Master's Thesis Academic Year 2015

Filtering Techniques and Search Heuristics Based on Placement Domain Analysis for the Two-Dimensional Irregular Packing Problem

Long Nguyen Huu

(Student ID No.: 81323900)

Advisor Dr. Masafumi Hagiwara

September 2015

Keio University

Graduate School of Science and Technology

School of Science for Open and Environmental Systems

August 2015

KEIO UNIVERSITY

Abstract

Graduate School of Science and Technology
School of Science for Open and Environmental Systems

Master of Science in Engineering

Filtering Techniques and Search Heuristics Based on Placement Domain Analysis for the Two-Dimensional Irregular Packing Problem

by Long Nguyen Huu

The two-dimensional irregular packing problem is a geometrical problem encountered in the clothing industry and in many common life situations. In this problem, a heterogeneous set of polygonal pieces must be placed into a rectangular container so that no polygon overlaps with any others. We consider two variants: the single knapsack problem, in which the container has a fixed size, and the open dimension problem, where the container has a variable length that must be minimized.

In this thesis, we propose a sequential approach to solve the irregular packing problem. First, we adapt a few search heuristics widely used in discrete constraint satisfaction problems. Second, we apply advanced filtering techniques to quickly eliminate inconsistent layouts. Both heuristics and filtering techniques are based on the analysis of the placement domain of each piece. With this approach, we can find packing layouts that are acceptable for casual applications relatively fast, but it requires a considerable amount of additional time and memory to obtain layouts of better quality, especially when pieces with complex shapes are involved.

Acknowledgements

First and foremost, I would like to thank Joanna Tomasik, my professor in Algorithms and Data Structure and at the same time my home country advisor, and Yolaine Bourda, professor in Software Engineering, for supporting me in my studies abroad. I want to express my gratitude to my research advisor, Dr. Masafumi Hagiwara, for providing me with all the resources I needed on Artificial Intelligence, and for showing me how to highlight the key points of my research. I am also grateful to professors Kyoko Inoue and Daniel Citterio for giving me crucial advice for my presentations.

I would also like to thank my laboratory colleagues Lucille Farrenq and Yang Ming for teaching me a few computer hacks, as well as An Nguyen and Wen Pin for sharing with me their thoughts on the world of research.

I want to congratulate the communities of *Stack Overflow* and *TeX* - *LaTeX* on *Stack Exchange* for their continuous support to the world of software development and documentation. Without their help, I would not have been able to overcome numerous implementation issues during my research.

Finally, I would like to express my gratitude to the organizers and exhibitors of the Tokyo Puzzle Day for conveying me their love of puzzles.

Contents

List of Figures

List of Tables

Abbreviations

CFR	**C**ollision-**F**ree **R**egion
COP	**C**onstraint **O**ptimization **P**roblem
CSP	**C**onstraint **S**atisfaction **P**roblem
FC	**F**orward **C**hecking
IFP	**I**nner-**F**it **P**olygon
IFR	**I**nner-**F**it **R**ectangle
LCV	**L**east **C**onstraining **V**alue
MHKP	**M**ultiple **H**eterogeneous **K**napsack **P**roblem
MRV	**M**inimum **R**emaining **V**alues
NFP	**N**o-**F**it **P**olygon
ODP	**O**pen **D**imension **P**roblem
SKP	**S**ingle **K**napsack **P**roblem

Nomenclature

W	container width
L	container length
$C(W, L)$	container
$P_1, \ldots, P_n$	problem pieces
P	piece
P_{ref}	reference point of piece P
$\mathcal{A}(P)$	area of piece P
$O(P)$	set of orientations allowed for piece P
$D(P)$	placement domain of piece P
$D_\theta(P)$	translation domain of piece P for orientation θ
$\mathbf{u}$	piece translation
θ	piece rotation
$p = (\mathbf{u}, \theta)$	piece placement
F	current layout
$int(X)$	interior of set X
∂X	boundary of set X in the plan
X^{e}	exterior of set X
X^{c}	complementary of set X
$NFP(A, B, \theta)$	no-fit polygon of A and B when B has orientation θ
$IFP(C, B, \theta)$	inner-fit polygon of B inside C when B has orientation θ
$CFR(C, F, P)$	collision-free region of P inside C with layout F

Chapter 1

Introduction

1.1 Problem overview

The irregular packing problem, sometimes called nesting or cutting problem, is a geometrical problem in which pieces of different shapes must be placed inside a container so that no pieces overlap any others. In this thesis, we consider the two-dimensional irregular packing problem with polygonal pieces and a rectangular container. Pieces may be non convex and contain holes. In addition, the pieces can be rotated, but only a finite set of orientations is allowed for each piece. Piece mirroring is not considered in this thesis, but can be handled in the same way as piece orientations. We deal with two variants of the problem, as classified by Wäscher et al. [1]: the Single Knapsack Problem and the Open Dimension Problem.

The original Single Knapsack Problem (SKP) is an optimization problem in which as many pieces as possible must be placed inside a single container of fixed size. In this thesis, we use a slightly different definition of the SKP where all the pieces must be placed so that the problem is considered solved. Our motivations are that we work with puzzle problems that require all the pieces to be placed, and that it makes the definition of the SKP closer to the definition of the Open Dimensional Problem. Nevertheless, for problem instances that are too hard to solve, it is useful to count the number of pieces placed in the container as a measurement of the progression of the search.

In the Open Dimension Problem (ODP), the container is a rectangle with a fixed width and a variable length. The objective is to pack the pieces in the container while minimizing its length. Most of the benchmarks available to test packing algorithms are ODPs.

1.2 Applications

The Single Knapsack Problem is encountered in everyday life, for instance when packing luggage in a travel case or a car trunk. However, in most cases, a three-dimensional representation and physics considerations are needed in order to find convincing solutions. Packing puzzles are more accurately represented by SKPs, and many instances are in two dimensions. Specialized puzzle solvers use combinatorial approaches that are more efficient than generic packing algorithms. Nevertheless, packing and dissection puzzles are good benchmarks for research on SKP because we know whether a solution exists or not for a given container length. In addition, the solutions to dissection puzzles are compact, and filtering techniques are effective on problems with strong constraints. Figure 1.1 shows a possible solution to the well-known Tangram dissection puzzle with a square container.

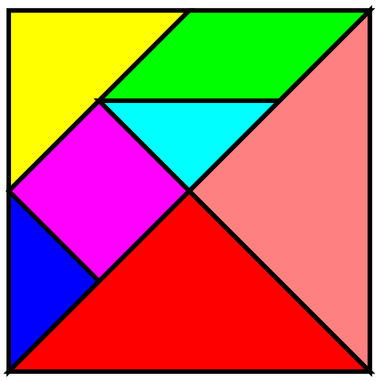


FIGURE 1.1: Tangram puzzle as a Single Knapsack Problem

The Open Dimension Problem models cutting problems present in the metal and cloth industry, where specific patterns must be cut from a material sheet. The unused parts of the sheet are often too small to be reused and become waste. Therefore, minimizing the length of the sheet used for cutting minimizes the cost of the production. The Shirts and Trousers data sets from the ESICUP benchmarks [2] contain such cloth patterns.

1.3 Problem definition

In this section, we give a formal definition to the Single Knapsack and the Open Dimension Problems. We can formulate the SKP as a Constraint Satisfaction Problem (CSP) and the ODP as a Constraint Optimization Problem (COP). We start by defining some problem data common to both variants.

1.3.1 Problem data

Pieces

The problem pieces are noted $(P_1, \dots, P_n)$. The geometry of a piece is a *polygon with holes*. It is a closed set characterized by a polygonal exterior boundary and zero, one or more polygonal holes. For each piece P, a reference point P_{ref} is defined as the point that matches the origin of the plan O when P is not translated. When P is translated by $\mathbf{u}$, $\overrightarrow{OP_{ref}} = \mathbf{u}$. P_{ref} may or may not belong to the piece geometry. See Figure 1.2 for an example of pieces with their reference points. By abuse of notation, we use P to indicate at the same time the problem data, the geometry and, when the context is clear, the translated and rotated geometry of piece P.

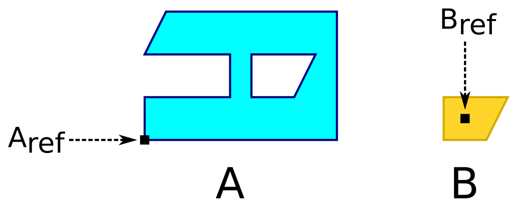


FIGURE 1.2: Two pieces A and B and their reference points

Although irregular packing deals with a heterogeneous set of pieces, it is common that several pieces share the same geometry. In this thesis, we consider each piece as a different entity. In practice, geometrical computations should be shared between pieces that have the same shape.

Pieces are subject to a *placement*, a combination of a translation and a rotation, noted $p = (\mathbf{u}, \theta)$. For each piece P, a finite set of orientations $O(P)$ is allowed. Pieces are always rotated around their reference points.

Container

The container is a rectangle of width W, length L, and is noted $C(W, L)$. In our model, the width is the size along the Y axis and the length is the size along the X axis. The edges of the container are aligned with the X and Y axes and its bottom-left corner is always at the origin.

1.3.2 Constraint problem formulation

Constraint satisfaction problems and constraint optimization problems have the following common characteristics:

- A finite set of *variables* is defined.
- A *value* can be assigned to each variable, and this value must live in a *domain* specific to the variable.
- An *assignment* is a tuple of variable $\leftarrow$ value associations. It is *complete* if all variables are associated to a value, otherwise it is *partial.* The term *assignment* is sometimes used to indicate a single variable $\leftarrow$ value association.
- *Constraints* impose a condition on an assignment. Unary constraints give a condition on the value assigned to one variable, while binary constraints give a condition on the relationship between the values assigned to two variables.
- A *model* is a complete assignment that satisfies all the problem constraints.

The objective of a CSP is to find any model, while the objective of a COP is to find a model that minimizes some *cost* function of the assignment. Now that we have introduced the characteristics of a CSP and a COP, we can define them for the irregular packing problem.

Single Knapsack Problem

The problem is defined as a CSP with the following characteristics:

- Variables: pieces $\langle P_1, \ldots, P_n \rangle$

  By abuse of notation we note $P_1, \ldots, P_k$ an arbitrary sequence of pieces, which is not necessarily in the same order as in the definition of the variables.

- Values: placements $\langle p_1, \ldots, p_n \rangle$ where $p_i = (\mathbf{u_i}, \theta_i)$ is the placement of piece P_i

  The placement domain of a piece P is noted $D(P)$. $D(P) \subseteq \mathbb{R}^2 \times O(P)$ and it represents, for each orientation allowed for P, the set of possible translations of P, which is also the set of possible positions of P_{ref}.

  To simplify notations, we define the placement function pl as follows:

  $$\forall p = (\mathbf{u}, \theta) \in \mathbb{R}^2 \times O(P), pl_p(P) = t_{\mathbf{u}}(r_\theta(P))$$

  where $t_{\mathbf{u}}$ is the translation of vector $\mathbf{u}$ and r_θ the rotation of angle θ

- Unary constraints: all the pieces must be inside the container of fixed dimensions, i.e.

  $$\forall i \in [1..n], pl_{p_i}(P_i) \subseteq C(W, L)$$

- Binary constraints: two different pieces must not overlap, i.e.

  $$\forall i, j \in [1..n], i \neq j, int(pl_{p_i}(P_i)) \cap int(pl_{p_j}(P_j)) = \emptyset$$

  where $int(X)$ is the interior of the set X.

Open Dimension Problem

The ODP is a COP with the same variables, values, domains and constraints as above. However, the length L is variable, so the unary constraint depends on the value of L considered at a given state in the search. In addition, a last characteristic is added:

- Cost: L must be minimized.

Additional notations

We use the term *layout* to designate a partial or a complete assignment of placements for the pieces. By abuse of language, we also designate the union of the pieces placed in the container based on a given assignment as a *layout*.

Since the placement domain of a piece P contains a set of possible translations for each orientation of P, it cannot be directly used in geometrical operations. For this reason, we define the translation domain $D_\theta(P)$ as the set of possible translations of P for a fixed orientation θ:

$$D_\theta(P) = \{\mathbf{u} \in \mathbb{R}^2, (\mathbf{u}, \theta) \in D(P)\}$$

For the Open Dimension Problem, the domains and translation domains are always considered for a given container length L.

We call pieces that are not placed in the current layout *remaining pieces*. In order to complete a layout, one must place all the remaining pieces.

1.4 Literature review

In this section we survey a few approaches to the 2D irregular packing problem. Search methods can be classified in two main groups: sequential searches and local searches. In a sequential search, the initial state of the search is an empty layout. Pieces are then placed one by one until the layout is complete. At each step, pieces must not overlap, and most algorithms are allowed to backtrack. Bennell and Song [3] use a beam search where the global evaluation of a placement is based on a single-pass layout completion. The method is cheap in computation time and memory usage, and can be applied to both the Single Knapsack and the Open Dimension Problems.

In a local search approach, the initial state is a complete layout where some pieces may overlap with each others. The length of the container is alternatively increased to make it easier to find a solution, and decreased to force the algorithm to find better solutions. At each step, pieces are moved and swapped until they do not overlap anymore, or until the amount of overlap is minimized. Sato et al. [4] use a two-level algorithm for the ODP, applying simulated annealing to the length of the container while replacing the pieces to exactly fitting positions. Imamichi et al. [5] analyze the way two pieces overlap to find the shortest translation required to separate them. Elkeran [6] creates, in a preprocessing step, clusters of two pieces if their shapes match well, then combines a Cuckoo search and a guided local search to gradually improve the solutions.

Physics-based methods can be used in either sequential or local searches. Wauters et al. [7] present a shaking algorithm, a physics-based dynamic local search, for packing problems with rectangular pieces.

Hybrid methods also exist: a sequential search is used to find a layout with little or no overlap, on top of which a local search is applied in order to ensure that the overlap remains low or null while decreasing the container length.

Chapter 2

Geometric tools

In this chapter, we introduce a few common geometric tools and models used in research on packing: the *no-fit polygon*, the *inner-fit polygon* and the *collision-free region*. Those tools allow us to determine whether problem constraints are violated and to place pieces so that the constraints are satisfied.

2.1 Collision detection methods overview

In order to determine whether there is an overlap between two pieces and to measure this overlap, a range of methods are available. Static collision detection methods measure overlap for a given set of objects in a fixed position, whereas dynamic collision detection can resolve collisions when objects are moving in real-time. Dynamic collision detection is used in physics simulation and therefore useful in dynamic searches such as shaking. However, our approach being based on a sequential and static search, only static collision detection is used in this paper.

To determine when an overlap occurs, the most common tools are direct trigonometrical computations, phi-functions and no-fit polygons. A direct computation verifies the intersection between each edge of each polygon and is expensive. A phi-function measures the equivalent of a signed distance between two geometrical objects, the distance being negative when the objects are overlapping. Finally, the no-fit polygon provides the set of positions for which two polygons are overlapping. Computing a no-fit polygon is expensive for polygons with complex shapes, but we choose this method as it is the only one that gives us directly the set of feasible placements for a piece. Inner-fit polygons and collision-free regions are related notions, as explained in the following sections.

2.2 No-fit polygon

The no-fit polygon (NFP) of two geometrical objects A and B with given orientations is the set of relative positions $\overrightarrow{A_{ref}B_{ref}}$ for which A and B are overlapping. Because touching positions are excluded, the NFP is an open set and its boundary is the set of touching positions.

Since $NFP(A,B)$ consists of positions of B relatively to A, we can consider that A is fixed. When computing an NFP it is convenient to assume that piece A is not translated, i.e. its reference point is at the origin, and that piece B is moving. In addition, we assume that A and B have been rotated and we omit the rotation parameter. The NFP of A and B can then be expressed as:

$$NFP(A,B) = \{\mathbf{u} \in \mathbb{R}^2, int(A) \cap t_{\mathbf{u}}(int(B)) \neq \emptyset\} \tag{2.1}$$

The no-fit polygon of two pieces is represented on Figure 2.1. The boundary is shown as a dotted line and the interior is shown in transparent red. The white square inside the hole of piece A represents an isolated point excluded from the NFP, a position in which B exactly fits.

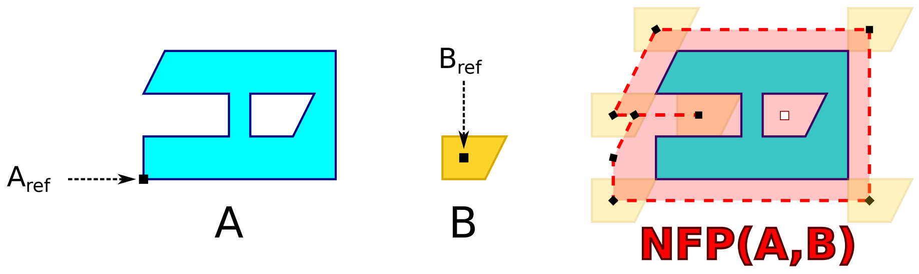


FIGURE 2.1: No-fit polygon of A and B, obtained by letting B orbit around A

The polygons considered in an NFP have an antisymmetric relation: a translation $\mathbf{u}$ of B relatively to A corresponds to a translation $-\mathbf{u}$ of A relatively to B. Hence, for given rotations:

$$NFP(A,B) = -NFP(B,A) \tag{2.2}$$

Since the NFP is based on relative positions, if static piece A is translated by a vector $\mathbf{u}$, the NFP is translated by the same vector:

$$NFP(t_{\mathbf{u}}(A), B) = t_{\mathbf{u}}(NFP(A, B)) \tag{2.3}$$

This expression is especially useful when computing the NFP between a piece A already placed in the container and a new piece to place B.

When computing the NFP for the same pieces with different piece orientations, it is useful to use orientation parameters:

$$NFP(A, B, \theta_A, \theta_B) = NFP(r_{\theta_A}(A), r_{\theta_B}(B)) \tag{2.4}$$

Again, it is more convenient to work with one piece with a fixed orientation. By convention, we assume that A is not rotated ($\theta_A = 0$) and B has an orientation $\theta_B = \theta$. Hence we define $NFP(A, B, \theta) = NFP(A, B, 0, \theta)$. Unlike with translations, the NFP is not preserved if both pieces are rotated by the same angle. Instead, we use the formula:

$$NFP(A, B, \theta_A, \theta_B) = r_{\theta_A}(NFP(A, B, \theta_B - \theta_A)) \tag{2.5}$$

When it is clear from the context what the orientations of A and B are, or when we describe a generic property that is valid for any orientations of A and B, we simply write $NFP(A, B)$.

The boundary of the NFP of A and B, $\partial NFP(A, B)$, is the set of relative positions for which A and B are touching, i.e. their intersection is not empty and is included in their boundaries. Its exterior, $NFP(A, B)^{\mathsf{e}} = \overline{NFP(A, B)}^{\mathsf{c}}$, is the set of relative positions for which A and B are separated i.e. they do not intersect at all.

To sum up, for a translation t of B relatively to A:

- A and B are overlapping iff $t \in NFP(A, B)$
- A and B are touching iff $t \in \partial NFP(A, B)$
- A and B are separated iff $t \in NFP(A, B)^{\mathsf{e}}$

Therefore, once the NFP of two pieces has been computed, checking for overlap is reduced to a point-in-polygon test. In practice, we do not check for overlap after placing a piece;

instead, we place each new piece on the boundary or in the exterior of its NFP with other pieces, so that we ensure that there is no overlap at first.

There are different ways to compute an NFP. Stoyan and Ponomarenko [8] observe that an NFP is a Minkowski sum where one object is reversed:

$$NFP(A,B) = int(A) \oplus -int(B) \tag{2.6}$$

where $X \oplus Y = \{x+y, x \in X, y \in Y\}$.

The Minkowski sum can be computed very easily for convex polygons, using the constructive algorithm of Cunninghame-Green [9]. In this algorithm, edges of A and $-B$ are concatenated by increasing angle of orientation, and the resulting polygon is the Minkowski sum. For non-convex polygons, we can apply a convex decomposition, then compute the NFP between each pair of convex parts obtained, and finally recompose the NFP between the non-convex polygons. This is possible by using the following decomposition formula of an NFP.

Let A and B be two polygons composed of a finite number of closed polygons:

$$\begin{cases} A = A_1 \cup \cdots \cup A_p \\ B = B_1 \cup \cdots \cup B_q \end{cases}$$

Then the NFP of A and B can be reconstructed as:

$$NFP(A,B) = \bigcup_{\substack{i=1..p \\ j=1..q}} NFP(A_i, B_j) \tag{2.7}$$

Indeed, A and B are overlapping iff any of their respective parts are overlapping with each other. Note that set interiors are used in the definition of the NFP, so the proof is not immediate.

Convex decomposition is expensive ($O(n^4)$ for n vertices, as stated by Hert [10]), and polygon triangulation is cheaper ($O(nr^2)$ for r non-convex vertices) but creates more parts, hence requires more union operations to reconstruct the NFP. Besides, decomposing a polygon with holes is NP-hard except for approximate decompositions, Lien and Amato [11]. Therefore, for complex non-convex polygons, it may be advantageous to apply other methods to compute NFPs.

One technique is to compute the convolution of two polygons by concatenating their edges in a specific order. The method is similar to Cunninghame-Green, but the resulting polygon may have self-intersections due to windings around some points, all windings being clockwise. Points with a non-zero winding number inside the convoluted geometry are the points of the NFP. Wein [12] explains the base of the algorithm in the user manual of the Computational Geometry Algorithms Library (CGAL), in the section *2D Minkowski Sums*. Behar and Jyh-Ming Lien [13] propose an optimized approach that spare the computation of convoluted edges not used in the final shape.

Burke et al. [14] describes another method to compute $NFP(A, B)$ called *orbital sliding*. The method consists in two parts. First, B is set in contact with A from the exterior and orbits around A. We track the positions occupied by B_{ref} to trace the exterior boundary of the NFP. Second, B is warped in holes of A where it can fit, or, if B is bigger, warped so that A is inside one of its holes. Then, the orbiting process resumes to trace the remaining parts of the boundary of the NFP. Eventually the boundary is completed and the NFP is deduced. See Figure 2.2 for a visualization of the process.

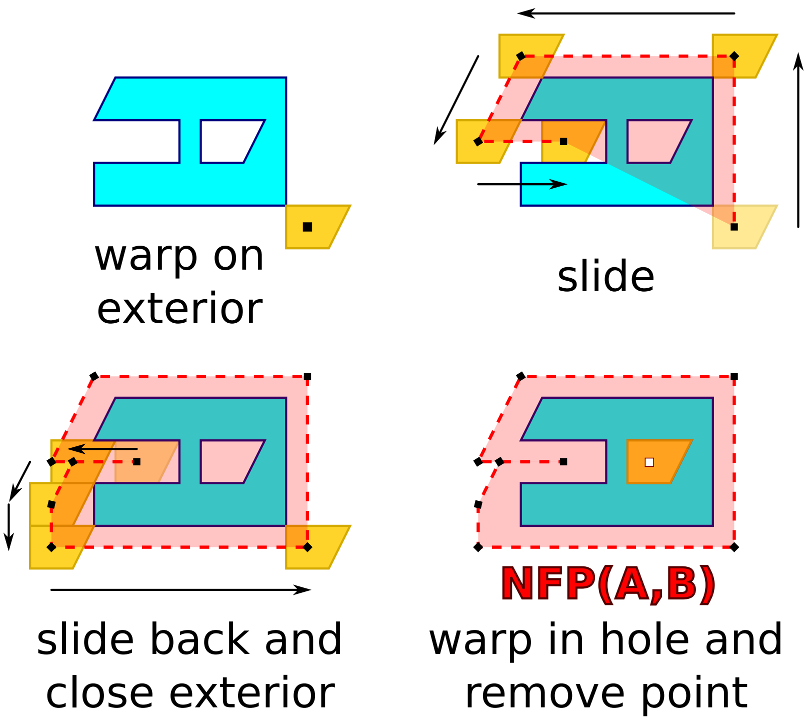


FIGURE 2.2: No-fit polygon construction with Burke's orbital sliding

For all three methods, extra caution must be taken when dealing with geometries where B has exact fitting or sliding positions inside A. An *exact fit* is defined as a position from which a piece cannot move at all, and an *exact slide* as a position from which a piece can only move in a finite number of directions. Such positions correspond to degenerated points or edges that are excluded from the geometry. Figure 2.3 shows one exact fit and one exact slide position.

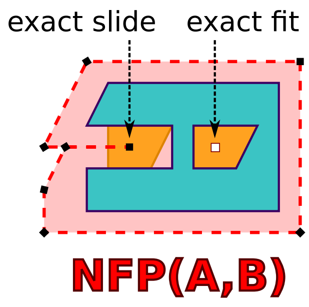


FIGURE 2.3: Exact fit and slide positions in a no-fit polygon

Degenerated cases also appear in the geometrical sets presented in the following sections.

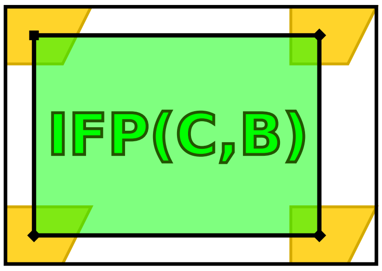
2.3 Inner-fit polygon

The inner-fit polygon (IFP) is the reversed notion of the NFP. The IFP of a contained object A for a container C is the set of positions of A_{ref} for which A lies inside C, including when A touches the boundary of C. Therefore, an IFP is a closed set. Since the container is always fixed in our problem, positions and relative positions of A_{ref} are the same, and they are equal to the translations of A from the origin. Note that there is no symmetric nor antisymmetric relation between the object parameters of an IFP. We will always write the container parameter first.

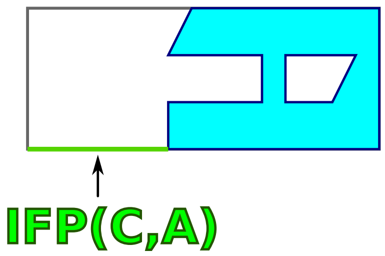
$$IFP(C, A) = \{\mathbf{u} \in \mathbb{R}^2, t_{\mathbf{u}}(A) \subseteq C\} \tag{2.8}$$

Again, we can write $IFP(C, A, \theta) = IFP(C, r_\theta(A))$ but we omit the orientation parameter when the context is clear.

The IFP is used to verify the unary constraints of the problem. In addition, the IFP provides the set of valid positions of a piece when the layout is empty, making it an essential part of the computation of the placement domains.



(A) Inner-fit polygon of B in C: a rectangle



(B) Inner-fit polygon of A in a modified C: a line

FIGURE 2.4: Inner-fit polygon as inner-fit rectangles

For a translation t of A:

- A is strictly contained in C iff $t \in int(IFP(C, A))$
- A is touching C from inside iff $t \in \partial IFP(C, A)$
- A is protruding from C iff $t \in IFP(C, A)^{\mathsf{e}}$

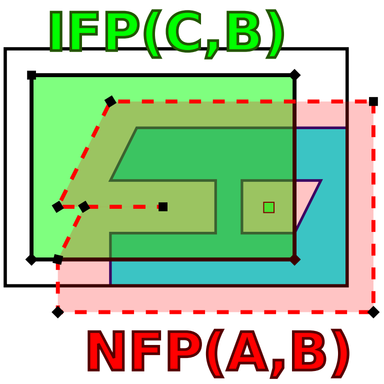
As we only work with rectangular containers, our IFPs are always rectangles or degenerated rectangles. Such an IFP is sometimes called inner-fit rectangle (IFR) in the literature. IFRs are very cheap to compute. To compute IFRs we use the concept of *axis-aligned bounding box* or simply *bounding box* of a set X, which is the smallest axis-aligned rectangle that contains X. Depending on the size of the bounding box of a piece A compared to a rectangular container C, $IFP(C, A)$ may be:

- the empty set
- a point (only one position allowed)
- a line (the polygon can only slide in one direction)
- a rectangle, smaller than the container

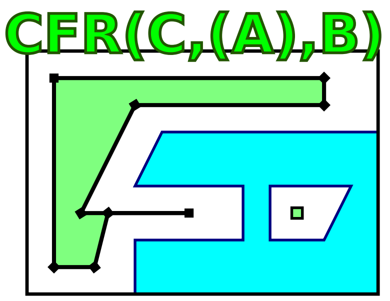
In general, an IFP may present degenerated edges or points, included in the geometry. IFRs are only degenerated in the case of a point or a line. Two different cases are illustrated in Figure 2.4.

2.4 Collision-free region

The collision-free region (CFR) of a piece P, with a given orientation, inside a container C where other pieces $P_1, \ldots, P_k$ have been placed, is the set of translations of P for



(A) $IFP(C,B)$ and $NFP(A,B)$ superposed



(B) $CFR(C,(A),B)$ with degenerated edges and vertices

FIGURE 2.5: Collision-free region computation

which P is inside the container and does not overlap with other pieces:

$$CFR(C,(P_1,\ldots,P_k),P) = \{\mathbf{u} \in \mathbb{R}^2, t_{\mathbf{u}}(P) \subseteq C \setminus \bigcup_{i=1..k} int(P_i)\} \tag{2.9}$$

P is used instead of $int(P)$ in the right-hand side expression because

$$int(P) \cap int(P_i) = \emptyset \Leftrightarrow P \cap int(P_i) = \emptyset$$

A piece placed in its collision-free region satisfies both the unary and binary constraints of the problem. Since the IFP represents unary constraints and the NFP represents binary constraints, we can reformulate the expression:

$$CFR(C,(P_1,\ldots,P_k),P) = IFP(C,P) \setminus \bigcup_{i=1..k} NFP(P_i,P) \tag{2.10}$$

We define $CFR(C,(P_1,\ldots,P_k),P,\theta) = CFR(C,(P_1,\ldots,P_k),r_\theta(P))$. When the orientation of P is clear, we write $CFR(C,(P_1,\ldots,P_k),P)$, and when the layout is given by the context, we write $CFR(P,\theta)$ or simply $CFR(P)$.

Like IFPs, a CFR can present degenerated edges and vertices including in its geometry. Sato et al. [4] note that even if the IFP is a normal rectangle and all the NFPs are non-degenerated, after the union and difference operations in equation 2.10, the CFR may be degenerated. An example of CFR is shown on Figure 2.5.

By computing the IFP of each piece with the problem container and the NFP of each couple of piece, we can compute CFR of any piece at any step of the search, at the cost of non-manifold union and difference operations. In section 3.5.1 Domain revision, we reformulate an incremental expression for the CFR.

Once CFRs are known, we can define a piece translation domain as the CFR of this piece to ensure that its future placement will satisfy all the problem constraints:

$$D_\theta(P) = CFR(P, \theta)$$

Implementation note 1. In order to deal with the potential degenerated parts in NFPs and CFRs, we designed a special class of polygons detailed in section 4.3 Non-manifold geometry.

Chapter 3

Hybrid graph-constrained search

3.1 Process overview

The approach we propose is a sequential search as defined in section 1.4 Literature review. Based on previous research on the definition of discrete constraint satisfaction problems as graph search problems, we design a framework that combines the characteristics of a graph search and a classical constrained search with continuous domains. We fill the components of the framework with data and methods specific to the irregular packing problem. Our approach is similar to other methods using graph-specific techniques such as the beam search of Bennell and Song [3], but we put more emphasis on filtering and heuristics that are specific to constraint problems.

The graph search components allow a nonlinear search over the space of packing layouts, while the constrained search components allow us to heavily filter the layouts and to apply heuristics based on the placement domains of each piece. Below, we describe the global process and the components used in our approach.

During the search, we build a graph where each node represents a layout. Following the terminology of graph and constrained search, we sometimes use the terms *state* and *assignment* to designate a layout contained in a node. A node also contains useful information for heuristics and filtering such as the last piece that has been placed in the layout, where it has been placed, the domains of the remaining pieces (see section 4.4 Incremental cache) and derived attributes to easily access the cost and the remaining pieces at the current state. Unexplored nodes are stored in a *fringe* and a new node is popped at each iteration (see 3.4.1 Fringe). Search heuristics give priority to certain kind of layouts (see 3.4.2 Heuristics). A popped node is filtered, and if it is not globally consistent it is dropped for CSPs, or the cost limit is increased for COPs (see 3.5 Filtering). If the node is not dropped, it is expanded to produce new nodes representing *successor layouts*. A successor layout is obtaining by placing a new piece on the layout. When expanding a node, all the remaining pieces are considered. For each of them, a value picker selects a finite set of placements among the placement domain and those placements are used to generate the successor layouts (see 3.3 Placement picker).

In our approach, we do not move or remove a piece to obtain a successor layout, so local searches are not possible. Furthermore, unlike other approaches, we do not choose which piece to place first, then where to place this piece, but generate a range of successors with different placements for each remaining piece at once. Then we leave the choice of the next piece and its placement to the heuristics.

We describe the high-level algorithms used for the Single Knapsack and the Open Dimension Problems in Algorithm 1 and Algorithm 2. Since our algorithms are generic, they can be applied to any Constraint Satisfaction Problem and Constraint Optimization Problem respectively. Lines specific to a CSP are written in blue, and lines specific to a COP are written in bronze. In addition to the *problem*, we pass a *strategy* parameter that contains our module settings for the search: `picker`, `heuristics` and `filter`. The modules are described in the following sections. Note that we added a precomputation step to obtain all the NFPs before the search begins. The precomputation is explained in the next section.

The iteration process is similar to a normal graph search, except that we apply filtering after popping a node, and that in order to generate successors we must pick specific values in the domain of each unassigned variable. For the COP, we must also manage the dynamic length of the container, represented by *cost limit*.

3.2 Precomputation

Since NFPs only depend on the shape of the pieces and their computation is expensive (see 5.2.1 Precomputation), we precompute the NFP between each pair of pieces $NFP(P_i, P_j)$, avoiding redundant computations when several pieces have the same shape.

Algorithm 1 Constraint satisfaction problem graph search

```
1:  function CSP-GRAPH-SEARCH(problem, strategy)
2:      precompute NFPs
3:      create fringe based on strategy.heuristics
4:      create node with empty layout for problem and push it to fringe
5:      while fringe is not empty do
6:          pop node from fringe                ▷ break if max iterations/time reached
7:          if node is model then
8:              return node
9:          apply strategy.filter to node
10:         if node is proven not globally consistent then
11:             continue
12:         successors ← EXPAND(node,strategy.picker)              ▷ pick values here
13:         push successors to fringe
14:     return null
```

Algorithm 2 Constraint optimization problem graph search

```
1:  function COP-GRAPH-SEARCH(problem, strategy)
2:      precompute NFPs
3:      create fringe based on strategy.heuristics
4:      create node with empty layout for problem and push it to fringe
5:      best model ← null
6:      while fringe is not empty do
7:          pop node from fringe                ▷ break if max iterations/time reached
8:          if node is model then
9:              if node.cost < best cost then
10:                 best model ← node
11:                 best cost ← node.cost
12:             continue
13:         apply strategy.filter to node
14:         if node is proven not globally consistent then
15:             extend cost limit
16:         if node.cost ≥ best cost then
17:             continue          ▷ ignore nodes that cannot improve the solution
18:         successors ← EXPAND(node)                               ▷ pick values here
19:         push successors to fringe
20:     return best model
```

In addition, we use the antisymmetry (2.2) and the relative rotation (2.5) formulas to optimize the computations.

3.3 Placement picker

Placement domains are continuous and a graph search is discrete, so we need to discretize the domains. We use an approach called *value picking*, in which we only pick a finite sample of domain values to generate successor layouts.

Remark 3.1. Another approach for discretization is called *domain splitting* and consists in partitioning the domain space so that each node represents a range of possible layouts. The process comes from discrete constraint problems where splitting a finite set is relatively easy, and its application to continuous constrained optimization is explained by Pedamallu et al. [15]. However, it is more difficult to use inference and filtering with domain splitting, because no specific placement is chosen.

To generate the successors of a node n, for each remaining piece P and for each orientation $\theta \in O(P)$, we pick a finite set of positions $\mathbf{u}$ from $D_\theta(P)$ the translation domain associated to this orientation. Then, for each placement $p = (\mathbf{u}, \theta)$ of P, we create a clone (deep copy) of n and we add P at p in the cloned layout. This process corresponds the EXPAND function in Algorithms 1 and 2, and takes a picker function as parameter. We describe it in Algorithm 3.

Algorithm 3 Placement picking

```
1: function EXPAND($node$, $picker$)
2:     $successors \leftarrow$ empty list
3:     for $P$ in $node$.remaining_pieces do
4:         for $\theta$ in $O(P)$ do
5:             $placements \leftarrow$ PICKER$(P, \theta)$
6:             for $p$ in $placements$ do
7:                 $s \leftarrow$ copy($node$)
8:                 assign $p$ to $P$ in $s$
9:                 append $s$ to $successors$
10:    return $successors$
```

Some placement picking strategies are exclusively based on the shape of the translation domain, while others choose positions that maximize some evaluation criterion. Most strategies presented in the literature pick points on the boundary of the translation

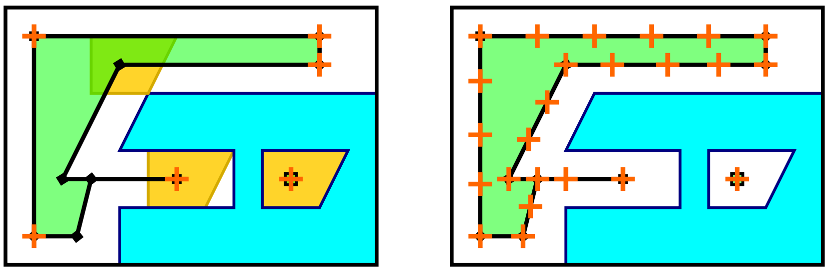

(A) Convex vertex picker (B) Regular vertex picker

FIGURE 3.1: Placement pickers based on domain shape.
A cross indicates a picked position.

domain, because the boundary represents positions where P touches other pieces and/or the container, which often contributes to a better packing.

Placement picker based on domain shape

For picking based on the shape of the domain, we use:

- *vertex picking*: we pick all the vertices of the translation domain. A variant, suggested by Sato et al. [4], consists in picking only convex vertices. This is because concave vertices correspond to positions where P touches other pieces at only one point, which is often undesirable. The disadvantages of vertex picking are that irregular domain shapes with many vertices produce many similar successors, and that positions in the middle of an edge of the domain are ignored. See Figure 3.1a for an illustration of a few positions.

- *regular picking*: we pick evenly spaced points on the boundary of the translation domain. We can either include all domain vertices or skip some of them in the parts of the boundary that are dense in vertices. In our implementation of regular picking, we pick all the vertices plus evenly spaced points on each edge of the translation domain. The regular picker is effective on problems where some pieces have long edges, but tends generates many similar successor layouts. See Figure 3.1b for an example.

Placement picker based on evaluation optimization

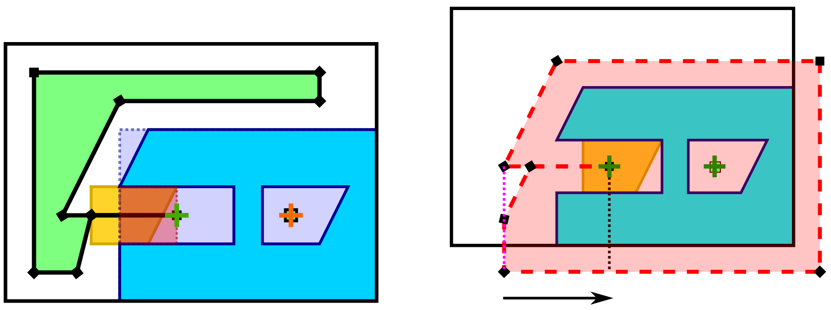
Strategies that maximize an evaluation criterion pick a few positions that are likely to result in a good packing. As they produce few successors for the search, the computations are lighter, but the picker may miss other potentially interesting positions.

A few strategies based on an evaluation criterion are listed below. We denote P the next piece to place, $P_1, \ldots, P_k$ the pieces already placed in the layout and $F = \bigcup_{i=1..k} P_i$ the geometry of the current layout.

- *minimum layout bounding box* (Min-BB): we pick positions of P for which the bounding box of the successor layout has the smallest area. To find these positions, we let P orbit around F while staying inside the container, following the path $\partial NFP(F, P) \cap IFP(C, P)$. We record a key position each time the bounding box of P starts or stops affecting the bounding box of $F \cup P$ in one direction. By studying the quadratic expression of the area of the bounding box of $F \cup P$, we observe that the positions that minimize it are always among the key positions. This picker tends to make the packing have a rectangular shape.

- *maximum bounding box overlap*: we pick the positions that maximize the area of the intersection between the bounding box of P and the union of the bounding boxes of $P_1, \ldots, P_k$. We define the same key positions as for the minimum layout bounding box picker above. The bounding box overlap area is either maximized at a key position, or between two key positions, in which case we obtain the maximizing positions by studying the quadratic polynomial expression of the area. This picker tends to make the bounding box of pieces fit inside each other.

- *dominant point* (DP): we pick the position of P that enters the deepest inside the convex hull of $NFP(F, P)$. Elkeran [6] describes the method of the dominant point, but applies it in a preprocessing phase to create clusters of two pieces. The dominant point picker has a similar effect to the maximum bounding box overlap picker, but is based on the movement of P around F rather than its shape, and is not axis-dependent.


(A) Maximum bounding box overlap (piece shown in intermediate position)

(B) Dominant point

FIGURE 3.2: Placement pickers based on domain shape.
A cross indicates a picked position.

3.4 Fringe and heuristics

3.4.1 Fringe

All the fringes we use are based on the data structure of a priority queue. In a classic priority queue, all the items are automatically sorted according to a priority passed with the pushed items. In our fringe, the priority is automatically computed by an evaluation function: the search heuristics. When a layout node is pushed to the priority queue, its heuristic is immediately calculated and the node is sent to the right position in the queue. The nodes with the lowest heuristics are considered as the best nodes, and placed at the head of the queue.

Fringe type

We designed a few variants of the priority queue and used them as fringes. The variants are listed below, with parameters written between square brackets.

- *normal fringe*: the capacity of the fringe is unlimited and the node with the lowest heuristic is popped. Layouts considered the best are always explored first.
- *bounded fringe* [`limit`]: the capacity of the fringe is limited to `limit`. If the maximum capacity is reached, the worst layouts are dropped first. It is useful to reduce memory footprint and heuristic computation time; but if `limit` is set too low, interesting layouts may be lost.

- *random fringe* [`random span`]: the node returned by a pop is randomly chosen among the best `random span` nodes. This fringe allows us to try layouts with heuristics equal or slightly higher than the heuristic of the best node, and reduce the impact of small variations in heuristic values.

Number of fringes

For a given fringe type, we choose how many fringes to use. In most cases, only one fringe is enough. When a search is trapped in a branch with layouts of low quality, using multiple fringes allows to increase the diversity of the layouts explored.

The fringes are handled in the following way:

1. *single fringe*: all the nodes are pushed to and popped from the same fringe, so all layouts are compared at the same time. If the fringe is empty at some point, the search stops.
2. *multiple fringes* [`K`, `M`]: `K` fringes of the same type are managed in parallel. Each fringe initially contains a different set of nodes prepared for a multi-start search. Nodes are popped from each of the fringes in turn, and successors are pushed on the same fringe as their parent. In addition, if one of the fringes is empty at some point in the search, it will receive all the but the best `M` nodes of another non-empty fringe. This ensures that all the `K` fringes are used at any time during the search.

3.4.2 Heuristics

Search heuristics allow us to to explore a range of layouts efficiently by providing problem-specific information on what we consider a good layout is. Unlike picking, heuristics do not restrict the set of placements to try for each piece, so that layouts scored with an average or low quality will eventually be explored within the maximum number of iterations allowed for the search. Evaluations used in placement pickers can also be used for search heuristics. Furthermore, it is easier to score a given layout than to find a placement that maximizes that score, hence more complex evaluations can be used for heuristics than for placement picking.

We use sub-heuristics that we categorize in three groups:

- *Depth-based heuristics*: heuristics that depend on the depth in the search tree, i.e. the number of assigned variables. Although we call this a group, we use only one heuristic of this kind: the depth-first heuristic, that gives a higher priority to nodes with more assignments. This is because in constrained search, the path taken to assign all the variables does not matter; instead, we want to find a complete assignment as fast as possible.
- *Variable-based heuristics*: heuristics that depend on the variable assigned in the node, but not on its value. In the packing problem, it helps us defining which pieces should be placed first. In addition, it can return different evaluations for different orientations of the same piece.
- *Value-based heuristics*: heuristics that depend on the value assigned in this node. Since the value alone is often meaningless, such heuristics actually depend on both the variable and the value assigned, as well as on the state resulting from the assignment.

All those sub-heuristics can be combined to form a complete heuristic. In this thesis, we combine them by priority order, using the order depth → variable → value and sometimes depth → value → variable or simply depth → value.

Since there is only one heuristic in the first group, we detail variable-based and value-based heuristics in the following sections.

Variable-based heuristics

In most sequential searches from the literature, a new piece is chosen at each assignment step, then a placement is chosen for this piece. Using variable-based heuristics allows us to define which piece to place next, but since all the remaining pieces are ordered by priority, if placing the piece with the best heuristic first does not work, the search will continue, trying other pieces first.

The main variable-based heuristics we use are:

- *Bigger first* (Big): we place the biggest pieces in priority. The size of a piece can be defined by measurements such as its area, its bounding box area or its diameter.

Bigger first is the most common method used in the literature to choose the next piece to place, and comes from the idea that big pieces tend to have fewer available positions, making it easier to try them all. In addition, big pieces restrict the domain of remaining pieces faster, so that if a search branch has to fail, it fails early.

- *Minimum Remaining Values* (MRV): the Minimum Remaining Values evaluation is widely used in discrete constrained search problems and returns the number of remaining values in the domain of a variable. Since our domains are continuous, we cannot count how many values they have. Instead, we adapted our MRV heuristic to return the size of the domain of a variable, defined as the tuple *(area of closed interior, length of boundary lines, number of degenerated vertices)*. Tuples are compared in order, so the lines and vertices are compared only if two pieces have the same domain area. This often happens because the domains contain only degenerated parts. At the beginning of the search, MRV behaves like Bigger first, but after that MRV distinguishes a number of cases that Bigger first does not. See Figure 3.3 for a case where Bigger first returns the same value for two orientations of the same piece, but MRV does not.

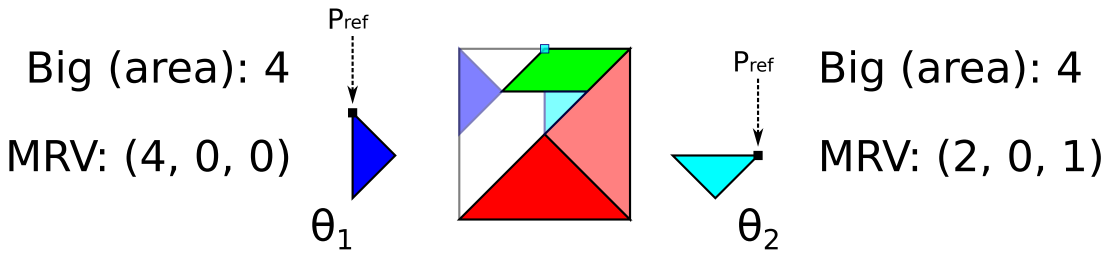


Figure 3.3: Minimum Remaining Values selects the small triangle P for orientation θ_2 because the domain (in cyan) is smaller than the domain for θ_1 (in dark blue)

Value-based heuristics

When used after a variable-based heuristic in order of priority, value-based heuristics selects, for a given piece, the placements that maximize some evaluation first. Used alone, a value-based heuristics will also choose the piece to place. As remarked by Bennell and Song [3], the drawback is that maximizing an evaluation for a partial layout does not guarantee the best results for the final layout. For instance, minimizing the length of the layout at each assignment step will result in placing small pieces first, but big pieces will eventually have to be placed. In this case, it is important to balance the heuristics with more considerations. Bennell and Song [3] suggest to normalize a measurement on the partial layout by a measurement on the placed piece, such as dividing the increase of length for the layout by the length of the placed piece. We can also take the progression of the search into account, for instance by considering the total area of the placed pieces to normalize measurements on the partial layout.

Some of the value-based heuristics we use are listed below, where P denotes the new piece placed in the layout evaluated by the heuristic.

- *Maximum bounding box overlap* (Max-BBO): we maximize the area of the intersection between P and the union of the bounding boxes of the other pieces, as we did for the picker of the same name. We also define a balanced version as suggested by Bennell and Song [3], where the intersection area is normalized by the area of the new piece placed.
- *Minimum convex hull waste* (Min-CH): we minimize the unused space inside the convex hull of the layout after P is placed. It is equivalent to maximizing the usage ratio inside the convex hull of the layout. Figure 3.4 shows the unused space in the convex hull of some layout. This is a generic and intuitive way to obtain a good packing without trying to place the smaller or the bigger pieces first. Other kinds of hull can be used, but convex hull and bounding box are the most used in the literature, bounding box being cheaper to compute but axis-dependent. Whichever hull is chosen, normalizing this heuristic by the sum of the areas of the placed pieces allows us to have an estimation of the waste/usage of the final layout.
- *Exact fit* (Exact): exact fit and exact slide positioning have been used by Sato et al. [4] to place a piece at a position from where the degrees of freedom of movement are the most limited. Such placements correspond to degenerated edges and vertices in the piece translation domain, and were illustrated in Figure 2.3.
- *Least Constraining Value* (LCV): we adapted the Least Constraining Value heuristics used in discrete constraint satisfaction problems. Our LCV maximizes the size of the domains of remaining pieces after P is placed. We also use a relative version,

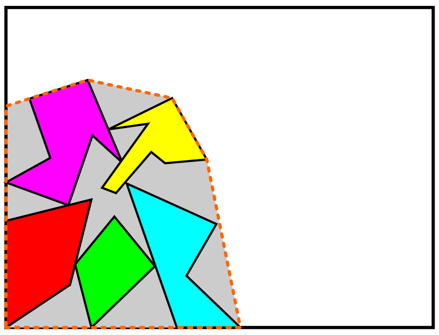
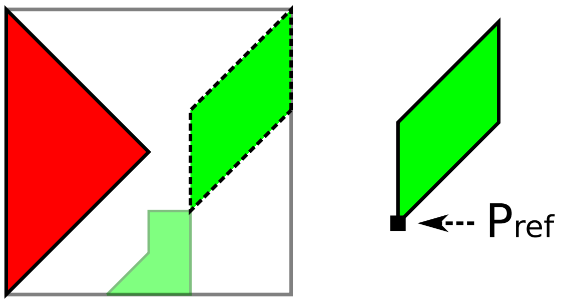


FIGURE 3.4: Minimum convex hull waste: convex hull of the layout shown in dotted orange line, unused space shown in gray

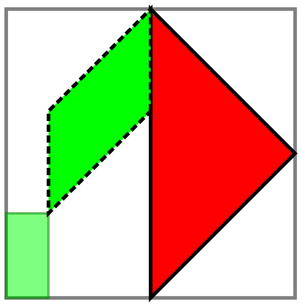
where we must minimize the reduction of the domain size of the remaining pieces. We use the domain definition of a domain size as in MRV. Unlike MRV, we do not work with a specific piece and orientation, so we average the domain sizes over all possible orientations of each remaining piece, and we sum the averaged sizes over all the remaining pieces. To use LCV, all the domains in a layout must be computed when the layout is pushed to the fringe, since the heuristic of a node is computed as soon as it enters the fringe. In comparison, other heuristics update placement domains later, when the layout node is popped, and are therefore much cheaper. See section 4.4 Incremental cache for optimization notes. The effect of LCV is that remaining pieces have bigger remaining domains after the placement, making it easier to place more pieces afterward, at least in short-term.

- *Min length*: we minimize the length of the layout as much as possible. It is the most direct attempt to limit the cost of the solution, but does not guarantee that pieces will fit well together. The effect of the heuristic is that pieces are stacked to the left, *a priori* is an untidy manner. It is a very cheap heuristic in terms of computation. The problem is that, unless a variable-based heuristics such as Bigger first is used before, all the small pieces will be placed first and the end of the search will be difficult, with only big pieces remaining.

- *Minimum dead area* (Min-DA): minimize the area of wasted regions or *dead regions* in the remaining space available in the container. The concept of *dead regions* is explained in section 3.5.2 Dead region analysis.



(A) Placement on the left: preferred because remaining domain is bigger



(B) Placement on the right: leave smaller domain

FIGURE 3.5: Least Constraining Value applied to the placement of the big triangle (Tangram problem). To simplify, only the domain of the remaining parallelogram is considered, shown in pale green.

3.5 Filtering

Filtering is a fundamental component of constrained search. In Constraint Satisfaction Problems, filtering consists of two phases: *domain revision* and *dropping*. For Constraint Optimization Problems, we replaced dropping with another phase, the *cost limit extension*.

- *domain revision*: each time a variable has been assigned, we revise the domains of unassigned variables by removing values that does not satisfy some problem constraints. The most basic revision is Forward Checking (FC) and ensures arc consistency between the newly assigned variable and all the unassigned variables bound to it by a binary constraint.

- *dropping*: in a Constraint Satisfaction Problem, when a new node is popped, its assignment is examined to see whether it is globally consistent, i.e. it is possible to complete the assignment while satisfying all the problem constraints. If the assignment is proven not globally consistent, the node is dropped and the search continues with other nodes. In most cases, dropping occurs because one of the variables has an empty domain, but more advanced methods can detect non-consistency earlier.

- *cost limit extension*: in a Constraint Optimization Problem, non consistency can also be proven for a given cost limit (a given container length in the packing problem). If a state is inconsistent, it is always possible to extend the cost limit so that the state becomes consistent again. In the irregular packing problem, this means that instead of dropping a layout node when it is not consistent, we extend the length of the container and continue the search. However, the solution may be expensive after many extensions.

We have adapted both phases to our graph search. Domain revision is typically done after a layout node is popped, but may be done earlier if a heuristic requires it. For instance, with the LCV heuristic, domains are revised for successor layouts before they are pushed to the fringe. The computation for domain revision is described in the next section, and the procedure to revise the domains when needed is explained in Section 4.4 Incremental cache. The dropping phase is more regular and always occur after a node is popped. This is because dropping requires to revise the placement domains first, and it is cheaper to wait for the nodes to be sorted and then rejecting non consistent layouts among the best than checking them all when they are pushed to the fringe. Advanced elimination procedures are described in section 3.5.2 Layout dropping and 3.5.3 Cost limit extension describes the process of extending the container length.

3.5.1 Domain revision

In the packing problem, all the pieces are bound to every other by a binary constraint, forming a network of constraints also called *rhizome*. Therefore, when applying Forward Checking, the domains of all the remaining pieces are revised. Since the collision-free region is the set of positions of a piece that satisfy all the problem constraints, applying Forward Checking is equivalent to ensuring that the translation domain of each piece is equal to its CFR.

Let $P_1, \ldots, P_k$ be the sequence of pieces previously placed. Let P be a remaining piece in a given orientation for which we want to apply Forward Checking, i.e. we want to determine $CFR(C, (P_1, \ldots, P_k), P)$.

We have seen in equation 2.10 how to compute the CFR from an IFP and a union of NFPs. To avoid expensive union operations, we compute the CFR incrementally, with the initialization and update formulas:

$$\begin{cases} \mathit{CFR}(C, (), P) = \mathit{IFP}(C, P) \\ \mathit{CFR}(C, (P_1, \dots, P_{k+1}), P) = \mathit{CFR}(C, (P_1, \dots, P_k), P) \setminus \mathit{NFP}(P_{k+1}, P) \end{cases} \tag{3.1}$$

where $\mathit{NFP}(P_{k+1}, P)$ takes the placement of P_{k+1} into account with equation (2.3), and the difference operation is detailed in section 4.3 Non-manifold geometry, equation (4.14).

Once the collision-free region has been computed for piece P with orientation θ, we set the translation domain $D_\theta(P) = \mathit{CFR}(P, \theta)$.

Note that filtering techniques that enforce consistency of higher order also exist; they would ensure that it will be possible to place a certain number of pieces from a given state. However, because the constraints of the packing problem form a rhizome and every piece can be placed anywhere in the container, it is difficult to predict the influence of a piece on another several steps ahead.

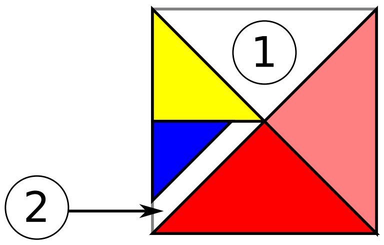
3.5.2 Layout dropping

In Constraint Satisfaction Problems, when an assignment is not globally consistent, it can be rejected from the search. In the next subsections, we describe three methods to detect non-consistency in a layout.

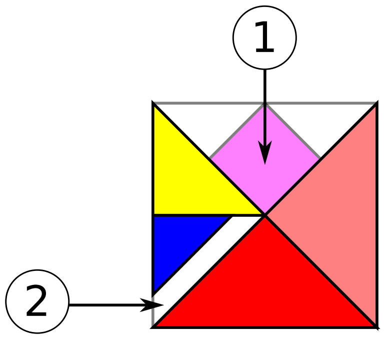
Empty domain

If during domain revision, the domain of one of the remaining pieces was reduced to a empty set, then the layout is non-consistent and the node can be dropped immediately.

Furthermore, we can prove that the layout is globally not consistent in other specific situations. First, we introduce a few geometrical concepts to support our analysis of the placement domains.



(A) Free space composed of two free regions



(B) Square bottom vertex placed at the junction of two free regions

FIGURE 3.6: Free space decomposed in two free regions

Free space and free region

In this section, we introduce the concepts of *free space* and *free region.* We define the *free space* as the set of points that are not covered by any pieces in the container. Unlike the CFR, the points in the free space do not represent the translations of a piece, but actual points in the container. Therefore, the free space is not related to a specific piece and is directly visible on the layout. For a layout with placed pieces $(P_1, \ldots, P_k)$, the free space S inside the container C is:

$$S(P_1, \ldots, P_k) = C \setminus \bigcup_{i=1..k} P_i \tag{3.2}$$

Again, we can compute the free space incrementally:

$$\begin{cases} S() = C \\ S(P_1, \ldots, P_{k+1}) = S(P_1, \ldots, P_k) \setminus P_{k+1} \end{cases} \tag{3.3}$$

The free space is an open set and does not contain the boundaries of the pieces and of the container. The free space is composed of a finite number of open polygonal parts that we call *free region.*

Because free regions are isolated from each other, a new piece can and must be placed in only one free region at a time. Since pieces are closed geometries, the interior of a piece may be included in a free region, whereas the geometry of a piece may be included in the closure of a free region. See Figure 3.6a for an illustration of a free space composed of two free regions.

We want to know in which free regions a remaining piece P can fit. We call such regions the *fit regions* of P. Similarly to free space, free regions represent actual points in the plane, but we can relate them to the CFR of a piece in order to find its fit regions. A free region R is a fit region of P iff the CFR of P contains a translation for which P is located inside R.

Let P be a piece with a given orientation, $\mathbf{u} \in \mathbb{R}^2$, R a free region. $t_{\mathbf{u}}(P)$ is located inside R iff P is placed in its CFR and one of its interior point belongs to R:

$$int(t_{\mathbf{u}}(P)) \subseteq R \Leftrightarrow \mathbf{u} \in CFR(P) \wedge \exists M \in int(P), M \in R \tag{3.4}$$

Let $M \in int(P)$, and we define $\mathbf{v} = \overrightarrow{P_{ref}M}$. The CFR of P is the set of all the feasible positions of its reference point. By translating $CFR(P)$ by $\mathbf{v}$, we obtain the set of feasible positions of the interior point M that we call an *interior point translation domain* $IPD = t_{\mathbf{v}}(CFR(P))$. Therefore:

$$R \text{ is a fit region of } P \Leftrightarrow (t_{\mathbf{v}}(CFR(P))) \cap R \neq \emptyset \tag{3.5}$$

Figure 3.7 shows an example of a piece P where the reference point is outside the geometry. By translating the CFR of P, we obtain the IPD of P. Here, the IPD has only one part, which is contained in the only fit region of P, the upper free region.

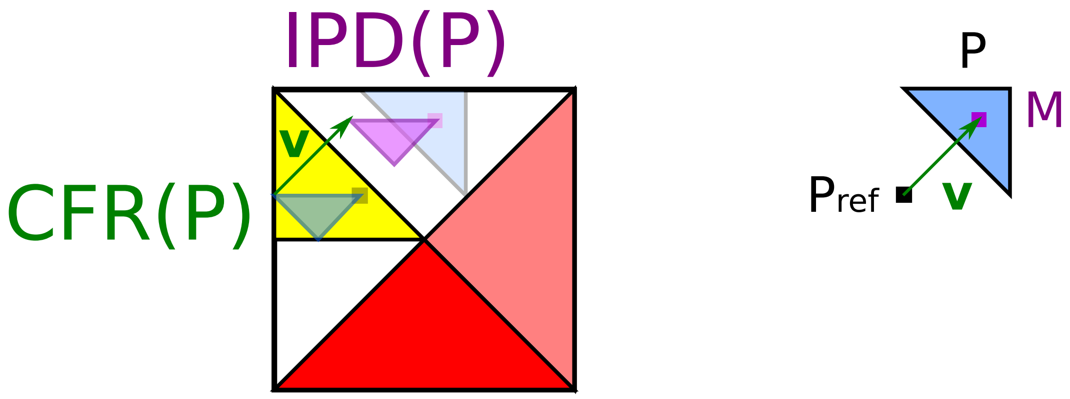


FIGURE 3.7: Interior point translation domain of triangle piece P in purple. The collision-free region of P, in light green, is translated by $\mathbf{v} = \overrightarrow{P_{ref}M}$, resulting in the IPD.

It is worth noting that we work with open sets and interior points because if two regions are contiguous, closed sets and boundary points are not enough to determine inside which region a piece is located. For instance, in 3.6b, we study the fit regions of the square piece. The bottom vertex of the square belongs to the closure of both regions 1 and 2 at their junction point, and testing the position of this vertex against the free regions is ambiguous. In contrast, the center of the square is an interior point and only belongs to region 1.

Algorithm 4 describes the process of determining the fit regions of each remaining piece. The function returns a dictionary containing the list of fit regions per piece, where pieces are identified by a unique ID. The first phase of the algorithm consists in precomputing all the interior point translation domains, and the second phase uses equation 3.5 to verify if a piece can fit in a region.

Algorithm 4 Determine fit regions for each remaining piece

```
1: function GET-FIT-REGIONS($node$)
2:     $IPD \leftarrow$ empty dict                    ▷ dictionary of interior point domains per piece ID
3:     for $P$ in $node$.remaining_pieces do
4:         for $\theta$ in $O(P)$ do
5:             $M \leftarrow$ a point of $int(P)$
6:             $\mathbf{v} \leftarrow \overrightarrow{P_{ref}M}$
7:             $IPD[P.id, \theta] \leftarrow t_{\mathbf{v}}(node.CFR(P, \theta))$       ▷ CFR for the layout in $node$
8:     $fit\_region\_dict \leftarrow$ dict of empty list for each piece ID
9:     for $R$ in node.free_regions do                 ▷ free regions for the layout in $node$
10:        for $P$ in $node$.remaining_pieces do
11:            for $\theta$ in $O(P)$ do
12:                if $IPD[P.id, \theta] \cap R \neq \emptyset$ then
13:                    append $R$ to $fit\_region\_dict[P.id]$
14:                    break  ▷ piece can fit for one orientation, no need to check others
15:    return $fit\_region\_dict$
```

Dead region analysis

We define a *dead region* as a free region in which no piece can fit. When a dead region is found, we can consider it as a waste from the point of view of the packing. The remaining pieces must occupy the other regions or *living regions*. We apply a simple area analysis to check whether the remaining pieces can possibly fit in those living regions. We define the *living area margin* LAM as the difference between the area of the living regions LR_i

and the area of the remaining pieces Q_i:

$$LAM = \sum_i \mathcal{A}(LR_i) - \sum_i \mathcal{A}(Q_i)$$

If the margin is positive or null, then the remaining pieces fit "in area" in the living regions, i.e. the total area of the living regions is enough to contain all the remaining pieces. However, this study does not take the shape of the pieces into account, hence we cannot guarantee that the pieces can geometrically fit as well. If the margin is **negative**, then the area of the living regions is not enough to contain all the remaining pieces, and the layout is proven **globally non-consistent**: we can drop the layout node. See figure 3.8 for a representation of a dead region. On the figure, the only living region, in white, is not enough to contain all the remaining pieces in area, because the Tangram only has compact solutions, i.e. solutions with no waste.

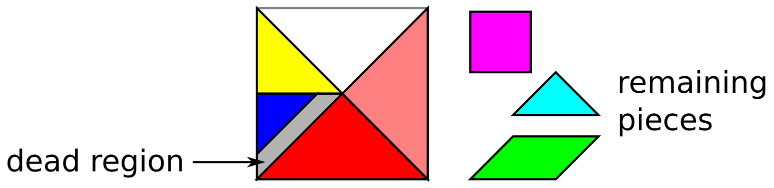


Figure 3.8: Dead region in gray: none of the remaining pieces can fit there

In addition, dead region analysis can useful for heuristics: as stated in section 3.4.2 Value-based heuristics, we can define *Minimum dead area*, a value-based heuristic that returns the sum of the areas of the dead regions in a layout. With this heuristic, a search will avoid layouts that create too much waste. However, the risk is that the search artificially avoids creating dead regions by placing the pieces so that there is always a small space that links all the parts of the free space together, and that dead regions can never be isolated. Another disadvantage of this heuristic is that it requires to compute more placement domains, similarly to LCV, so it increases the heuristics computation time.

Area knapsack analysis

We can go further with our domain analysis. Again, we use the actual geometry of the remaining pieces to determine their respective fit regions. Then, only considering their areas, we try to distribute the pieces into the free regions to see if they can fit without overflowing in area, this time considering each region as a separate container.

This problem is itself a variant of the irregular packing problem called the One-Dimensional Multiple Heterogeneous Knapsack Problem or 1D MHKP (Wäscher et al. [1]): one-dimensional objects called *orders* have a certain length and must be packed into a set of one-dimensional containers called *stocks* that have different lengths. In our case, the orders are the remaining pieces, the stocks are the free regions, and the lengths are the areas of the pieces and the regions. In other words, we have reduced the problem of 2D packing to a simpler 1D sub-problem. If the 1D problem is infeasible i.e. it has no solutions, then it is impossible to solve the 2D problem. Indeed, the existence of a packing of pieces into the free regions implies the existence of a distribution of their areas into the areas of the free regions. However, since the 1D problem has looser constraints than the 2D problem, the existence of a solution to the 1D problem does not guarantee the existence of a solution to the 2D problem.

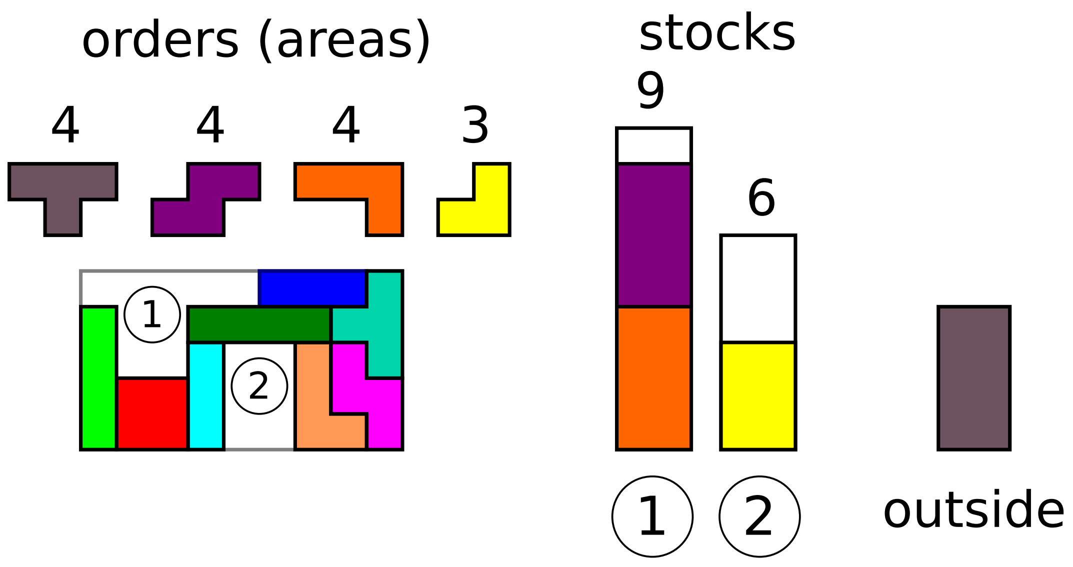


FIGURE 3.9: Area knapsack: the remaining pieces cannot fill the free regions in area

On figure 3.9, the container has two free regions numbered 1 and 2. Region 1 has an area of 9, region 2 an area of 6. The remaining pieces are shown around the container and we assume that they cannot be rotated. The fit region analysis shows that only the yellow (elbow-shaped) piece can fit in region 2. However, its area is only 3, so it can only fill half of region 2. The three other pieces can all fit in region 1, but their total area is $3 * 4 = 12$, which is superior to the area of region 1. Therefore, it is impossible to place all the pieces in the free regions. This layout is globally not consistent.

As always, non-consistent layouts are dropped. The 1D MHKP is discrete and can be either solved or proven to be infeasible very fast. Therefore, we have an efficient filter method for dense packing problems where isolated regions tend to appear quickly.

Note that the Area knapsack analysis includes the inference and conclusion of the Dead region analysis since a dead region is a special case where a free region *stock* cannot receive any piece *order*. Therefore, filtering with the Area knapsack analysis allows to drop at least as many nodes as with the Dead region analysis.

Implementation note 2. The 1D MHKP is a discrete constraint optimization problem, and is therefore easily tackled with a graph search. For this reason, we reused our framework for hybrid graph-constrained search for the sub-problem of Area Knapsack, which required almost no extra implementation. This is why we often introduce generic notions of node, heuristics, filter and cost that can be applied to any Constraint Satisfaction and Optimization Problem, and then specialize the program modules for specific problems.

3.5.3 Cost limit extension

In Constraint Optimization Problem, instead of dropping an assignment that is not consistent for a given cost limit, we can extend the cost limit instead. In the packing problem, it is equivalent to extending the length of the container L when a layout is not consistent with the current L. By giving more space to the remaining pieces, we increase the chance to find a complete and valid layout. Whereas local searches gradually increase and decrease L, we only extend L. We choose the next value of L so that the layout cannot be proven non-consistent anymore by the analysis used for layout dropping. For each of the analyses described in the above sections, we explain the length extension procedure.

- Empty domain: If the domain of piece P is empty, we extend L so that $CFR(P)$ contains at least one point, for at least one orientation. This corresponds to allowing P to be placed on the very right of the container. On figure 3.10, the square cannot fit anywhere in state 1. We extend to length so that the square can be placed to the right of the container, in its orientation that has the smaller length.

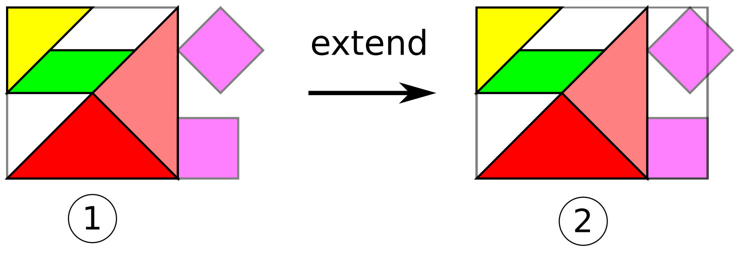


FIGURE 3.10: Length extension for empty domain of square piece

- Dead region analysis: If the living area margin LAM is negative, consider the area overflow $AO = -LAM$. We want to add the area AO to the container area so that the free regions are enough to container the remaining pieces in area. Since the container has a width W, we have to increase the length L by at least AO/W. However, if AO/W is too small, this will produce another dead region, so in this case we must extend the container so that the smallest piece in length can be placed to the right of the container. On figure 3.11, the area overflow is equal to area of the dead region because the solution to the Tangram with its original container length is compact. But an extension of AO/W would create another dead region, so we extend L by the length of the square instead.

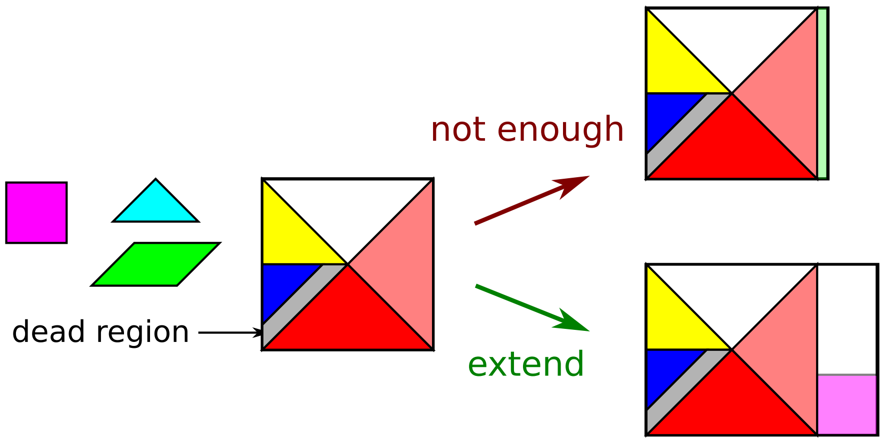


FIGURE 3.11: Length extension: area compensation (in pale green) is not enough, must extend for square

- Knapsack area analysis: if there is no solution to the area knapsack problem, we extend the container length just enough so that there is one. We have to complete the assignments to the knapsack problem as much as possible to see which pieces among the shortest are left outside the container in the end. Then, we have to extend L so that those pieces can fit in length and in area on the right of the container. For instance, Figure 3.12 shows the container extension required if the case shown in Figure 3.9 happened in the Open Dimension problem. Note that the orange (¬-shaped) and the purple (s-shaped) pieces seem to overlap because the area knapsack does not consider the actual geometries of the pieces.

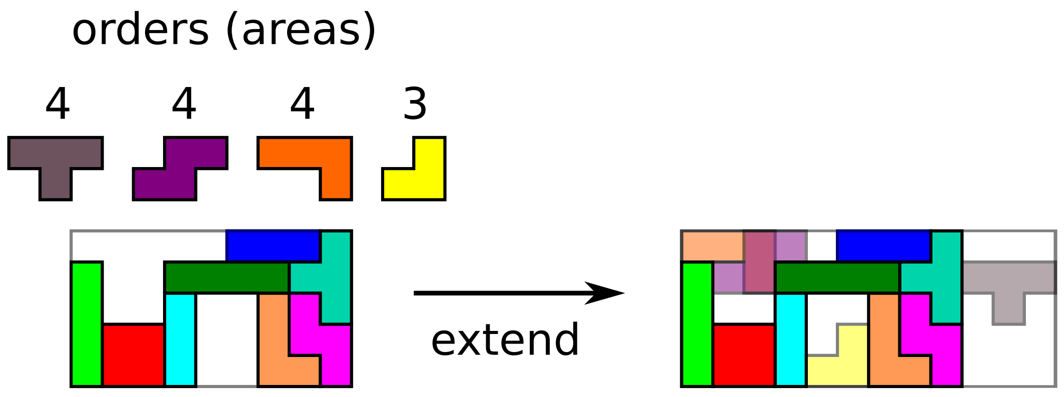


FIGURE 3.12: Length extension: 1D knapsack problem must have a solution

This approach allows us to work with a fixed container length as in the Single Knapsack Problem, until a layout is proven non-consistent for the current length and we have to extend it. The advantage of limiting L is that the search will not spread the layout to the right, as long as there is enough space left in the container. The disadvantage is that the search will give up as soon as a layout is proven non-consistent, and extend L immediately instead of backtracking. In order to balance this behavior, we continue searching for better solutions once a solution is found. To ensure this, we use an additional filter exclusive to Open Dimension searches, that drops any layout with a length equal or superior to the length of the best solution. This can be seen on line 16 of Algorithm 2.

We can sum up the length extension process as follows:

1. Initialize L with the smallest length possible L_m

   We compute the length that a hypothetic rectangular layout where all pieces are packed with a usage of 100% would give. Since in such a packing, the sum of all the piece areas is equal to the area of the rectangle, we deduce the minimum length $L_m = \sum_{i=1}^{n} \mathcal{A}(P_i)/W$

2. If a layout is proven non-consistent for the current L, increase L so that it cannot be proven non-consistent anymore
3. Continue the search normally as in a Single Knapsack Problem, but reject layouts where L has been extended to a value equal or higher than the length of the best solution
4. If a model layout with a lower length than the previous solutions is found, record it as the new best solution

Unlike local search techniques found in the literature, our approach does not allow to reduce the container length, because we do not include any algorithm to relocate pieces that would protrude from the container after this operation.

3.6 Example

An example of search tree with a compact puzzle is shown on Figure 3.13. The puzzle pieces have been taken from the Himilk chocolate puzzle of Hanayama and Meiji [16].

The search tree shown starts in the middle of the search, and only a few branches are shown. All branches eventually end with a node drop due to either the presence of an empty domain, a negative living area margin after the dead region analysis or an infeasible area knapsack problem.

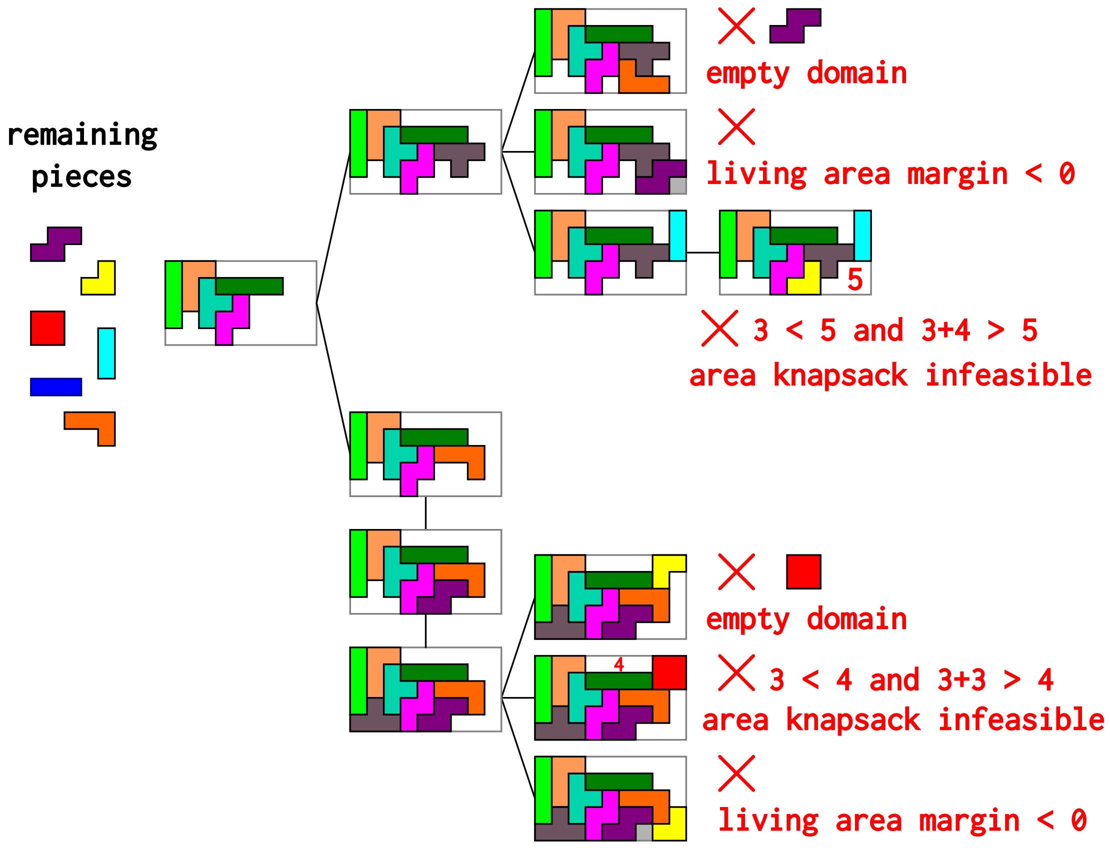


FIGURE 3.13: Example of search tree. Dead regions in gray, piece with empty domain shown after cross symbol, area comparisons shown for area knapsack

Chapter 4

Implementation

4.1 Problem representation

Problem data is stored in an XML file that follows the XML Schema suggested by Martins et al. [17] in 2006 and improved gradually since. The format, called NestingXML, is available as an XSD (XML Schema Definition) file on the website of the ESICUP (Euro Special Interest Group on Cutting and Packing) [18], on the NestingXML page [19].

The format defines the shape of the container and the pieces, the number of pieces of each type to pack, their allowed orientations, metadata to help computations such as precomputed NFPs and sometimes a few solutions found by other researchers.

4.2 Language and libraries

We developed our program in Python 2 for fast prototyping. We used the following packages:

- Geometrical computations: *Shapely*

  Shapely [20] is a Python package for spatial analysis based on GEOS (Geometry Engine, Open Source) [21], a C++ port of the Java Topology Suite [22]. GEOS is specialized in geographical analysis and widely used in world imaging applications. Shapely 1.5 does not allow the developer to choose a specific precision model and thus we think that it is not suited for geometrical computations that require a high precision. Nevertheless, rounding and merging techniques allowed us to use the package for the packing problem.

- XML parsing: *generateDS*

  generateDS generates a Python class from an XSD file, and parses XML files that follow the specification of the XSD to an instance of the generated class. generateDS is based on lxml, an XML parser for Python that is not aware of XML Schema. We used generateDS to parse benchmark problems provided by the ESICUP, and to generate the XML files for our own toy problems, such as the Tangram.

- GUI: *Tkinter* and *matplotlib*

  Tkinter [23] is a basic GUI toolkit for Python. It does not feature advanced capabilities and graphics but was enough for our purpose. We built a single window application that loads a problem from an XML file and runs a search. It is possible to tune the search parameters introduced in this thesis. See Appendix: Application GUI for a view of the application window. We used Matplotlib [24], a Python 2D plotting library, to plot the search steps and the results.

In the literature, many implementations are done in C++, in particular with the C++ library CGAL (Computational Geometry Algorithms Library) [25]. CGAL provides exact precision calculation and a variety of geometric tools, such as Minkowski sums and Nef polygons, that can make the computations of no-fit polygons and collision-free regions easier.

4.3 Non-manifold geometry

In order to handle degenerated edges and vertices present in no-fit polygons and collision-free regions, we need a class of objects that supports non-manifold geometrical operations. One solution is to use *Nef polygons*.

4.3.1 NFP, IFP and CFR as Nef polygons

Nef polygons, designed by Nef [26], represent the set of polygons obtained through a combination of set intersection and complement operations applied to a finite set of half-planes. A brief description of the concept is done by Seel [27] and its implementation for the CGAL library is detailed by Seel [28]. In this implementation, Nef polygons are represented by a plane map (V, E, F) of vertices, edges and faces, each edge and vertex marked as either *included* or *excluded*. Nef polygons handle the kind of degenerated parts that we can find in no-fit polygons, inner-fit polygons and collision-free regions. They may be bounded or unbounded. An example of unbounded Nef polygon is shown on Figure 4.1, with included parts colored.

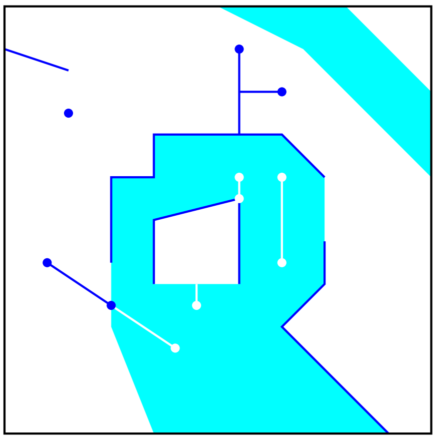
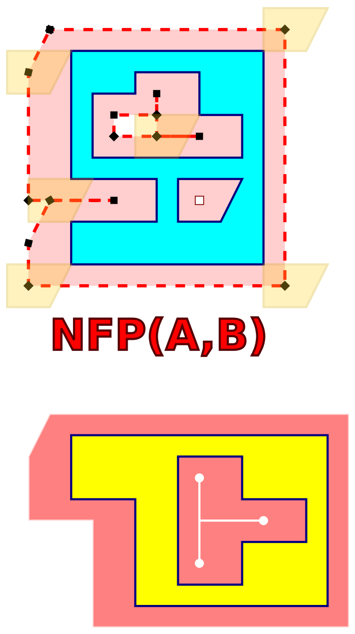


Figure 4.1: Example of Nef polygon. Included edges and points are represented by darker lines and dots, excluded parts are in white. The frame represents positions at the infinity and edges touching the frame represent affine rays or affine lines

No-fit polygons, inner-fit polygons and collision-free regions are instances of Nef polygons. The set of NFPs is the set of open Nef polygons and the set of IFPs and CFRs is the set of closed Nef polygons. Furthermore, when pieces and containers are bounded, NFPs and IFPs/CFRs are bounded open and bounded close Nef polygons respectively. Although physical pieces and containers are always bounded, it is useful to consider unbounded geometries because (1) with them the set of Nef polygons is closed under the complement operation and (2) they can be used for intermediate computations.

A few NFPs are represented on Figure 4.2 and an IFP is represented on Figure 4.3 with the same graphical conventions as on Figure 4.1. For the IFP, a non-rectangular container was exceptionally chosen to illustrate a non-trivial case.



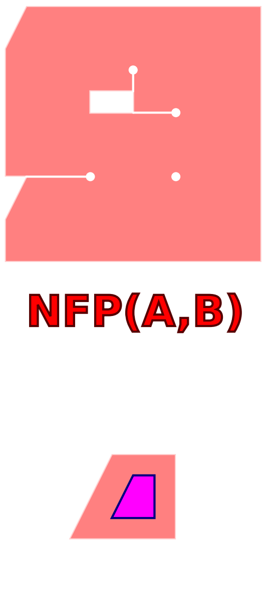


FIGURE 4.2: No-fit polygon of piece B with a modified piece A and with two other pieces, as open Nef polygons

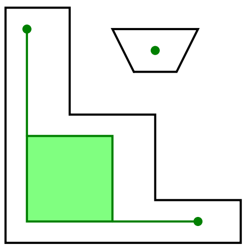


FIGURE 4.3: Inner-fit polygon of piece B inside irregular container, as a closed Nef polygon

If an implementation of Nef polygons is available, it is possible to apply unary or binary operation on NFPs, IFPs and CFRs as Nef polygons. In order to obtain similar results with a simple geometrical library that only supports bounded closed polygons, closed lines and points, we implemented a class that represents a subset of bounded Nef polygons that we called *Polygon with Boundary*.

4.3.2 Polygon with Boundary

We define a *multi-polygon with holes* as a union of zero, one or more closed polygons with polygonal holes. Polygons with Boundary represent the set of multi-polygons with holes from which a finite number of closed segments and points may have been subtracted or added. This definition does not only limit the scope of geometries available compared to Nef polygons, but also give us clues on how to model the data structure of a Polygon with Boundary.

An example of bounded Polygon with Boundary is shown on Figure 4.4.

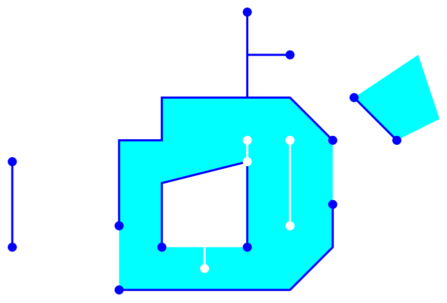


FIGURE 4.4: Example of bounded Polygon with Boundary

Data structure

The data structure of a Polygon with Boundary PWB contains three components: a *regularized set*, a *subtractive boundary* and an *additive boundary*.

Regularized set The regularized set $RS(PWB)$ is the closure of the interior of the set. It corresponds to the original multi-polygon with holes from which we subtracted or added boundaries, and contains all the non-degenerated parts of the geometry. We call $\partial RS(PWB)$ the *natural boundary* of PWB.

Subtractive boundary The subtractive boundary $SB(PWB)$ contains all the segments and points that should be removed from the regularized set. We enforce that only actual points of the regularized set are subtracted: $SB(PWB) \subseteq RS(PWB)$ (inclusion condition). The subtractive boundary is closed. The parts of the natural boundary that are excluded from PWB should all be stored in the subtractive boundary.

Additive boundary The additive boundary $AB(PWB)$ contains all the segments and points to add to the regularized set. Furthermore, all the parts of the natural boundary that are included in PWB should be stored in the additive boundary. Note that this is redundant with knowing the subtractive boundary, but more convenient for computations. Apart from parts on the natural boundary, the additive boundary should be outside the regularized set: $AB(PWB) \subseteq \partial RS(PWB) \cup RS(PWB)^{\mathsf{e}}$ (inclusion condition). The additive boundary is closed.

The actual geometry of a Polygon with Boundary is

$$PWB = (RS(PWB) \setminus SB(PWB)) \cup AB(PWB) = (int(PWB) \setminus SB(PWB)) \cup AB(PWB) \tag{4.1}$$

due to the redundancy between the subtractive and the additive boundaries.

Figure 4.5 shows the decomposition of the previous example of Polygon with Boundary.

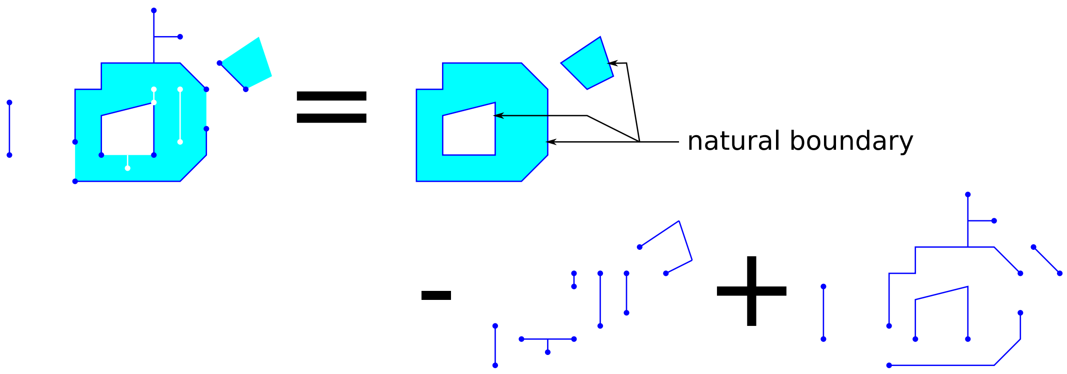


Figure 4.5: Decomposition of Polygon with Boundary with regularized set and boundaries

The boundary of a Polygon with Boundary is given by both boundaries:

$$\partial PWB = AB(PWB) \cup SB(PWB)$$

which includes the natural boundary.

Finally, we can isolate each component:

$$RS(PWB) = \overline{int(PWB)}$$

$$SB(PWB) = RS(PWB) \setminus PWB$$

$$AB(PWB) = PWB \setminus int(PWB)$$

Geometric tools as Polygons with Boundary

In this section, we revisit the NFP, IFP and CFR as Polygons with Boundary.

NFP An NFP is an open Polygon with Boundary (OPWB). The regularized set represents the set of overlap and touch positions. The subtractive boundary always include the natural boundary, and there is no additive boundary. Figure 4.6 shows the decomposition of one of the NFPs used in the previous examples. NFPs as Polygons with Boundary are characterized by:

$$\begin{cases} \partial NFP \subseteq SB(NFP) \\ AB(NFP) = \emptyset \end{cases} \tag{4.2}$$

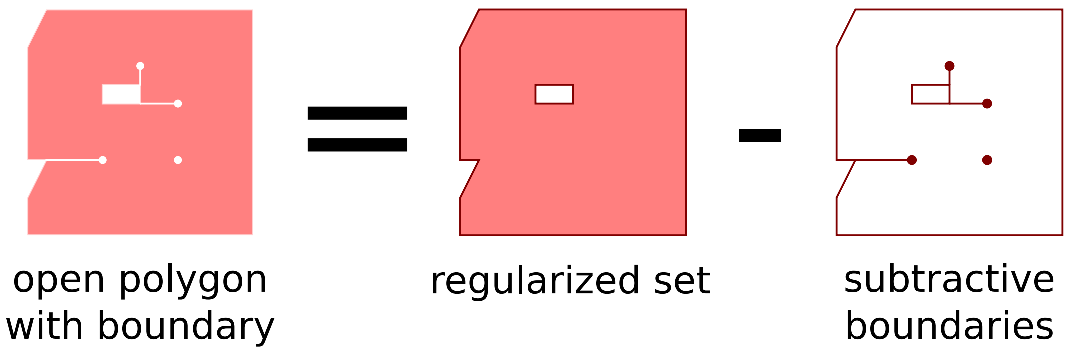


Figure 4.6: Decomposition of a no-fit polygon as a Polygon with Boundary

IFP/CFR An IFP or a CFR is a closed Polygon with Boundary (CPWB). The interior represents the set of non-touching contained positions. There is no subtractive boundary, hence the boundary is entirely given by the additive boundary, that represent touching positions inside the container. Therefore, the additive boundary always includes the natural boundary. Figure 4.7 shows the decomposition of the IFP in the previous example. To sum up, IFPs and CFRs as CPWB are characterized by:

$$\begin{cases} \partial CPWB \subseteq AB(CPWB) \\ SB(CPWB) = \emptyset \end{cases} \tag{4.3}$$

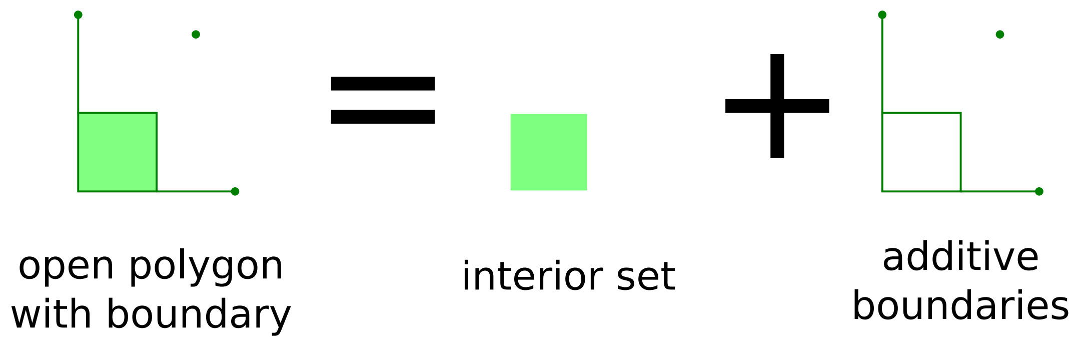


Figure 4.7: Decomposition of an inner-fit polygon as a Polygon with Boundary

Polygon with Boundary operations

Polygons with Boundary are not closed under the union, intersection and difference operations. For instance, by removing a point from an additive boundary edge we can obtain a half-open segment, which is not a legal component in our model. It is possible to ensure that the result of an operation is still a Polygon with Boundary by regularizing it, i.e. taking the closure of the interior of the additive and the subtractive boundaries. In this section, we choose another method. Since we are only interested in NFPs, IFPs and CFRs, we only consider operations on open Polygons with Boundary and closed Polygons with Boundary. Each group is closed under the union and intersection operations, because additive and subtractive boundaries are not mixed together. In addition, the complement link both sets together, as seen in equation (2.10).

We can define operations on Polygons with Boundary by providing the expression of each component, RS, SB and AB, of the result. In the formula below, R and S are two OPWB and T and U are two CPWB. For each operation, the expression of each component of the resulting Polygon with Boundary is given, omitting the subtractive boundary for CPWB and the additive boundary for OPWB, as they are respectively empty. All the formulas have been written so that the decomposition equation (4.1) and the inclusion conditions on the subtractive and additive boundaries are verified. For each formula, we provide figures with actual NFPs, IFPs and CFRs.

Union The union of two OPWB is:

$$R \cup S = \begin{cases} RS(R \cup S) = RS(R) \cup RS(S) \\ SB(R \cup S) = (SB(R) \setminus RS(S)) \cup (SB(S) \setminus RS(R)) \cup (SB(R) \cap SB(S)) \end{cases} \tag{4.4}$$

The difference operations mean that subtractive lines and points in one OPWB are covered by the regularized set of the other OPWB, except if both geometries share the same subtractive parts, in which case the last intersection $SB(R) \cap SB(S)$ is added to the resulting subtractive boundary.

This operation is used for the NFP unions of the CFR formula (2.10), and to reconstruct the NFP between two decomposed pieces, using equation (2.7). A case is illustrated on Figure 4.8, where a NFP is reconstructed from a piece decomposition.

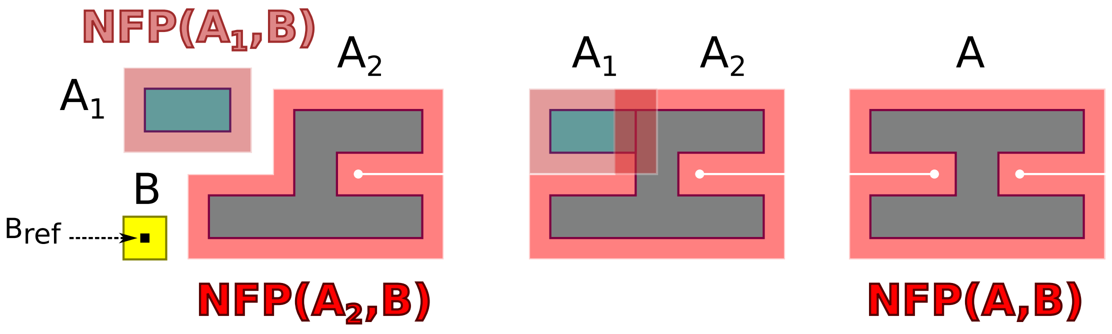


Figure 4.8: Reconstruction of $NFP(A, B)$ with OPWB union for $A = A_1 \cup A_2$

The union of two CPWB is:

$$T \cup U = \begin{cases} RS(T \cup U) = RS(T) \cup RS(U) \\ AB(T \cup U) = (AB(T) \setminus RS(U)) \cup (AB(U) \setminus RS(T)) \end{cases} \tag{4.5}$$

This time, the difference operations are only required to ensure the inclusion condition of the additive boundary. Without them, in the resulting union, the additive boundary may contain parts inside the regularized set.

Unfortunately, the application of this equation is very restricted. The set of positions in $IFP(C_1, B) \cup IFP(C_2, B)$ is the set of positions for which B is inside C_1 *or* inside C_2, and not the set of positions for which B is inside $C_1 \cup C_2$ as we would expect. See Figure 4.8 for an example. Therefore, it is not possible to merge containers with this formula, but the formula can still be used with individual containers or holes in a polygon. Instead, we must use the IFP/NFP duality (explained in paragraph 4.3.2 Complement) and compute $NFP(C_1, B) \cup NFP(C_2, B)$.

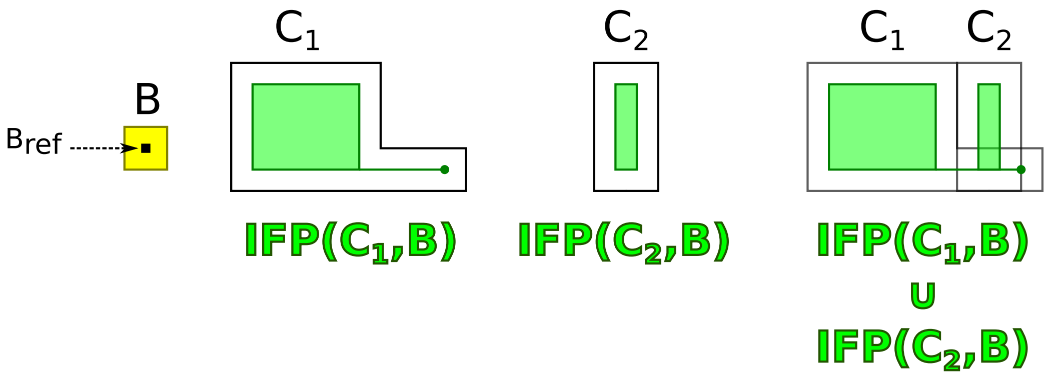


FIGURE 4.9: Union of $IFP(B, C_1)$ and $IFP(B, C_2)$ as CPWB

Intersection The intersection between two OPWB is:

$$R \cap S = \begin{cases} RS(R \cap S) = \overline{int(RS(R) \cap RS(S))} \\ SB(R \cap S) = (SB(R) \cup SB(S)) \cap RS(R \cap S) \end{cases} \tag{4.6}$$

The regularization operation is required in the first expression because the intersection of two polygons may be reduced to a line of a point, and only the boundary should contain such geometries.

The formula can be directly used to determine the no-fit polygon with the intersection of two pieces. Although this does not correspond to a physical reality, this is useful to infer information on feasible positions even when one of the pieces does not have a determined placement. For instance, on Figure 4.10, the F-shaped piece may have two different orientations, represented by A_1 and A_2, but the translation is known. We are certain that the space $A = A_1 \cap A_2$ will be occupied, so we can already use $NFP(A, B)$ to reduce the domain of B. In our approach, we do not use such inference.

The intersection between two CPWB is:

$$\begin{cases} RS(T \cap U) = \overline{int(RS(T) \cap RS(U))} \\ AB(T \cap U) = (AB(T) \cap RS(U)) \cup (AB(U) \cap RS(T)) \cup (AB(T) \cap AB(U)) \end{cases} \tag{4.7}$$

The operation is used in particular when the length of the container decreases. The new collision-free region of a piece can be deduced by computing the intersection of the old

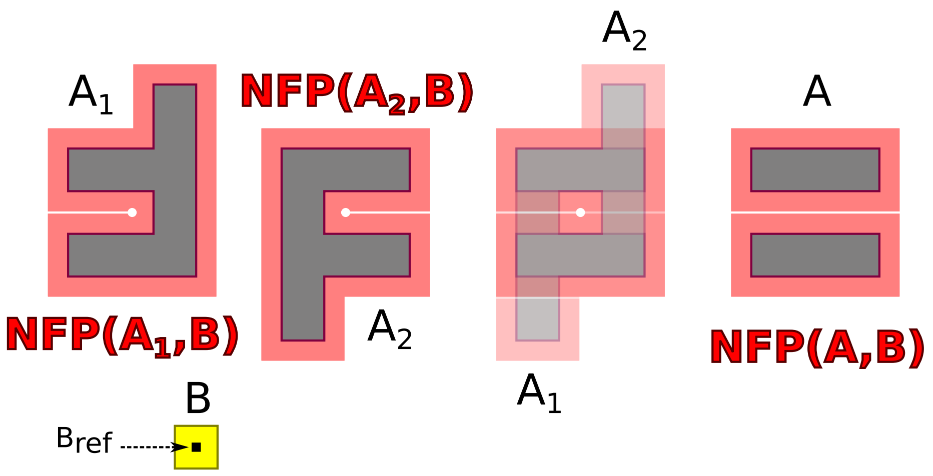


FIGURE 4.10: $NFP(A, B)$ as OPWB intersection for $A = A_1 \cap A_2$

CFR and the new IFP. Figure 4.11 illustrates this case. In our algorithm, the container never shrinks. Nevertheless, to compute the CFR for any container length we use the following trick: we first compute the CFR for a very high length. Then we reduce the container length and update the CFR for this new length. Therefore, in our approach, we use the formula each time the length of the container increases or the CFR is updated after a new piece is placed.

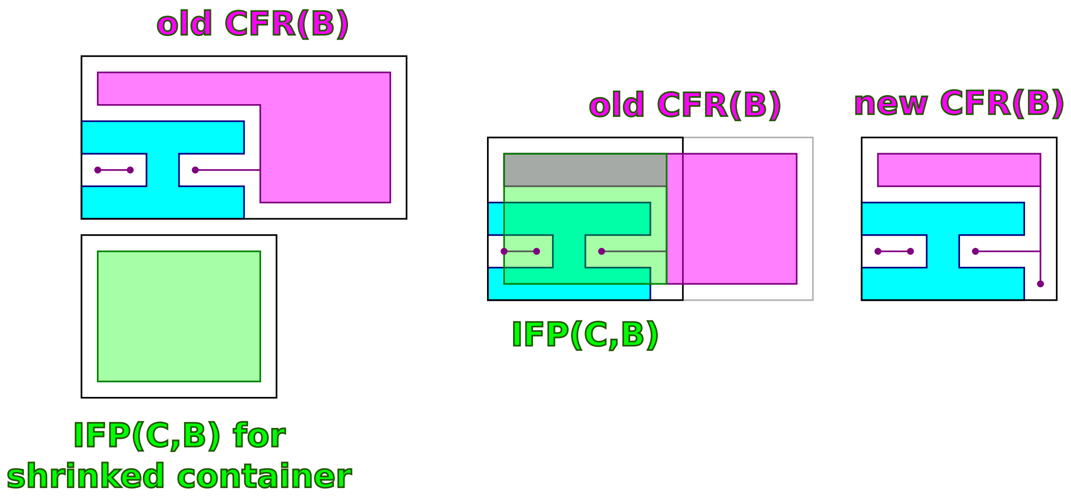


FIGURE 4.11: $CFR(B)$ updated after decrease of container length with CPWB intersection

Complement The complement of a bounded set is an unbounded set, so Polygons with Boundary are not closed under the complement operation. However, as explained previously with Nef polygons, it is useful to consider unbounded Polygons with Boundary, with unbounded regularized sets and/or some boundaries being rays or lines, in order to define the complement operation and deduce the formula of the difference operation. Therefore, the following formulas are not used in our program but they can help to understand the duality between OPWB and CPWB. Since the complement of an open set is closed and vice versa, the complement operation transforms an OPWB into a CPWB and vice versa.

The complement of an Open Polygon with Boundary is a Closed Polygon with Boundary:

$$R^{\mathsf{c}} = \begin{cases} RS(R^{\mathsf{c}}) = \overline{RS(R)^{\mathsf{c}}} \\ AB(R^{\mathsf{c}}) = SB(R) \end{cases} \tag{4.8}$$

The complement of a Closed Polygon with Boundary is an Open Polygon with Boundary:

$$T^{\mathsf{c}} = \begin{cases} RS(T^{\mathsf{c}}) = \overline{RS(T)^{\mathsf{c}}} \\ SB(T^{\mathsf{c}}) = AB(T) \end{cases} \tag{4.9}$$

The complementary operation has a meaning by itself for NFPs and IFPs. Let A and B be two pieces, and let $p \in \mathbb{R}^2 \times O(B)$ a placement of B. Then

$$\begin{aligned} p \notin \mathit{NFP}(A,B) \Leftrightarrow \mathit{int}(A) \cap t_{\mathbf{u}}(\mathit{int}(B)) = \emptyset &\Leftrightarrow t_{\mathbf{u}}(\mathit{int}(B)) \subseteq \mathit{int}(A)^{\mathsf{c}} \\ &\Leftrightarrow t_{\mathbf{u}}(\mathit{int}(B)) \subseteq \overline{A^{\mathsf{c}}} \\ &\Leftrightarrow t_{\mathbf{u}}(B) \subseteq \overline{A^{\mathsf{c}}} \\ &\Leftrightarrow p \in \mathit{IFP}(\overline{A^{\mathsf{c}}}, B) \end{aligned}$$

Therefore, there is a duality between NFPs an IFPs. The complementary of an NFP is:

$$\mathit{NFP}(A,B)^{\mathsf{c}} = \mathit{IFP}(\overline{A^{\mathsf{c}}}, B) \tag{4.10}$$

and the complementary of an IFP is:

$$\mathit{IFP}(A,B)^{\mathsf{c}} = \mathit{NFP}(\overline{A^{\mathsf{c}}}, B) \tag{4.11}$$

On Figure 4.12, an NFP is shown along with its complementary IFP.

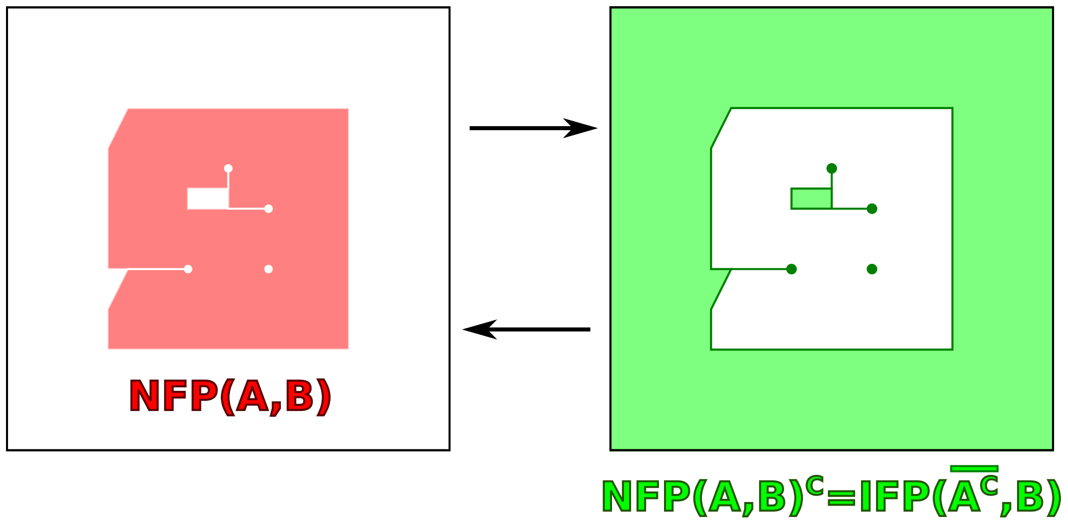


FIGURE 4.12: Polygon with Boundary duality with reciprocal complement operation

Similarly, the complementary of the collision-free region of a piece is the set of positions at which a piece overlaps with other pieces or protrude from the container.

Difference The difference $X \setminus Y$ is the intersection of X and Y^c. Since we have defined the intersection operation between two Polygons with Boundary of the same type and the complement operation switches between OPWB and CPWB, we define the difference operation between an OPWB and a CPWB. The final formulas of the difference operations do not contain complementary operations, and are therefore applicable to the set of bounded Polygons with Boundary, that we actually implemented in our program.

Subtracting a CPWB T from an OPWB R gives an OPWB:

$$\begin{cases} RS(R \setminus T) = \overline{RS(R) \setminus RS(T)} \\ SB(R \setminus T) = (SB(R) \cup AB(T)) \cap RS(R \setminus T) \end{cases} \tag{4.12}$$

The operation can be used to compute the NFP of a piece with a hole. For instance, let us consider a piece Q from which we subtract a hole H (without the boundary) to obtain a new piece A: $A = Q \setminus int(H)$. Then the set of no-fit positions of another piece B with A is the set of no-fit positions of B with Q, excluding the fit positions in the hole H:

$$NFP(A, B) = NFP(Q \setminus int(H), B) = NFP(Q, B) \setminus IFP(H, B) \tag{4.13}$$

An illustration is given on Figure 4.13.

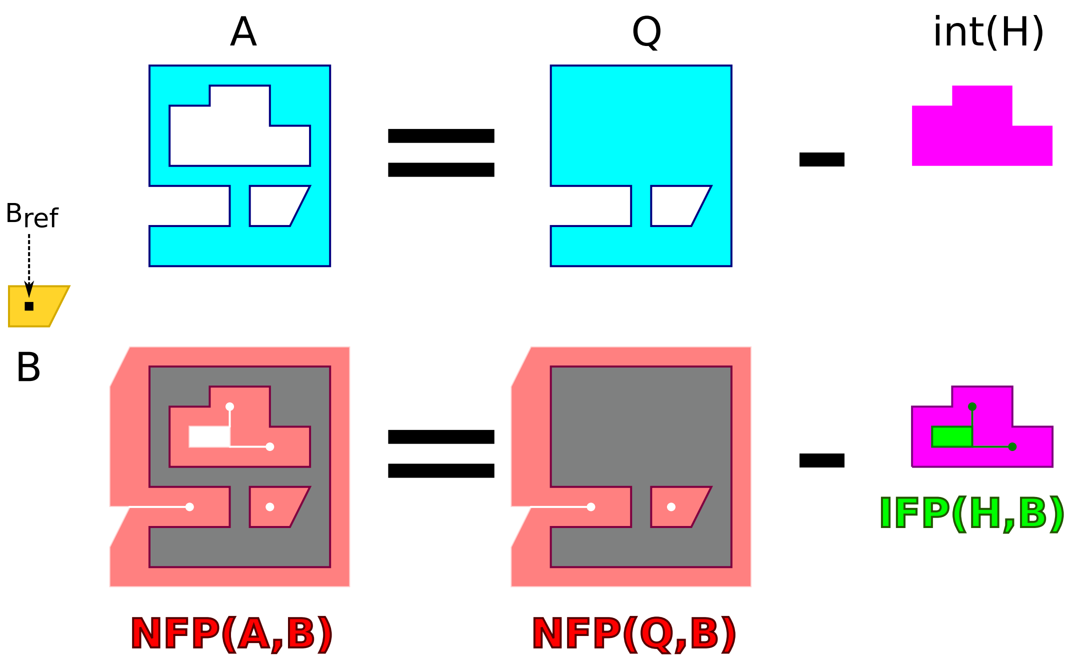


FIGURE 4.13: NFP with a piece minus a hole, using the OPWB - CPWB difference

Subtracting an OPWB R from a CPWB T gives a CPWB:

$$\begin{cases} RS(T \setminus R) = \overline{RS(T) \setminus RS(R)} \\ AB(T \setminus R) = (AB(T) \setminus RS(R)) \cup (SB(T) \cap RS(R)) \cup (AB(T) \cap SB(R)) \end{cases} \tag{4.14}$$

An illustration is given on Figure 4.14. This is the most used operation because it a CFR is a CPWB and the incremental CFR formula 3.1 requires to subtract an NFP from a CFR.

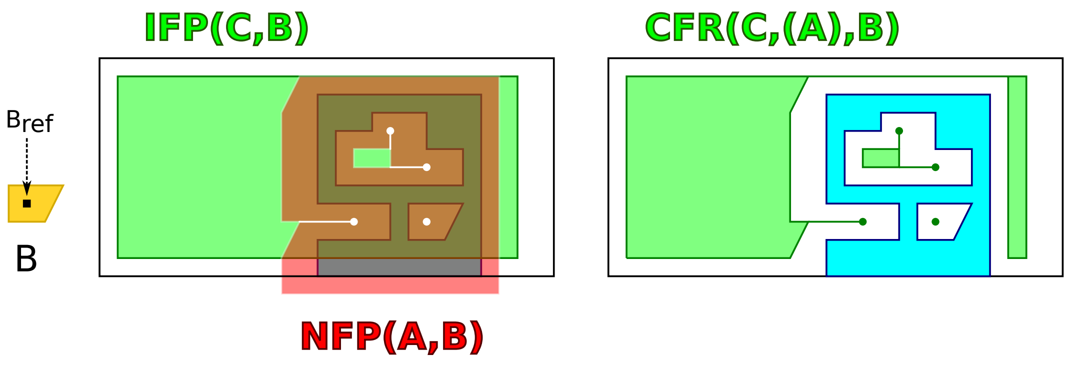


FIGURE 4.14: Subtracting an NFP (OPWB) from an IFP (CPWB)

Finally, we illustrate the computation of a collision-free region when multiple pieces are present in Figure 4.15, using equation 2.10.

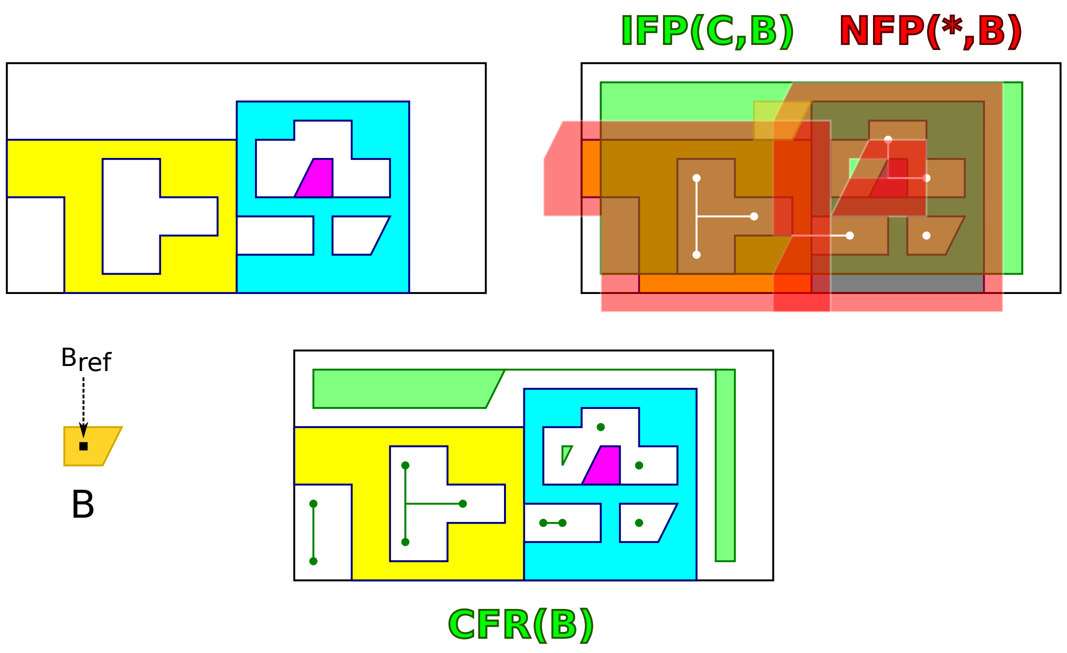


Figure 4.15: Computing CFR as PWB from IFP and NFP with 3 pieces

4.4 Incremental cache

Geometrical computations are the most expensive in our program. In order to minimize them, we combined two common optimization practices in software development:

- using a cache, to store geometries once computed
- lazy evaluation, to only compute geometries when needed

The no-fit polygons and the collision-free regions are the most expensive to compute, but inner-fit polygons and free regions can be cached as well. Since placement domains are equal to the collision-free regions, they can also be accessed from the cache.

Lazy evaluation is useful to do geometrical computations only when heuristics and filtering require data to be up-to-date. For instance, the LCV heuristic presented in section 3.4.2 requires up-to-date placement domains in the cache of a node as soon as it is pushed to the fringe, whereas other heuristics need to know them later, when the node is popped. Thanks to lazy evaluation, LCV can access up-to-date information, but when heuristics are used, unneeded computations are delayed. The ratio between the number of popped and pushed nodes at one stage of the search, before backtracking, is equal to the branching factor of the search, which is in the order of *average number of placements* *picked per piece* $*$ *number of pieces*, around 100 for the Tangram problem. Lazy evaluation allow heuristics other than LCV to reduce the number of geometrical computations of that ratio.

4.5 Source code

The source code is expected to be pushed to a Git repository hosted by Github by the end of February 2016. The Python code for the solver will be available at `https://github.com/hsandt/hipps` (Hygracose-based Irregular Packing Problem Solver) and the Python code for the underlying framework will be available at `https://github.com/hsandt/hygracose` (Hybrid Graph-Constrained Search framework). If for some reasons, the source code was not available at the expected date, visitors will be notified by a message on the README page.

The first repository already contains the XML file for the Tangram problem, the original NestingXML and a modified XML Schema Definition that supports non-degenerated NFPs and IFPs. Appendix A.2 provides more information on this new format.

Chapter 5

Results

5.1 Benchmarks overview

We considered 9 benchmarks created by other researchers and available on the website of the ESICUP at Data Sets > 2D > Irregular [2]. Those benchmarks are among the most used by other researchers, and results of other approaches can be found, with final container length / usage (percentage of occupied space) and computation time. We also corrected a few mistakes that we found in the XML files distributed on the ESICUP website. A partial erratum can be found in Appendix: A Benchmarks.

All the benchmarks from the ESICUP have been solved as Open Dimension problems, but for some we have fixed the length of the container after the ODP search to do a more restrictive Single Knapsack search, in order to discover solutions with lower lengths. This process can also be automated as a two-level search, for instance using dichotomy on the container length to find the minimum length for which the SKP can be solved within a given time limit.

In addition, in order to compare different settings with our approach efficiently, we ran a number of searches on a much smaller problem, the Tangram problem. The concept of the game emerged in China during the Tang dynasty (618 - 907). Although the original Tangram consists in creating various shapes with a set 7 pieces, we only worked with the square container problem, since our research is focused on rectangular containers. This toy problem is not available on the ESICUP website so we generated our own XML file to describe it. The file is available as `tangram.xml` on our GitHub repository at `https://github.com/hsandt/hipps`, where the source code of the program is also expected to be pushed.

5.2 Numerical results

5.2.1 Precomputation

During the precomputation phase, we compute the inner-fit polygons of each piece inside the container, and the no-fit polygons between each pair of pieces. IFPs with rectangular containers are cheap to compute, and must be computed repeatedly in Open Dimension Problems because the container length changes dynamically during the search. For this reason, the precomputation of IFPs does not noticeably improve efficiency, but could reveal useful in Single Knapsack problems with complex containers. For instance, in actual Tangram problems, a polygonal shape, our container, must be reproduced by placing Tangram pieces together.

Even if IFPs are computed, NFP computations amount to 99% of the precomputation time. To compute the NFPs we used the Cunninghame-Green algorithm [9] combined with a simple triangulation to have a convex decomposition of the pieces, as described in section 2.2 No-fit polygon. Table 5.1 shows the precomputation time for our benchmarks, along with information of the problem pieces. The number of piece types, the total number of pieces and the average number of vertices per piece type are written in order in the first columns. Since NFPs are more complex when the shapes of the pieces are more complex, these measurements are strongly related to the computation time of the NFPs.

All precomputation times are negligible compared to the search time shown in the next sections, except for Jakobs1 and Jakobs2. The reason is probably that the piece shapes, although not very complex, are not suited for triangulation. For instance, triangulation is very ineffective with "+"-shaped pieces, producing 9 triangle parts instead of 5 convex parts with an average convex decomposition, and 3 convex parts with an optimal decomposition.

We expect a convex decomposition to be faster than a triangulation because it creates fewer convex parts, hence fewer unions are required to reconstruct an NFP (although

TABLE 5.1: Precomputation time per benchmark

Data set	Piece types	Total	Vertices	Orientations (°)	Time (s)
Tangram	7	7	3.3	Any step of 45°	1.5
Albano	8	24	7.25	0, 180	13.5
Dagli	10	30	6.30	0, 180	6.6
Dighe1	16	16	3.87	0	0.5
Dighe2	10	10	4.70	0	0.4
Fu	12	12	3.58	0, 90, 180, 270	3.7
Jakobs1	25	25	5.60	0, 90, 180, 270	261.1
Jakobs2	25	25	5.36	0, 90, 180, 270	232.2
Shapes2	7	28	6.29	0, 180	4.9
Shirts	8	99	6.63	0, 180	8.5

convex decomposition itself is more complex, as explained in the No-fit polygon section). The specialized method of orbital sliding provided by Burke et al. [14] is another cheaper alternative.

It is also possible to store the NFPs after computing them the first time, so that we do not have to recompute them for any future searches on the same data set. The structure of the NFPs can be stored in the XML file of the problem, in the `<nfps>` element, and most benchmarks provide them. If NFPs are entirely generated by parsing NFP data in the XML file, precomputation time is reduced to less than 1 second for simple data sets such as Dighe1 and Dighe2, and a few seconds for more complex data sets.

Note that the current XML Schema cannot handle degenerated parts in NFPs, so we designed a new Schema to handle them. The parts changed in the new XSD file can found in Appendix: A Benchmarks, and the file itself is available at `https://github.com/hsandt/hipps` as `nesting_degenerated.xsd`.

5.2.2 Tangram problem

Single Knapsack Tangram

We applied our approach with various settings on the Tangram problem as a Single Knapsack Problem. The abbreviations are explained in Table 5.2 and the results are shown in Table 5.3.

All the heuristics are ordered by depth → variable → value. The depth-first search heuristic is always used, and the basic Forward Checking and empty domain layout dropping are used in all settings. Some heuristics use a variable-based heuristic without a value-based heuristic and vice-versa.

Table 5.2: Table abbreviations

Vertex	Vertex picker
Min-BB	Minimum layout bounding box picker
Var.	Variable-based heuristics
Big	Biggest piece first, in area
MRV	Minimum Remaining Value
Value	Value-based heuristics
FRA	Filter by free region / dead region analysis
Iter.	Number of iterations (nodes popped)
Exp.	Number of nodes expanded (popped and not dropped)
Fringe	Maximum fringe size (number of nodes stored)
Layout	Layout completion (full if all pieces are placed)

If we compare strategies with different variable-based heuristics, we can see that MRV, that was designed as a generalization of Biggest piece first, is a much more effective heuristics than the latter. All other settings equal, the number of iterations and expanded nodes as well as the computation time are roughly divided by 3 with MRV compared to Biggest piece first.

If we compare value-based heuristics, other heuristics equal, we observe that:

- The choice of the picker radically changes the performance, with the minimum layout bounding box picker offering an edge over the classic vertex picker, and the dominant point picker providing the best results whichever the heuristics.
- Cheap heuristics such as Min-CH (minimum convex hull) and Exact (exact fit and slide) reduce the computation time and the number of iterations / expanded nodes when there are no variable heuristics. When MRV is used, they do not improve the search and when DP (dominant point picker) is used, they worsen the search.

TABLE 5.3: Results for fixed size Tangram

Results of our hybrid graph-constrained search with various strategies for the fixed size Tangram problem

Strategy settings				Result				
Picker	Var.	Value	Filter	Iter.	Exp.	Time (s)	Fringe	Layout
Vertex		LCV	FRA	5000	417	81.3	425	6/7
Vertex				3400	306	39.6	323	full
Vertex	Big			3175	283	36.2	321	full
Vertex			FRA	2076	155	28.3	323	full
Vertex	Big		FRA	1759	134	24.6	321	full
Vertex		Min-CH	FRA	1710	138	22.1	348	full
Min-BB	MRV		FRA	792	56	12.2	273	full
Vertex		Exact		722	57	9.3	342	full
Vertex	MRV			716	86	8.9	316	full
Vertex	MRV	∅ / Exact	FRA	543	47	7.2	316	full
Vertex	MRV	LCV	FRA	401	36	20.0	317	full
DP		Min-CH	FRA	250	32	6.2	263	full
DP		Exact	FRA	149	20	4.0	272	full
DP		Exact		179	24	3.8	272	full
DP	MRV		FRA	127	18	3.1	263	full
DP			FRA	107	17	3.3	256	full
DP	MRV	LCV	FRA	93	15	14.7	261	full

This is probably due to the restricted nature of the SKP, as Min-CH is performing better on the ESICUP benchmarks.

- LCV is an expensive heuristic as expected. Without MRV or DP to limit the number of nodes expanded, domain computations on each expanded node are too expensive and the search does not complete in time (1st row). Otherwise it still multiplies the computation time by a factor 2 to 5. However, it reduces the number of iterations and expanded nodes by 10% to 40%. In other words, the search is more informed and makes better decisions but takes more time to run. For a casual application, a computation time under 4 seconds is expected, so only the best results with DP would work and LCV would be rejected.

- Filtering reduces the number of expanded nodes as expected, and since the successors of the dropped nodes are ignored as well, the number of iterations also decreases. Filtering does not help the search to explore better layouts first, but reduces the computation time by around 15%.
- The fringe size, that gives an indication of the memory used, is stable and decreases gradually with better heuristics, picker and filtering. The picker is a critical factor on the fringe size since it directly affects the branching factor of the graph search.

Open Dimension Tangram

After solving the Tangram as an SKP, we ran searches on the Tangram problem as an Open Dimension Problem. The search runs unaware of the optimal length, 8. Results are shown in Table 5.4.

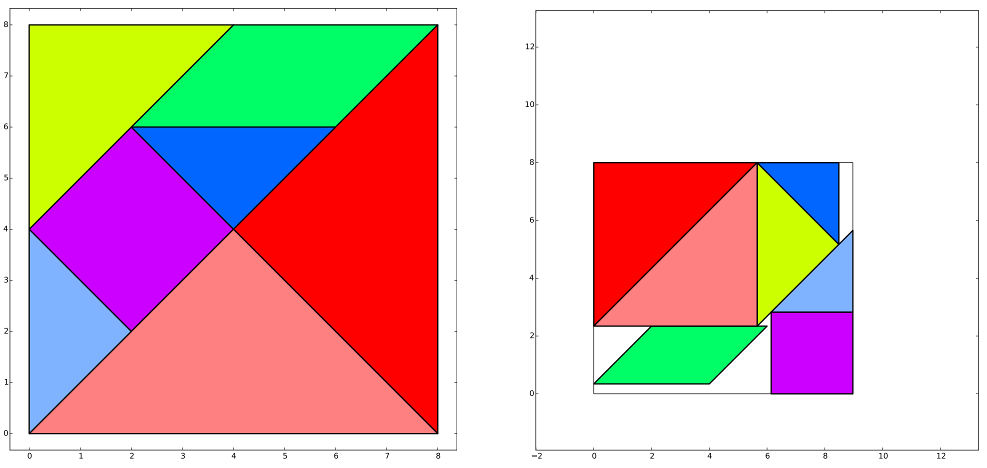
The last column in the table shows the best length found at the end of the search. Our algorithm extends the length of the container unnecessarily and ends with a sub-optimal packing with a length of 8.97. However, the sequential placement of the pieces itself is meaningful, with the big triangles compacted into a square first, see Figure 5.1b.

Table 5.4: Results for variable size Tangram

Results of our hybrid graph-constrained approach with various strategies for the variable size Tangram problem

Strategy settings				Result					
Picker	Var.	Value	Filter	Iter.	Exp.	First	Time (s)	Fringe	Length
Vertex	MRV	Max-BBO		5000	12+	12.1	151	540	10.83
Vertex	MRV	Min-DA	FRA	5000	609	116	315	555	10.0
Vertex	MRV	Max-BBO		5000	14+	3.8	284	356	9.66
Vertex		LCV	FRA	4516+	241+	271	294	336	8.97
DP		LCV	FRA	47+	13+	15	69	237	8.97

Since search in COP continues after a solution is found, we recorded both the time where the best solution was first found, *First* in the table, and the total time of the search, both in seconds. For other measurements, we added a "+" sign to indicate that the value written was reached when the best solution was found, but increased later. The (Time/First time) ratio should give an idea of the difference between values for the first solution and values at the end of the search.



(A) Tangram as an SKP (B) Tangram as an ODP

FIGURE 5.1: Results for the Tangram problem

Again, the best results are obtained with the LCV heuristic (length of 8.97), but the DP picker is needed to limit the expansion of the graph and keep the computation time around 1 minute. Min-DA is also expensive as it requires early domain computation, and it provides medium results with a length of 10.

5.2.3 ESICUP Benchmarks

Results

We recorded the best results that we could find with our approach, trying different strategies. More information on the strategies used can be found under the visual representation of the results.

This time we score the resulting layouts by usage, the ratio of the total area of the pieces on the area of the container with the best length found. All usages are written in percentage. Usage is inversely proportional to the length of the container.

Computations have been done under Linux Ubuntu on a computer with a processor Intel Core i7, 2.5GHz, with single threading. Geometrical computations were handled by Shapely and GEOS. Results are shown in Table 5.5. Final layouts are shown on Figure 5.2. Not all the layouts shown correspond to the best solution we have found.

The other strategies mentioned are:

Table 5.5: Results for ESICUP benchmarks

Results for various approaches on benchmarks from the ESICUP

Data set	Proposed		Beam search		Cuckoo	ILSQN	Exact
	Usage	Time (s)	Usage	Time (s)	Usage	Usage	Usage
Albano	74.77	946	87.88	5460	89.58	87.14	**89.21**
Dagli	72.25	1867*	87.97	17331	89.51	85.80	**88.36**
Dighe1	68.05	88	**100.00**	1.4	**100.00**	90.49	**100.00**
Dighe2	100.00	166	**100.00**	0.3	**100.00**	84.21	**100.00**
Fu	74.03	4.1 - 334	90.28	1192	92.41	87.57	**91.96**
Jakobs1	66.22	43.4	85.96	2193	**89.10**	84.78	89.09
Jakobs2	66.37	497	80.40	75	87.73	80.50	**84.83**
Shapes2	66.26	2657	81.29	5603	84.84	81.72	83.30
Shirts	88.51	515	89.69	6217	88.96	88.12	87.59

- Beam search: devised by Bennell and Song [3], sequential search with low memory usage
- Cuckoo: devised by Elkeran [6], local search with container length reduction and a Cuckoo search on the position of the polygons
- ILSQN: devised by Imamichi et al. [5], an Iterated Local Search with overlap evasion via penetration depth estimation
- Exact: local search combining exact fit and exact slide heuristic and simulated annealing for the evolution of the container length, devised by Sato et al. [4]

Cuckoo, ILSQN and Exact fit strategies all spend 1200 seconds on Albano, Dagli, Shapes2 and Shirts, and 600 seconds on the other data sets.

Remark 5.1. Beam Search was executed with a C++ program but on a Pentium D processor. It would probably have a significant improvement in speed if it was executed on a modern machine. However, Cuckoo, ILSQN and Exact fit searches are local search that run for a given time and then stop, returning the best solution found. With better processing power, those searches would still run the same time, but probably obtain better solutions.

Remark 5.2. For Dagli, after the search on the Open Dimension Problem that lasted around 1800 seconds, we ran a new search as a Single Knapsack Problem, setting the container length to a lower value. The second run took 65s and found a slightly better packing than the ODP search, hence the total time 1867s. Note that the process of applying a strict SKP search for different lengths can be automated, but requires a huge amount of time in general.

Remark 5.3. For Fu, we wrote the time the best solution was found before the total search time because the difference was huge. The dominant point picker was used for this search.

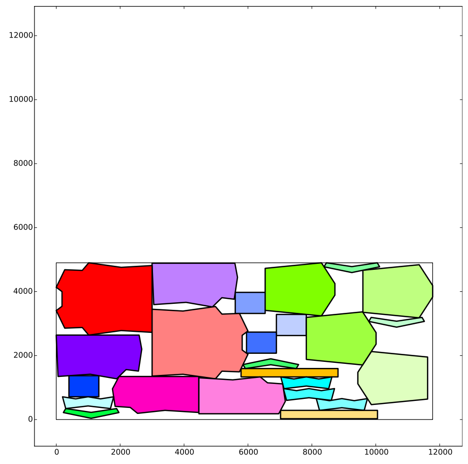
Remark 5.4. Shapes2 is available under the name Blaz1 in the ESICUP benchmark XML files.

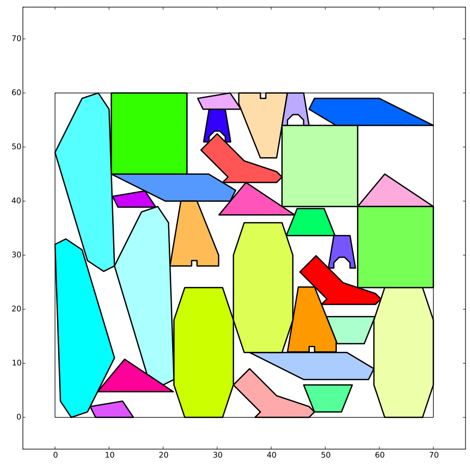
Observations

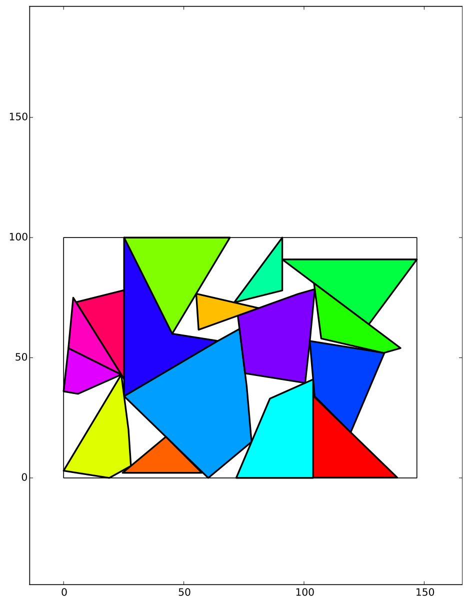
Our results do not compete with other methods in terms of length/usage, but most searches are executed under 1000s. Although our search is not local, the way we tackle Open Dimension problems with container length extension allow us to obtain a solution with a high length very fast, then to explore better solutions gradually over time. This is in contrast with the beam search, another sequential search, which never backtracks but put more emphasis on the evaluation at each step. We consider that our algorithm would be appropriate for casual software applications, for which the user wants to find a relatively good layout in a limited amount of time, without seeking an industrial quality.

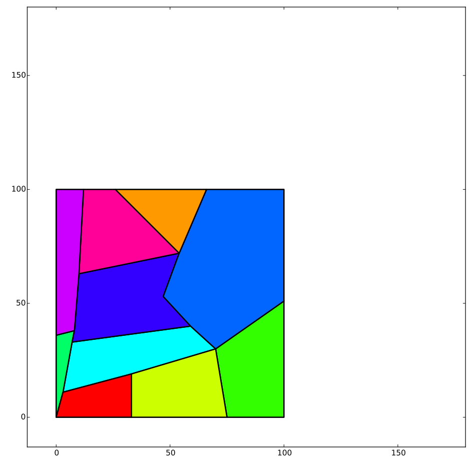
The proposed method is effective in average for dense problems where pieces have different scales, such as Shirts, and problems with few simple polygons, such as Dighe2, which is a dissection puzzle slightly more complex than the Tangram. However, the resulting layouts have a very low usage on data sets with many square-like shapes, such as Jakobs1 and Jakobs2.

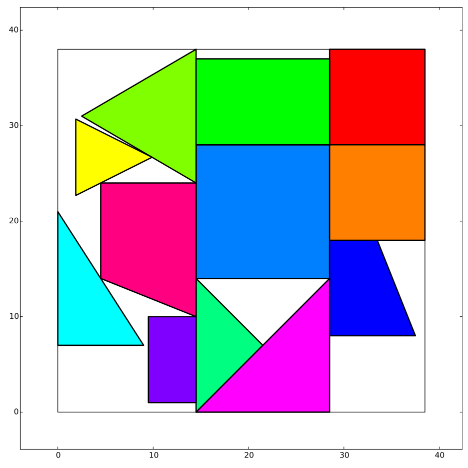
Actual results depend on the heuristics chosen, but generally speaking, a vertex picker combined with an MRV heuristic is stable, and completing it with a value-heuristic such as maximum bounding box overlap limits the amount of waste for data sets of average difficulty. The efficiency of evaluation-based pickers such as *minimum bounding box* and *dominant point* strongly depends on the data set. Min-BB picker found the best layout for Shirts, but failed to find good solutions with Dagli. The dominant point picker found its best solution very fast with Fu, although the final usage is low compared to other

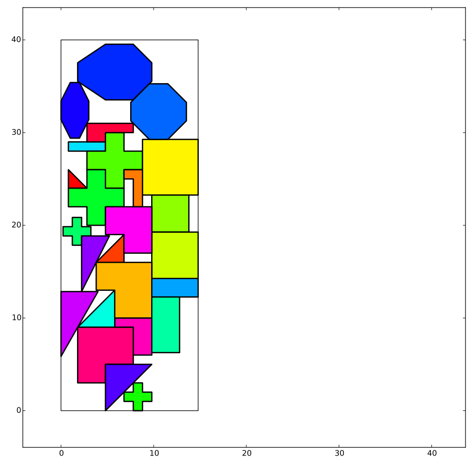

(A) Albano: MRV, Max-BBO (usage 73.90%)

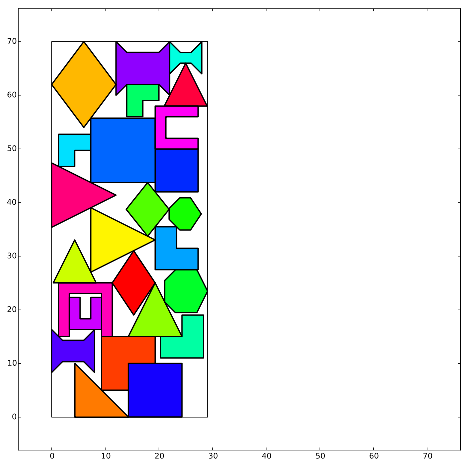

(B) Dagli: MRV, Max-BBO in SS for length 70 (usage 72.25%)

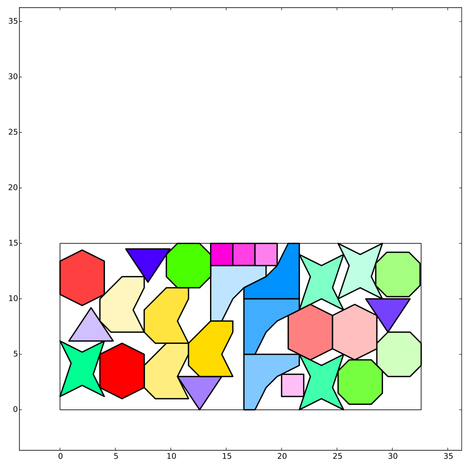

(C) Dighe1: 2 bounded fringes (500), dominant point, MRV, LCV (usage 68.05%)


(D) Dighe2: MRV, mDA (usage: 100%)


(E) Fu: dominant point, MRV, Max-BBO (usage: 74.03%)


(F) Jakobs1: dominant point, Max-BBO (usage 66.22%)


(A) Jakobs2: MRV, Max-BBO (usage 66.37%)


(B) Shapes2 (Blaz1): MRV, mDA, Min-CH (usage 66.26%)

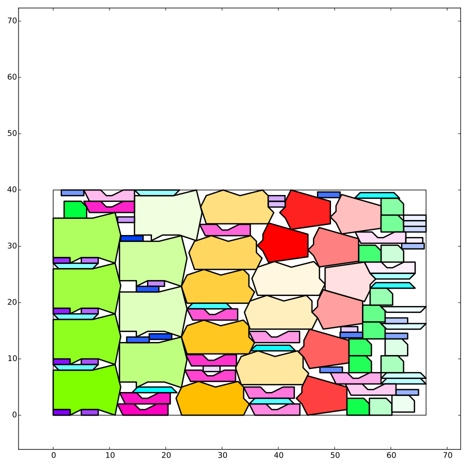

(C) Shirts: bounded fringe (500), Min-BB picker, Max-BBO (usage 81.51%)

FIGURE 5.2: Results with the benchmarks of the ESICUP

researchers' approaches. A possible reason is that because shapes in Dagli are round-like, strategies that try to maximize penetration and fitting between piece shapes do not work well and end picking an arbitrary placement instead, missing many placements that could lead to a layout of equivalent or better quality.

Finally, regular picker, multiple and randomized fringe have been used in order to increase the diversity of the layouts. In most searches, results were either not as good as with more classical searches, or equivalent but required more computation time. For Dighe1, however, where we are far from the optimal usage of 100%, a multi-search on 3 fringes returned better results on the 2 extra fringes. The best result was finally obtained with a double bounded fringe, the size limit on the fringe contributing to reducing both the memory footprint and the computation time.

5.2.4 Complexity

Computation time being implementation-dependent and machine-dependent, we provided the number of iterations, expanded nodes and nodes stored in the fringe to facilitate comparisons with other algorithms. However, each iteration contains a number of operations that depend on the exact implementation of the algorithm. The actual time of an operation depends, again, on its implementation in the library used and on the machine on which it is run. However, by providing the number of each type of operations, in particular geometrical operations, we hope to help other researchers to compare their results with ours in a way that is implementation and machine-independent. We also hope to release the source code of our program, as explained in section 4.5 Source code, so that researchers can run it on their computers.

Algorithm analysis

In this section, we analyze the flow of the main iteration cycle and theoretical complexity of each process block.

From the algorithms of the CSP and COP searches, Algorithm 1 and Algorithm 2, presented in 3 Hybrid graph-constrained search, we built the flowchart on Figure 5.3.

There are 4 exit points possible during an iteration loop. We call *exit point* an instruction at which the program may drop the node and restart the loop from the beginning (`continue` instruction) or jump to the point after the end of the loop (`break` instruction, only after finding a solution in an SKP or reaching the maximum number of iterations). Depending on the moment an iteration exits, the number operations of an iteration will vary. The exit conditions are:

1. Model found: happens only once for an SKP but may happen many times in an ODP. In addition, for ODPs, if a node has a current cost higher than the best solution found, it is immediately dropped even if its layout is still incomplete. On our benchmarks, in average, this happens in 80% of the iterations.

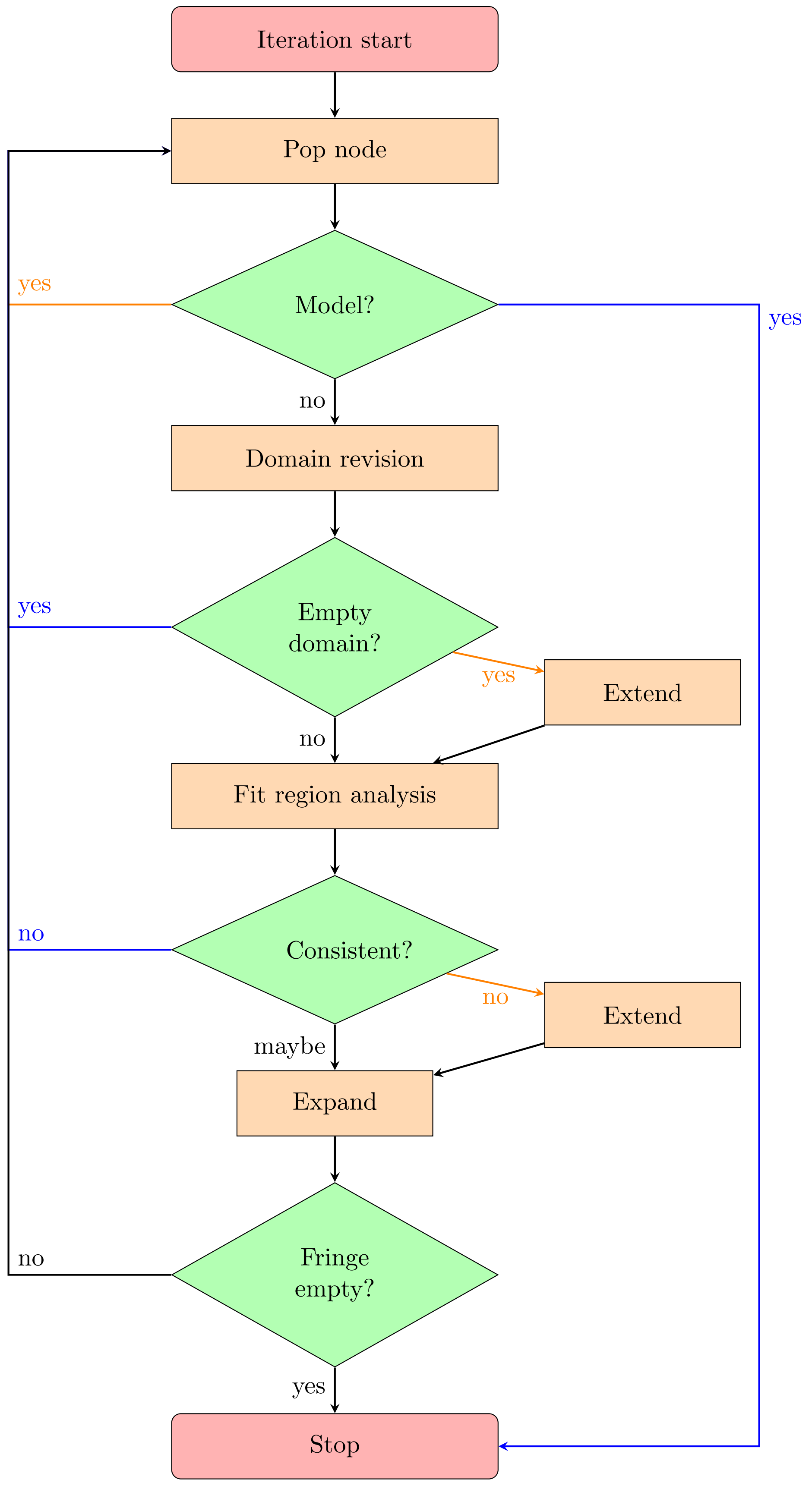


FIGURE 5.3: Flowchart of an iteration for a CSP (black and blue arrows) and a COP (black and orange arrows)

2. Empty domain: for an SKP, the most common cause of early `continue` because, when the layout is about to be completed, there is little space left for the remaining pieces. 85% of the nodes are dropped due to an empty domain in searches on the Tangram problem as an SKP. For an ODP, the iteration continues but requires container extension, which should be taken into account in the operations count.
3. Non-consistency: for an SKP, this happens for 5% of the iterations. For an ODP, the iteration also continues with container extension.
4. Empty fringe: in practice, it is checked at the beginning of the loop. The fringe can only be empty once, at which point the search ends.

In the following part, we study the complexity of each process block in the flowchart. For a given iteration step, we note Z the number of remaining pieces and Θ the average number of orientations per piece.

1. Pop node: we use a Python heap [29] for which popping has $O(n \log n)$ comparisons.
2. Domain revision: Unless LCV or Min-DA is used, domain revision occurs after a node is popped to prepare the coming Empty Domain test. During domain revision, we reduce the domains of each remaining piece, for each allowed orientation, following the iterative CFR formula 3.1. This makes $Z\Theta$ times the following operations:
   - 1 PWB translation, because in the term $NFP(P_{k+1}, P)$ of formula 3.1, P_{k+1} has already been placed so we must use equation 2.3 to obtain the NFP with a translated piece
   - 1 PWB difference, composed of 2 polygonal set differences, 2 intersections, 2 unions and 1 closure (equation 4.14)
   - 1 CPWB intersection to adapt the CFR to the current container length, as explained after equation 4.7 (Open Dimension only). It requires 4 polygonal intersections, 2 unions, 1 regularization.
3. Extend 1 (ODP only): 1 CPWB intersection to adapt the CFR to the new container length

4. Fit region analysis: Algorithm 4 in section 3.5.2 Layout dropping shows that we need, in the worst case:
   - $Z\Theta$ PWB translations
   - $Z\Theta r$ polygonal intersections for r the number of free regions
5. Consistency test: complexity depends on the filter used.
   - For the dead region analysis: we must compute the sum of the area of all the living regions and all the remaining pieces. Assuming the area each piece is precomputed, we need to compute at most r polygonal areas. Other operations in the analysis are cheap in comparison.
   - For the area knapsack analysis: the complexity comes from the 1D Knapsack subproblem, also solved with a constrained graph search. The number of nodes to explore depends on the situation. There at most $(r+1)^Z$ partial assignments possible, but in average we can expect $(r+1)^Z/r$ assignments to be valid. In addition, all operations are additions, subtractions or comparisons, hence each iteration is cheap.
6. Extend 2 (ODP only): 1 CPWB intersection to adapt the CFR to the new container length
7. Expand: $Z\Theta d$ node expansions, where d is the average number of vertices per translation domain. In one expansion, a node is cloned and a new assignment is added, but computations are minimized by using shallow clones for geometrical data. If a heuristic that requires to know placement domains early such as LCV is used, domain revision (step 2) is applied for each expanded node, hence $Z\Theta d$ times. In this case, domain revision will not be applied again on the next step 2.

In our SKP searches, 10% of the iterations are completed until node expansion, and in our ODP searches, in average, 20% of the iterations are complete. Assuming their is no container extension and we use a fit region analysis but no advanced heuristic such as LCV, the total operations are: $O(n \log n)$ comparisons, $2Z\Theta$ translations, $2Z\Theta$ differences, $Z\Theta(r+2)+4$ intersections, $2Z\Theta+2$ unions, $Z\Theta+1$ closures, 1 interior. Differences, intersections and unions are the most expensive, so we can consider a complexity of $O(Z\Theta)$ polygonal set operations, with a factor in the order of the number of free regions for the current layout. The factor would be 6 without the free region analysis, but can go up to 18 with two container extensions. If LCV or the Minimum dead area heuristic is used, domain revision occurs during step 7 and the complexity becomes $O(Z^2\Theta^2 d)$ because it is multiplied by the branching factor $Z\Theta d$, which is in the order of 100 for the Tangram problem, 1000 for the Shirts problem.

The complexity of each operation depends on the number of vertices of the pieces, NFPs, IFPs and CFRs, which is not easy to predict.

Operations count

To complete the analysis of the theoretical complexity, we provide the actual count of polygonal operations for the searches we have done on the different benchmarks. Table 5.6 shows the number of iterations and the number of expanded nodes in the second column, and the average number of operations per node expanded in the other columns. The most common PWB operation, $IFP \setminus NFP$, is shown in the third column. Following columns show the count for elementary polygonal operations (intersection, union, difference and translation), either applied as part of a PWB operation or elsewhere. The search parameters are similar to those that provided the best results in sections 5.2.2 Tangram problem and 5.2.3 Results, although they slightly differ for some benchmarks. In addition, we stopped the search shortly after finding the best solution this time. The NFP precomputation phase is included in the count, but should be negligible compared to runtime operations.

We also provide the average time of a single operation in milliseconds for each benchmark in Table 5.7. The computation times are machine-dependent but allow us to compare the difficulty of each benchmark and the time of the different operations. The total number of vertices among all the pieces is also displayed, to give an idea on the complexity of each data set.

Note that the intersection, union and difference operations require approximately the same amount of time with the Shapely package, whereas translations take more time. We expected translations to be cheaper, and we suppose that the slow down is due to the implementation of the translation operation. Indeed, the translation of a collection

TABLE 5.6: Average number of operations per node expanded for each benchmark

Data set	Iter./Exp.	IFP \ NFP	Inter	Union	Diff	Transla
Tangram (SKP)	127/18	101	800	529	233	273
Tangram (ODP)	47/13	613	3922	2585	1251	1317
Albano	1000/160	129	2323	976	322	720
Dagli	31/30	586	7130	3731	1307	1791
Dighe1	314/97	37	366	204	87	148
Dighe2	5026/1342	13	280	115	37	112
Fu	13/12	22	438	253	90	322
Jakobs1	200/33	92	2918	4078	2432	1264
Jakobs2	1500/224	53	1471	869	411	614
Shapes2	30/28	3491	40494	21076	7168	10561
Shirts	100/99	98	1250	715	256	1675

TABLE 5.7: Average time per operation for each benchmark, in milliseconds

Data set	Total vertices	IFP \ NFP	Inter/Union/Diff	Translations
Tangram (SKP)	23	0.67	0.08	0.25
Tangram (ODP)	23	1.27	0.13	0.24
Albano	164	2.83	0.34	0.97
Dagli	186	1.82	0.24	0.78
Dighe1	62	1.48	0.21	0.44
Dighe2	47	1.41	0.16	0.40
Fu	43	1.44	0.16	0.35
Jakobs1	150	0.47	0.24	0.47
Jakobs2	134	3.19	0.30	0.61
Shapes2	176	2.11	0.28	0.74
Shirts	599	5.20	0.74	0.43

of objects is written as a pure Python recursion in Shapely, whereas other operations delegate calls to the dynamic library GEOS.

Chapter 6

Conclusion

6.1 Conclusion

We built a generic framework to apply methods specific to constrained problems inside a classical graph search. To use this framework for the two-dimensional irregular packing problem, we used value pickers to discretize the problem and we designed heuristics and filtering techniques based on the analysis of placement domains, introducing the concept of free and fit regions.

Picker We introduced a few placement picking strategies. Pickers based on domain shape resulted in average but stable results, whereas evaluation-based pickers were very effective on a few benchmarks but were too restrictive on most of them. Nevertheless, placement pickers allowed us to reduce the memory footprint and make searches faster.

Heuristics Our search heuristics are decomposed in three steps: depth-first search, heuristics that depend on the variable to assign and heuristics that depend on the value to assign. The variable-based heuristic Minimum Remaining Values proved more effective than the classic approach to place bigger pieces first, while remaining cheap to compute. Most of our value-based heuristics work well when piece shapes fit well together, but fail in problems containing many round or square pieces.

Filter In Single Knapsack instances of the packing problem, the filtering techniques we presented considerably reduced the number of nodes expanded, compared to basic Forward Checking. In Open Dimension instances, filtering takes the role of computing how much length extension is required to continue the search when a layout is not consistent, and allows us to safely search as if the container length was fixed the rest of the time. However, since filtering reduces memory usage and is essentially a tool that optimizes performance. Nevertheless, filtering can indirectly help to find better solutions, since the freed computation time and memory can be allocated to the exploration of more layouts.

6.2 Discussion

We encountered many limitations when applying our method to the Tangram testbed and to the benchmarks of the ESICUP. Some can be addressed by improving the picker, heuristic and filtering modules, and some improvements require changes in the search framework itself.

Firstly, the concept of graph search itself is to generate new successors from each explored node and to score each of them. As the packing problem deals with continuous domains, we had to discretize it by picking placements on the domain boundary. However, picking only vertices from the boundary is limited and picking points spaced with a small interval often produces too many successors. In addition, picking to maximize an evaluation criterion is also too restrictive, as positions in second rank are completely ignored. A more flexible approach would be to define potential positions to pick, and then to pick a reasonable number of positions among them based on a few evaluation criteria and some randomization.

Secondly, we have systematically used depth-first a the highest priority heuristic in our searches. Because of this, our searches were often trapped in branches with low quality layouts and backtracked very slowly, almost canceling the benefits of variable-based and value-based heuristics. To counter this phenomenon, we have experimented more diverse searches by distributing nodes between multiple fringes, but each branch eventually got trapped in a series of similar layouts. Again, we suggest to use more randomization in the search, distributing nodes between fringes based on their heuristic values, so that each fringe receives a mix of good and bad layouts rather than only the best or the worst. Alternatively, we could use mixed heuristics instead of ordered heuristics, by combining the different kinds of sub-heuristics in one expression, using for instance a weighted sum. However, our experiments with mixed heuristics showed that, without a good normalization coefficient to set all the sub-heuristics at the same level, the result of the comparisons of the mixed heuristic only depends on one sub-heuristic.

Thirdly, we designed two value-based heuristics that are based on the placement domains of the remaining pieces: the Least Constraining Value and the Minimum dead area. They were very effective when combined with restricted placement pickers and the Minimum Remaining Values heuristic, but difficult to use with benchmarks of big size because they require to compute many placement domains early. Doing so multiplies the computation cost by the branching factor of the graph, which is around 100 for a problem as small as the Tangram. We suggest to use lazy heuristic evaluation to address this issue. In our approach, we only used lazy evaluation when caching geometries, but the method is applicable to heuristics as well. The main idea is that we do not need to know all the heuristics of all nodes in the fringe in the detail. For ordered heuristics in particular, heuristics are compared one by one, by decreasing priority. Since most of our heuristics use the depth $\rightarrow$ variable $\rightarrow$ value priority system, it is not necessary to compute value-based heuristics such as LCV unless the best nodes in the fringe have the same depth and the same variable heuristic values, which only happens for a few nodes at each step of the search.

Finally, there are many aspects of space analysis that are yet to be used for filtering. In our approach we could only study isolated regions composing the free space in the container. By studying spatial properties such as the set of positions covered by the geometry of a piece when it is moving inside its collision-free region, it would be possible to detect unreachable parts of the free space even when they are not separated from other regions. Geometrical operations such as Minkowski sums, until now used only in the preprocessing step of no-fit polygon computation, could be efficiently used for this purpose.

Appendix A

Benchmarks

A.1 ESICUP data sets XML errata

polyXX.xml In `poly1a.xml`, `poly2b.xml`, `poly3b.xml`, `poly4b.xml` and `poly5b.xml`, in the `<nfps>` section, the IDs of the polygons for which NFPs are defined are incorrectly offset by -1. The polygons used for the piece shapes starts at index 1 but the NFPs are written for polygons starting at index 0. `polygon0` is only used for the rectangular container so this cannot be correct, and all indices should be increased. We did not use those data sets in our benchmarks.

shirts.xml The quantity of `piece4` is 1 in the XML file but the PDF and TXT files state that it should be 15. In our benchmarking tests we fixed the quantity to 15. Note that the piece IDs in the XML are often offset by -1 compared to the PDF. Consequently, what is called `piece4` in the XML is actually `PIECE 5` is the PDF. In addition, a few NFPs have only 2 vertices, which means some NFPs are reduced to a line. However, by looking at the shape of the pieces, there does not seem to be any exact fit/slide positions, therefore we believe that some of the NFPs defined in the XML file are incorrect. We computed the NFPs ourselves during the precomputation phase.

jakobs1.xml The XML defines an NFP with only 2 vertices for pieces 1 and 8 but those pieces do not have any exact slide positions, as in `shirts.xml`. We recomputed the NFPs in the precomputation phase.

jakobs2.xml In this benchmark pieces 8 and 17 can fit with an exact slide. Therefore $NFP(17, 8)$ contains degenerated lines, visible on the left of Figure A.1. The NFP provided by the XML file represents the degenerated line by an edge that goes backward onto itself in `nfpPolygon6531`. Although this is a legitimate representation, some geometrical libraries do not support self-intersecting edges. Therefore it is impossible to parse the XML to obtain valid NFP geometries, without using some extra processing to handle degenerated edges. For an XML format that supports both degenerated edges and vertices, see Appendix A.2. Alternatively, after parsing the XML file, we could apply extra processing by resolving self-intersections in the NFP and producing valid lines instead, as we would with the orbital sliding approach of Burke et al. [14].

Other data sets We experimented issues with the NFPs provided by other data sets such as Fu, Mao, Marques and Swim: the pieces were overlapping in the final layouts, which means that the reconstructed NFPs were incorrect. However, we have not confirmed whether the error lied in NFP data of the XML or in our parser and reconstruction algorithm. Nevertheless, we obtained valid layouts for Dighe1 and Dighe2, which means that our parser worked at least for those data sets.

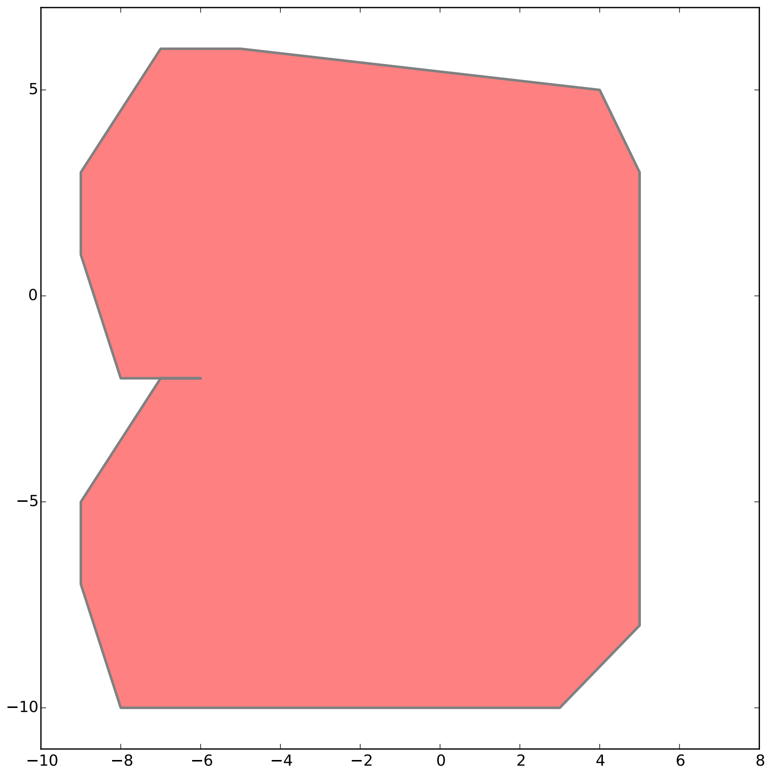


FIGURE A.1: NFP of piece 17 (angle 90°) vs piece 8 (angle 270°) in Jakobs2

A.2 New NestingXML for degenerated no-fit polygons

In order to store NFPs with degenerated edges, we propose a few improvements to the current NestingXML. Those suggestions should be considered carefully and adapted to the needs of other researchers because they can break compatibility with the previous format. Obviously, parsers can be programmed so that they can handle both the original and the new format.

First, it is noteworthy that polygons in NestingXML have no holes; they only have an exterior boundary. Instead, a piece with a hole is composed of two polygons: a positive polygon (`type=1` in the `componentType` element) to indicate the exterior and a negative polygon (`type=-1`) for the hole. To compute the NFP of a piece with another piece that contains a hole, we can use equation (4.13). If a piece is composed of multiple separated polygons, although rare, we can apply the NFP union formula 2.7. This means we can recompose the NFP between 2 pieces composed of multiple polygons. However, in practice, the objective of storing precomputation results is to reduce the amount of computation, so we propose to store the geometries of the NFPs between two pieces instead of 2 polygons. In this manner, a parser can immediately retrieve data on $NFP(P, Q)$ and the only computations required would be the actual construction of the geometry from this data.

We replace the elements `staticPolygon` and `orbitingPolygon` in `nfpType` in the original NestingXML:

```
<xs:complexType name="nfpType">
   <xs:sequence>
      <xs:element name="staticPolygon" type="nfpPolygonType"
   minOccurs="1" maxOccurs="1" />
      <xs:element name="orbitingPolygon" type="nfpPolygonType"
   minOccurs="1" maxOccurs="1" />
      <xs:element name="resultingPolygon" minOccurs="1"
   maxOccurs="1">
      <xs:complexType>
         <xs:attribute name="idPolygon" type="xs:IDREF" />
      </xs:complexType>
      </xs:element>
   </xs:sequence>
```

```
</xs:complexType>
```

Listing A.1: Original NestingXML nfpType

with the elements `staticPiece` and `orbitingPiece`:

```
<xs:complexType name="nfpType">
   <xs:sequence>
      <xs:element name="staticPiece" type="nfpPieceType"
   minOccurs="1" maxOccurs="1" />
      <xs:element name="orbitingPiece" type="nfpPieceType"
   minOccurs="1" maxOccurs="1" />
      <xs:element name="resultingPolygonWithBoundary" minOccurs
   ="1" maxOccurs="1">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="component" type="componentType
   " minOccurs="0" maxOccurs="unbounded" />
               <xs:element name="boundaryComponent" type="
   boundaryComponentType" minOccurs="0" maxOccurs="unbounded"
   />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:sequence>
</xs:complexType>
```

Listing A.2: New NestingXML nfpType

where `nfpPieceType` is similar to `nfpPolygonType`, but is linked to a piece ID:

```
<xs:complexType name="nfpPieceType">
   <xs:attribute name="idPiece" type="xs:IDREF" />
   <xs:attribute name="angle" type="xs:decimal" />
   <xs:attribute name="mirror" type="mirrorType" />
</xs:complexType>
```

Listing A.3: New NestingXML nfpPieceType

In addition, we have changed `resultingPolygon` to `resultingPolygonWithBoundary` in order to store degenerated edges and vertices similarly to our Polygon with Boundary model, described in section 4.3 Non-manifold geometry. For IFPs, we store additive boundaries, and for NFPs, we store subtractive boundaries. To have lighter XML files, it is also possible not to store the natural boundary, i.e. the boundary of the regularized set, and to generate it later instead.

The regularized set of `resultingPolygonWithBoundary` is represented by positive and negative polygons with `componentType`s, exactly as a piece is represented in the original NestingXML. We have simply factorized the type as a shared complex type instead of a nested type:

```
<xs:complexType name="componentType">
   <xs:attribute name="type" type="xs:integer" />
   <xs:attribute name="idPolygon" type="xs:IDREF" />
   <xs:attribute name="xOffset" type="xs:decimal" />
   <xs:attribute name="yOffset" type="xs:decimal" />
</xs:complexType>
```

Listing A.4: NestingXML componentType

Therefore, a `resultingPolygonWithBoundary` can have zero, one or more `componentType` children to represent its regularized set. If it has 0 such components, it is entirely composed of degenerated edges and vertices.

In addition, `resultingPolygonWithBoundary` has `boundaryComponentType` children to represent its degenerated parts. A `boundaryComponentType` can itself contains references to multiple edges and vertices, therefore only one `boundaryComponentType` child is enough to define any degeneracies in an NFP.

The degenerated edges and vertices are stored as follows:

- A `boundaryComponentType` contains a reference to a `boundaryType`.
- A `boundaryType` refers to zero, one or multiple `LinesType` (present in the original NestingXML), that represent degenerated edges, and `PointsType`, that represent degenerated vertices.

```
<xs:complexType name="boundaryComponentType">
   <xs:attribute name="idBoundary" type="xs:IDREF" />
   <xs:attribute name="xOffset" type="xs:decimal" />
```

```
    <xs:attribute name="yOffset" type="xs:decimal" />
</xs:complexType>

<xs:complexType name="boundaryType">
    <xs:sequence>
        <xs:element name="lines" type="LinesType" minOccurs="0"
    maxOccurs="1" />
        <xs:element name="points" type="PointsType" minOccurs="0"
     maxOccurs="1" />
        <xs:element name="xMin" type="xs:decimal" minOccurs="0"
    maxOccurs="1" />
        <xs:element name="xMax" type="xs:decimal" minOccurs="0"
    maxOccurs="1" />
        <xs:element name="yMin" type="xs:decimal" minOccurs="0"
    maxOccurs="1" />
        <xs:element name="yMax" type="xs:decimal" minOccurs="0"
    maxOccurs="1" />
        <xs:element name="perimeter" type="xs:decimal" minOccurs=
    "0" maxOccurs="1" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" />
</xs:complexType>

<xs:complexType name="PointsType">
    <xs:choice minOccurs="1" maxOccurs="unbounded">
        <xs:element name="point">
            <xs:complexType>
                <xs:attribute name="n" type="xs:integer" />
                <xs:attribute name="x0" type="xs:decimal" />
                <xs:attribute name="y0" type="xs:decimal" />
            </xs:complexType>
        </xs:element>
    </xs:choice>
</xs:complexType>
```

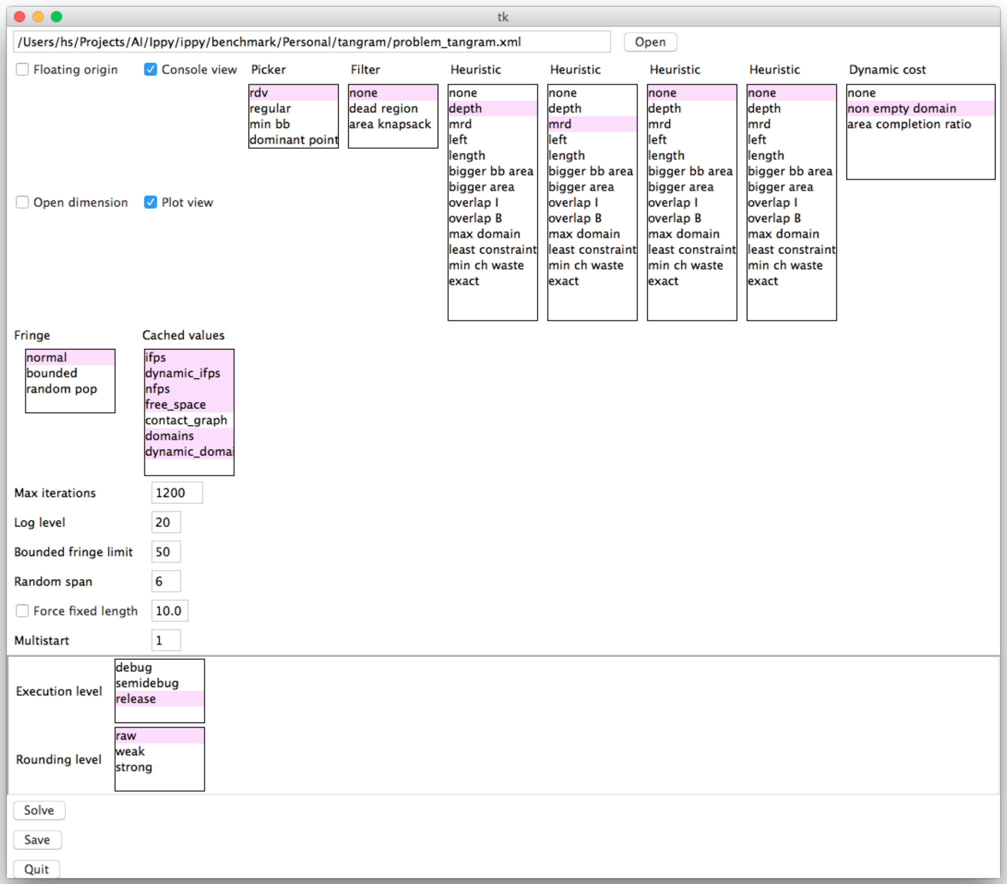
Listing A.5: New NestingXML boundary-related types

Once the content of `LinesType` and `PointsType` elements have been defined, it is possible to define the additive/subtractive boundary of any IFP/NFP respectively.

The complete XSD file can be found at https://github.com/hsandt/hipps under the name `nesting_degenerated.xsd`. The XSD file for the original NestingXML can be downloaded from the ESICUP [19] or at the same address as above, under the name `nesting.xsd`.

Appendix B

Application GUI



/Users/hs/Projects/AI/lppy/ippy/benchmark/Personal/tangram/problem_tangram.xml

Console view ✓

Plot view ✓

Max iterations	1200
Log level	20
Bounded fringe limit	50
Random span	6
Force fixed length	10.0
Multistart	1

FIGURE B.1: Application GUI

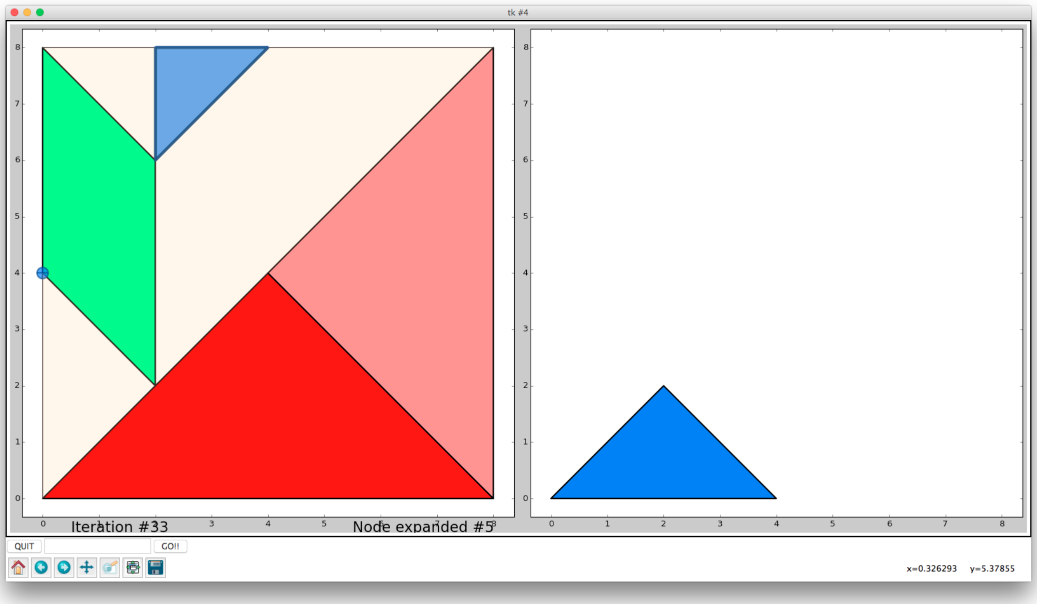


FIGURE B.2: Plotting the next piece (right subplot) and its domain for an orientation of -90° in light blue (left subplot). The blue circle on the left indicates a degenerated (additive) vertex.

Bibliography

[1] G. Wäscher, H. Haußner, and H. Schumann. "An improved typology of cutting and packing problems". In: *European Journal of Operational Research* 183.3 (2007), pp. 1109–1130. ISSN: 0377-2217. DOI: http://dx.doi.org/10.1016/j.ejor.2005.12.047. URL: http://www.sciencedirect.com/science/article/pii/S037722170600292X.

[2] *ESICUP: 2D Irregular Packing Benchmarks*. URL: http://paginas.fe.up.pt/~esicup/tiki-list_file_gallery.php?galleryId=2.

[3] J. A. Bennell and X. Song. "A beam search implementation for the irregular shape packing problem". In: *Journal of Heuristics* 16.2 (Sept. 2008), pp. 167–188. ISSN: 1381-1231, 1572-9397. DOI: 10.1007/s10732-008-9095-x. URL: http://link.springer.com/article/10.1007/s10732-008-9095-x.

[4] A. K. Sato, T. C. Martins, and M. S. G. Tsuzuki. "An algorithm for the strip packing problem using collision free region and exact fitting placement". In: *Computer-Aided Design* 44.8 (Aug. 2012), pp. 766–777. ISSN: 0010-4485. DOI: 10.1016/j.cad.2012.03.004. URL: http://www.sciencedirect.com/science/article/pii/S0010448512000565.

[5] T. Imamichi, M. Yagiura, and H. Nagamochi. "An iterated local search algorithm based on nonlinear programming for the irregular strip packing problem". In: *Discrete Optimization* 6.4 (Nov. 2009), pp. 345–361. ISSN: 1572-5286. DOI: 10.1016/j.disopt.2009.04.002. URL: http://www.sciencedirect.com/science/article/pii/S1572528609000218.

[6] A. Elkeran. "A new approach for sheet nesting problem using guided cuckoo search and pairwise clustering". In: *European Journal of Operational Research* 231.3 (Dec.

2013), pp. 757–769. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2013.06.020. URL: http://www.sciencedirect.com/science/article/pii/S0377221713005080.

[7] T. Wauters, J. Verstichel, and G. Vanden Berghe. “An effective shaking procedure for 2D and 3D strip packing problems”. In: *Computers & Operations Research* 40.11 (Nov. 2013), pp. 2662–2669. ISSN: 0305-0548. DOI: 10.1016/j.cor.2013.05.017. URL: http://www.sciencedirect.com/science/article/pii/S0305054813001548.

[8] Y. Stoyan and L. Ponomarenko. *Minkowski sum and hodograph of the dense placement vector function*. Technical Report SER. A10. SSR Academy of Science, 1977.

[9] R. Cunninghame-Green. “Geometry, Shoemaking and the Milk Tray Problem”. In: *New Scientist* 1677.1 (Aug. 1989), pp. 50–53.

[10] S. Hert. “2D Polygon Partitioning”. In: *CGAL User and Reference Manual*. 4.6.1. CGAL Editorial Board, 2015. URL: http://doc.cgal.org/4.6.1/Manual/packages.html#PkgPolygonPartitioning2Summary.

[11] J.-M. Lien and N. M. Amato. “Approximate convex decomposition of polygons”. In: *Computational Geometry*. Special Issue on the 20th ACM Symposium on Computational Geometry 35.1–2 (Aug. 2006), pp. 100–123. ISSN: 0925-7721. DOI: 10.1016/j.comgeo.2005.10.005. URL: http://www.sciencedirect.com/science/article/pii/S0925772105001008.

[12] R. Wein. “2D Minkowski Sums”. In: *CGAL User and Reference Manual*. 4.6.1. CGAL Editorial Board, 2015. URL: http://doc.cgal.org/4.6.1/Manual/packages.html#PkgMinkowskiSum2Summary.

[13] E. Behar and Jyh-Ming Lien. “Fast and robust 2D Minkowski sum using reduced convolution”. In: IEEE, Sept. 2011, pp. 1573–1578. ISBN: 978-1-61284-456-5 978-1-61284-454-1 978-1-61284-455-8. DOI: 10.1109/IROS.2011.6094482. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6094482.

[14] E. K. Burke et al. “Complete and robust no-fit polygon generation for the irregular stock cutting problem”. In: *European Journal of Operational Research* 179.1 (May 2007), pp. 27–49. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2006.03.011. URL: http://www.sciencedirect.com/science/article/pii/S0377221706001639.

[15] C. S. Pedamallu, L. Özdamar, and T. Csendes. "An Interval Partitioning Approach for Continuous Constrained Optimization". In: *Models and Algorithms for Global Optimization*. Ed. by A. Törn and J. Žilinskas. Optimization and Its Applications 4. Springer US, 2007, pp. 73–96. ISBN: 978-0-387-36720-0 978-0-387-36721-7. URL: http://link.springer.com/chapter/10.1007/978-0-387-36721-7_5.

[16] Hanayama and Meiji. *Himilk chocolate puzzle, sweet difficulty*. URL: http://www.hanayama-toys.com.

[17] R. B. Martins, M. A. Carravilla, and C. Ribeiro. "Solving combinatorial problems: an XML-based software development infrastructure". In: *XATA 2006: XML: Aplicações e Tecnologias Associadas: 4ª Conferência Nacional, 9 a 10 de Fevereiro de 2006*. 2006. URL: http://repositorio-aberto.up.pt/handle/10216/15746.

[18] *ESICUP: Euro Special Interest Group on Cutting and Packing*. URL: http://paginas.fe.up.pt/~esicup/tiki-index.php.

[19] *ESICUP: NestingXML*. URL: http://paginas.fe.up.pt/~esicup/tiki-index.php?page=NestingXML.

[20] *The Shapely User Manual — Shapely 1.2 and 1.3 documentation*. URL: http://toblerity.org/shapely/manual.html.

[21] *GEOS (Geometry Engine, Open Source)*. URL: http://trac.osgeo.org/geos/.

[22] *JTS Topology Suite*. URL: http://tsusiatsoftware.net/jts/main.html.

[23] *TkInter - Python Wiki*. URL: https://wiki.python.org/moin/TkInter.

[24] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[25] The CGAL Project. *CGAL User and Reference Manual*. 4.6.1. CGAL Editorial Board, 2015. URL: http://doc.cgal.org/4.6.1/Manual/packages.html.

[26] W. Nef. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*. Beiträge zur Mathematik, Informatik und Nachrichtentechnik, Bd. 1. Bern: Herbert Lang, 1978. ISBN: 3216050004.

[27] M. Seel. "2D Boolean Operations on Nef Polygons". In: *CGAL User and Reference Manual*. 4.6.1. CGAL Editorial Board, 2015. URL: http://doc.cgal.org/4.6.1/Manual/packages.html#PkgNef2Summary.

[28] M. Seel. *Implementation of Planar Nef Polyhedra*. Research Report MPI-I-2001-1-003. Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany: Max-Planck-Institut für Informatik, Aug. 2001.

[29] *8.4. heapq — Heap queue algorithm — Python 2.7.10 documentation*. URL: https://docs.python.org/2/library/heapq.html.