A Fast 2D Packing Procedure for the Interactive Design and Preparation of Laser-cut Objects for Fabrication

by

Adrian Reginald Sy

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . .
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by . . . .
Stefanie Mueller
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

A Fast 2D Packing Procedure for the Interactive Design and Preparation of Laser-cut Objects for Fabrication

by

Adrian Reginald Sy

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The 2D packing problem is commonly encountered in various CNC manufacturing workflows such as laser cutting. The goal of this optimization problem is to place a set of irregular polygons onto a material sheet without overlapping and in a way that minimizes wasted material.

In this thesis, we delve into a fast 2D packing algorithm that automatically arranges parts of a user's design onto their available material sheets to provide live feedback on how much material they have left. To do this, we make use of the concept of the no fit polygon to generate candidate placements, and we utilize custom heuristics and performance optimization to improve packing quality and runtime .

Our technical evaluation shows that the implementation is fast enough to work at interactive speeds and produce solutions of similar quality to other tools and algorithms that perform automated packing.

Thesis Supervisor: Stefanie Mueller
Title: Assistant Professor

Acknowledgments

First of all, I would like to extend my deepest gratitude to my advisor, Professor Stefanie Mueller. She has been especially supportive during these trying times and has always been available to give guidance and instruction. I would also like to give the hugest of thank yous to Ticha Sethapakdi and Daniel Anderson without whom this thesis could not have been possible. Ticha was the mastermind of Fabricaide and its user interface, and also somehow simultaneously manages to be one of the coolest and one of the most awkward people I know. Dan was always available to provide guidance and support in wrangling with C++ and other difficult aspects of the implementation. Even outside the context of working on Fabricaide, Ticha and Dan have been great friends and role models throughout the time I got to know them. I could not have asked for a better group of mentors and colleagues than Professor Mueller, Ticha and Dan.

I am extremely grateful to my friends and family both here in the MIT community as well as back home in the Philippines all the way at the other side of the world. I am thankful for all the love and concern I've received from them in spite of current circumstances.

Finally, I would like to thank the rest of the Human Computer Interaction Engineering Group for creative such a fun and welcoming environment. Be it sharing and discussing ideas during lab meetings or hitting the dance floor with Just Dance for celebrations, I have never had a boring moment with this great group of people.

Language and Figure Note

Throughout this thesis, I use the word "we" to signify that this work has been a group effort. However, the work that is provided in detail is my own. Figures and images were jointly created by the project team.

Contents

List of Figures

List of Tables

Introduction

In the context of laser cutting, minimizing the amount of material wasted when cutting two-dimensional (2D) parts from 2D sheets is an important task. The problem of placing small irregular shaped parts onto larger sheets to produce an overlap-free arrangement is often referred to as the nesting problem or the packing problem.

In recent years, several research projects have investigated how to help laser cutter users organize finished designs onto their existing materials. Tools, such as VisiCut [32] and PacCam [35], for instance, provide interfaces that help users with positioning their designs onto material sheets once they are ready to fabricate them. Other tools such as Deepnest [34] and SVGnest [33] automate the packing process within a sheet given a finished design. While such interfaces aid users in planning their design at the time of fabrication, tools that do so during the design phase itself are under-explored.

To explore solutions to this problem that operate during the design phase, we built Fabricaide, an end-to-end system that uses its knowledge of the user's available material sheets to interleave the design creation and fabrication preparation processes. At the core of Fabricaide, we implement a custom 2D packing algorithm that optimizes the placement of parts onto material sheets with pre-existing holes at interactive speeds. This allows users to receive continuous feedback on how the parts of their designs can be placed into their available sheets or receive warnings in cases where packing is not feasible.

While the nesting problem that our algorithm tries to solve has been the focus of research for other industry contexts such as that of garment and leather manufacturing, the constraints we need to address differ from other applications. Firstly, the base surfaces in consideration are bounded rectangles which may include holes of irregular shapes. Secondly, the parts a user would like to cut may require more than a single material sheet, thus requiring considering multiple base surfaces. Lastly, the packing algorithm must run within interactive speeds to be able to display continuous feedback to a user as they are creating their designs. These constraints are not the case in other use cases of the nesting problem such as in garment manufacturing where the goal is to minimize the length of a single strip of material of fixed width, and the packing is typically run offline.

In this thesis, we discuss the custom 2D packing algorithm that drives Fabricaide, its implementation, and its application within Fabricaide. We assess the algorithm's performance and how it compares against similar tools and algorithms through a technical evaluation, as well as a qualitative user study to see its performance in practice in the context of Fabricaide.

Related Work

Research related to our work falls under two categories: tools that assist in the part placement process of laser cutting and approaches on 2D part placement.

2.1 Preparing Designs for Laser Cutting

Several research projects have explored how to help users prepare finished designs for fabrication.

To assist in the process of aligning designs onto sheets with existing holes, VisiCut [32] and PacCam [35] make use of an overhead camera to capture the inside of the laser cutter and display the camera image within the laser cutting software, allowing users to manually align the parts of their design to accommodate for holes. VAL [40] takes this a step further by using an overhead projector to project the output onto the sheet before cutting, allowing users to better understand the scale and placement of their design on the sheet. MARCut [25] moves the positioning process entirely into the physical space by asking users to place physical markers on the sheet to indicate where the shape should be cut. Koo et al. [26] incorporate the notion of fabrication constraints to minimize wastage for furniture design. Given a design, their software suggests modifications that preserve the structural properties of the furniture but require less material wastage to produce.

While these methods bring together the placement constraints and the parts to be placed by either showing the sheets and holes in the digital space or letting users display the placements in the physical space, most of these projects still requires the user to manually choose how to place each part.

Furthermore, as the methods above help users with placement after they have finished their design, they come at the expense of users not knowing whether their design is feasible with the available materials until after they are ready to prepare it for fabrication. Without the foresight for how their design can fit onto the material sheet, designers have to go through a trial-and-error process of adjusting and checking the parts of their design until it is feasible.

2.2 Arranging Parts of a 2D Design

Producing overlap-free arrangements of 2D geometries is broadly known as the 2D *packing* problem, the *nesting* problem, the *irregular cutting stock* problem, or the *pattern marking* problem, depending on the field, and has a range of applications in different domains that use cutting machines. The packing problem, even in the 1D case, is NP-hard [19]. The earliest-known attempt to solve this problem algorithmically is a paper of Kantorovitch [24], which gives a solution for packing orthogonal rectangles. Art [4] describes an algorithm for packing convex polygons onto rectangular sheets, applied to garment manufacturing. Baldacci et al. [6] use a rasterization approach to represent pieces of leather and shapes to be cut out of them to handle the nesting problem in the context of leather manufacturing.

A number of algorithms for packing polygons (see e.g. [2, 31, 7, 30]) involve a concept called the *no fit polygon* (NFP) [4], a useful tool for detecting overlap and producing candidate placement locations for sets of polygons. The NFP of a pair of polygons, each of which is in a fixed orientation, represents the set of possible relative arrangements of the polygons such that they touch but do not overlap (Figure 3-1). Approaches for using the NFP for polygon packing typically select an initial order and orientations of the polygon, then place them one after another, calculating the NFP to determine candidate locations. Modern approaches to the problem have tried integrating techniques such as heuristic search [3], and meta heuristics [9, 12, 10].

Packing non-convex polygons is much harder than packing convex polygons, and early solutions were only capable of handing few polygons at a time. An early approach of Daniels and Milenkovic [15] based on Mixed-integer Programming has been improved by Fischetti and Luzzi [17] to make it more practical. Work by Burke and Kendall [11, 13] uses metaheuristic algorithms and an approximate NFP to give one of the first highly practical solutions to the problem, on which some popular open-source software projects, such as Deepnest [34] and its browser-based counterpart SVGNest [33], are based.

The above algorithms, however, are not designed for situations where the target sheets may contain holes or defects that must be avoided. Early work by Heistermann and Lengaeur [22] approaches this in the context of leather manufacturing. Babu and Babu [5] give a heuristic algorithm for 2D packing that can also handle defects. Baldacci et al. [6] further explore this problem and develop a layered algorithm that uses heuristics to distribute parts onto multiple sheets, and a guided local search based on Lagrangian relaxation to optimize the placement of parts onto an assigned sheet.

These algorithms tend to be targeted towards industrial manufacturing settings which prioritize material-saving over interactivity, so none of them attempt to achieve interactive speeds. For an interactive design tool, our main goal is to achieve interactive speeds to provide the user with continuous feedback, with material-saving being a secondary goal.

Packing Algorithm

3.1 Preliminaries

3.1.1 No Fit Polygons

Many algorithms for general non-convex 2D packing are based on the concept of the *no fit polygon* (NFP).

We first review the NFP and describe the general techniques that are used to design algorithms for 2D packing. Given a fixed-position polygon A and a polygon B to place, we choose a vertex of B as the *reference vertex* p. The NFP of A and B is then defined as the set of p's location under all arrangements of B such that B and A touch but do not overlap. See Figure 3-1 for an example. Observe, importantly, that the interior of the no fit polygon denotes all of the points at which it would be illegal to place B, else it would overlap with A, and the exterior denotes all of the points at which placing B would cause it to not touch A. Placing B precisely on the boundary of the NFP means that the two will touch but not overlap.

More generally, given a set of polygons $P_1, ..., P_n$ and a polygon P, the boundaries of the union of the NFPs of P and $P_1, ..., P_n$ give the locations at which P can be placed such that it touches at least one of but does not overlap any of $P_1, ..., P_n$. See Figure 3-2 for an example. Note that since the boundaries of the NFPs (the candidate locations at which to place the new polygon) is an infinite set, the algorithms typically consider only the vertices as candidate locations. Lastly, to prevent the newly placed polygon from leaving the bin, an *inner fit polygon* (IFP) is computed, which is the interior analog of the NFP, i.e. it is the set

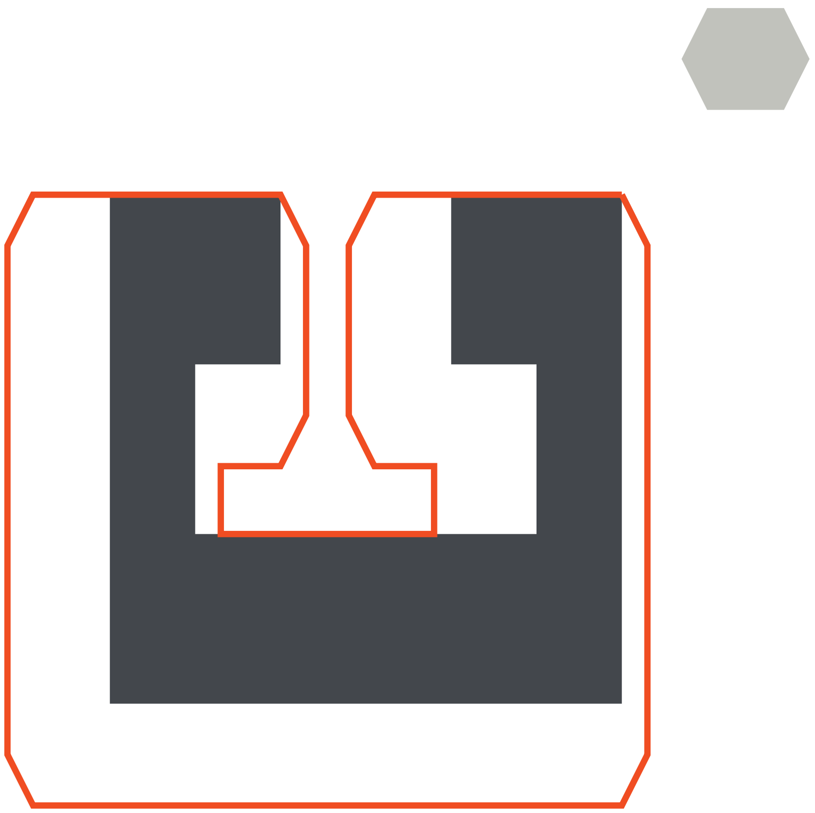


Figure 3-1: A fixed-position polygon (black), a polygon to place (gray), and their NFP (red). The reference vertex of the polygon to place is the bottom-left corner.

of points at which a polygon can be placed inside another without leaving it. See Figure 3-3 for an example.

Computing NFPs efficiently has been the subject of much research. The most successful approaches have been the orbital method [28, 39, 13], methods based on Minkowski Sums [29, 7, 8], and decomposition-based approaches [27, 37]. These approaches can also be extended to shapes that are polygons with internal holes [7].

3.1.2 Applying NFPs to the Packing Problem

Armed with the no fit polygon technique, algorithms for optimizing packings often follow a greedy strategy combined with meta heuristics [11]. Given a fixed ordering of a set of polygons, the algorithms place each one sequentially. When placing a new polygon, its NFPs with respect to the previously placed ones are computed in order to determine the candidate locations (Figure 3-2). A candidate location is then selected by minimizing some

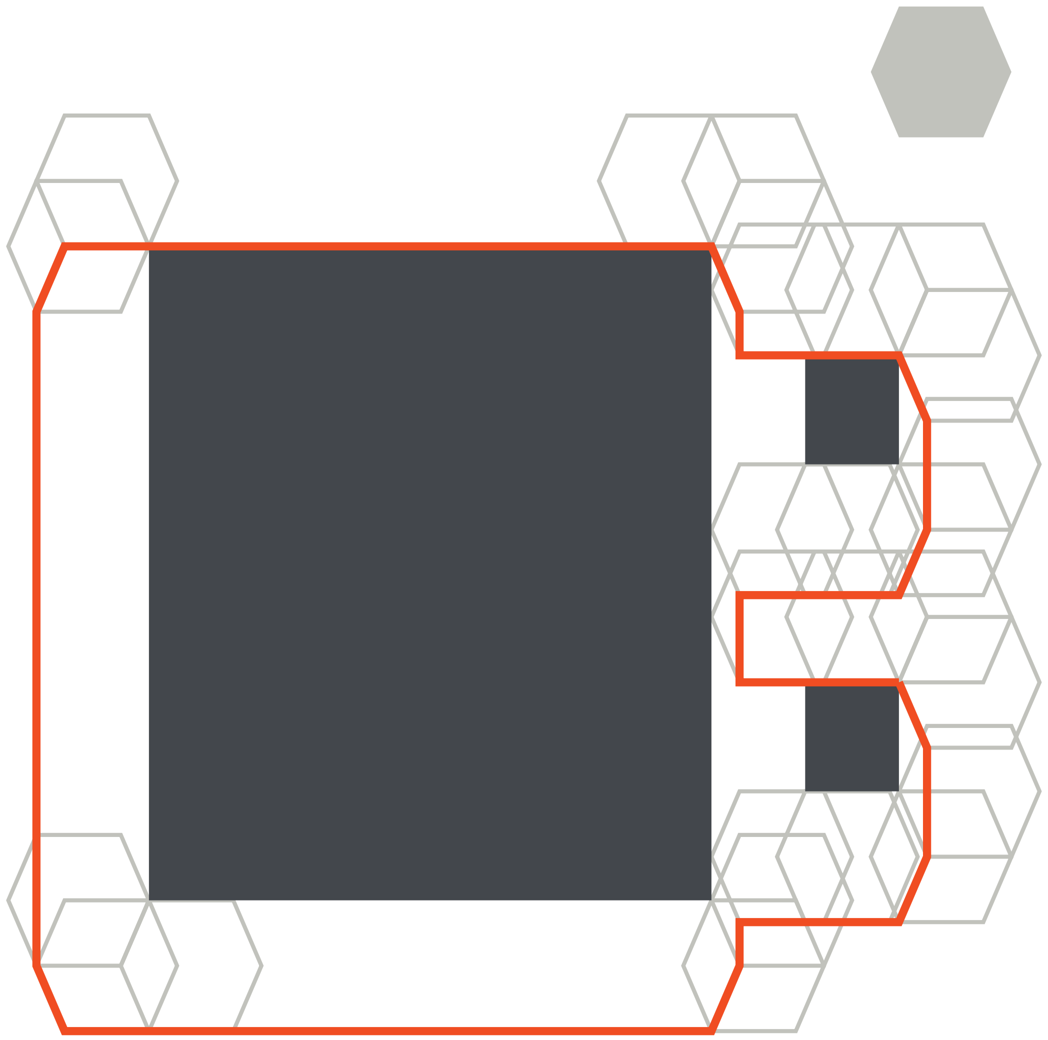


Figure 3-2: A set of fixed-position polygons (black), a polygon to place (gray), the union of the NFPs (red), and the resulting candidate placement locations (gray outlines). The reference vertex of the polygon to place is the bottom-left corner.

objective score, often the size of the bounding box or convex hull of the placed polygons, although more complicated heuristics have also been used [21]. Finally, a metaheuristic, such as a genetic algorithm or local search is employed to optimize the fixed order of the polygons and their rotations.

3.1.3 Bin Selection

The packing problem that our algorithm covers is a specific case of the general *bin packing problem* where items of different sizes must be packed into a finite number of bins of fixed size. Generally, the metric to be minimized is the number of bins used.

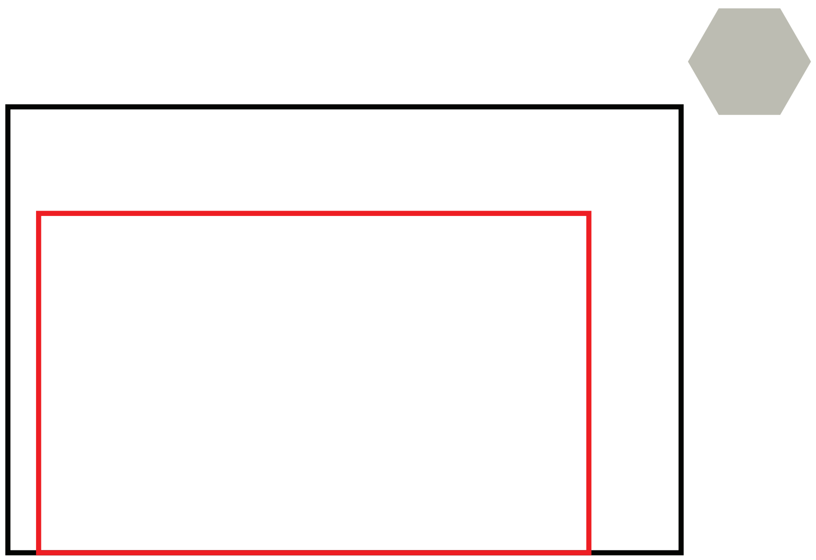


Figure 3-3: A fixed-position polygon (black), a polygon to place (gray), and their IFP (red). The reference vertex of the polygon to place is the bottom-left corner.

For this application, since parts to be placed can span more than a single bin, it is also important to consider how to select the bin in which each part is placed. In the context of the 1D bin packing problem, Johnson explores several variations of the algorithm with different bin selection strategies [23]. Some of the more well-known approaches are Next-fit and First-fit.

For Next-fit, the parts to be placed are considered one at a time and placed onto the first sheet. Once a part is unable to fit within a bin, that bin is considered closed and all succeeding parts will no longer consider that bin. Because each bin can only fail to fit at most one part, the runtime is proportional to the sum of the number of bins and the number of parts.

On the other hand, first-fit tries to place the part in the first bin that can fit the part. In the worst-case, the algorithm can try to place every part into every bin increasing the runtime of the algorithm depending on the complexity of the attempted placement. On the other hand, first-fit tends to give better quality solutions due to checking more bins compared to next-fit.

3.2 Overview

We implemented a custom 2D packing algorithm which is based on the NFP approach [4] with custom heuristics and some additional performance optimizations. The performance-critical part of the algorithm is implemented in C++ using Boost Python [1] to provide a Python interface. We use the CGAL [36] library for computational geometry primitives. Pre- and post-processing, such as discretizing SVG elements into polygons, applying dilation to account for laser cutter tolerance, and formatting the output is implemented in Python.

At a high level, our algorithm is an application of the no fit polygon approach in the more general setting in which the target bins can vary in shape and size and contain arbitrary existing holes. It utilizes the classic first-fit decreasing heuristic for bin packing [23] generalized to the 2D case. Given a set of parts to place, the algorithm greedily places each part in decreasing order of size (biggest to smallest by the area of their bounding box) onto the first sheet on which it fits, using NFPs to find feasible candidate locations, and a heuristic quality score to select the best candidate location. Given a set of placed polygons P and holes H, our main heuristic is to minimize the sum of the area of the bounding box of the placed parts and the area of the bounding box of both the placed parts and the holes in the sheet.

We improve the performance of the algorithm by caching the computations of NFPs, not just within a run, as has been done before by Burke and Kendall [11], but across separate runs of the algorithm. Since subsequent runs of the algorithm will likely be ran on designs that have only changed a small amount, most of the same polygons will be present.

3.3 Implementation

For performance purposes, the core of the implementation is written in C++ and makes use of the CGAL computational geometry library for Minkowski sum calculation and polygon representation [38]. The packing algorithm is made accessible through a Python interface that performs preprocessing on SVG files that interacts with the C++ library via Boost-Python bindings [1].

3.3.1 Sheets with Holes

One of the problems we aim to solve is to be able to pack parts of a users design even when the sheets being packed into contain pre-existing holes. Generalising the NFP approach to handle arbitrary bins with existing holes consists of two steps. First, the placement routine must be generalised to account for the differing bin shapes and existing holes. To do this, for each part to be placed, the NFPs of that part with respect to the holes must also be calculated in addition to the NFPs with respect to previously placed parts. Taking the union of these NFPs as in the original approach generates candidate locations that will overlap with neither the previously placed parts nor the holes. Second, the objective score must be adjusted in order to ensure that good quality placements are chosen, since the standard objective functions will not necessarily yield good results in this context.

3.3.2 Basic Part Placement Algorithm

Pseudocode for the building block of our part placement algorithm is shown in Algorithm 1. Given a sheet and a part to be packed into the sheet, it determines the best placement of that part that minimizes the score of the resulting configuration as calculated by a heuristic function SCORESHEET.

We make use of the following primitives. UNION computes the union of a list of polygons (possibly with holes), while DIFFERENCE computes the difference of a pair of polygons. These functions are standard in typical computational geometry libraries. IFP and NFP compute the inner fit polygon and no fit polygon of the given polygons respectively.

Such functions are not typically included in most computational geometry libraries, but are implementable in terms of Minkowski sums. For this purpose, we use the Minkowski sum algorithm implemented in CGAL [38].

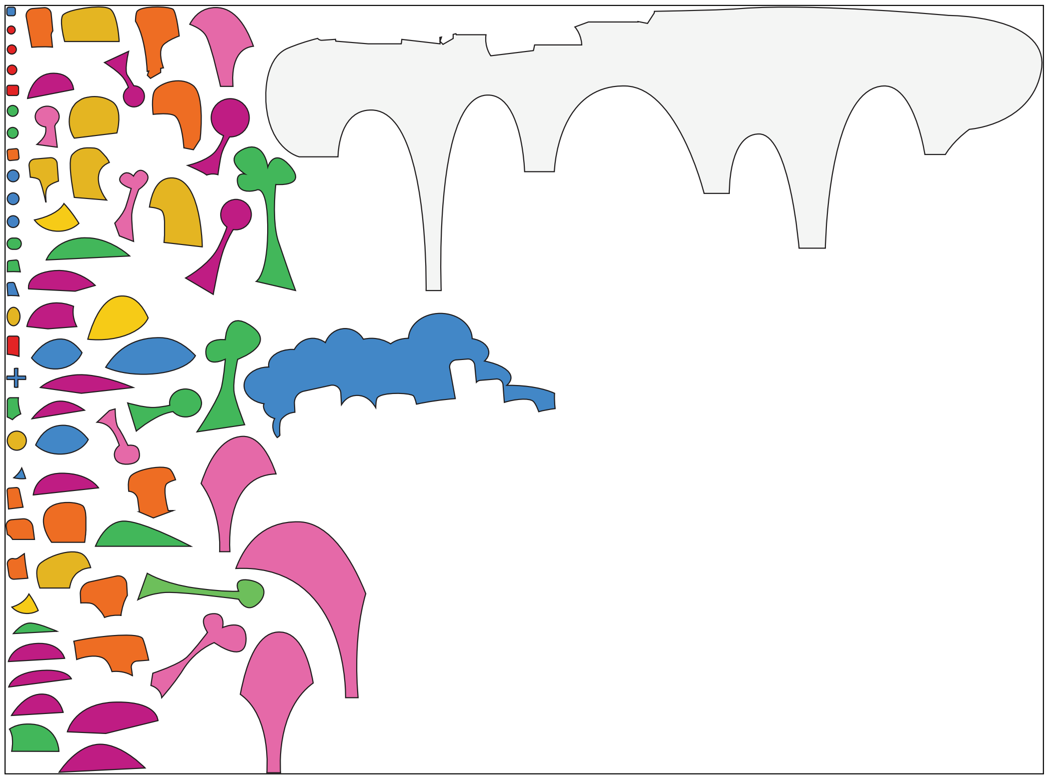
Algorithm 1 Basic Part Placement Algorithm

```
1:  procedure COMPUTEPLACEMENT(sheet, part)
2:      local binNFP = IFP(sheet.boundary, part)
3:      // Compute the candidate positions
4:      local nfps = empty list
5:      for each shape in sheet.holes ∪ sheet.parts do
6:          nfps.push(NFP(shape, part))
7:      local shapesNFP = UNION(nfps)
8:      local candidates = DIFFERENCE(binNFP, shapesNFP)
9:      // If the part fits nowhere, fail the packing
10:     if candidates is empty then return false
11:     // Select the best position on the sheet
12:     sheet.parts.push(part)
13:     local best_position = part.position
14:     local best_score = ∞
15:     for each polygon in candidates do
16:         for each vertex in polygon do
17:             part.position = vertex
18:             score = SCORESHEET(sheet)
19:             if score ≤ best_score then
20:                 best_position = vertex
21:                 best_score = score
22:     part.position = best_position
23:     return best_position
```

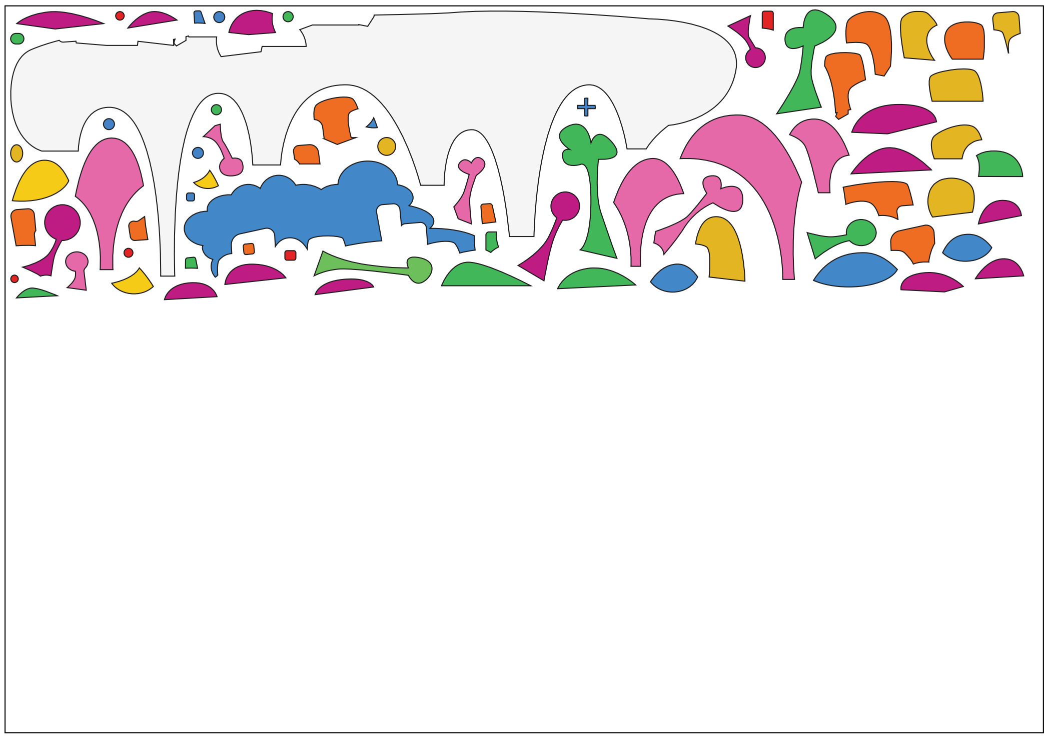
The runtime for this procedure scales linearly with the number of previously placed parts and holes. This is because a NFP must be computed for each of part and hole, and the union of these NFPs must be computed.

3.3.3 Order and Rotations of Parts

While the part placement method in Algorithm 1 optimizes part placement for a single part, it does not take into account the order of the parts nor does it take into account possible rotations of parts for better part placement. Depending on the order in which parts are placed, the packing may become noticeably inefficient (Figure 3-4). As such, we also need to optimize the placement order and rotations of the shapes.



(a) Parts packed in increasing bounding box size order. 52.64% of material remaining.



(b) Parts packed in decreasing bounding box size order. 59.15% of material remaining

Figure 3-4: Example of sub-optimal packing of one of P1's designs if shape order is poorly chosen for greedy part placement.

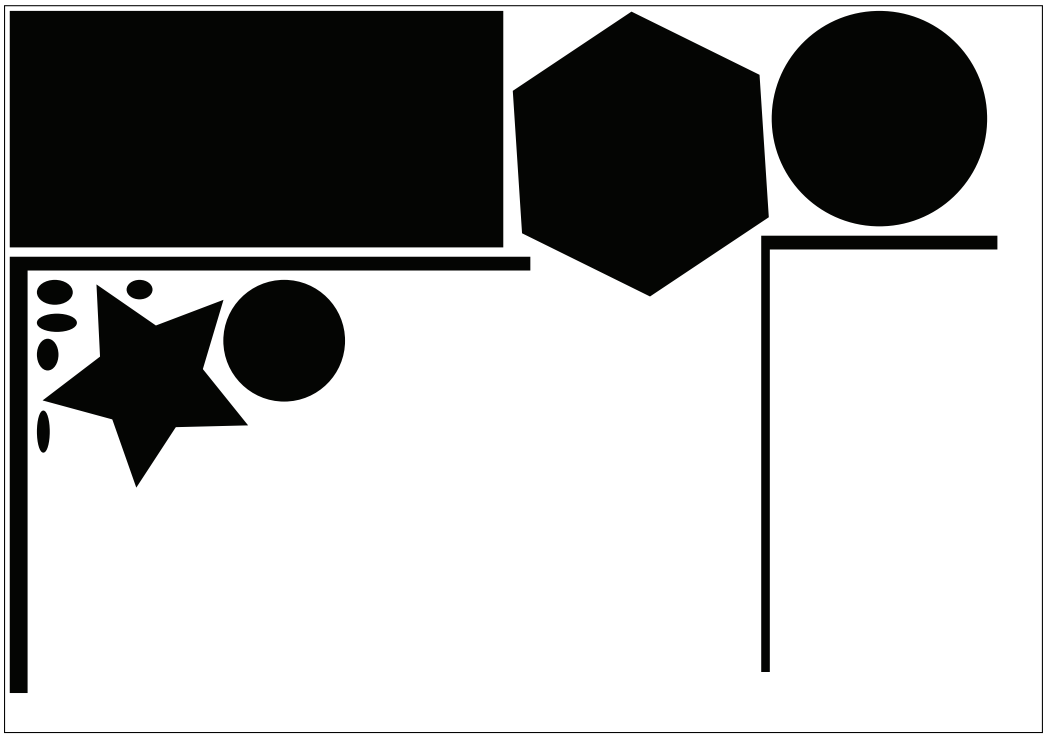
One consideration was to use an online algorithm for part placement, where the parts are considered in the order in which they are drawn. While this is in line with an interactive system due packing parts as they are drawn or modified by the user, this does not work very well in the context of continuous feedback. Unlike other contexts where online approaches are used such as the bin packing problem, not only are new parts added but previously inserted parts can also be modified or deleted, with the feedback encouraging users to edit their design when errors occur. This would require frequent repacking of all parts after a part that is deleted or modified. As such, we opted for an offline approach where we repack all parts every time the algorithm is run.

Previous work on the packing problem such as that of Burke and Kendall [14] and open-source packing software such as Deepnest [34] and SVGNest [33] utilize metaheuristic methods such as genetic algorithms or local search to optimize the order the parts to be placed, i.e. the part placement algorithm is run on arbitrarily many different orderings of parts to search for better quality solutions. In this case, the search space is over all possible orderings of parts ($O(n^n)$ where n is the number of parts) as well as all possible rotations for each part ($O(r^n)$ where r is the number of rotations to be considered).

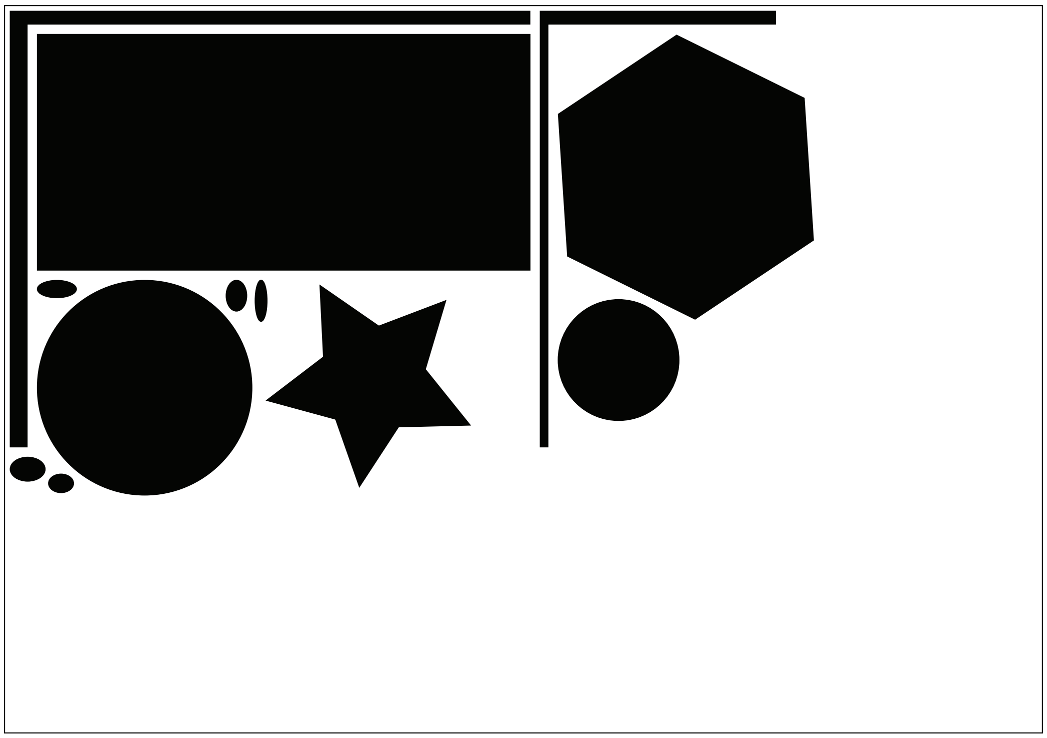
Since finding an optimum solution for maximum space-saving is a secondary goal as opposed to the primary goal of attaining interactive speeds for continuous feedback, we opted against using a metaheuristic approach for packing parts because of the large number of iterations that would be required to cover a sufficiently large fraction of the search space.

We decided on using a modified first-fit decreasing heuristic for bin packing [23] where parts are ordered in decreasing size based on the sizes of their bounding boxes then greedily packed into the sheet. As opposed to using the area of the part directly, we opted to use the area of the bounding box instead. This makes it such that parts with relatively small areas but large holes or spaces within their bounding boxes are placed early so smaller shapes can fill up these spaces and conserve more usable space (Figure 3-5).

In addition, we also implemented the option to consider rotations of a part within the greedy part placement by exhaustively checking a tunable number of rotations during the insertion of each part (Algorithm 2). Since the parts are placed one at a time, the runtime only scales linearly with the number of rotations to be tested.



(a) Parts packed in decreasing area order.



(b) Parts packed in decreasing bounding box size order.

Figure 3-5: Example of sub-optimal packing when the area is used as opposed to the bounding box when there are shapes with large empty areas within their bounding boxes.

Algorithm 2 Rotated Part Placement Algorithm

```
 1: procedure COMPUTEROTATEDPLACEMENT(sheet, part,rotations)
 2:     // Compute the candidate placements for each rotation
 3:     local candidate_placements = empty list
 4:     for each rotation in rotations do
 5:         part.rotation = rotation
 6:         local best_position = ComputePlacement(sheet, part)
 7:         if best_position then candidate_placements.push((best_position, rotation))
 8:     // If the part fits nowhere, fail the packing
 9:     if candidate_placements is empty then return false
10:     // Select the best position and rotation among the candidates
11:     local best_position
12:     local best_rotation
13:     local best_score = $\infty$
14:     for each (position, rotation) in candidate_placements do
15:         part.position = position
16:         part.rotation = rotation
17:         score = SCORESHEET(sheet)
18:         if score $\leq$ best_score then
19:             best_position = position
20:             best_rotation = rotation
21:             best_score = score
22:     part.position = best_position
23:     part.rotation = best_rotation
24:     return sheet
```

3.3.4 Heuristics

As previously mentioned, the heuristic function, SCORESHEET, used to calculate the score for a sheet with holes H and placed parts P is the following:

$$\textsc{Area}(\textsc{BBox}(P)) + \textsc{Area}(\textsc{BBox}(P \cup H)),$$

where BBOX computes the bounding box of the given set of polygons.

To break ties in the case of multiple equal scores, the algorithm selects the placement location (x, y) that minimizes the value of $x + y$. This has the effect of packing shapes more tightly towards the upper-left corner.

An initial consideration was just to use the bounding box of both the holes and the parts, i.e. $\textsc{Area}(\textsc{BBox}(P \cup H))$, but in sheets where the holes were spaced apart, it resulted in placements of parts that could be anywhere within the bounding box of the holes which was a problem when holes were spaced widely apart (Figure 3-6). In contrast, the chosen heuristic function avoids this problem by having the component $\textsc{Area}(\textsc{BBox}(P))$ to encourage new parts to be placed tightly together, as well as the component $\textsc{Area}(\textsc{BBox}(P \cup H))$ to encourage parts to be placed tightly near existing holes.

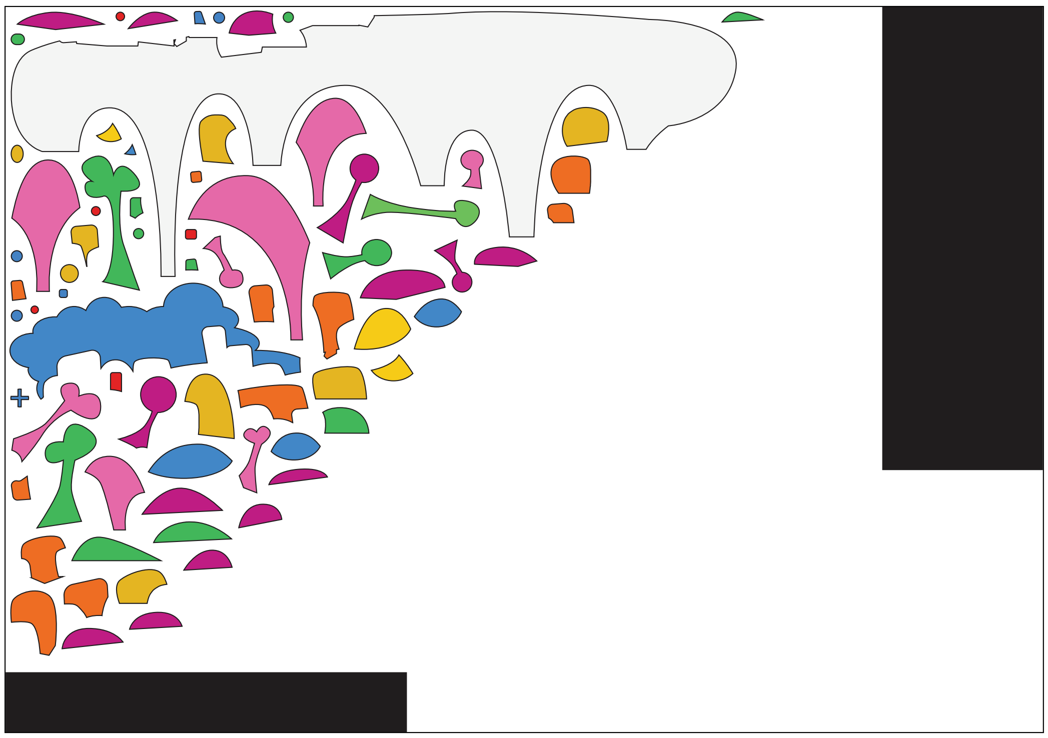
On the other hand, we also considered more complex heuristics such as using a clustering algorithm to determine clusters of shapes and holes, then minimizing the number of clusters and area covered by each cluster. We opted against taking this approach as these were computationally expensive metrics to compute which would have to be recomputed for every candidate location for every part that is placed. Bounding boxes on the other hand are easy to compute due to simply being max and min calculations across the coordinates of the vertices.

3.3.5 Placement across Multiple Sheets

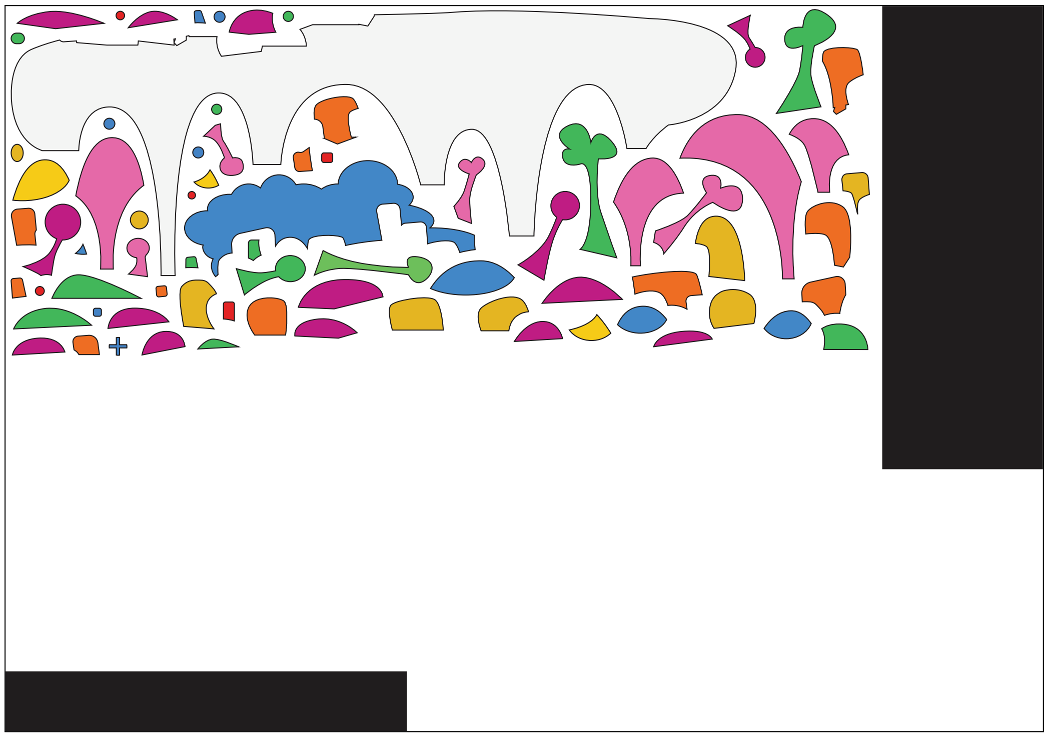
In addition to considering the order in which the parts are to be greedily placed into sheets, we also need to consider how to decide which sheet a part should be packed into. A simple approach is shown in Algorithm 3, where the algorithm uses a next-fit approach to choosing the sheet to pack each part in.

As previously mentioned, one benefit of the next-fit approach is that each sheet will only fail a packing once so the number of times the $\textsc{ComputePlacement}$ has to be run is bounded by the number of parts plus the number of sheets. On the other hand, this may not necessarily yield good results in certain situations. For example, suppose a user wants to place two large shapes and many small shapes into several sheets but the two largest shapes are unable to fit into the first sheet. This would result in the first shape being packed into the first sheet on its own while the algorithm will attempt to pack the remaining shapes into the second sheet and onwards, effectively wasting any remaining space in the first sheet that could have been packed with some of the smaller shapes. This problem is made more likely due to the parts being packed in decreasing size order.

To account for this, we opted to use a first-fit approach instead and check all prior sheets starting from the first sheet until a fit is found as opposed to just the most recently used sheet



(a) Parts packed when the heuristic is just $\textsc{Area}(\textbf{BBox}(P \cup H))$.



(b) Parts packed when the heuristic is $\textsc{Area}(\textbf{BBox}(P)) + \textsc{Area}(\textbf{BBox}(P \cup H))$.

Figure 3-6: Example of sub-optimal packing when the heuristic is only the overall bounding box for a sheet with holes (black).

Algorithm 3 Next-Fit Part Placement Algorithm

```
 1: procedure PlaceParts(sheets, parts)
 2:     local used_sheets = empty list
 3:     while parts is not empty do
 4:         // If we run out of sheets, the parts do not fit
 5:         if sheets is empty then return false
 6:         local sheet = sheets.first
 7:         local part = parts.first
 8:         local best_placement = ComputePlacement(sheet, part)
 9:         // If the part fits nowhere, go to the next sheet
10:         if best_placement is false then
11:             if sheet.parts is not empty then
12:                 used_sheets.push(sheet)
13:             sheets.remove(sheet)
14:             continue
15:         // Remove the part from the list of parts to be considered
16:         parts.remove(part)
17:     return used_sheets
```

when attempting to place a new shape. While at worst, this would require checks against all holes and previously placed parts across sheets for each new part, the number of sheets is usually small and it is only necessary to check sheets until a valid placement is found. The modified algorithm is shown in Algorithm 4.

3.3.6 Partial Packing

To provide more useful feedback for material warnings in the context of Fabricaide, the algorithm has also been modified to output partial results in cases where not all parts could be placed. This is done by simply keeping track of parts for which no valid placement is found as shown in Algorithm 5.

Algorithm 4 First-fit Part Placement Algorithm

```
 1: procedure PLACEPARTS(sheets, parts)
 2:     local used_sheets = empty list
 3:     for part in parts do
 4:         local part_placed = false
 5:         for sheet in sheets do
 6:             local best_placement = COMPUTEPLACEMENT(sheet, part)
 7:             // If the part fits nowhere, go to the next sheet
 8:             if best_placement is false then
 9:                 continue
10:             // Once the part is placed, break out of the loop
11:             part_placed = true
12:             break
13:         // If all the sheets have been checked, the part does not fit
14:         if part_placed is false then return false
15:     for sheet in sheets do
16:         if sheet.parts is not empty then
17:             used_sheets.push(sheet)
18:     return used_sheets
```

3.3.7 Complexity Analysis

The completed algorithm is shown in Algorithm 5. For analyzing the complexity of the algorithm, we assume that the parts have a constant number of vertices and that each sheet has the same number of holes. We also make simplifying assumptions that the complexity of UNION is linear with respect to the number of polygons we are taking the union of, DIFFERENCE is a constant time operation, and the number of candidate points scales linearly with the number of combined NFPs.

We let h be the total number of pre-existing holes, n be the number of parts to place, and r be the number of rotations to be tested.

We note that the bulk of the computation happens in the repeated calculations of COMPUTEROTATEDPLACEMENT. As previously mentioned, the runtime of COMPUTEROTATEDPLACEMENT is proportional to $k+h'$ where k is the number of parts that have already been placed in that sheet and h' is the number of holes in that sheet. Summing this up as k ranges from 0 to $n-1$ and assuming the worst-case of having to generate NFPs against all pre-existing holes, we get an upper bound complexity of $O(n^2+hn)$ when there is only one rotation to consider.

Since the complexity scales linearly with the number of rotations, we get a theoretical runtime of $O(r(n^2 + hn))$.

Algorithm 5 Complete Part Placement Algorithm

```
 1: procedure PLACEPARTS(sheets, parts, rotations)
 2:     // Sort the parts in decreasing order of bounding box area
 3:     parts.sort(part => -AREA(BBOX(part)))
 4:     local used_sheets = empty list
 5:     local failed_parts = empty list
 6:     for part in parts do
 7:         local part_placed = false
 8:         for sheet in sheets do
 9:             local best_placement = COMPUTEROTATEDPLACEMENT(sheet, part, rotations)
10:             // If the part fits nowhere, go to the next sheet
11:             if best_placement is false then
12:                 continue
13:             // Once the part is placed, break out of the loop
14:             part_placed = true
15:             break
16:         // If all the sheets have been checked, the part does not fit
17:         if part_placed is false then failed_parts.push(part)
18:     for sheet in sheets do
19:         if sheet.parts is not empty then
20:             used_sheets.push(sheet)
21:     return used_sheets, failed_parts
```

3.4 Implementation Optimizations

To further improve the performance of the algorithm, we implemented several optimizations in both the algorithm itself, as well as in the preprocessing phase when taking in user input.

3.4.1 Caching of NFPs

The computation of the NFP is one of the most computationally expensive steps in the algorithm due to the complexity in computing Minkowski sums. To be specific, in the case of computing the Minkowski sum of two non-convex polygons with m and n vertices respectively, the complexity of computing the Minkowski sum is $O(m^2n^2)$ [18]. Furthermore, due to chosen offline approach of repacking all the parts every time the user adds or modifies a part, the NFPs of the unchanged parts would be redundantly computed.

Since the algorithm requires the calculation of the NFP of each part with every part preceding it, the number of NFP computations necessary whenever a part is added would be $\binom{n}{2}$ where n is the number of parts being placed. This results in $O(n^3)$ NFP computations across the entire design process in the case where a user adds one part at a time.

Burke and Kendall avoid some of the computational cost by implementing caching of NFPs within each run of the algorithm [11]. In several industry use cases including laser cutting to a lesser extent, there are usually multiple copies of each part thus allowing the computation of NFPs of those shapes to be reused for repeated parts.

Our algorithm also utilizes caching of NFPs but preserves the NFP cache through multiple runs of the algorithm. In the context of Fabricaide, typically only one part is added or modified at a time in between runs of the algorithm. As such, if all prior NFP computations are cached, the only NFPs that need to be computed when a new part is added are NFPs of that part with the other shapes. This reduces the number of NFP computations from $\binom{n}{2}$ to $n-1$. Similarly, the overall number of computations scales quadratically as opposed to cubically with the number of parts.

We note that the complexity of the algorithm does not change as we still need to compute the unions of the NFPs regardless of whether or not they are cached. We opted against caching the unions of the NFPs due to memory concerns. While the number of NFPs we would need to store is at most quadratic to n (one NFP for every pair of parts) where n is the number of parts being placed, the maximum number of unions we would need to store is at worst proportional to the number of subsets of the parts placed which is exponential with respect to n (one deterministically generated arrangement of parts for every subset of parts).

3.4.2 Incremental Heuristic Calculation

In the sheet scoring part of Algorithm 1, the heuristic has to be calculated for every candidate vertex for every inserted part. While computing the bounding box of the union of all inserted parts is a simple computation, this computation scales linearly with the number of parts.

To improve the runtime of this process, we modified the heuristic calculation function to incrementally store the bounding box of all previously inserted parts. This modification makes it such that each new bounding box calculation would only need to compare the coordinates of the previous bounding box and the vertices of the new part as opposed to having to check all the coordinates of all the parts. Assuming a constant number of vertices for each part, this effectively reduces the runtime complexity from $O(n)$ to $O(1)$.

Although there is only one scoring function used by our implementation, should there be a better heuristic to measure how good a packing is, a similar incremental approach could potentially be used to improve runtimes.

3.4.3 Shape Dilation and Simplification

In addition to reducing the number of times the NFP needs to be computed, the runtime of the NFP calculation is also dependent on the complexity of the parts. Computing the NFPs of polygons with fewer vertices is easier than that of polygons with more vertices.

As such, we preprocess the shapes by dilating them and applying polygon simplification with tunable dilation distance and curve tolerance parameters. The preprocessing is done in Python using the Shapely [20] library. By default shapes are dilated by 5 points with a curve tolerance of 2.5 points to account for required distance between placed parts for laser cutting kern tolerances.

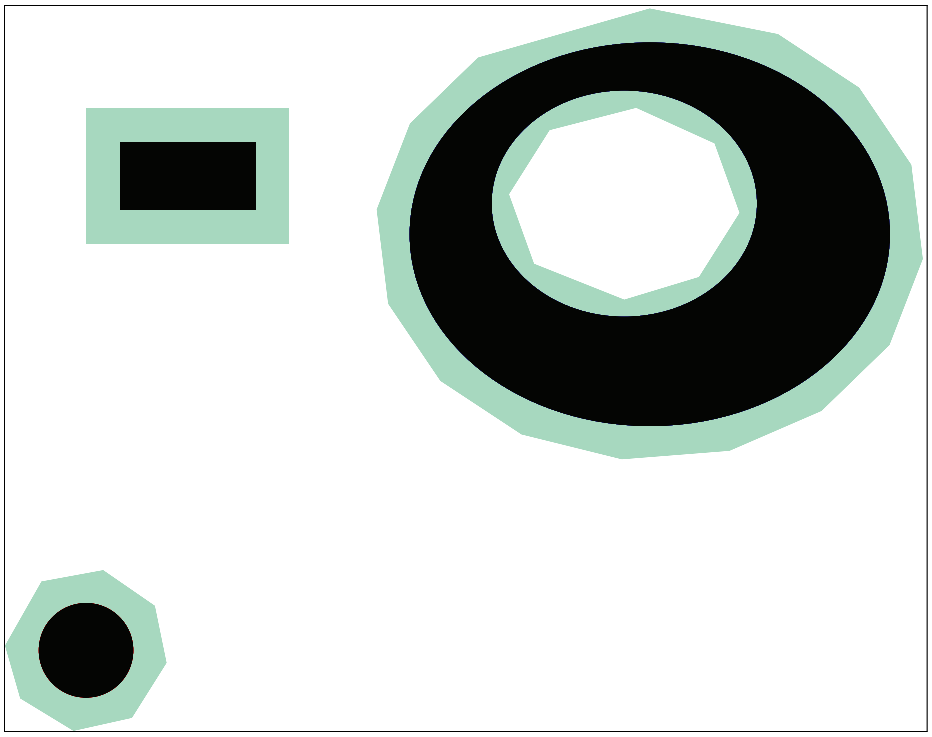


Figure 3-7: Original parts (black) with their dilated and simplified versions (green).

3.5 Technical Evaluation of Part Placement Algorithm

In the context of Fabricaide, the part placement algorithm needs to be fast enough to not be a hindrance to interaction without overly sacrificing the quality of the solutions. In this section, we present a quantitative evaluation of the performance of our implementation to demonstrate its suitability to this task.

3.5.1 Benchmark Dataset

To ensure that our benchmarks use shapes that are typical of laser-cut designs, we create datasets using the designs created by our user study participants, which, in total, contained 246 parts. To build our dataset, we created 10 sets of 100 randomly sampled parts of the users' designs. When simplified with the default curve tolerance of 2.5pts (0.88 mm) and ilation width of 5pts (1.76 mm), the parts had an average of 27 vertices, with 54% of them being non-convex, and 13% containing composite holes. The target sheet for each set is a 5000pts by 5000pts blank sheet which is sufficiently large so as to not constrain the solution.

3.5.2 Benchmark Setup

We ran our experiments on a personal desktop computer equipped with an AMD Ryzen 5 2600X processor and 16GB of DDR4 main memory. Since the packing procedure is designed to run seamlessly in the background without lagging the design tool, we avoid the use of multithreading. For each of the 10 sets of parts in our dataset, we compared the runtime performance of the algorithm with and without cross-run caching. In both cases, we measured the time taken to pack only one part (representing the first step in the design process), then pack two parts, and so on (representing an extension of the design over the entire design process). This simulates how the algorithm would typically work during the design process where users would draw one part at a time. Furthermore, we also evaluate how much cross-run caching improves performance, and observe the scaling of the algorithm with respect to the number of parts to be placed. Rotations of parts were not considered, since they just increase the runtime linearly and hence are easy to factor in if desired.

3.5.3 Evaluation of Runtime

The results of our experiment are shown in Figure 3-8. For each number of parts placed, we report the average runtime over the 10 sets. The scaling of the runtime appears approximately quadratic; packing twice as many parts takes roughly four times as long. This matches the theoretical complexity of the algorithm as shown in Subsection 3.3.7, which, assuming parts have a constant number of vertices, is quadratic in the number of parts.

Enabling cross-run caching results in a 1.8–2x observed speedup on our benchmark data. We note that this number is dependent on the complexity of the parts in the design. To observe this, we repeated this experiment but used 10 times the dilation and discretization tolerance, which resulted in the dataset having an average of just 6 vertices per polygon, 35% of which were non-convex. In this setting, caching produced a speedup of around 1.5x. This shows that caching is useful, and even more so as the designs become more complex.

We observe that our algorithm can easily handle 100 complex parts with caching in just a few seconds, which is far more parts than in any of our user study participants' designs, which had an average of 9. The largest had 74 parts, but spread over 11 materials, which are packed separately. This demonstrates that our system is capable of packing at interactive speeds for typical laser-cut designs using modest hardware.

Note that we do not consider benchmarking on sheets with holes because the performance of the algorithm on such instances is comparable if not faster than an instance with an equivalent number of polygons on a blank sheet, since candidate placements do not have to be generated for holes. For example, the time required to pack 50 parts onto a sheet with 50 holes will be faster than the time to pack 100 parts onto a blank sheet. Our experiment therefore provides an upper bound of the running time for such instances.

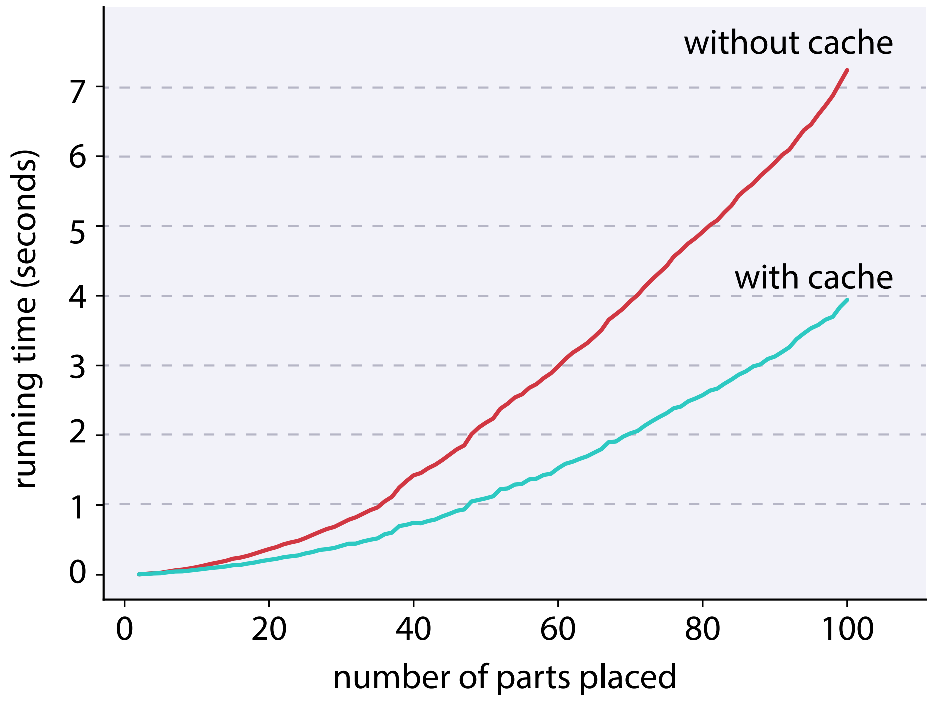


Figure 3-8: Average running times of the packing routine for various numbers of parts to place.

3.5.4 Evaluation of Solution Quality

To ensure that our algorithm has not sacrificed solution quality to achieve its speeds, we compared the runtime and quality to Deepnest [34], a state-of-the-art open-source nesting tool. Deepnest uses a metaheuristic search, i.e., it can run for an arbitrarily long time searching for better quality solutions. We therefore performed two comparisons. For each set of parts in our dataset, we measured the time taken for Deepnest to produce its first feasible solution, and we recorded the best solution obtained after running it for 5 minutes. Deepnest was configured with equivalent parameter settings (curve tolerance, dilation, etc.) to ours to ensure consistency. To compare solution quality, we measure the area of the bounding box of the placed parts, which is a fair comparison since this value is used by both our and Deepnest's objective functions. Deepnest was unable to find any solutions for one of our input sets (perhaps due to a bug), so we consider only the remaining 9.

On average, Deepnest found its first feasible solution after 31.3 seconds, which is 4.3x longer than our algorithm without any cross-run caching at 7.3 seconds. The average quality of the first solution was also 8% worse than that of our algorithm. The best solutions found by Deepnest after 5 minutes were, on average, just 3% better than those of our algorithm.

3.5.5 Additional Evaluation for Standard Datasets

We also compare our packing algorithm against TOPOS, a constructive algorithm for the nesting problem [30] using datasets commonly used within the literature. TOPOS's part placement strategy places one part at a time then uses a variety of heuristics to choose the next part to be placed and the placement of that part. Using different combinations of heuristics, there are 126 variations of the TOPOS algorithm.

To do this, we use five datasets used by several authors in the literature. SHAPES0 is a custom dataset of simple shapes created by the authors. SHAPES1 is the same as SHAPES0 but allowing rotations of 0° and 180°. SHAPES2 is a dataset created by Blacewitz [9] in a paper exploring using tabu search for the part packing problem. SHIRTS is a dataset of parts of shirts taken from the garment industry and was used in a study by Dowsland et al. [16]; rotations of 0° and 180° area allowed. TROUSERS contains simplified parts of trousers which is also taken from the garment industry and also allows rotations of 0° and 180°. Some metadata about the datasets used are included in Table 3.1.

The parts are placed on a blank sheet of fixed width and sufficiently large length to not constrain solutions. The metric used to determine the effectiveness of packing is the length of the bounding box of the packed parts. This metric is standard in several industries where the materials used are typically stored in long rolls such as metal and garment manufacturing.

Dataset	Number of Unique Parts	Total Number of Parts	Average Number of Vertices per Part	Rotations	Sheet Width
SHAPES0	4	43	8.88	0°	40 pts
SHAPES1	4	43	8.88	0° and 180°	40 pts
SHAPES2	7	28	6.29	0° and 180°	15 pts
SHIRTS	8	99	6.05	0° and 180°	40 pts
TROUSERS	17	64	6.06	0° and 180°	79 pts

Table 3.1: Metadata on nesting problem datasets

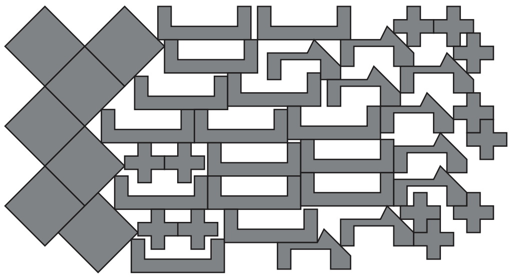
We test our packing algorithm on the same datasets with no dilation and curve tolerance and compare our results with the best results reported for TOPOS. We note that the results for each dataset for TOPOS reflects one of the 126 different variations of the TOPOS algorithm and the results reported are that of the best for that specific dataset. We run each test of our algorithm 10 times with cross-run caching disabled to calculate the average runtime across those runs. The packing algorithm is deterministic so the solution quality should not change. The results are shown in Table 3.2.

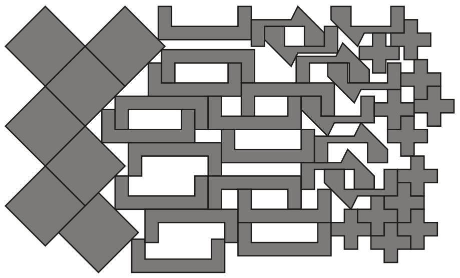
Dataset	Average Runtime	Length	TOPOS Best Length
SHAPES0	0.222s	75.5 pts	66.75 pts
SHAPES1	2.261s	67.5 pts	61.00 pts
SHAPES2	0.575s	29.83 pts	28.90 pts
SHIRTS	7.150s	67.26 pts	66.44 pts
TROUSERS	2.02s	283.6 pts	263.17 pts

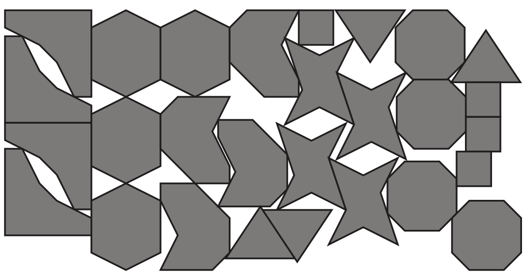
Table 3.2: Results comparing our algorithm to TOPOS's best results

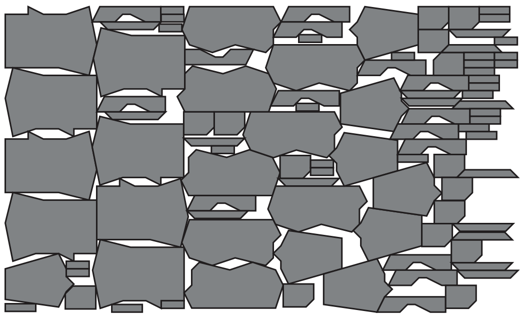
For solution quality, our algorithm performed reasonably well across all five datasets, at worst being 13.1% worse than the best run of TOPOS for SHAPES0 and at best performing only 1.2% worse than TOPOS for the SHIRTS dates. This shows that choice of heuristics for our packing algorithm is a reasonable catchall even for datasets not specific to the laser cutting setting.

We did not have access to TOPOS's implementation so consistent runtime comparisons are not possible due to difference in hardware. Even when comparing to the reported runtimes in the TOPOS paper, because the variation of TOPOS reported was different across the five datasets, the runtimes were anywhere between 10.31 to 155.86 times that of the corresponding runtime measured for our algorithm.


(a) SHAPES0


(b) SHAPES1


(c) SHAPES2


(d) SHIRTS

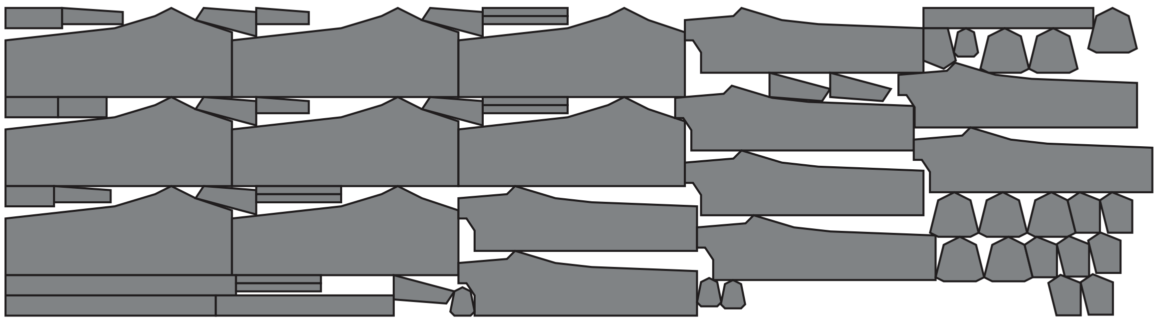
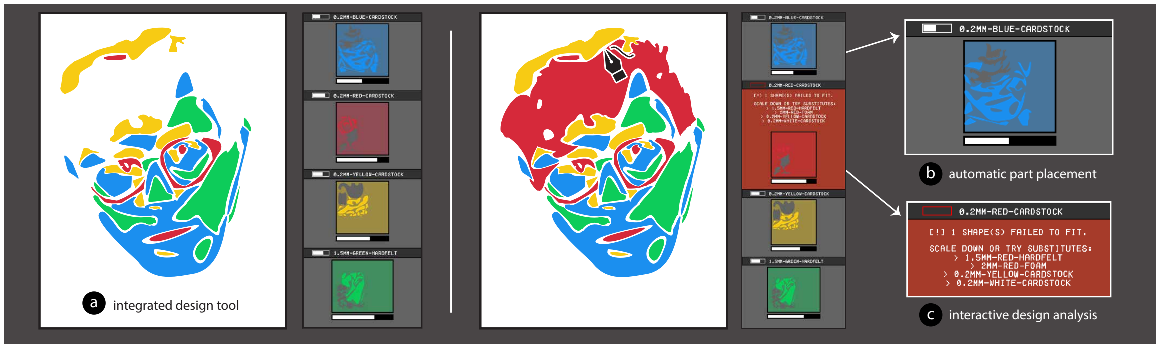

(e) TROUSERS

Figure 3-9: Packings of the test datasets generated by our algorithm

Fabricaide System Overview



```
[!] 1 SHAPE(S) FAILED TO FIT.

SCALE DOWN OR TRY SUBSTITUTES:
> 1.5MM-RED-HARDFELT
> 2MM-RED-FOAM
> 0.2MM-YELLOW-CARDSTOCK
> 0.2MM-WHITE-CARDSTOCK
```

Figure 4-1: As the user (a) develops their design, Fabricaide (b) decides good placements of parts onto material sheets given its knowledge of the user's available materials, and is able to (c) provide warnings if the user has insufficient material for a design while suggesting substitute materials. This design was made by P5 in our user study.

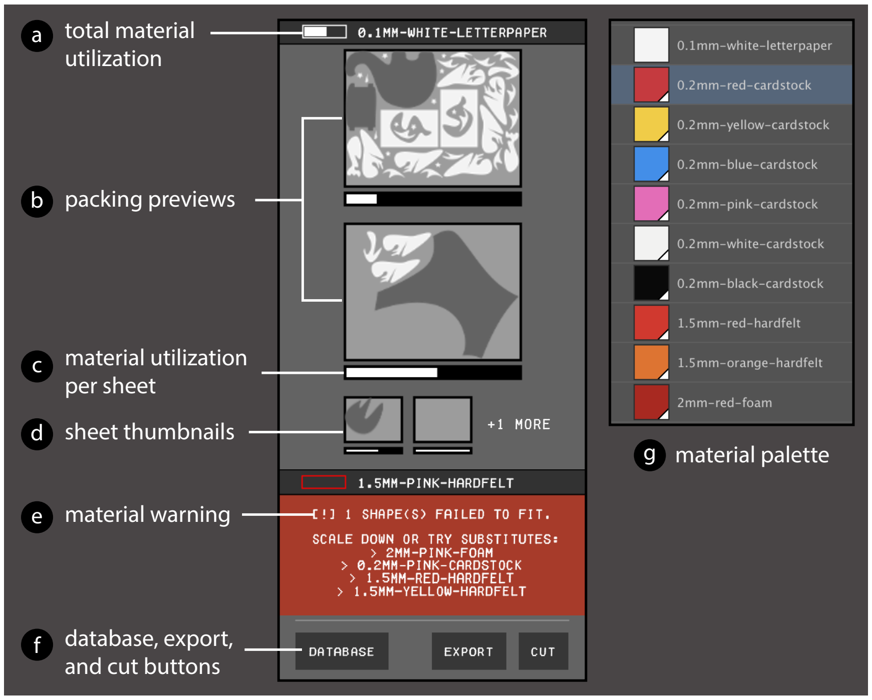
To set the context for the packing algorithm, we describe the core features of Fabricaide and discuss its implementation.

4.1 Core System Features

Fabricaide runs in conjunction with Adobe Illustrator, a 2D vector drawing program. The workflow enabled by Fabricaide is an iterative process that cycles between drawing shapes, assigning materials, and viewing packing previews. We elaborate on those features below.

4.1.1 Drawing Parts and Assigning Materials

Parts can be made using any drawing tool in Illustrator, provided that the part drawn is a closed path. Users can specify the material for a part through Fabricaide's *material palette*



```
[!] 1 SHAPE(S) FAILED TO FIT.

SCALE DOWN OR TRY SUBSTITUTES:
> 2MM-PINK-FOAM
> 0.2MM-PINK-CARDSTOCK
> 1.5MM-RED-HARDFELT
> 1.5MM-YELLOW-HARDFELT
```

Figure 4-2: An overview of the Fabricaide UI.

(Figure 4-2g), which is a collection of Illustrator swatches. Each swatch has a unique name and color that identifies a material with a specific type (e.g., wood or plastic), color, and thickness. To assign materials to parts of their design, users assign a swatch to the "fill" attribute before or after the part is drawn. If the fill color of a part does not correspond to any material in the database, Fabricaide will use a configurable default material.

4.1.2 Live Packing Previews

Fabricaide uses live *packing previews* to show users how the parts are placed (Figure 4-2b). When the user creates their first part with a particular material assignment, Fabricaide tries to find a good placement for it on a sheet corresponding to that material, optionally considering rotations of the part. If the packing algorithm places it successfully, the resulting placement is shown on its respective material sheet and displayed to the user (Figure 4-3a). The packing preview renders the design parts in full opacity, the sheet in partial opacity, and the holes of the sheet in dark gray. When additional new parts for a particular material are added to the design, or an existing part is modified, the packing algorithm is re-run for all parts corresponding to that material (Figure 4-3b). Fabricaide only re-generates packing

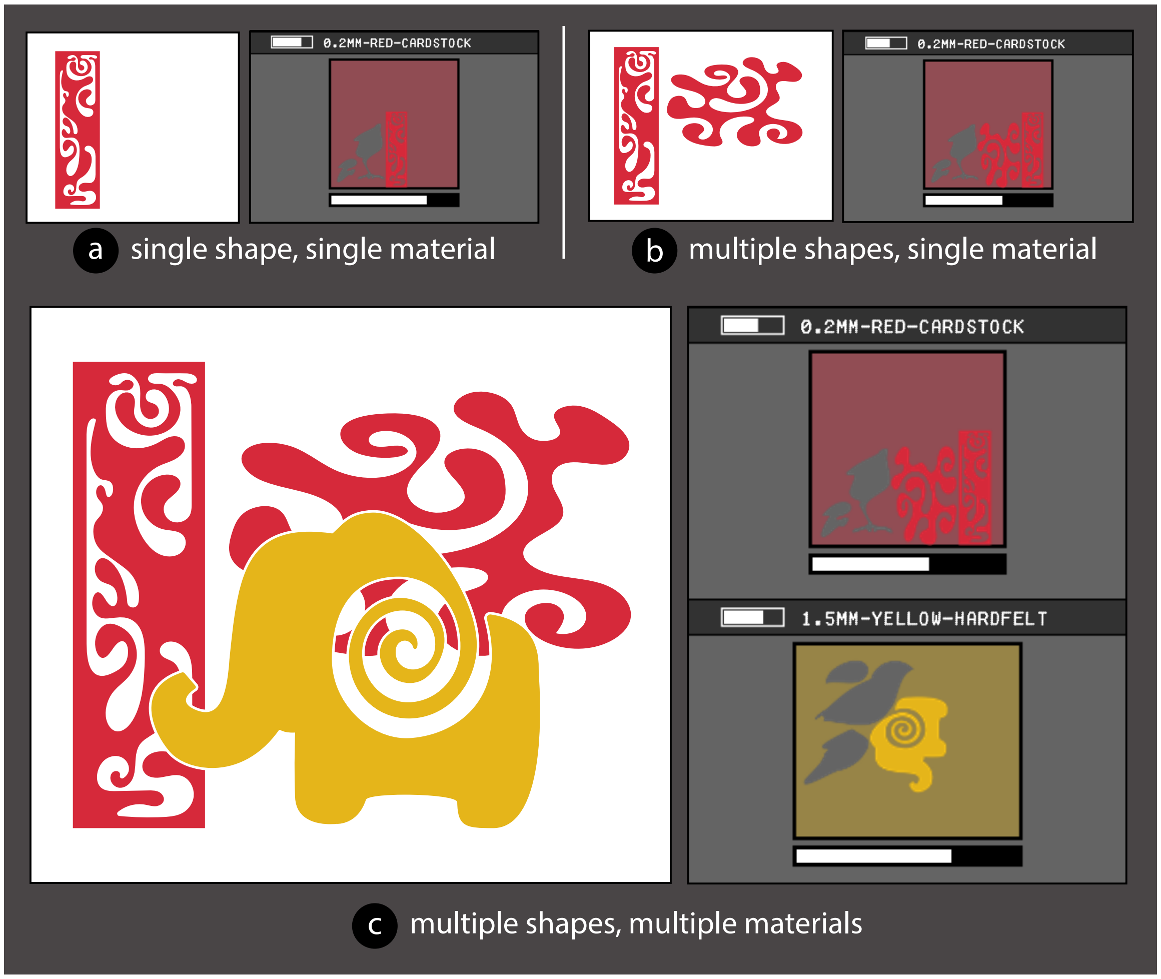

0.2MM-RED-CARDSTOCK
a single shape, single material
0.2MM-RED-CARDSTOCK
b multiple shapes, single material
0.2MM-RED-CARDSTOCK
1.5MM-YELLOW-HARDFELT
c multiple shapes, multiple materials

Figure 4-3: A demonstration of live packing in Fabricaide using designs from P2 in our user study.

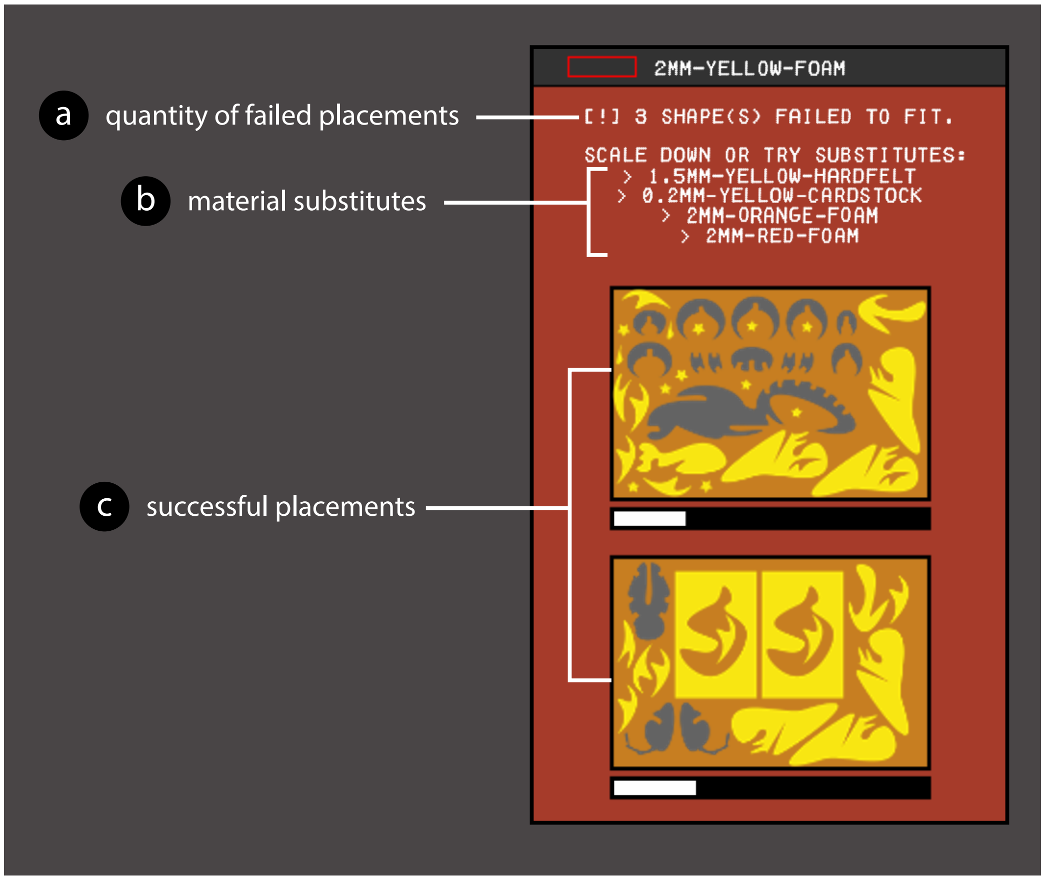
configurations for materials that have been affected; in other words, changing the shapes for one material does not affect the packing configurations for other materials (Figure 4-3c).

If there are more sheets for that material that can be used, Fabricaide additionally shows the user up to two *sheet thumbnails* (Figure 4-2d) as well as the number of sheets remaining.

While live packing is typically always running as long as Fabricaide is open, Fabricaide also offers the option to disable live packing and instead have the user refresh the packing on demand.

4.1.3 Material Warnings and Suggested Substitutes

If the packing algorithm is unable to place some of the parts in any of the sheets, the panel will alert the user with a *material warning* (Figure 4-4). The user is shown a preview of the parts that were successfully placed in their design (Figure 4-4c) as well as the number of parts that could not be placed (Figure 4-4a). If the amount of material is insufficient, the warning suggests material substitutes (Figure 4-4b) with up to two same-color substitutes and up to two same-thickness substitutes.



[!] 3 SHAPE(S) FAILED TO FIT.
SCALE DOWN OR TRY SUBSTITUTES:
> 1.5MM-YELLOW-HARDFELT
> 0.2MM-YELLOW-CARDSTOCK
> 2MM-ORANGE-FOAM
> 2MM-RED-FOAM

Figure 4-4: A material warning for 2mm yellow foam. It shows that a) three shapes that could not be placed, b) a list containing two color substitutes and two thickness substitutes, and c) the parts in the design that were successfully placed.

4.1.4 Material Utilization

In addition to showing how the space on a material sheet is being used through packing previews, Fabricaide also offers the notion of *material utilization* to show an approximation of the amount of remaining usable space across different materials and their sheets. Simply computing the total area remaining on the sheet would not suffice because the arrangement of the pieces affects how usable the remaining space actually is. Material utilization therefore also accounts for the tightness of the packing, since a sheet with parts tightly packed together leaves more available space for future parts than one with parts that are spread out. The calculation of material utilization is described in more detail in the Implementation section.

Whenever a sheet is re-packed by the packing algorithm, Fabricaide also recomputes the material utilization of the resulting packing. Fabricaide conveys material utilization to the user in the form of material utilization bars that approximate the percentage of remaining usable material per material type (Figure 4-2a), as well as per individual sheet (Figure 4-2c). A more filled bar indicates that the sheet has a greater amount of usable space.

4.2 Implementation

The Fabricaide interface is implemented in Processing, and is designed to be run in parallel with Adobe Illustrator. Illustrator scripts are used to periodically export the current document as an SVG file so that it can be continuously analyzed, and to switch the currently opened document when transitioning between designing and editing packing configurations.

A local web server implemented in Python handles communication with the Fabricaide interface and the background data processing of the exported document and is responsible for preprocessing the design file, sending it to the packing algorithm, and then generating the resulting images that are displayed in the UI.

4.2.1 Splitting a Design into Constituent Materials

Fabricaide first splits a design into its individual materials by analyzing the fill attribute of the shapes in the exported SVG file and checking which material it maps to in the database.

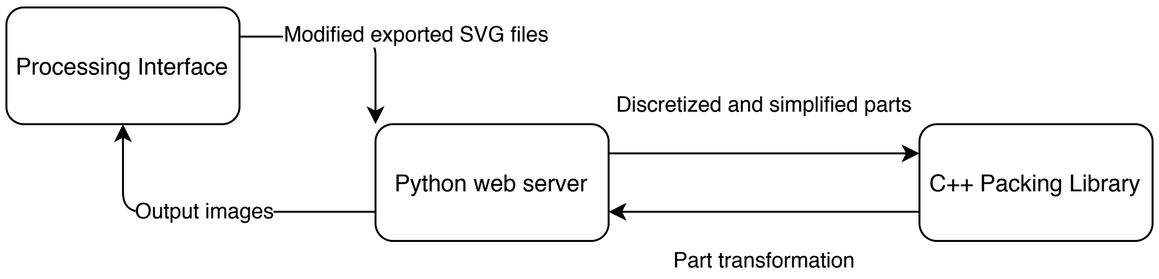


Figure 4-5: System diagram showing the interactions between the different parts of Fabricaide.

If a part has a fill that is not mapped to a specific material, it is assigned to a default material. The result is a set of SVG files, each containing the parts assigned to a particular material.

4.2.2 Calculating Material Utilization

Since many CNC manufacturing workflows involve cutting parts from long rolls of fixed-height material, a common metric of material utilization that has been used is the width of the section of sheet that contains the placed parts, i.e. the length of sheet that must be rolled off the roll. This favors tight packings over ones with parts spread far out. This, however, does not give sensible values if we consider fixed-size material sheets with arbitrary pre-existing holes, since the holes themselves may be spread out, and it is usually preferable to place new parts close to the existing holes.

To address the limitation above, the material utilization metric is computed as follows: we consider every hole and newly placed part on the sheet, dilate them by some fixed amount, and compute the area of their union. The dilation is set to a default of 20 points or approximately 7 mm to accommodate for the recommended minimum amount of spacing between shapes to be laser cut (2mm) and the recommended minimum width of a shape that can be cut (typically between 2mm to 5mm depending on material thickness). A smaller area indicates a better packing. This metric has the effect of penalizing parts that are placed far away from others (Figure 4-6a), and rewarding tightly packed configurations (Figure 4-6b).

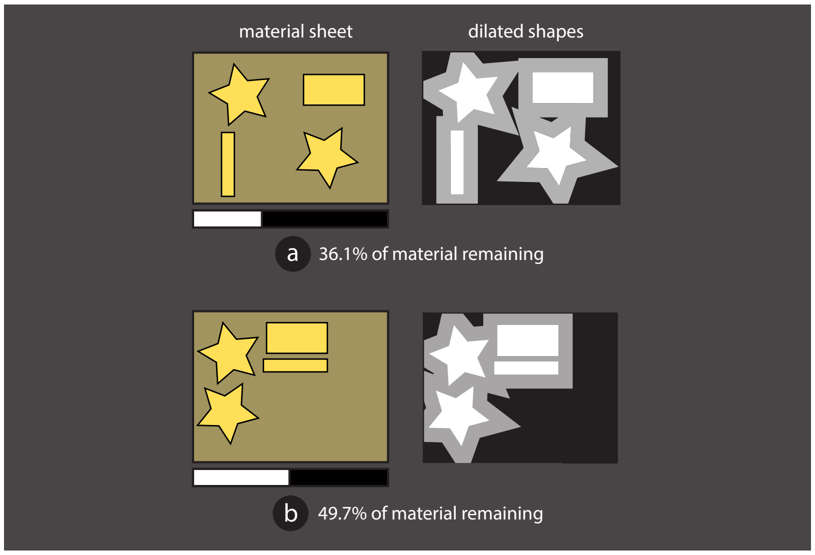


Figure 4-6: Calculating material utilization. (a) shows the effect of a poor packing configuration on the utilization bar for a material sheet while (b) shows the effect of a good packing configuration.

4.3 Qualitative Evaluation of Fabricaide

To test how well Fabricaide and the packing algorithm worked in practice, we ran a qualitative user study with designers, in which they used Fabricaide over the period of one week.

4.3.1 User Study Procedure

We asked the six participants of our user study (referred to as P1 through P6) to install the Fabricaide software onto their computers and provided them with a reference guide for the tool. To simplify the installation by removing the need for users to install library dependencies, the back-end code was containerized using Docker.

To get a sense of when participants might prioritize continuous packing, we additionally gave them the option to disable live packing and only refresh on demand.

4.3.2 User Study Findings

During the user study, we observed two main design approaches in which users would use the packing previews differently depending on how the design was created.

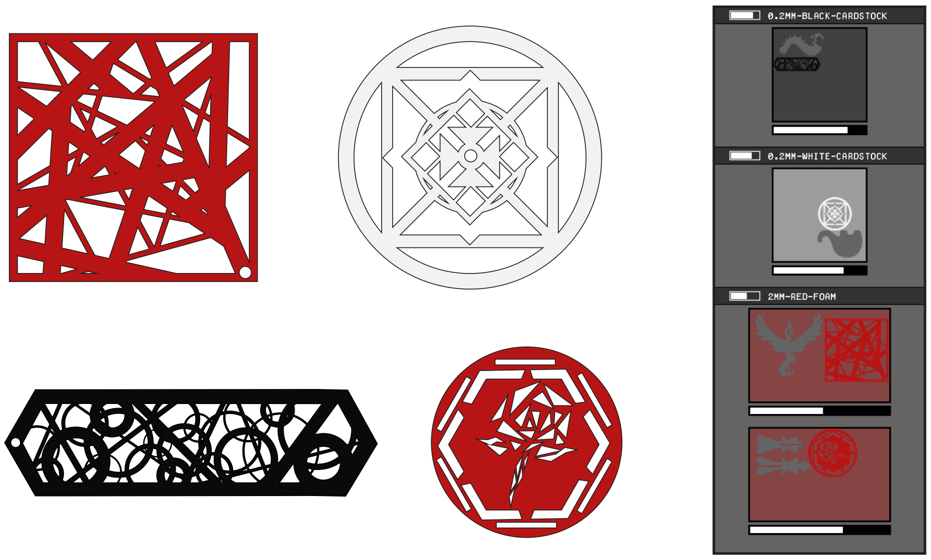


Figure 4-7: Four abstract designs by P6, created using a strictly subtractive approach. Her process involved starting with basic geometric shapes and refining them by removing sections of the geometry.

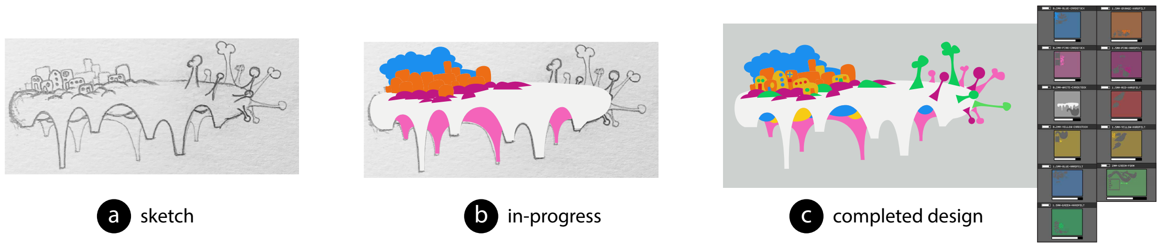


Figure 4-8: A complex design by P1 that has 74 parts and uses 11 different materials.

For the first kind of design approach, users tended to first draw larger initial shapes which served as the base which they then refined by removing portions of the geometry (Figure 4-7). In this type of design process, users tended to only pay attention to the packing previews at the beginning of the design process.

On the other hand, the second approach was more additive in nature where the user would progressively add shapes to their design. Users who designed in this way tended to check the packing previews more frequently. For example. P1's process involved creating an initial sketch on paper and drawing over it in Illustrator (Figure 4-8). She checked Fabricaide whenever she would finish drawing a section of her design to confirm that it had no issues and that there was enough material.

In both cases, users were able to effectively make use of the packing previews and get feedback within a reasonable amount of time during the times they paid attention to the interface. We do note that we were unable to tell from user feedback how fast the packing algorithm worked for users using subtractive approaches as they did detail work and either paused or stopped paying attention to the packing previews. In these cases, the added complexity of the parts created may lead to performance hits in the algorithm that were not noticed by the user.

Furthermore, users with lower performance machines (e.g., older machines with limited RAM) caused Fabricaide to lag as their designs became more complex (P3, P5). This caused to frequently leave the automated packing paused and only refresh later on. We only discovered after the fact that this was due to the high memory footprint of Docker on machines with less RAM as opposed to the performance of the algorithm itself. In any case, it is important to consider how to make interactive systems accessible to users with low performance hardware. A potential solution would be to offload the computationally intensive tasks to the cloud.

Conclusions

We presented a custom 2D packing algorithm that optimizes the placement of parts onto material sheets with preexisting holes at interactive speeds for the laser cutting use case. In a technical evaluation, we were able to show that our algorithm is capable of handling typical laser-cut designs at interactive speeds on modest hardware. We were also able to show that the quality of the packing solutions was not overly sacrificed by comparing it against similar tools and algorithms.

Future Work

In this section, we discuss some of the limitations of the current packing algorithm and some ideas on how these may be addressed in the future.

Packing Heuristics

As the definition of what can be considered *usable* material varies depending on how a user typically likes to place their parts as well as what kinds of parts a user tends to cut, it is hard to create a catch-all heuristic that handles all possible cases. For example, in Figure5-1, while the heuristic is the sum of the area of the overall bounding box and the area of the bounding box of the placed parts, it can be argued that users would rather have parts placed below the horizontal hole so that the continuous available area in the upper left is larger.

An approach that can be considered to address this limitation would be to use a weighted heuristic function where the weights of each component of the function can be tuned by the

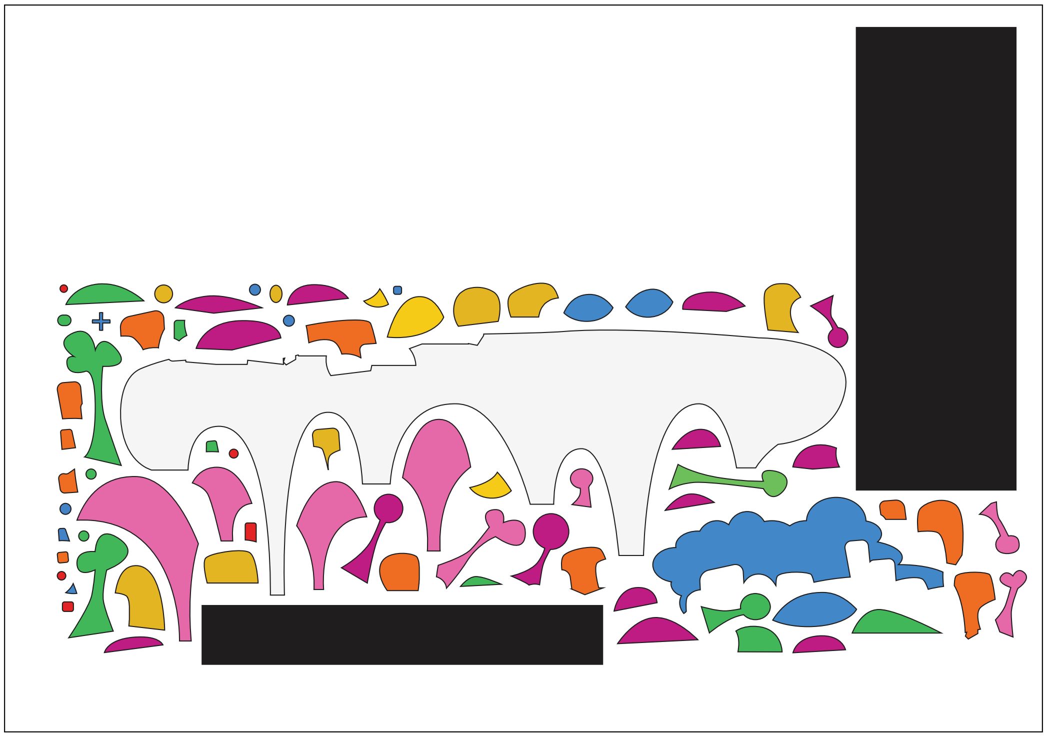


Figure 5-1: Example of an arguably sub-optimal packing when the holes (black) are not flush against the edge of the material.

user. The area of the overall bounding box and the area of just the placed parts are two such possible components, though modifications such as examining the perimeter as opposed to the area is also a possibility.

The performance of other more complex heuristics or heuristic components can be examined as well though each would come with tradeoffs.

NFP Caching

For NFP caching, our current implementation can at worst store the NFPs for every possible pair of different parts, where rotations of a part are considered to be distinct from one another. This results in having to compute $(nr)^2$ where n is the number of unique parts and r is the number of rotations to consider.

We note however that some of these NFPs can be generated without explicitly calculating for the NFP. For example, given two parts that have been rotated by the same amount, their NFP can be generated by computing the NFP generated by the two unrotated parts, then rotating it by that same amount. This allows the algorithm to save on redundant computation on memory,

Taking advantage of the rotational properties of NFPs, the upper bound for NFPs that need to be computed can be reduced to $(\frac{n(n+1)}{2})(\lceil\frac{m+1}{2}\rceil)$ [39].

Parallel Processing

The current packing algorithm runs serially to avoid lagging the design tools it is used with, but higher performance machines may be able to take advantage of parallel processing to speed up runtime.

While the placement of parts still has to be done serially due to the constraints for a part to be placed depending on all previously placed parts, the computation of NFPs and evaluation of candidate placements should be easily parallelizable.

Bibliography

[1] David Abrahams, Ralf W Grosse-Kunstleve, and Operator Overloading. 2003. Building hybrid systems with Boost. Python. *CC Plus Plus Users Journal* 21, 7 (2003), 29–36.

[2] Michael Adamowicz and Antonio Albano. 1976. Nesting two-dimensional shapes in rectangular modules. *Computer-Aided Design* 8, 1 (1976), 27–33.

[3] Antonio Albano and Giuseppe Sapuppo. 1980. Optimal allocation of two-dimensional irregular shapes using heuristic search methods. *IEEE Transactions on Systems, Man, and Cybernetics* 10, 5 (1980), 242–248.

[4] Richard Carl Art Jr. 1966. *An approach to the two dimensional irregular cutting stock problem.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[5] A Ramesh Babu and N Ramesh Babu. 2001. A generic approach for nesting of 2-D parts in 2-D sheets using genetic and heuristic algorithms. *Computer-Aided Design* 33, 12 (2001), 879–891.

[6] Roberto Baldacci, Marco A Boschetti, Maurizio Ganovelli, and Vittorio Maniezzo. 2014. Algorithms for nesting with defects. *Discrete Applied Mathematics* 163 (2014), 17–33.

[7] Julia A Bennell, Kathryn A Dowsland, and William B Dowsland. 2001. The irregular cutting-stock problem—a new procedure for deriving the no-fit polygon. *Computers & Operations Research* 28, 3 (2001), 271–287.

[8] Julia A Bennell and Xiang Song. 2008. A comprehensive and robust procedure for obtaining the nofit polygon using Minkowski sums. *Computers & Operations Research* 35, 1 (2008), 267–281.

[9] J Błażewicz, P Hawryluk, and Rafal Walkowiak. 1993. Using a tabu search approach for solving the two-dimensional irregular cutting problem. *Annals of Operations Research* 41, 4 (1993), 313–325.

[10] Edmund Burke, Robert Hellier, Graham Kendall, and Glenn Whitwell. 2006. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research* 54, 3 (2006), 587–601.

[11] Edmund Burke and Graham Kendall. 2002. A New Approach to Packing Non-Convex Polygons Using the No Fit Polygon and Meta-Heuristic and Evolutionary Algorithms. In *Adaptive Computing in Design and Manufacture V*. Springer, 193–204.

[12] Edmund Burke, Graham Kendall, and Nottingham Uk Nottingham. 1999. Applying simulated annealing and the no fit polygon to the nesting problem. In *Proceedings of the world manufacturing congress*. Citeseer.

[13] Edmund K Burke, Robert SR Hellier, Graham Kendall, and Glenn Whitwell. 2007. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research* 179, 1 (2007), 27–49.

[14] Edmund K Burke, Graham Kendall, and Glenn Whitwell. 2009. A simulated annealing enhancement of the best-fit heuristic for the orthogonal stock-cutting problem. *INFORMS Journal on Computing* 21, 3 (2009), 505–516.

[15] Karen Mcintosh Daniels and Victor J Milenkovic. 1995. *Containment algorithms for nonconvex polygons with applications to layout*. Ph.D. Dissertation. Citeseer.

[16] KA Dowsland, WB Dowsland, and JA Bennell. 1998. Jostling for position: local improvement for irregular cutting patterns. *Journal of the Operational Research Society* 49, 6 (1998), 647–658.

[17] Matteo Fischetti and Ivan Luzzi. 2009. Mixed-integer programming models for nesting problems. *Journal of Heuristics* 15, 3 (2009), 201–226.

[18] Efi Fogel, Dan Halperin, and Christophe Weibel. 2007. On the exact maximum complexity of Minkowski sums of convex polyhedra. In *Proceedings of the twenty-third annual symposium on Computational geometry*. 319–326.

[19] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.

[20] Sean Gillies and others. 2007–. Shapely: manipulation and analysis of geometric objects. (2007–). `https://github.com/Toblerity/Shapely`

[21] A Miguel Gomes and José F Oliveira. 2006. Solving irregular strip packing problems by hybridising simulated annealing and linear programming. *European Journal of Operational Research* 171, 3 (2006), 811–829.

[22] Joerg Heistermann and Thomas Lengauer. 1995. The nesting problem in the leather manufacturing industry. *Annals of Operations Research* 57, 1 (1995), 147–173.

[23] David S Johnson. 1973. *Near-optimal bin packing algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[24] Leonid V Kantorovich. 1960. Mathematical methods of organizing and planning production. *Management science* 6, 4 (1960), 366–422.

[25] Takashi Kikuchi, Yuichi Hiroi, Ross T. Smith, Bruce H. Thomas, and Maki Sugimoto. 2016. MARCut: Marker-based Laser Cutting for Personal Fabrication on Existing Objects. In *Proceedings of the TEI '16: Tenth International Conference on Tangible, Embedded, and Embodied Interaction (TEI '16)*. ACM, New York, NY, USA, 468–474. DOI:http://dx.doi.org/10.1145/2839462.2856549

[26] Bongjin Koo, Jean Hergel, Sylvain Lefebvre, and Niloy J Mitra. 2016. Towards zero-waste furniture design. *IEEE Transactions on Visualization and Computer Graphics* 23, 12 (2016), 2627–2640.

[27] Zhenyu Li and Victor Milenkovic. 1995. Compaction and separation algorithms for non-convex polygons and their applications. *European Journal of Operational Research* 84, 3 (1995), 539–561.

[28] Anantharam Mahadevan. 1984. Optimization in computer-aided pattern packing (marking, envelopes). (1984).

[29] Victor Milenkovic, Karen Daniels, and Zhenyu Li. 1991. Automatic marker making. In *Proceedings of the Third Canadian Conference on Computational Geometry*. Simon Fraser University, 243–246.

[30] José F Oliveira, A Miguel Gomes, and J Soeiro Ferreira. 2000. TOPOS–A new constructive algorithm for nesting problems. *OR-Spektrum* 22, 2 (2000), 263–284.

[31] José Fernando C Oliveira and José A Soeiro Ferreira. 1993. Algorithms for nesting problems. In *Applied simulated annealing*. Springer, 255–273.

[32] Thomas Oster and Rene Bohne. 2012. Visicut: A userfriendly tool to prepare, save and send Jobs to Lasercutters. (2012).

[33] Jack Qiao. 2019. SVGNest: Open Source nesting. https://svgnest.com/. (2019).

[34] Jack Qiao. 2020. Deepnest - open source nesting software. https://deepnest.io/. (2020).

[35] Daniel Saakes, Thomas Cambazard, Jun Mitani, and Takeo Igarashi. 2013. PacCAM: Material Capture and Interactive 2D Packing for Efficient Material Usage on CNC Cutting Machines. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 441–446. DOI:http://dx.doi.org/10.1145/2501988.2501990

[36] The CGAL Project. 2020. *CGAL User and Reference Manual* (5.0.2 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0.2/Manual/packages.html

[37] PD Watson and AM Tobias. 1999. An efficient algorithm for the regular W_1 packing of polygons in the infinite plane. *Journal of the Operational Research Society* 50, 10 (1999), 1054–1062.

[38] Ron Wein, Alon Baram, Eyal Flato, Efi Fogel, Michael Hemmer, and Sebastian Morr. 2019. 2D Minkowski Sums. In *CGAL User and Reference Manual* (5.0 ed.). CGAL Editorial Board. https://doc.cgal.org/5.0/Manual/packages.html#PkgMinkowskiSum2

[39] Glenn Whitwell. 2004. *Novel heuristic and metaheuristic approaches to cutting and packing*. Ph.D. Dissertation. University of Nottingham.

[40] K. Winge, R. Haugaard, and T. Merritt. 2014. VAL: Visually Augmented Laser cutting to enhance and support creativity. In *2014 IEEE International Symposium on Mixed and Augmented Reality - Media, Art, Social Science, Humanities and Design (ISMAR-MASH'D)*. 31–34. DOI:http://dx.doi.org/10.1109/ISMAR-AMH.2014.6935435