of intersection calculations needed to find a solution was used by Dickinson and Knopf [18] to compare their solution method with the one by Ikonen et al. [39]. Unfortunately we are not able to do the same since we do not make intersection calculations in the same way. Execution times can be compared and we will do so in a benchmark at the end of this chapter.

In the following we will try to gather some statistics about the average behavior of our solution method. A simple knapsack scheme has been implemented to pack as much as possible within a cylinder of height 5.5 and radius 3.5. Within 2 minutes 54 objects have been packed (the 8 Ikonen objects are simply repeatedly added one by one) and they fill 34% of the volume of the cylinder. About 3500 translations were done per minute. The packing can be seen from four different angles in Figure 8.8.

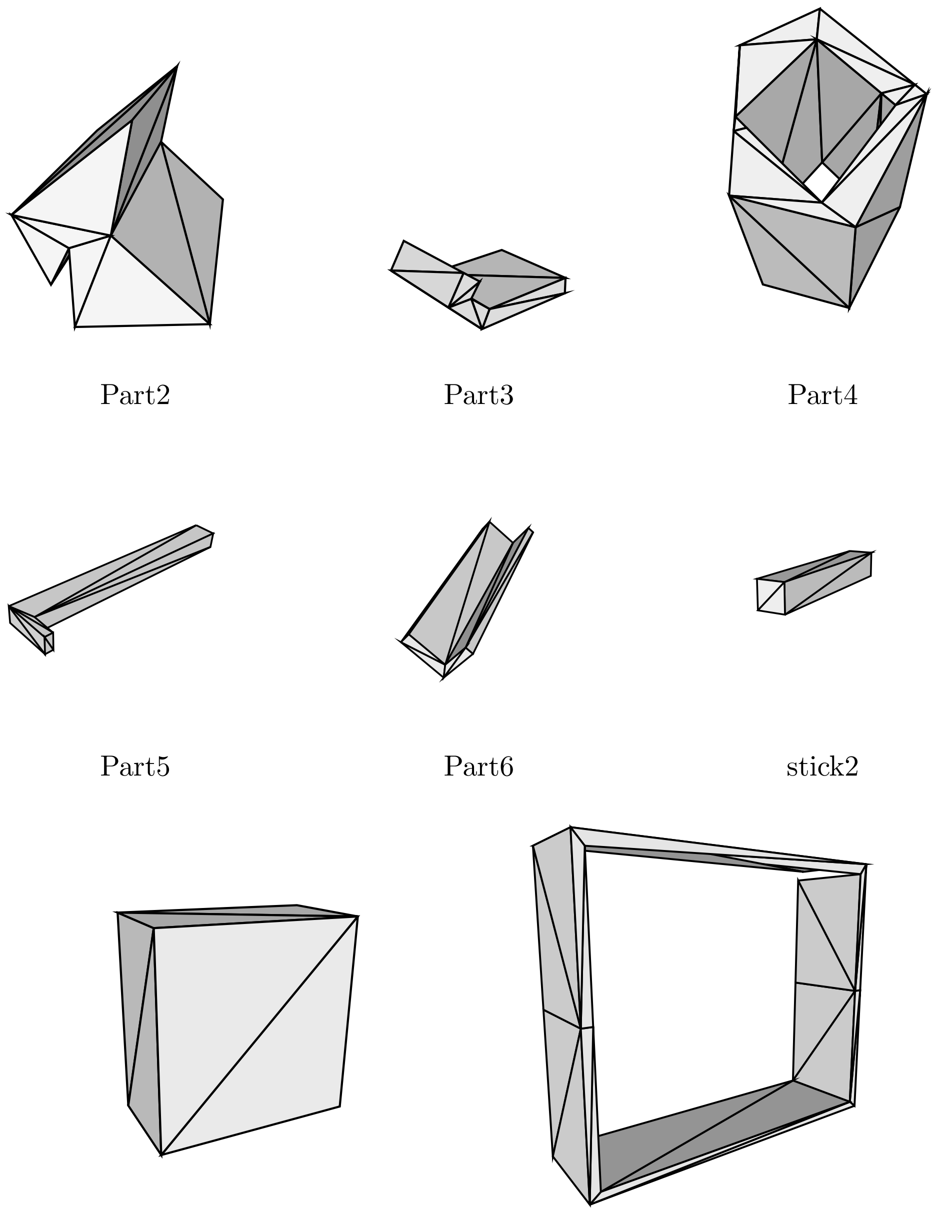


Figure 8.7: The Ikonen data set.

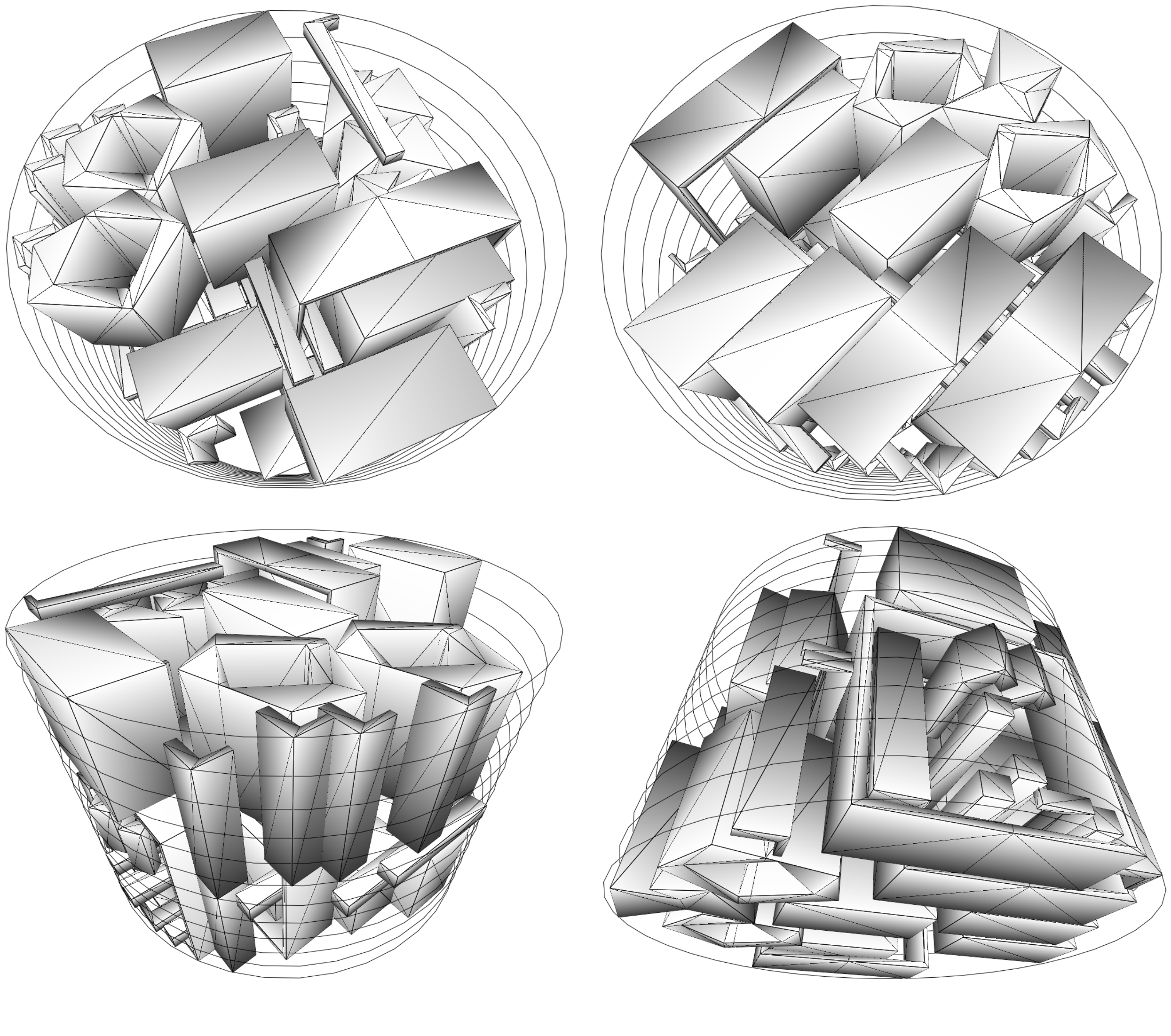


Figure 8.8: Various aspects of a solution produced by a simple knapsack scheme.

8.2.3 Benchmarks

We have already presented the only available set of data instances for which there exists comparable results 8.2.1. Two test cases are created by Dickinson and Knopf for their experiments.

- Case 1
  Pack 10 objects into a cylinder of radius 3.4 and height 3.0. The 10 objects are chosen as follows: 3 times Part2, 1 Part4 and 2 times Part3, Part5 and Part6. Total number of faces is 260 and 11.3% of the total volume is filled.
- Case 2
  Pack 15 objects into a cylinder of radius 3.5 and height 5.5. The 15 objects are chosen as in case 1, but with 5 more Part2. Total number of faces is 380 and 12.6% of the total volume is filled

Dickinson and Knopf report execution times for both their own solution method (serial packing) and the one by Ikonen et al. (genetic algorithm). They ran the benchmarks on a 200 MHz AMD K6 processor. The results are presented in Table 8.7 in which results from our algorithm is included. Our benchmarks were run on a 733MHz G4.

A random initial placement might be a problem since it could quite likely contain almost no overlap. Then it would not say much about our algorithm — especially the GLS part. To make the two cases a bit harder we doubled the number of objects. The results can be seen in Table 8.7 where it is compared with the results of Ikonen et al. and Dickinson and Knopf. Even considering the difference in processor speeds there is no doubt that our method is the fastest. The placements can be seen in Figure 8.9.

Test	Ikonen et al.	Dickinson and Knopf	3DNest
Case 1	22.13 min.	45.55 sec.	3.2 sec. (162 translations)
Case 2	26.00 min.	81.65 sec.	8.1 sec. (379 translations)

Table 8.7: Execution times for 3 different heuristic approaches. Note that the number of objects are doubled for 3DNest.

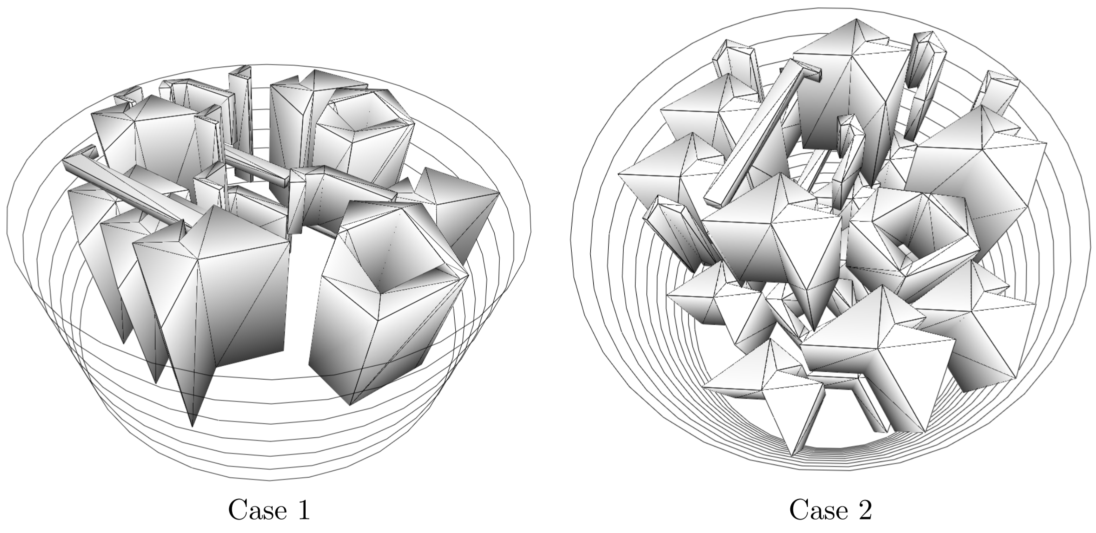


Figure 8.9: The above illustrations contain twice as many objects as originally intended in Ikonens Case 1 and 2. They took only few seconds to find.

Chapter 9

Conclusions

The nesting problem has been described in most of its variations. Various geometric approaches to the problem has been described and a survey has been given of the existing literature concerning the nesting problem in two and three dimensions.

The major theoretical contributions of this thesis in relation to previous work is the following. A proof of correctness of the translation algorithm presented by Egeblad et al. [28], a generalization of the algorithm which among other things provides an algorithm for rotation and an easy generalization to three-dimensional nesting.

Furthermore, a comprehensive implementation of a two-dimensional nesting solver has been created and it can handle a wide range of problem variations with various constraints. It performs very well compared with other solution methods even though it has not been optimized to its full potential. A three-dimensional nesting solver has also been implemented proving in practice that a fast translation method is also possible in this case. The results are promising although only limited comparisons have been possible.

Subjects for future work include three-dimensional rotation and additional experiments to optimize the solution strategies for various problem types.

Acknowledgements

The authors would like to thank the following people. Jens Egeblad for his participation in the project on which this thesis is based, Ralf Heckmann for allowing us to access SCAIs commercial nesting program, Julia Bennell for providing 2D data instances and for mailing hard-to-obtain articles, and finally, John Dickinson for providing the 3D data instances.

Bibliography

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

[2] A. Albano. A method to improve two-dimensional layout. *Computer Aided Design*, 9(1):48–52, 1977.

[3] A. Albano and G. Sappupo. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Transactions on Systems, Man and Cybernetics*, 5:242–248, 1980.

[4] R. C. Art, Jr. An approach to the two dimensional, irregular cutting stock problem. Technical Report 36.Y08, IBM Cambridge Scientific Center, September 1966.

[5] T. Asano, A. Hernández-Barrera, and S. C. Nandy. Translating a convex polyhedron over monotone polyhedra. *Computational Geometry*, 23:257–269, 2002.

[6] J. A. Bennell and K. A. Dowsland. A tabu thresholding implementation for the irregular stock cutting problem. *International Journal of Production Research*, 37:4259–4275, 1999.

[7] J. A. Bennell and K. A. Dowsland. Hybridising tabu search with optimisation techniques for irregular stock cutting. *Management Science*, 47(8):1160–1172, 2001.

[8] J. Blazewicz, P. Hawryluk, and R. Walkowiak. Using a tabu search approach for solving the two-dimensional irregular cutting problem. *Annals of Operations Research*, 41:313–325, 1993.

[9] J. Blazewicz and R. Walkowiak. A local search approach for two-dimensional irregular cutting. *OR Spektrum*, 17:93–98, 1995.

[10] E. K. Burke and G. Kendall. Applying ant algorithms and the no fit polygon to the nesting problem. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence (AI'99)*, volume 1747, pages 454–464. Springer Lecture Notes in Artificial Intelligence, 1999.

[11] E. K. Burke and G. Kendall. Applying evolutionary algorithms and the no fit polygon to the nesting problem. In *Proceedings of the 1999 International Conference on Artificial Intelligence (IC-AI'99)*, volume 1, pages 51–57. CSREA Press, 1999.

[12] E. K. Burke and G. Kendall. Applying simulated annealing and the no fit polygon to the nesting problem. In *Proceedings of the World Manufacturing Congres*, pages 70–76. ICSC Academic Press, 1999.

[13] J. Cagan, D. Degentesh, and S. Yin. A simulated annealing-based algorithm using hierarchical models for general three-dimensional component layout. *Computer Aided Design*, 30(10):781–790, 1998.

[14] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, and W. Wright. Simplification envelopes. *Computer Graphics*, 30(Annual Conference Series):119–128, 1996.

[15] W. Cook and A. Rohe. Computing minimum-weight perfect matchings. *INFORMS Journal on Computing*, 11:138–148, 1999.

[16] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry, Algorithms and Applications*. Springer-Verlag, 1997.

[17] J. K. Dickinson and G. K. Knopf. Generating 3d packing arrangements for layered manufacturing. In *International Conference on Agile, Intelligent, and Computer-Integrated Manufacturing*, Troy, New York, 1998. Rensselaer Polytechnic Institute.

[18] J. K. Dickinson and G. K. Knopf. Serial packing of arbitrary 3d objects for optimizing layered manufacturing. In *Intelligent Robots and Computer Vision XVII*, volume 3522, pages 130–138, 1998.

[19] J. K. Dickinson and G. K. Knopf. A moment based metric for 2-D and 3-D packing. *European Journal of Operational Research*, 122(1):133–144, 2000.

[20] J. K. Dickinson and G. K. Knopf. Packing subsets of 3d parts for layered manufacturing. *International Journal of Smart Engineering System Design*, 4(3):147–161, 2002.

[21] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri. Computing the intersection-depth of polyhedra. *Algorithmica*, 9:518–533, 1993.

[22] K. A. Dowsland and W. B. Dowsland. Solution approaches to irregular nesting problems. *European Journal of Operational Research*, 84:506–521, 1995.

[23] K. A. Dowsland, W. B. Dowsland, and J. A. Bennell. Jostling for position: Local improvement for irregular cutting patterns. *Journal of the Operational Research Society*, 49:647–658, 1998.

[24] K. A. Dowsland, S. Vaid, and W. B. Dowsland. An algorithm for polygon placement using a bottom-left strategy. *European Journal of Operational Research*, 141:371–381, 2002.

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

[26] H. Dyckhoff, G. Scheithauer, and J. Terno. Cutting and packing: An annotated bibliography. In M. Dell'Amico, F. Maffioli, and S. Martello, editors, *Annotated Bibliographies in Combinatorial Optimizations*, pages 393–412. John Wiley & Sons, 1996.

[27] J. Edmonds and E. L. Johnson. Matching, Euler tours and the Chinese postman. *Mathematical Programming*, 5:88–124, 1973.

[28] J. Egeblad, B. K. Nielsen, and A. Odgaard. Metaheuristikken *Guided Local Search* anvendt på pakning af irregulære polygoner, 2001.

[29] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems, part II: The Rural Postman Problem. *Operations Research*, 43:399–414, 1995.

[30] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Petropolitanae*, 8:128–140, 1736.

[31] R. J. Fowler, M. S. Paterson, and S. L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[32] O. Færø, D. Pisinger, and M. Zachariasen. Guided local search for the three-dimensional bin packing problem. To appear in *INFORMS Journal on Computing*, (2002).

[33] M. Garey and D. Johnson. *Computers and intractability. A Guide to the theory of NP-completeness.* W. H. Freeman and Company, New York, 1979.

[34] A. M. Gomes and J. F. Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141:359–370, 2002.

[35] R. Heckmann and T. Lengauer. A simulated annealing approach to the nesting problem in the textile manufacturing industry. *Annals of Operations Research*, 57:103–133, 1995.

[36] R. Heckmann and T. Lengauer. Computing closely matching upper and lower bounds on textile nesting problems. *European Journal of Operational Research*, 108:473–489, 1998.

[37] J. Heistermann and T. Lengauer. The nesting problem in the leather manufacturing industry. *Annals of Operations Research*, 57:147–173, 1995.

[38] E. Hopper and B. C. H. Turton. A review of the application of meta-heuristic algorithms to 2D strip packing problems. *Artificial Intelligence Review*, 16:257–300, 2001.

[39] I. Ikonen, W. E. Biles, A. Kumar, J. C. Wissel, and R. K. Ragade. A genetic algorithm for packing three-dimensional non-convex objects having cavities and holes. In *Proceedings of the 7th International Conference on Genetic Algortithms*, pages 591–598, East Lansing, Michigan, 1997. Morgan Kaufmann Publishers.

[40] P. Jain, P. Fenyes, and R. Richter. Optimal blank nesting using simulated annealing. *Journal of mechanical design*, 114:160–165, 1992.

[41] S. Jakobs. On genetic algorithms for the packing of polygons. *European Journal of Operational Research*, 88:165–181, 1996.

[42] J. K. Lenstra and A. H. G. Rinnooy Kan. On general routing problems. *Networks*, 6:273–280, 1965.

[43] Z. Li and V. Milenkovic. Compaction and separation algorithms for non-convex polygons and their applications. *European Journal of Operational Research*, 84:539–561, 1995.

[44] A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141:241–252, 2002.

[45] H. Lutfiyya, B. McMillin, P. Poshyanonda, and C. Dagli. Composite stock cutting through simulated annealing. *Journal of Mathematical and Computer Modelling*, 16(2):57–74, 1992.

[46] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983.

[47] V. J. Milenkovic. Rotational polygon containment and minimum enclosure using only robust 2D constructions. *Computational Geometry*, 13:3–19, 1999.

[48] V. J. Milenkovic. Densest translational lattice packing of non-convex polygons. *Computational Geometry*, 22:205–222, 2002.

[49] V. J. Milenkovic, K. M. Daniels, and Z. Li. Automatic marker making. In *Proceedings of the Third Canadian Conference on Computational Geometry*, pages 243–246. Shermer, Ed., Simon Fraser University, Vancouver, B.C., 1991.

[50] J. F. Oliveira and J. S. Ferreira. Algorithms for nesting problems. *Applied Simulated Annealing*, pages 255–273, 1993.

[51] J. F. Oliveira, A. M. Gomes, and J. S. Ferreira. TOPOS - a new constructive algorithm for nesting problems. *OR Spektrum*, 22:263–284, 2000.

[52] T. Osogami. Approaches to 3D free-form cutting and packing problems and their applications: A survey. Technical Report RT0287, IBM Research, Tokyo Research Laboratory, 1998.

[53] S. Skyum. A simple algorithm for computing the smallest enclosing circle. *Information Processing Letters*, 37(3):121–125, 1991.

[54] Y. Stoyan, M. Gil, T. Romanova, J. Terno, and G. Scheithauer. Construction of a Φ-function for two convex polytopes, 2000.

[55] V. E. Theodoracatos and J. L. Grimsley. The optimal packing of arbitrarily-shaped polygons using simulated annealing and polynomial-time cooling schedules. *Computer methods in applied mechanics and engineering*, 125:53–70, 1995.

[56] Chris Voudouris and Edward Tsang. Guided local search and its application to the traveling salesman problem. *European Journal of Operational Research*, 113:469–499, 1999.

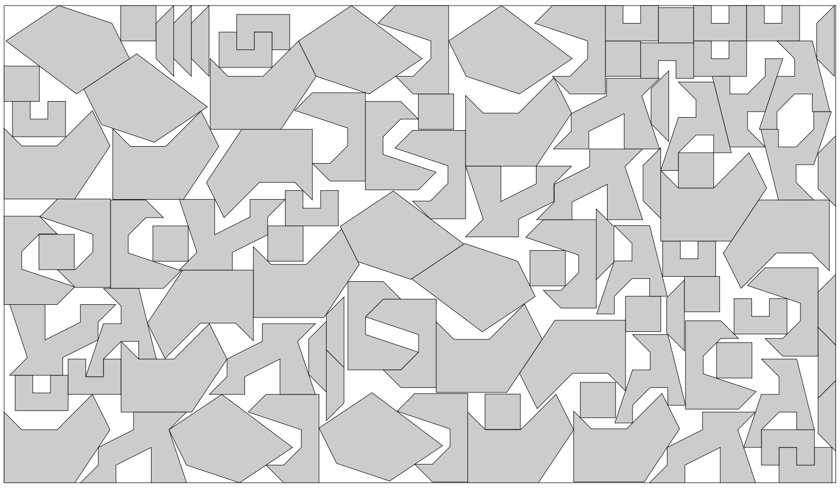
[57] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *H. Maurer (Ed.), New Results and New Trends in Computer Science, LNCS 555*, pages 359–370. Springer-Verlag, 1991.

[58] X. Yan and P. Gu. A review of rapid prototyping technologies and systems. *Computer Aided Design*, 28(4):307–318, 1996.

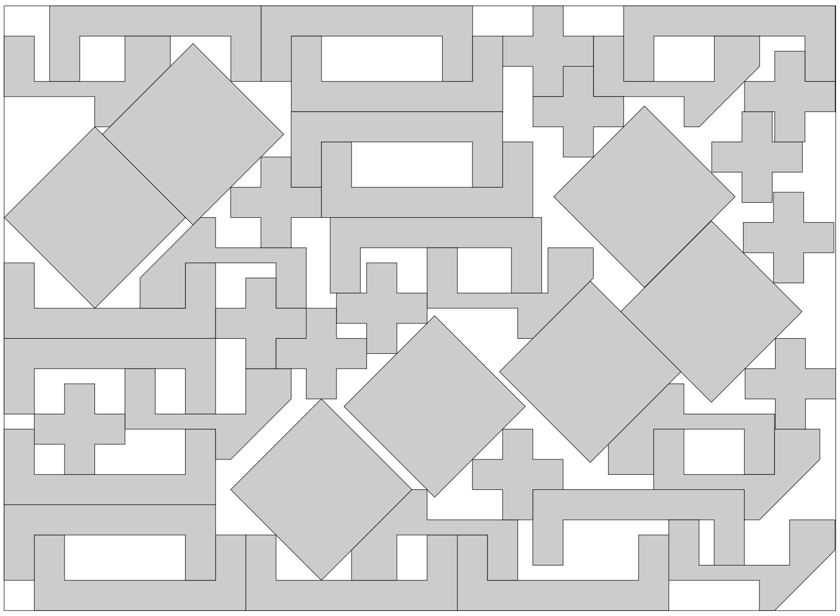
Appendix A

Placements

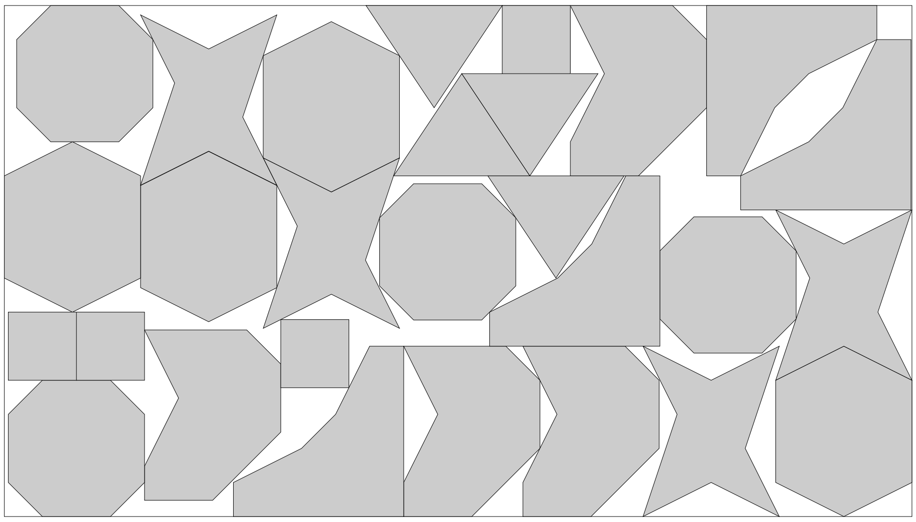
Artificial



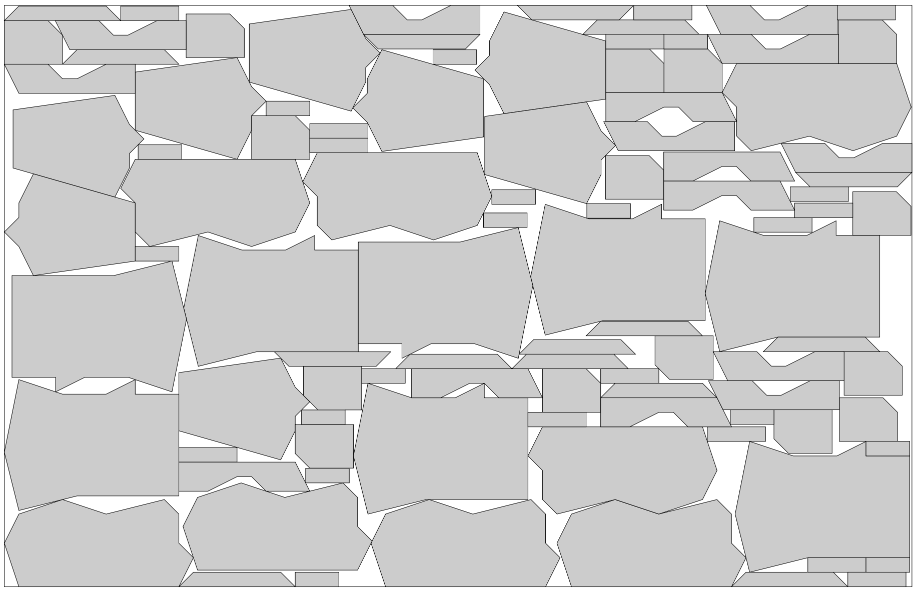
Shapes0



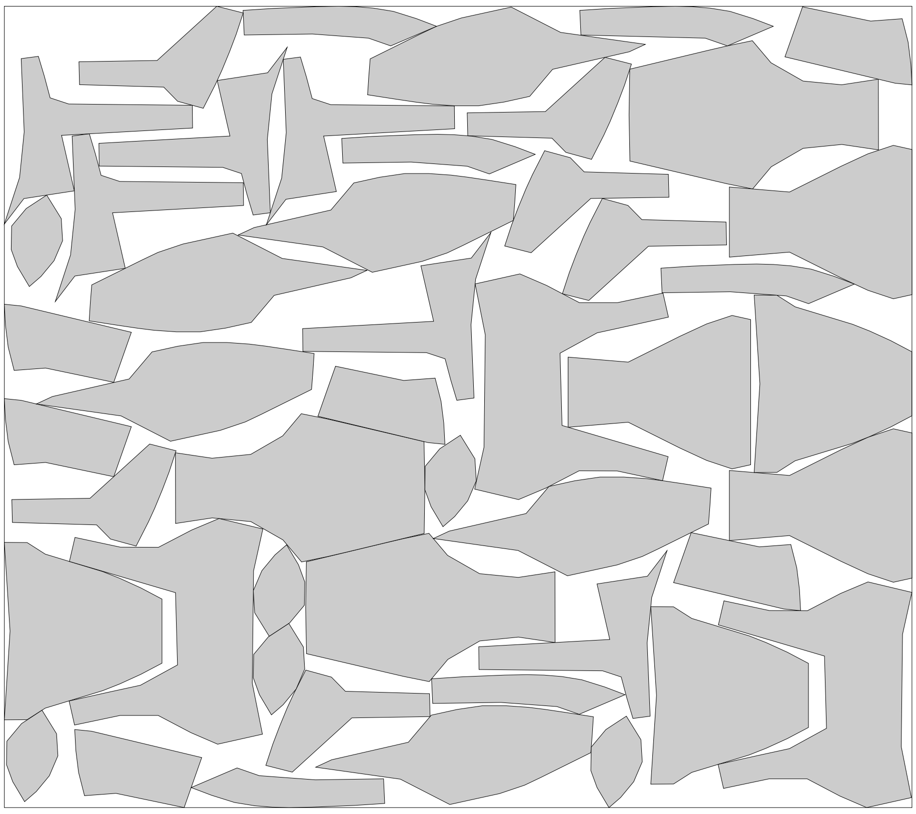
Shapes2



Shirts



Swim



Trousers