Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Computational approaches to polyomino tiling problems

Bachelor Thesis

2022
Dávid Mišiak

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Computational approaches to polyomino tiling problems

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	doc. RNDr. Ján Mazák, PhD.

Bratislava, 2022
Dávid Mišiak

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Dávid Mišiak
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Computational approaches to polyomino tiling problems
Výpočtové riešenie úloh o dláždení polyominami

Anotácia: Problém vydlážditeľnosti zadanej mriežky sadou polyomín sa často objavuje v rekreačnej matematike a matematických súťažiach. Hoci je tento problém vo všeobecnosti NP-úplný, v praxi môžu byť niektoré prístupy k riešeniu (napríklad konverzia na problém splniteľnosti a následné využitie SAT solvera) účinné pre niektoré vstupy. V tejto práci podrobnejšie preskúmame niekoľko výpočtových prístupov k tomuto problému.

Cieľ: Implementovať viacero algoritmov na riešenie problému dláždenia polyominami, porovnať ich na rôznych vstupoch a identifikovať zaujímavé triedy vstupov vo vzťahu k použitým algoritmom. Preskúmať možnosti použitia ofarbovacích invariantov na dôkaz neexistencie dláždenia.

Vedúci: doc. RNDr. Ján Mazák, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 28.10.2021

Dátum schválenia: 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Dávid Mišiak
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Computational approaches to polyomino tiling problems

Annotation: Tilings of various boards by polyominoes often appear in recreational mathematics and in mathematical competitions. While the problem is NP-complete in general, some instances can be solved quickly in practice (for instance, by a conversion into a SAT instance and usage of an efficient SAT solver). This thesis aims to investigate several computational approaches to the problem.

Aim: The goal is to implement several algorithms solving chosen variants of polyomino tiling problems, compare these algorithms on a variety of inputs, and identify interesting classes of inputs. We will also consider employing colouring invariants for proving impossibility of tiling.

Supervisor:	doc. RNDr. Ján Mazák, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.

Assigned: 28.10.2021

Approved: 04.11.2021 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank my supervisor doc. RNDr. Ján Mazák, PhD. for his guidance and approach suggestions.

Abstrakt

Problém dláždenia polyominami je typickým príkladom problému založeného na prehľadávaní. V tejto práci preskúmame niekoľko vhodných prístupov k riešeniu problému dláždenia: jednoduché prehľadávanie s návratom, použitie Algoritmu X navrhnutého Donaldom Knuthom, redukciu problému dláždenia na problém splniteľnosti booleovskej formuly, redukciu na celočíselné lineárne programovanie či použitie constraint satisfaction solvera. Popíšeme užitočné implementačné techniky pre tieto prístupy a vytvoríme nástroj Tiler, v ktorom spojíme popísané algoritmy so špičkovými solvermi. Na záver detailne porovnáme ich výkon na zbierke inštancií problému dláždenia a identifikujeme triedy inštancií, ktoré sú vhodné pre jednotlivé algoritmy.

Kľúčové slová: dláždenie polyominami, SAT, lineárne programovanie, ofarbovacie invarianty

Abstract

The problem of tiling a board by polyominoes is a typical example of a search-based problem. Our work explores several suitable approaches to solving the tiling problem: simple backtracking, using Algorithm X proposed by Donald Knuth, translating the tiling problem to the boolean satisfiability problem, translating it to integer linear programming, and using a constraint satisfaction solver. We describe useful implementation techniques for these approaches and create a command-line tool Tiler that integrates the solving algorithms with state-of-the-art solvers. Finally, we prepare a detailed comparison of their performance on a collection of problem instances and identify classes of tiling instances that are better suited for one algorithm or another.

Keywords: polyomino tiling, SAT, linear programming, coloring invariants

Contents

List of Figures

Introduction

Tiling problems have been studied extensively in past decades [1, 2, 3]. The difficulty of a specific problem class highly depends on restrictions that are in place. For instance, the tileability of a simply connected board by an unlimited number of bars of length 2 or 3 can be decided in linear time [4]. On the other hand, there are tiling problems on infinite boards which are undecidable [5].

In this thesis, we will focus on the tileability of finite boards with a set of polyominoes, given their shapes and counts. We can easily see that this problem is decidable and NP. Furthermore, it can be shown that this problem is NP-complete [3]. Because of this, all our algorithms will inherently have exponential worst-case time complexity.

Since tiling problems are popular in recreational mathematics and mathematical competitions, there are user-friendly web applications capable of solving them [6, 7].

Among scientific and professional software, the SageMath [8] system is probably the most notable framework that features a dedicated polyomino tiling solver [9]. The solver provides a rich API to define a board and polyominoes of arbitrary shape and dimension and specify whether rotations and reflections are allowed when solving. The solver is capable of finding the number of all solutions as well as displaying the solutions. There are other convenient functionalities, such as animations of the solving process.

Another project that aims to solve tiling instances is the `polyomino` Python package [10]. It is best suited for command line usage—it supports visualizing solutions textually (using ASCII characters).

All previously mentioned solvers have a common trait: They employ Algorithm X proposed by Donald Knuth [11] [1]. Algorithm X is a general algorithm for solving the exact cover problem. It is natural to formulate a polyomino tiling instance as an exact cover instance—we will explore this algorithm further in section 2.3.

Compared to the described tools, in our work we aim to examine many more different approaches to solving, see chapter 2. Our main focus will be on conversions to other problems (such as SAT or ILP) where we can utilize efficient solvers. We will also briefly explore the usability of coloring invariants for proving the impossibility of tiling. This method is often used when solving tiling problems in mathematical competitions.

[1] Although one of them supports conversion to a SAT instance as an alternative [7].

Chapter 1

Terminology

In this chapter, we will establish basic terminology and precisely define the problem of our interest.

The most fundamental entities in this work are unit squares in a square lattice. Two unit squares are *adjacent* if they share a common edge. The underlying square lattice can be considered infinite in all directions, even though we will examine only finite tiling problems.

Definition 1 (Region)**.** A finite geometric shape consisting of unit squares arranged in a square lattice is called a *region*.

The size $|r|$ of a region r is the number of unit squares it occupies.

A region is *connected* if every pair of its squares is connected by a path composed of consecutively adjacent squares contained in the region; otherwise, the region is *disconnected* and its maximal connected subregions are called *components*.

A connected region is *simply connected* (i.e. without holes) if its complement in the infinite square lattice is connected.

Two regions in a square lattice differing in translation (shifting in any direction), rotation or reflection (flipping over) are considered different.

Definition 2 (Top-left unit square)**.** The region's leftmost unit square in its topmost row is the region's *top-left unit square*.

The relative position of two regions can be unambiguously expressed as the pair (d_x, d_y): offsets of their top-left unit squares along the x-axis and y-axis.

Definition 3 (Tile)**.** A *polyomino tile*, or simply a *tile*, is an equivalence class on the set of all connected regions. Two regions are considered equivalent (thus being the same tile) if they differ only in translation, rotation, and optionally, reflection as well.

A tile's rotations and reflections are called *tile variants*—an example of those is shown in fig. 1.1.

The only polyomino tile consisting of a single unit square is called *monomino*; the one consisting of two unit squares is called *domino*. Subsequently, there are *trominoes*,

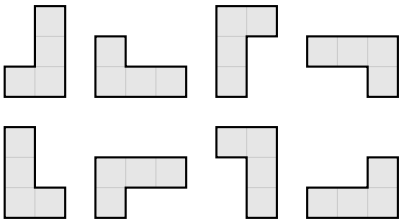


Figure 1.1: An example of tile variants

tetrominoes, *pentominoes*, and so forth. All polyominoes with size up to 5 are listed in fig. 1.2. We assigned them names consisting of a number and a letter representing the tile's size and shape. Notice that if we allow reflections, some names correspond to a single tile, such as 4J and 4L. A different naming scheme for pentominoes was introduced by Golomb [12], but it does not distinguish between reflected shapes (e.g. 4J and 4L), therefore prompting us to suggest our own naming convention.

There has been substantial research done on the topic of polyomino enumeration and counting polyominoes of a fixed size [12, 13, 14], although this is out of the scope of this work.

Now we can formally define the concept of a polyomino tiling problem instance.

Definition 4 (Tiling problem instance)**.** A polyomino tiling problem instance is a triple (b, T, r), where:

- Board b is a connected region.
- Set T is a finite set of pairs (t_i, c_i), where t_i is a tile and c_i is the maximum count of its available instances (either a positive integer or infinity). No two tiles t_i, t_j are the same for any $i \neq j$.
- Boolean value r determines whether tiles differing in reflection are considered equivalent (that is, whether we are allowed to flip individual tile instances during the tiling process).

Definition 5 (Solution)**.** The solution of a tiling instance is a perfect partitioning of board b into subregions. Each subregion has to correspond to a tile from T (taking the value of r into account) and for each tile t_i, the number of the corresponding subregions may not exceed c_i.

If a solution exists, we say that b *is tileable by* T.

Example 1 (Tiling problem instance)**.** Is it possible to tile a 10×10 checkerboard by 25 instances of the 4×1 tetromino? (In this case, it is irrelevant whether reflections are allowed, the result will be the same.)

Some tiling instances may have multiple different solutions; we will however focus solely on the solvability. There are good reasons for this decision: tiling problems are

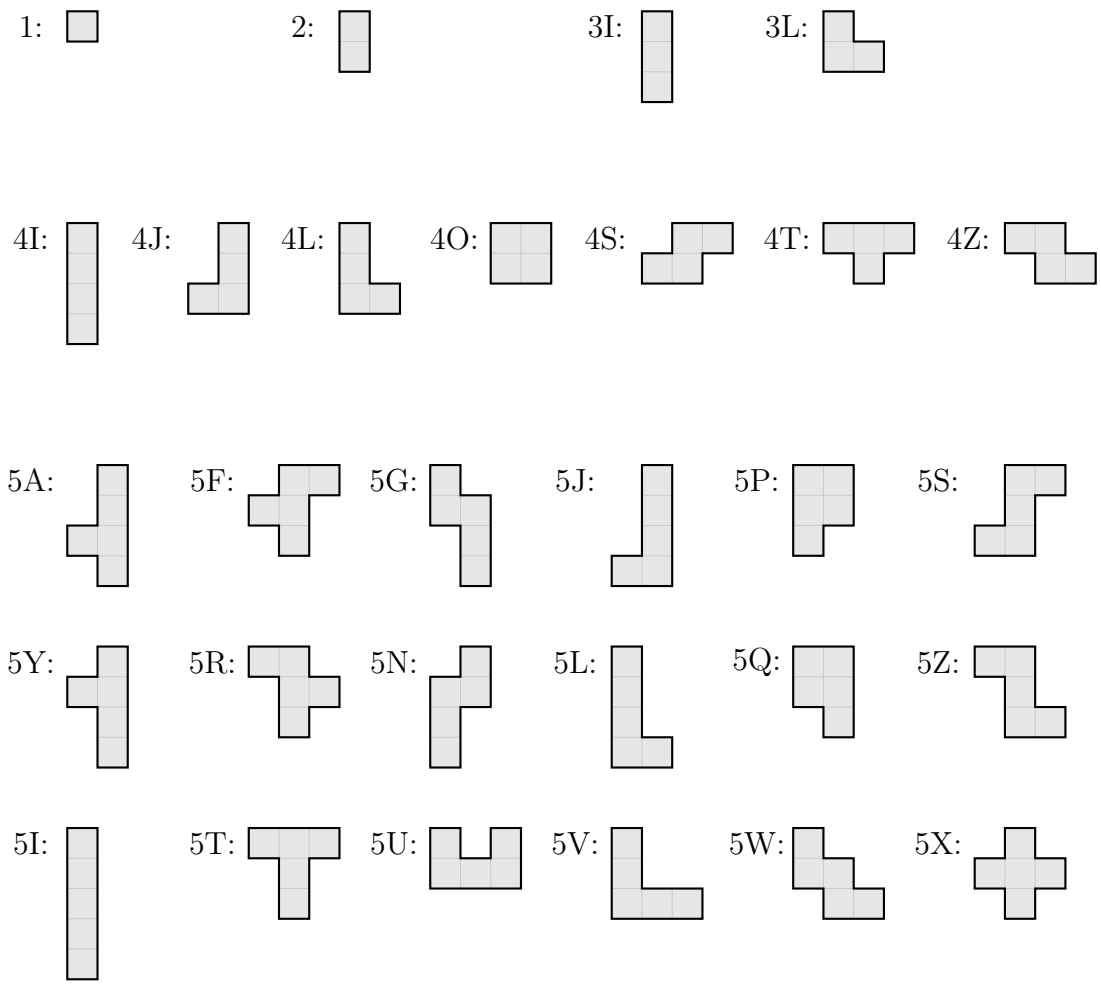


Figure 1.2: Monomino, domino, trominoes, tetrominoes, pentominoes

diverse enough by themselves, adding another aspect to consider might be counter-productive. Furthermore, some instances may have an immense number of solutions and searching for all of them would greatly prolong the computation, making it even infeasible, and thus effectively discard said instances in our performance comparisons.

A solution does not necessarily have to utilize all available tile instances. On the other hand, a sufficient number of tile instances needs to be available to cover the entire board—specifically, the following equation has to be satisfied for a problem instance to possibly be solvable:

$$|b| \leq \sum_{(t_i,c_i)\in T} |t_i| \cdot c_i \tag{1.1}$$

Chapter 2

Algorithms

In this chapter, we will analyze the selected approaches to solving the polyomino tiling problem and briefly describe their implementation.

2.1 Tiler CLI tool

In order to simplify the experimentation with the presented tiling algorithms, we decided to develop a user-friendly command line tool called *Tiler*. We selected C++ as the programming language for this task, motivated by its execution speed and the fact that virtually all libraries and toolkits utilized by our algorithms (such as the state-of-the-art SAT and ILP solvers) provide a C/C++ interface.

Tiler is capable of reading the problem instance definition—board shape and tile shapes and counts—either from command line arguments or from an input file. It accepts several shape input formats, including shapes given by name (see fig. 1.2) or specifying the shape by a grid of "x" and space characters.

On top of that, Tiler supports several optional parameters which enable the user to allow tile reflections, select the solving algorithm of their choice, set a time limit for solving, and control logging verbosity.

When the solving algorithm halts, Tiler prints "SOLVABLE" or "UNSOLVABLE", and if a solution exists, Tiler optionally enumerates the tile instance positions or generates an SVG image depicting one of the solutions.

The entire Tiler source code is attached in appendix A. Tiler's dependencies, input formats, and other similar topics are described in `README.md`.

In the following sections, we will focus on the solving algorithms as well as some of the important implementation aspects.

2.2 Simple backtracking

The main goal of this algorithm is simplicity so that we have an easy-to-understand reference point for other solvers when comparing their performances. The central idea is as follows: In one step, we will place a tile to the top-left corner of the board and fill the entire board in several steps of this kind.

More precisely, we define a procedure `place_tile` which takes a problem (b, T, r) and a *variant* v of tile t_i, where $(t_i, c_i) \in T$. The procedure aligns the top-left unit square of b with the top-left unit square of v. If v placed this way is a subregion of b, `place_tile` removes this subregion from b, creating a smaller board b'. Correspondingly, it creates T' by replacing (t_i, c_i) with $(t_i, c_i - 1)$ in T (or dropping this tile from T if $c_i = 1$). Finally, `place_tile` returns the new problem (b', T', r).

If v placed this way is not a subregion of b (i.e. there is at least one unit square that belongs to the placed v but not to b), `place_tile` fails.

At this point, the tiling algorithm is rather straightforward. We iterate over all variants v of all tiles from T and for each of them, we call `place_tile` on (b, T, r) and v. If it succeeds, we recursively perform this same procedure with the obtained subproblem (b', T', r).

If all of the recursive sub-calls fail, so fails the current call and the algorithm backtracks to the parent call. If at least one recursive call returns a valid solution of the respective subproblem, the current call can construct a solution of the current problem as a combination of tile variant placement realized by `place_tile` and the solution of the corresponding subproblem.

Ultimately, we need to describe the trivial case of the recursion: A recursive call is trivially successful if b is empty. In this case, b is clearly tileable by T regardless of the actual T.

In the end, we either obtain a complete solution, or the algorithm fails (no branch reaches the trivial case). If the algorithm returns a solution, we can see that it will indeed be correct. On the other hand, if a solution exists, we can gradually remove tile instances from its top-left corner. This sequence of tile variants uniquely determines a successful branch of recursive calls, which proves that the algorithm will find a solution in this case.

The exponential time complexity is evident from the execution flow—we are recursively traversing a virtual tree of subproblems with depth proportional to the size of the board and branching factor dependent on the number of tiles and their variants.

2.2.1 Tile variant selection strategies

In the algorithm above, we stated that we iterate over all variants of all tiles in each recursive call. However, the order in which we enumerate the available tile variants may affect the execution time for some solvable problem instances. Apart from the *default* ordering, which respects the input tile order, we implemented three other selection strategies: *frequent first*, which prefers tiles with the highest number of remaining tile instances (thus preserving flexibility for later); *rare first*, which favors a lower number of remaining tile instances instead; and the *fill top row* ordering, which selects the tile variant that can cover as many top row unit squares as possible.

It should be emphasized that these strategies will make no difference for many problem instances. For example, all strategies perform equally well on problem instances with a single tile shape or problem instances with all tile counts equal to one (the only exception is the *fill top row* strategy).

2.3 Algorithm X

The core idea of this approach is a conversion of the polyomino tiling problem to the NP-complete [15] problem of finding an exact cover.

Definition 6 (Exact cover)**.** Given a set X and a collection S of subsets of X, an exact cover C of X is a subcollection of S that partitions X.

Put differently, X is exactly covered by C if each element of X is contained in exactly one set included in C.

Donald Knuth formulated a relatively simple algorithm that enumerates the solutions of an exact cover instance [11, 16]. In his work, Knuth calls the elements of X *primary items* and the elements of S (i.e. sets of items) *options*. Furthermore, he introduces *secondary items*—elements that should be covered by *at most one* set included in C each (we can consider them optional). It is not difficult to see that this extension does not change the problem hardness.

We can construct a matrix A of 0s and 1s with items as columns and options as rows, where the value 1 means that the row's option contains the column's item and the value 0 means the opposite. Knuth then provides [11] the search procedure listed in algorithm 1. It is a simplified version of the algorithm which does not keep track of the removed rows—if it did, we could return the successful row sets as solutions.

The nondeterministic choice of row r represents trying all the available options sequentially, always with a fresh copy of the current matrix A (i.e. not affected by the sibling nondeterministic branches). If column c contains only 0s, this branch of the search yields no solutions.

Algorithm 1 Knuth's Algorithm X

If A is empty, the problem is solved.
Otherwise, choose a primary column c (deterministically).
Choose a row r such that $A[r, c] = 1$ (nondeterministically).
for column j in A such that $A[r, j] = 1$ **do**
    Delete column j from A.
    for row i in A such that $A[i, j] = 1$ **do**
        Delete row i from A.
    end for
end for
Repeat this algorithm recursively on the reduced matrix A.

The deterministic choice of c can be performed according to a heuristic function. For example, Knuth suggests choosing the column with the fewest 1s in the current matrix since such column can be viewed as "hard to cover" and requires only a small number of rows r to be examined.

The main accomplishment of Knuth's paper [11] is the concept of *dancing links*: An elegant implementation of the (usually sparse) matrix A using doubly linked links, which supports instant removing and reinserting of columns and rows.

2.3.1 Tiling problem translation

The translation of a tiling instance (b, T, r) to an exact cover instance is straightforward. We introduce a primary item for each unit square of board b and a secondary item for each instance of each tile. Subsequently, we add a single option for each possible position of each variant of each tile instance: The option shall include all unit square items covered by the instance's board placement and the corresponding secondary item of the tile instance.

If the Algorithm X identifies a correct exact cover, we have a tiling solution: Each primary item is covered by exactly one option, which translates to each unit square of the board being covered by exactly one tile instance; and each secondary item is covered by at most one option, which implies that each tile instance is placed on the board at most once. The other direction of the equivalence is analogous.

Moreover, we identified the following independent optimizations. Note that none of these optimizations affects the problem solvability.

Optimization 1 (Omitting initial tile instance positions)**.** The exact cover problem statement size as well as the according search space can be reduced by enforcing that the j-th instance of the tile t_i can appear only on the j-th position on board or further (with regards to a sensible position ordering, such as left-to-right and top-to-bottom),

for each $j \in \{1, \ldots, c_i\}$. This can be achieved by omitting the relevant primary items in the options.

Optimization 2 (Many tile instances)**.** If there is a tile (t_i, c_i) for which the equation $|b| \leq |t_i| \cdot c_i$ holds (the tile has enough instances to cover the entire board area), we can ignore the notion of tile instances. There is no need to encode that at most c_i instances can be utilized since no solution could possibly require more than c_i instances. In practice, this means we can act as if c_i was one and skip the creation of the secondary item. Since c_i is usually large if the condition holds, this optimization eliminates a considerable number of secondary items and redundant options.

Optimization 3 (Exact tile set)**.** If $|b| = \sum_i |t_i| \cdot c_i$ (indicating that all instances of all tiles must be used), we can replace all secondary items with primary items, hinting that all items must indeed be covered *exactly* once rather than *at most* once. This allows the solving algorithm to conclude that some branches lead to no solution sooner (because of a primary column containing no 1s—such column would be ignored if it was secondary).

2.3.2 Implementation

We implemented the translation algorithm with all the optimizations listed above.

To solve the exact cover problem, we decided to employ Knuth's implementation of Algorithm X [17] in order to minimize the possibility of implementing it incorrectly or inefficiently ourselves (though several other implementations are available [18, 19]). It was necessary to modify the code slightly, mainly to facilitate using it as a callable function instead of as a command line tool. Even though the original implementation searches for all solutions by default, it accepts a parameter specifying the solution count limit (set to one by us).

It should be noted that Knuth lists several problems that can be reduced to the exact cover problem in his work [11, 16], one of them being the problem of tiling 3×20 rectangular board by the twelve distinct pentominoes (each used exactly once) with reflections allowed. His translation matches our translation with opt. 3 applied; the other two optimizations are not applicable to this tiling instance. Knuth adds a few symmetry-breaking optimizations specific to the tiling instance (e.g. he fixes the rotation of the 5V pentomino).

The exact cover solving approach is similar to our simple backtracking (described in section 2.2) in the search structure, but there are a few differences: When choosing the next unit square to cover, Algorithm X picks the unit square with the fewest possible tile instance placements that can cover it, whereas simple backtracking covers the top-left corner of the remaining portion of the board. Also, Algorithm X should have another

performance advantage—the optimized data structures and procedures, thanks to the *dancing links* technique. On the other hand, the encoding of tile counts larger than one is rather cumbersome because of severe option duplication. Simple backtracking handles these cases much more smoothly.

2.4 Conversion to the SAT problem

The boolean satisfiability problem, abbreviated as SAT, is the problem of deciding whether a given boolean formula is satisfiable, in other words, whether there exists an assignment of *true* and $false$ values to the formula's boolean variables (so-called *interpretation*) resulting in the entire formula being *true*.

There are several high-quality solvers designed for solving SAT instances, such as CaDiCaL [20, 21] or CryptoMiniSat [22, 23]. In practice, they are often very powerful [24, 25] for solving a great range of constraint-based problems converted to SAT instances. Likewise, we will design an algorithm to translate a tiling instance to a SAT instance.

A *literal* is either a boolean variable or its negation. A *clause* is a disjunction of literals. We say that a formula F is in *conjunctive normal form* (CNF) if it is a conjunction of clauses:

$$F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{i,j}, \tag{2.1}$$

where $n \in \mathbb{N}$ is the number of clauses, $m_i \in \mathbb{N}$ is the length of the i-th clause and $l_{i,j}$ is the j-th literal in this clause.

SAT solvers usually require the input to be in CNF, hence we aim to produce a formula in this form.

2.4.1 AMO translation

In the simplest translation approach, we will heavily use constraints of the form "*at most one* of n literals should be *true*" and "*exactly one* of n literals should be *true*". Let us suppose that we have a way to express these constraints as sets of clauses (we will explore the available implementations below). We will refer to the encodings as $\mathrm{AMO}(\{x_1, \ldots, x_n\})$ and $\mathrm{EO}(\{x_1, \ldots, x_n\})$.

The translation is very similar to the exact cover translation described in section 2.3. Let v be a variant of a tile t_i. For each $j \in \{1, \ldots, c_i\}$ and each possible position (d_x, d_y) on the board b, let $r(v, d_x, d_y, j)$ be the subregion of b occupied by the j-th instance of v, where (d_x, d_y) are the relative offsets of top-left unit squares of b and the subregion.

Moreover, let $x(v, d_x, d_y, j)$ be a boolean variable representing the statement "the j-th instance of tile variant v is placed on the position (d_x, d_y) of the board b".

Now we can express all the tiling constraints as sets of boolean clauses:

1. "Each unit square is covered by exactly one tile instance." Let s be a unit square of b and let cover(s) be the set of all variables $x(v, d_x, d_y, j)$ for which $r(v, d_x, d_y, j)$ covers the unit square s. The desired set of clauses is as follows:

$$F_1 = \bigwedge_{s \in b} \mathrm{EO}(\mathrm{cover}(s)) \tag{2.2}$$

2. "Each tile instance is placed on the board at most once." Let placements(t_i, j) be the set of all variables $x(v, d_x, d_y, j)$ where v is a variant of t_i. The constraint is then equivalent to the following set of clauses:

$$F_2 = \bigwedge_{(t_i, c_i) \in T} \bigwedge_{j=1}^{c_i} \mathrm{AMO}(\mathrm{placements}(t_i, j)) \tag{2.3}$$

The final formula in CNF is $F = F_1 \wedge F_2$. We can see that if this formula is satisfiable, then the tiling instance is solvable because all requirements in the definition 5 are fulfilled (F_1 implies the perfect partitioning of b; F_2 ensures that for each tile t_i, c_i is not exceeded). Conversely, if the tiling instance has a solution, the corresponding interpretation of variables satisfies all the clauses.

Most SAT solvers can determine not only the satisfiability of a formula, but in case it is satisfiable, they return a successful interpretation, too. It is straightforward to transform such interpretation to a tiling instance solution—each variable $x(v, d_x, d_y, j)$ which is evaluated as *true* precisely identifies a placed tile in the solution.

Thanks to the resemblance between the AMO translation and the exact cover translation, we can employ all opts. 1 to 3. Opts. 1 and 2 directly reduce the search space, whereas opt. 3 provides additional information to the SAT solver that it cannot easily deduce by itself. It should be noted that one could take the opposite approach—instead of using the stricter EO constraint to express the second condition, the first condition could be relaxed by using an *at least one* constraint. This change preserves the translation correctness (if the assumption of opt. 3 is met) and shortens the boolean formula, but it prolongs the execution time, presumably because of the missing information that could otherwise be harnessed during the search process.

AMO and EO constraint encoding

A natural way how to express the AMO constraint is to state that for any pair of literals, at least one of them is $false$:

$$\mathrm{AMO}(\{x_1, \ldots, x_n\}) = \bigwedge_{1 \le i < j \le n} \neg x_i \vee \neg x_j \tag{2.4}$$

Encoding the EO constraint is as easy as adding a clause requiring at least one of the literals to be *true*:

$$\mathrm{EO}(\{x_1, \ldots, x_n\}) = \mathrm{AMO}(\{x_1, \ldots, x_n\}) \wedge \bigvee_{i=1}^{n} x_i \tag{2.5}$$

We can see that this is indeed a set of clauses under conjunction and that it is *true* if and only if exactly one of $x_1, \ldots, x_n$ is *true*.

However, a significant drawback of this encoding is the size of its output—there are $O(n^2)$ clauses. Since we intend to use this encoding many times in our tiling problem reduction, this could seriously harm the performance of SAT solvers.

Fortunately, there are more efficient encodings, such as the *sequential* encoding [26], which produces only $O(n)$ additional clauses, at the expense of $n - 1$ new auxiliary variables.

When encoding $\mathrm{AMO}(\{x_1, \ldots, x_n\})$ using the sequential encoding, we add auxiliary variables $y_1, \ldots, y_{n-1}$ and add binary clauses that force the following two properties to be satisfied:

1. "There is $j \in \{0, \ldots, n-1\}$ such that $y_1 = y_2 = \cdots = y_j = \mathit{false}$ and $y_{j+1} = y_{j+2} = \cdots = y_{n-1} = \mathit{true}$." We can secure this by adding implications of the form $y_i \implies y_{i+1}$, which can be rewritten as clauses:

$$S_1 = \bigwedge_{i \in \{1,\ldots,n-2\}} \neg y_i \vee y_{i+1} \tag{2.6}$$

2. "If $x_i = \mathit{true}$, then $y_{i-1} = \mathit{false}$ and $y_i = \mathit{true}$." This can be expressed by adding implications of the form $x_i \implies \neg y_{i-1}$ and $x_i \implies y_i$, which we can reformulate as:

$$S_2 = (\neg x_n \vee \neg y_{n-1}) \wedge (\neg x_1 \vee y_1) \wedge \bigwedge_{i \in \{2,\ldots,n-1\}} (\neg x_i \vee \neg y_{i-1}) \wedge (\neg x_i \vee y_i) \tag{2.7}$$

We can now observe that the formula $S_1 \wedge S_2$ does indeed enforce the AMO constraint as a natural consequence of the fact that the false-true boundary among the auxiliary variables can be present at most once. The first requirement secures the boundary existence and the second requirement ties its position to the x_i with the value *true*.

An example of the sequential encoding for $n = 7$ is provided in fig. 2.1. The gray circles represent the input literals, the white circles represent the auxiliary literals and the lines represent the binary clauses. A black square at the end of a line stands for a negation of the adjacent literal. The boolean values inside the circles show one of the satisfying interpretations.

There are several other AMO (and EO) encodings, such as the *commander* [27], *bimander* [28], *k-product* [29], and *binary* [30] encodings, as well as articles comparing

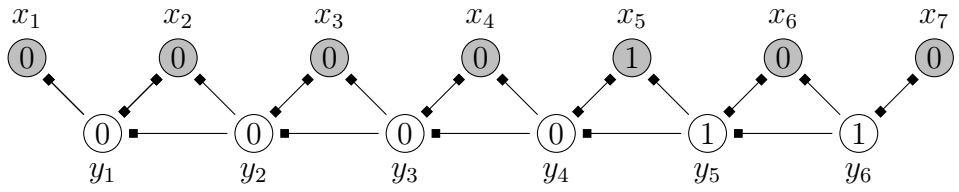


Figure 2.1: An example of the sequential AMO encoding

their properties [28]. Regarding the implementation in Tiler, the PBLib library [31, 32] for encoding pseudo-boolean constraints (including AMO and EO) allows exploring multiple encodings inexpensively in terms of our programming effort. It provides ready-to-use implementations of all the AMO and EO constraint encodings mentioned above, as well as the *auto* encoding that attempts to select the best encoding for the given number of literals.

2.4.2 AMO-ordered translation

If a tile has only one tile instance available or if opt. 2 is in place, we do not need to worry about the tile instance encoding overhead. However, if we encounter a tile with several tile instances, the AMO encoding treats the tile instances as different pieces (except for opt. 1), whereas in reality, the order of the tile instances does not matter. If we enforce the order of their positions on the board, we can further reduce the search space. Breaking the order symmetry is explicitly advised by some authors [33].

Fortunately, the sequential encoding of the AMO constraint can be extended effortlessly to enforce such ordering.

If we arrange the boolean variables $x_{j,1}, \ldots, x_{j,n}$ and $y_{j,1}, \ldots, y_{j,n-1}$ representing the placements of the j-th tile instance into a row for each $j \in \{1, \ldots, c_i\}$ (similarly to the arrangement in fig. 2.1), we can make two observations: Each row is shorter than the previous row by one variable, because of opt. 1. Subsequently, adding implications of the form $y_{j,i} \implies y_{j-1,i+1}$ enforces a condition similar to the first condition of the sequential encoding. For each *column* in the auxiliary variable matrix, there are a few $true$ values followed by $false$ values, which translates to "if a tile instance occupies a position on the board, all the previous instances reside on previous positions on the board". This clause structure is orthogonal to the structure that encodes the first sequential constraint.

We will call this tiling-to-SAT translation *AMO-ordered*. The boolean variables and clauses encoding a single tile with four instances and seven possible board placements are displayed in fig. 2.2. The scheme uses the same notation as fig. 2.1.

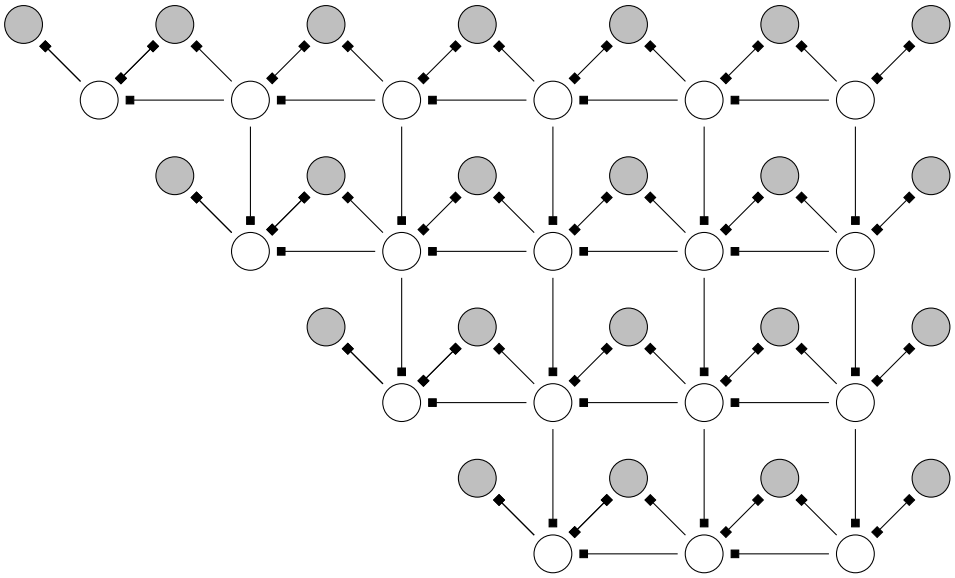


Figure 2.2: An example of the AMO-ordered encoding

2.4.3 AMK translation

Despite the AMO-ordered translation being arguably elegant, it has a massive flaw: The number of boolean variables required to encode all possible tile placements is multiplied by the tile instance count c_i, when in fact we only want to express that "the tile t_i is placed on the board at most c_i times". To encode such condition without a direct introduction of tile instances on the level of non-auxiliary variables, we need to use a more advanced encoding than AMO: the *at most k* encoding, which we will denote as $\mathrm{AMK}(\{x_1, \ldots, x_n\}, k)$.

With this new weapon in our arsenal, the tiling constraints described in section 2.4.1 can be encoded as follows (with the obvious modification of $x(v, d_x, d_y)$ and $r(v, d_x, d_y)$ which drops the concept of tile instances):

1. "Each unit square is covered by exactly one tile." Let s be a unit square of b and let $\mathrm{cover}(s)$ be the set of all variables $x(v, d_x, d_y)$ for which $r(v, d_x, d_y)$ covers the unit square s. The desired set of clauses is as follows:

$$F_1 = \bigwedge_{s \in b} \mathrm{EO}(\mathrm{cover}(s)) \tag{2.8}$$

2. "Tile t_i is placed on the board at most c_i times." Let $\mathrm{placements}(t_i)$ be the set of all variables $x(v, d_x, d_y)$ where v is a variant of t_i. The constraint is then equivalent to the following set of clauses:

$$F_2 = \bigwedge_{(t_i, c_i) \in T} \mathrm{AMK}(\mathrm{placements}(t_i), c_i) \tag{2.9}$$

PBLib delivers three AMK encodings: *BDD* [34], *cardinality networks* [35] and *auto* encoding (which tries to select the better of the former two for each input).

This approach supersedes opt. 1, so it is no longer needed (or even possible to encode). Opts. 2 and 3 are still in place.

2.4.4 Automated CNF symmetry breaking

Another potential improvement is breaking CNF symmetries generally. A CNF can be symmetric under permutation or negation of some variables. Breaking such symmetries can reduce the search space significantly, as we outlined in section 2.4.2.

One approach to breaking CNF symmetries in general is *static preprocessing*—analyzing the CNF formula before solving and adding a few clauses that break some or all CNF symmetries. One of the state-of-the-art symmetry breaking libraries is BreakID [36, 37, 38].

BreakID converts the formula to a graph, attempts to search for graph automorphisms and generates clauses that break them. This process induces some time overhead and eventually expands the CNF by the added clauses, but the SAT solving time can be greatly reduced for some input formulas thanks to this.

2.4.5 Implementation

Tiler's SAT approach implementation includes all the options examined in this section. This means that the user can select the SAT solver (either CaDiCaL or CryptoMiniSat), the symmetry breaking provider (either BreakID or none) and the translation type (either one of PBLib's AMO encodings, or the AMO-ordered translation, or one of PBLib's AMK encodings). This exposes 40 configuration combinations, all of them being available in Tiler.

2.5 Conversion to the ILP problem

Another famous problem that we will cover is *linear programming*, LP. A linear program is a set of real-valued variables, a set of linear equalities and inequalities which act as constraints that must be satisfied, and a linear objective function that is to be minimized or maximized. The linear program can either be infeasible (in that case, the objective function is of no importance), or the objective function is not bounded, or there is an assignment of values to the variables that achieves the optimal objective function value.

Additionally, if the variables are further limited to integer values, we call the problem *integer linear programming*, ILP. Terms *mixed integer programming* (only some variables are integers) and *binary integer programming* (the variables can only have values zero or one) are sometimes used, too.

Many optimization problems are linear in their nature and can thus be expressed as a linear program. Even though weakly polynomial algorithms such as the *interior-point method* have been developed, the *simplex method* is more efficient on some real-world problems, despite its exponential worst-case time complexity [39].

On the other hand, integer linear programming is NP-hard. For example, SAT can be converted to ILP by assigning a binary integer variable x_i to each boolean variable, constructing its boolean negation as $1 - x_i$ and representing each CNF clause by a linear inequality that sets the sum of the respective variables to be greater or equal to one (forcing at least one variable to have the value one). The resulting linear program is feasible if and only if the original boolean formula is satisfiable.

A commonly used ILP solving paradigm in practice is the *branch and bound* technique. Suppose we are solving a minimization problem. The search for a minimal solution is performed in recursive calls, traversing a virtual search tree. In each call, the integrality requirement is relaxed and the obtained LP instance is efficiently solved. This provides a lower bound for the objective function value in this branch, since no solution that obeys the integrality requirement can yield a better objective function value. If the lower bound is worse than a value achieved by a previously-known integer solution (or the LP is entirely infeasible), we can safely backtrack to the parent call. If the solution has integer values for all variables, we have a new minimal solution candidate. Otherwise, we select a variable and create new child branches by adding opposite linear inequalities, splitting the variable's available interval into two. When the search is completed, we either have a minimal solution, or we found no integer solution and the ILP instance is infeasible. The procedure can be fine-tuned in many ways (e.g. by careful selection of variables to branch on).

There are many advanced LP and ILP solvers on the market, such as an open-source solver Cbc [40] (developed under the auspices of the non-profit COIN-OR Foundation), or a commercial solver Gurobi [41].

2.5.1 Tiling problem translation

In section 2.4.3, we encoded the *at most k* constraints as boolean formulas. Such constraints are trivial to express as linear equalities and inequalities, so we can easily reformulate the AMK translation approach without the overhead caused by translating the constraints into boolean formulas.

Namely, each placement of each tile variant v is assigned a binary integer variable $x(v, d_x, d_y)$. Each constraint of the form EO(cover(s)) from eq. (2.8) can then be represented by the following linear equality:

$$\sum_{x_j \in \text{cover}(s)} x_j \quad = \quad 1 \tag{2.10}$$

Similarly, each constraint of the form AMK(placements(t_i), c_i) from eq. (2.9) can be rewritten as the following linear inequality:

$$\sum_{x_j \in \text{placements}(t_i)} x_j \quad \leq \quad c_i \tag{2.11}$$

In this case, the objective function is irrelevant, we are concerned solely by the feasibility of the resulting ILP instance. On the other hand, providing a meaningful objective function might help the ILP solver when computing the bound of the current branch. We suggest using the measure of board covering—the cumulative area of all tile variants placed on the board:

$$\sum_{v,d_x,d_y} |v| \cdot x(v, d_x, d_y) \tag{2.12}$$

If we task the ILP solver to minimize this value and the LP relaxation results in this value being larger than the board size, the solver can stop exploring the branch since it contains no solution.

Moreover, one can notice that when minimizing this value, the equality in eq. (2.10) can be replaced by inequality $\sum_{x_j \in \text{cover}(s)} x_j \geq 1$. When the solver deems the ILP instance feasible and returns the minimal value, we need to verify that the returned objective function value is equal to the board size. This value is equal to the sum of the left-hand sides of eq. (2.10), which guarantees that the other inequality direction in eq. (2.10) holds as well. In other words, if each unit square is covered by at least one tile instance and all the tile instances cover precisely the area of the board, then no unit square is covered by more than one tile instance.

All in all, we implemented the following tiling translation variants:

- `eq-ign` This is the original translation, with "= 1" in eq. (2.10) and no objective function.

- `eq-min` Equation (2.10) contains "= 1", but we also provide the objective function from eq. (2.12) for the solver to minimize.

- `geq-min` Equation (2.10) is weakened to "≥ 1" and we minimize the objective function from eq. (2.12).

- `leq-max` Being the opposite of `geq-min`, eq. (2.10) includes "≤ 1" and we task the solver to maximize the objective function from eq. (2.12).

As in the AMK translation, we can apply opts. 2 and 3 (opt. 1 is not relevant, since the tile instances are not encoded explicitly).

2.5.2 Implementation

We interfaced Tiler with both Cbc and Gurobi. Due to Gurobi being proprietary software (although free for academic use), building and using Tiler with Gurobi support requires Gurobi to be installed on the hosting operating system.

Both Cbc and Gurobi provide many configuration options. Tiler supports running the ILP solvers either with the default settings, or with a set of options determined empirically by us on a few tiling problems. For the purpose of tuning the solver settings, Gurobi features a convenient tool *grbtune* that tries several configurations and recommends the best ones.

To sum up, the user of Tiler is able to select the desired ILP solver, the default or adjusted solver settings and one of the tiling translation variants listed above.

2.6 Conversion to the CSP problem

The last domain we will examine is the realm of *constraint satisfaction problems*, CSP. This term encapsulates many kinds of satisfaction problems over a set of variables and constraints. The available constraint types vary in different contexts, but in general, they are more diverse than boolean and linear constraints that define the SAT and ILP problems discussed in previous sections.

Since the wider constraint palette provides greater flexibility and enables elegant expressing of many problems, as a consequence, there are many ready-to-use CSP modeling and solving tools. Arguably one of the most advanced toolkits is the MiniZinc modeling language and solver interface [42, 43]. The MiniZinc language allows defining boolean, integer and floating-point constants and variables, as well as arrays and sets of constants and variables. It also offers many high-level constraints, ranging from simple counting constraints to high-level scheduling and graph constraints. It supports expressions with universal or existential quantifiers, too.

The MiniZinc executable's input comprises a model file, which declares constants, variables and constraints, and a data file, which defines the constant values (alternatively, they can be inlined in the model file). After model and data validation, MiniZinc performs a step called *flattening*—compiling the model into a low-level language FlatZinc. The FlatZinc file is then passed to a CSP solver that supports the FlatZinc format. The MiniZinc binary package bundles several such solvers, e.g. Gecode [44], Chuffed [45] and Cbc [40], or the user can pass the file to a third-party FlatZinc-capable solver. When the solver manages to assign values to all variables, the values are returned to MiniZinc and subsequently presented to the user in an output format that can be specified in the model file.

The main advantage of this approach is the possibility to add new MiniZinc constraint

definitions without the need to update the underlying CSP solvers. Conversely, adding a new solver requires only implementing the limited number of FlatZinc features instead of the entire MiniZinc interface.

2.6.1 Tiling problem translation

Thankfully, MiniZinc exposes the `geost` constraint proposed by Beldiceanu et al. [46], which receives object shape definitions and ensures that no two objects overlap. It is designed to work in any number of dimensions; however, we are interested only in its two-dimensional form.

The `geost` constraint can be used to encode a variety of packing problems and polyomino tiling surely belongs among them. When using it, we must provide the following input data:

- Rectangle sizes. We define the shapes by decomposing them into rectangles, so this array holds the rectangle sizes. To simplify the translation, we will decompose the shapes into unit squares, so all sizes will be 1×1.
- Rectangle positions. The offsets of the rectangles (or in our case, unit squares) from the origin of the coordinate system. We list all unit squares of the board's bounding box—if there is a tile variant that would require more unit squares in some direction, it would not fit inside the board, and therefore can be ignored. Some of the unit squares may be unused, but it should not affect the solving performance noticeably.
- Shape definitions. Each shape is defined as a set of rectangles (specified as indices of rectangles in the previous two arrays). In our translation, we create a shape for each tile variant.

Besides the input data, `geost` takes two arrays of variables describing the geometric objects, which are then restricted by `geost` to forbid overlaps (so they serve as a 'return value'):

- Object shapes. An array of integer variables, where each variable represents a tile instance and is assigned a shape (index of a shape in the array of shape definitions).
- Object positions. An array of coordinate variable pairs, where each pair is assigned offsets from the origin of the coordinate system.

Apart from utilizing `geost` to prohibit object overlaps, we need to enforce two more restrictions. Firstly, we must ensure that the tile instance count c_i is obeyed for each tile (since `geost` could assign e.g. the same shape to all objects). This can be easily achieved by providing the allowed shapes for each object (i.e. tile instance) and adding a `forall` constraint to state that for each object, its shape must be one of the allowed ones.

Secondly, we need to encode the board shape somehow, since the `geost` constraint by itself could scatter the objects anywhere over the virtually infinite plane. For this purpose, we can employ the `geost_bb` modification of `geost`, which also forces all objects to be placed inside of a bounding box with the specified width w and height h. If our board has a rectangular shape, using `geost_bb` by itself is enough. If the board is not a rectangle, we need to ensure that the $w \cdot h - |b|$ unit squares missing from the board's bounding box will remain empty. We can achieve this by adding $w \cdot h - |b|$ new objects and fixing their shape to a single unit square and their positions to match the empty unit square positions.

This trick allows us to express that all unit squares are covered implicitly by simply enforcing that all tile instances are placed on the board. Unfortunately, it brings a disadvantage—this approach is applicable only to tiling problems with no redundant tile instances.

Regarding optimizations, the translation could be enhanced by adding constraints to encode opt. 1 (or even stronger, by enforcing the order of all instances for each tile); opt. 2 is not applicable to this approach; and opt. 3 is included in the fact that this approach supports *only* problems where all tile instances are used. Additionally, decomposing the shapes into rectangles instead of unit squares could improve the performance, too.

Alternative translations

The world of constraint solving is vast and therefore, it is no surprise that there are other ways to model polyomino tiling problems. Apart from models that essentially recreate the approaches we used in the previous section, there is at least one unique strategy worth mentioning, proposed by Lagerkvist and Pesant [47, 48]. They present a clever encoding of tiling problems as regular expressions, which can be solved by many CSP solvers, including MiniZinc. We believe that this approach deserves a closer look in future approach comparisons.

2.6.2 Implementation

We implemented the MiniZinc model described in section 2.6.1, utilizing the `geost_bb` constraint and decomposing shapes into unit squares. The model file can be found in appendix A in the `solvers/csp/minizinc_models.hpp` file. Although MiniZinc is intended to be used as a standalone command line tool, its authors made some effort to enable utilizing it as a callable library in C++ programs [49]. However, the library still requires reading many files with MiniZinc constraint definitions as the first step when solving, which adds a constant time overhead in the order of fractions of a second.

As the underlying CSP solver, Tiler currently supports Gecode, Chuffed and Gurobi.

Chapter 3

Results

In this chapter, we will present a hand-picked set of tiling problem instances and use them as a benchmark for measuring the speed of execution of our algorithms. Based on the results, we will select one solver configuration for each solving approach to represent the entire group of approach configurations. Finally, in sections 3.3 and 3.4, we will compare the performance of the representants on two interesting groups of problem instances—instances with many unique tile shapes and instances that can be proved to be unsolvable by coloring invariants.

3.1 Benchmarking problem instances

There are many tiling instance classes: Trivial ones, such as tiling a small board by the monomino; bigger instances with a compact board and many different tile shapes; intricate boards with ad-hoc tiles; instances with an elegant proof of the tiling non-existence; non-tileable instances where no such proof is known; and so on. When selecting problems, we strived to capture a wide range of problem types, but also to provide a problem instance in several sizes where possible (usually by scaling the board size). We included some instances published by Golomb [12] and Martin [50] (a detailed list of these instances can be found in `README.md` in appendix A). Altogether, we collected more than 200 problem instances. Each instance is stored in a file under the `problems` directory. The instances are grouped into subdirectories by their tile set characteristics and their file names hint at the board shape or other instance attributes. In the following text, we will identify problem instances by their file paths. As a visual aid, each file name ends with either `_s` if the instance is solvable or `_u` if it is unsolvable.

To streamline the benchmarking process, we used the *Benchmark* library created by Google [51]. It handles measuring the execution time for each problem/solver combination (that can be filtered conveniently by a regular expression) and repeats the measurement several times for enhanced accuracy if the execution time is too short. It

saves all the measurement data to a JSON file and continuously prints the results to the console.

We also created a Python script to either visualize the measured data as interactive plots or save the plots as image files. The JSON data file and several plots are attached in appendix B. Due to the exponential nature of tiling problems, the y-axis with execution durations uses a logarithmic scale.

3.2 Selecting approach representants

In order to compare the approaches efficiently, we will review each solving approach and select a solver configuration to represent it. We will discuss the differences between configurations in each group briefly. Appendix B provides the plots corresponding to each of the following subsections.

3.2.1 Simple backtracking and Algorithm X

Currently, Tiler implements four backtracking configurations, differing in the tile selection strategy: the *default*, which follows the input tile order, and the *frequent first*, *rare first* and *fill top row* strategies. On most problem instances in our collection, the non-default strategies bring no benefit—for example, if each tile has only one available instance, all the strategies except for *fill top row* behave the same. In fact, the non-default strategies are almost always slower than *default* by some factor, because they waste a portion of their execution time on selecting the next tile (essentially, sorting the tiles in each recursive call), whereas the *default* strategy omits this step entirely. Even though there are problems where a non-default strategy helps significantly (e.g. `mixed/1-3I-11x11_s`), we decided to select the `simple_default` solver configuration that implements the *default* strategy as the representant of our simple backtracking approach, since it has the best performance on almost all instances that we examined.

The Algorithm X approach (labeled as `dlx` based on Knuth's DLX abbreviation of the *dancing links* technique) usually performs similarly to `simple_default`. There are, however, problem instances where one or the other performs much better: Most notably, `dlx` always tries to cover the unit square with the fewest possible tile instance placements that can cover it, which enables `dlx` to solve the instances in the `3L/L-02` family easily, whereas `simple_default` examines many more unviable placements. To the contrary, tiles with higher tile instance counts are unnatural for `dlx` and introduce unnecessary symmetry, so problem instances such as `mixed/40s-4T-06x06_u` are better suited for `simple_default`.

Because of the differences, we will not collapse `simple_default` and `dlx` into a single representant, we will evaluate them separately instead.

3.2.2 Conversion to the SAT problem

Before we dive into comparing the configurations, it should be noted that on the instances with a single tile shape and different board sizes, the execution times of an individual configuration always form a smooth, increasing curve, so it is easy to compare the configurations with each other. However, on problem instances with no size scaling, the performances are rather unstable and there is usually no clear winner among the configurations. We will try to select an overall good configuration to represent the SAT approach.

As described in section 2.4.5, we implemented 10 different translation types: AMO translations (`amo-auto`, `amo-seq`, `amo-bimander`, `amo-commander`, `amo-kproduct` and `amo-binary`), the AMO-ordered translation, and the AMK translations (`amk-auto`, `amk-bdd` and `amk-card`). In general, AMO translations perform worse then AMK translations on problem instances with tile counts bigger than one (as expected). This difference is wiped by opt. 2 on problem instances where it is applicable. Even though there are a few cases where an AMO translation performs exceptionally well (such as the `sat_cms_no-breaker_amo-commander` solver on `many-unique/4r-5r-04x20'_s`), they are rather inferior on other problem instances, therefore we believe that `amk-card` is a "safe bet" when selecting the SAT representant.

Regarding the underlying SAT solver selection, CaDiCaL and CryptoMiniSat perform comparably on most problem instances.

To our surprise, adding the BreakID symmetry breaker harmed the performance of all configurations. Some translation types were more affected than others, but based on a few manual experiments conducted by us, the pattern seems to be always the same: BreakID spends some time searching for symmetries, then it either adds a few symmetry-breaking clauses or not, and then SAT solving occurs, with no performance benefit when compared to the alternative without BreakID. It is not clear to us what is the culprit of this issue (e.g. whether tiling problems are not a good fit for BreakID in general, or some other reason), but without further investigation, using configurations without BreakID is always a better idea.

Given the observations above, we selected `sat_cadical_no-breaker_amk-card` as the SAT representant.

3.2.3 Conversion to the ILP problem

Like the SAT configurations, the ILP configurations tend to have unstable performance on some problem instances; for example, setting a different seed for the solver's random number generator can affect the execution time massively. However, the Gurobi solver appears to suffer from this instability less than Cbc, which can be seen mainly on the instances from the `many-unique` class. Additionally, Gurobi's execution durations are

almost always lower than Cbc's, in line with Gurobi's better expected performance due to its commercial funding. Both solvers have a constant time overhead of approximately one millisecond on all problem instances (including trivial ones).

From the four translation variants we implemented—`eq-ign`, `eq-min`, `geq-min`, and `leq-max`—the variants with equality perform better most of the time, so providing more information to the solver turns out to be helpful once again. The `eq-ign` and `eq-min` variants are almost identical in their performance.

The last attribute of the ILP configurations is whether to keep the default solver parameters or choose the adjusted ones. Using the adjusted parameters with Cbc helps on some problem instances but does the opposite on other instances. Searching for the ideal parameter adjustments is therefore a tedious and tricky task. On the other hand, Gurobi provides the *grbtune* tool to automate the process, so using the parameters recommended by this tool either preserves or improves the execution times consistently, but only by a relatively small factor (this may however mean that the Gurobi's default parameters are already excellent for our problem).

The `ilp_gurobi_adjusted_eq-ign` now seems to be a solid representant of the ILP approach.

3.2.4 Conversion to the CSP problem

As we discussed in section 2.6, MiniZinc's design allows for great flexibility on both sides: Its users can utilize the expressive MiniZinc modeling language and the broad palette of available constraints, and the CSP solver developers need only to support the primitive FlatZinc input language. The central step performed by MiniZinc is the translation between the MiniZinc and FlatZinc languages.

Unfortunately, this approach turned out to be unfit for our tiling model. For many problem instances, the MiniZinc-FlatZinc translation took an unreasonable amount of time and the generated FlatZinc model was immense. To illustrate the scale of this issue, appendix A provides the `csp-demo` directory with a set of example files. The example problem instance is a 3×12 board with 10 small tiles, which is solved by all other solving approaches in tens of milliseconds. Sadly, the flattening process lasts tens of seconds instead, and the produced FlatZinc file contains tens of thousands of variables and constraints.

Even if we decompose the tiles into rectangles instead of unit squares (as mentioned in section 2.6.1; the revised model is attached in `csp-demo/rectangles.mzn`), the terrible execution time improves only by a factor of two. We did not discover an easy way how to improve the model significantly. The `geost` constraint implementation uses nested `forall` statements heavily to ensure no overlaps, which could explain the size of the generated FlatZinc file and possibly the long flattening duration.

Even though some CSP solvers are able to solve the massive FlatZinc model reasonably quickly (e.g. Chuffed can solve the example problem instance in approximately two seconds), the slow flattening step effectively disqualifies this approach from our benchmarks. Further investigation would be necessary to evaluate whether it is possible to make MiniZinc viable when solving polyomino tiling problems.

3.3 Case study: Many unique tiles

Perhaps the most commonly presented polyomino instances are the ones we will focus on in this section: Their tiles have many different polyomino shapes, each of them being available as a single tile instance. Furthermore, the boards of these problem instances are typically rectangles, skewed rectangles, or other visually appealing shapes, and the instances are usually solvable. A solution of one such instance is pictured in fig. 3.1.

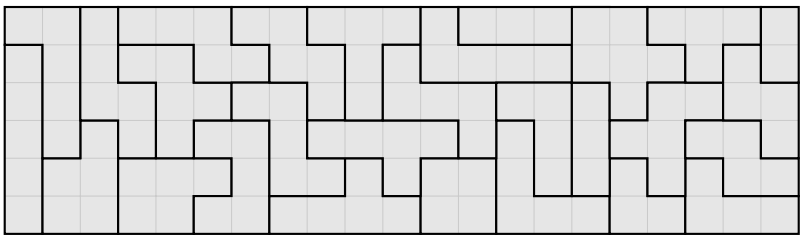


Figure 3.1: A solution of `many-unique/2-3-4-5-06x21_s` by the `dlx` solver

In section 3.2, we selected the four approach representants: `simple_default`, `dlx`, `sat_cadical_no-breaker_amk-card`, and `ilp_gurobi_adjusted_eq-ign`. Figure 3.2 displays their execution times on the problem instances selected for this comparison. It is easy to notice that `simple_default` and `dlx` are visibly correlated and on the same performance level, making them almost interchangeable in this context. On the other hand, the SAT and ILP solvers are slower roughly by a factor of 100 on most instances. This difference can be explained by a few speculations. Most likely, the reason for the slowdown is the fact that `simple_default` and `dlx` carry out the search quite efficiently, so SAT and ILP can perform a similar search at best, but they bear an approach-specific overhead. For instance, a unit square being covered by exactly one tile is intrinsic to `simple_default` and `dlx`, but the SAT solver has to work through many clauses that encode the EO constraint.

Interestingly, the three unsolvable problem instances were considerably harder for `simple_default` and `dlx` (since there was no way for them to quickly find one of the solutions), but posed no particular obstacle to the SAT and ILP solvers, so the performances of the four approach representants were closer to each other on these problem instances.

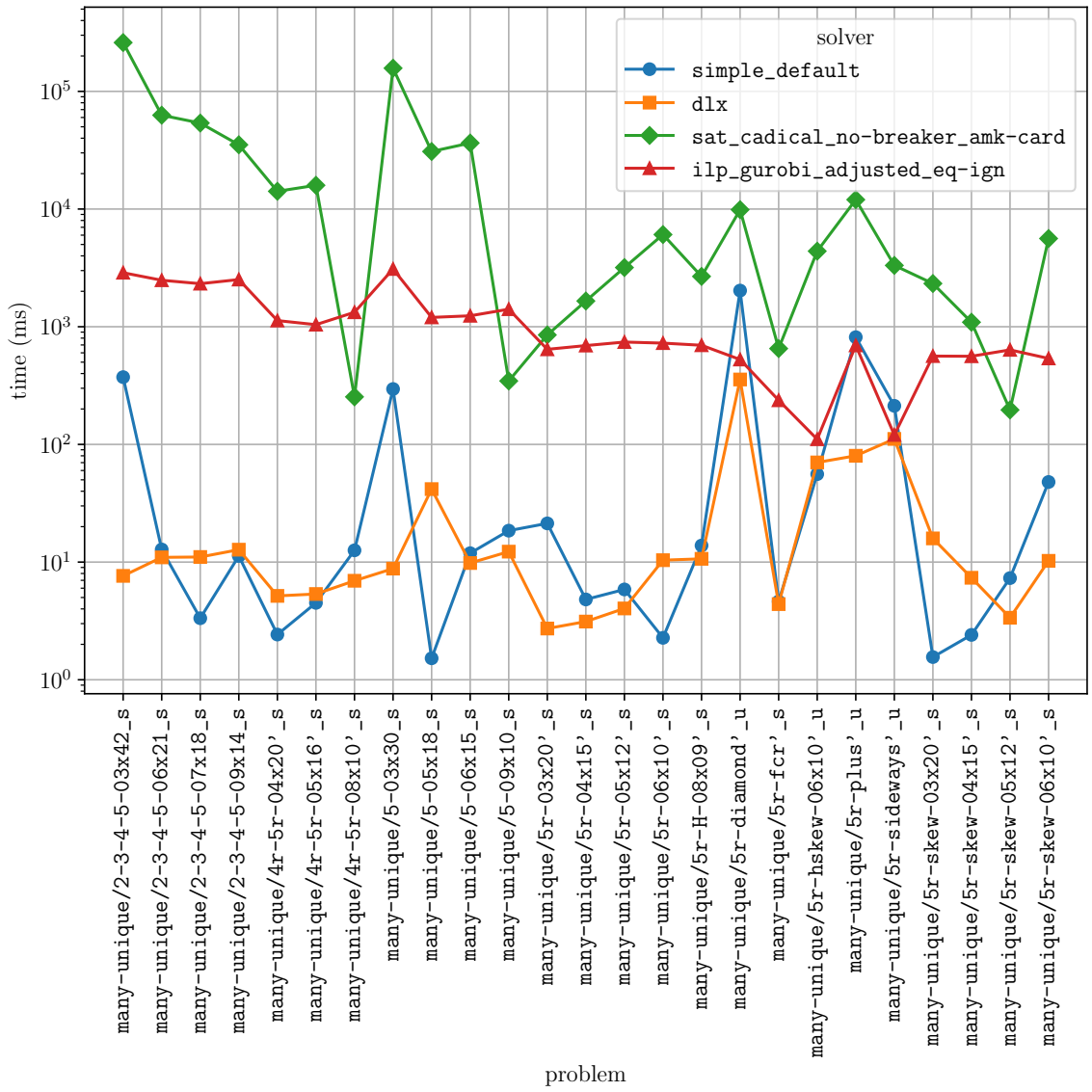
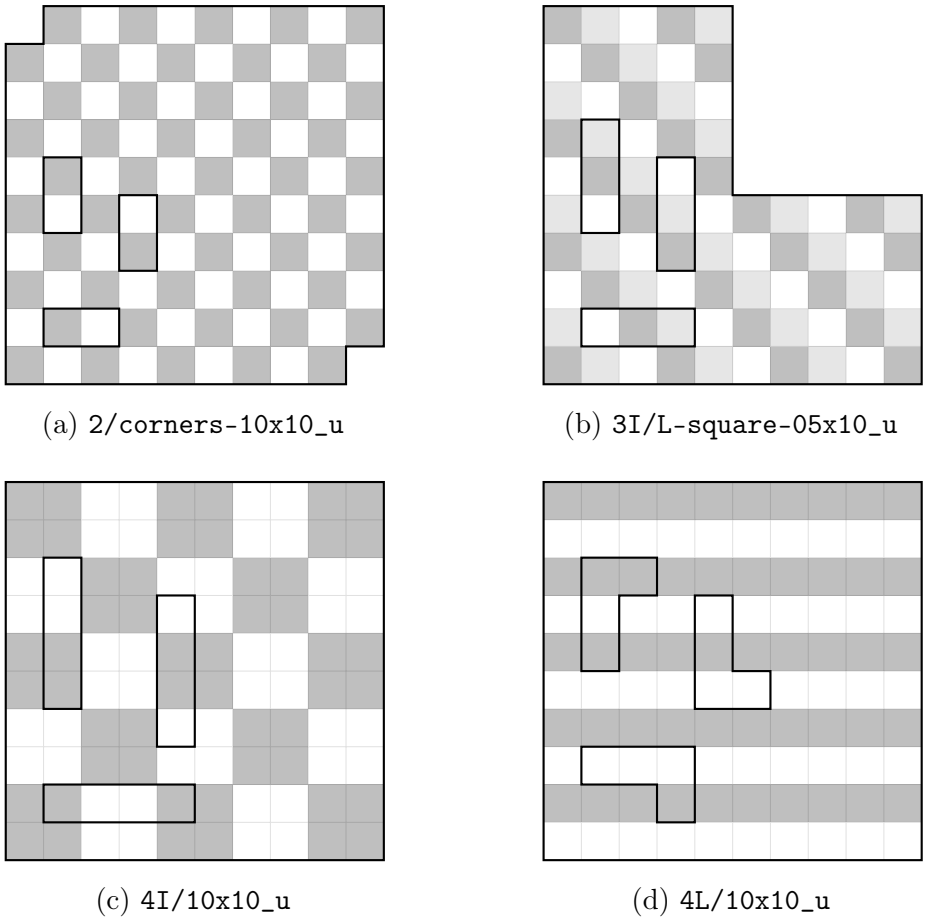


Figure 3.2: Many unique tiles: the benchmarking results

3.4 Case study: Coloring invariants

Another fascinating group of polyomino tiling instances are the unsolvable instances with coloring proofs. These instances usually include only one or two tile shapes. A generally known instance is the *mutilated chessboard problem* (`2/corners-10x10_u`), which asks whether dominoes can tile a 10×10 chessboard[1] without two diagonally opposite corner unit squares. The board is large enough to impair any kind of backtracking search, but fortunately, there is another approach we can utilize: If we apply the classic checkerboard pattern to the board, every domino placement will cover exactly one black and one white unit square (this is the so-called coloring invariant), as shown in fig. 3.3a. This implies that the number of black unit squares that are covered will always be

[1] Other even dimensions could be used instead.



(a) `2/corners-10x10_u` (b) `3I/L-square-05x10_u`

(c) `4I/10x10_u` (d) `4L/10x10_u`

Figure 3.3: Coloring invariants of selected tiling instances

equal to the number of white unit squares that are covered. However, the number of the board's black and white unit squares is different (since the two removed corner unit squares had the same color), which means that the board cannot be tiled by dominoes.

There are many other instances where coloring arguments are applicable. In our comparison, we included three more instances, all of them being displayed in fig. 3.3. The `3I/L-square-05x10_u` instance can be proved unsolvable by using three colors in a diagonal pattern. As before, each tile covers one unit square of each color regardless of the tile's position and rotation, but the board holds different numbers of unit squares of each color (in fig. 3.3b, the unit square counts are 26, 25, and 24).

Another colorable instance is `4I/10x10_u`, which can be solved by using a checkerboard pattern with squares of size 2×2 unit squares. This coloring produces a different number of black and white unit squares, while the tiles can still cover only the same number of black and white unit squares.

The last instance we will consider is slightly different from the previous ones. The `4L/10x10_u` instance uses a coloring consisting of black and white stripes, so the number of black and white unit squares on the board is the same this time. The coloring is not exactly an invariant with regards to the "L"-shaped tile: Each tile placement covers

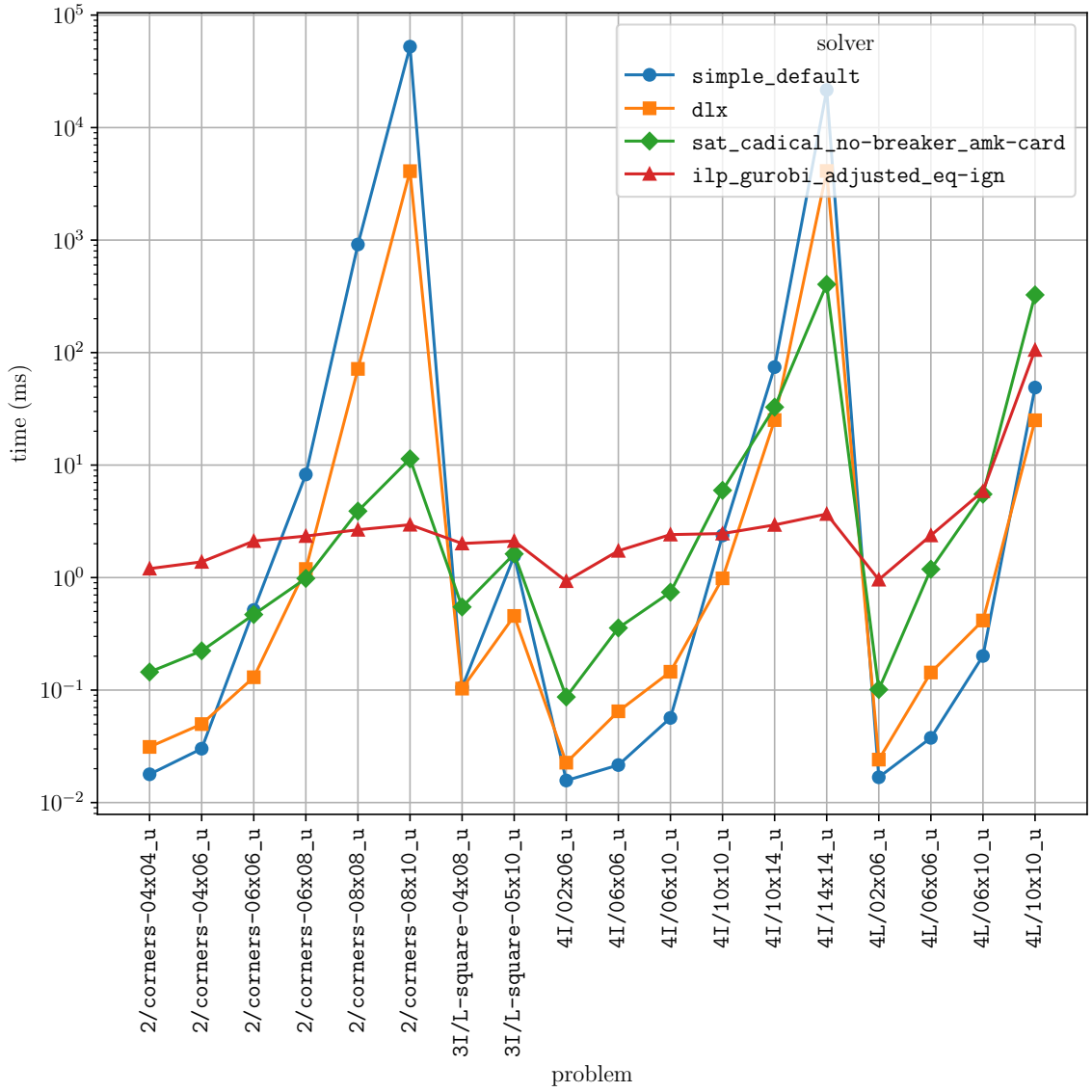


Figure 3.4: Coloring invariants: the benchmarking results

either three black and one white unit square, or the other way around. If we managed to place half of the tiles such that they cover three black unit squares and the other half of the tiles to cover three white unit squares each, there would be no contradiction to harvest. However, the number of tile instances is odd (25 in the case of a 10×10 board), which means that the tiles will necessarily cover a different number of black and white unit squares, and the problem instance is indeed unsolvable.

For each of the four instance types, we included several board sizes. We could only use the sizes 4×8 and 5×10 for the `3I/L-square` family, because 6×12 is trivially solvable and 7×14 is too large for most solvers. The benchmarking results of the approach representants are shown in fig. 3.4.

The performance of `simple_default` and `dlx` solvers matched our expectations: Their execution times grew quickly with the increasing board size in an exponential

manner.

The SAT approach had similar characteristics, although it had a less steep growth asymptote (at least on the first and third instance family). While further investigation would be needed to pinpoint the reason for this, it shows us that there are problem instances better suited for SAT than for `simple_default` or `dlx`, in contrast with the results in section 3.3.

Strikingly, the ILP solver exhibited fundamentally different results. Despite its longer startup time, its performance on bigger board sizes of the first three instance families was still in the order of milliseconds!

After inspecting Cbc and Gurobi solver logs, it turns out that they never execute the *branch and bound* search procedure—they claim the problem instance infeasible after solving the initial relaxed linear program.

This leads us to the explanation: The linear relaxation of our integer linear program allows the tile instances to be placed partially on several different board positions (for example, an instance could be placed on a position with the weight of 0.8 and on a different position with the weight of 0.2). However, the coloring argument holds even for this relaxed problem. Each tile instance still covers the same black area and white area, which is in contradiction with the board having different numbers of black and white unit squares. Since the relaxed problem is infeasible, the ILP solvers can detect the infeasibility quickly and declare the entire problem unsolvable.

Thanks to this, even instances with a 100×100 board (which translates to thousands of variables and constraints) can be solved in seconds or less.

Sadly, this impressive performance is not retained on the fourth problem instance family. Indeed, if we allow non-integer placements, the odd number of tile instances is of no use, since one tile could be equally much on two positions, one predominantly black and one predominantly white. Because of this, the ILP solvers must perform the *branch and bound* procedure, which results in their performance being comparable to the other three solving approaches.

Conclusion

We implemented five solving approaches (simple backtracking, Algorithm X, and conversions to SAT, ILP and CSP) in several configurations each. We created a collection of tiling instances and used them to compare the performance of the solving approaches.

Overall, the execution times depend strongly on the specific problem instance. The CSP approach appears to be too inefficient due to the design of the MiniZinc toolkit that we decided to use. The traditional approaches—simple backtracking and Algorithm X—perform better than SAT and ILP on problem instances with many unique tile shapes. On the other hand, translation to SAT often outperforms the traditional approaches on unsolvable instances with a single tile shape, and ILP solvers even manage to discover coloring invariants on some instances and thus solve them instantly.

Our work provides several opportunities for further research: One could focus on any particular approach and explore the possible ways how to optimize it even more. The CSP approach deserves special attention since the world of constraint solving offers many other tools and methods that could perhaps result in a much better performance.

Ultimately, we identified at least one tiling instance with a coloring invariant that cannot be exploited by an ILP solver. Such instances are easy to solve for a human (thanks to the coloring invariants), but none of our solvers is capable enough to solve them quickly. Therefore, crafting an algorithm that could recognize more coloring invariants would be a great contribution to our portfolio of solving approaches.

Bibliography

[1] M. Brand. Small polyomino packing. *Information Processing Letters*, 126:30–34, June 2017.

[2] H. R. Lewis. Complexity of solvable cases of the decision problem for the predicate calculus. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 35–47, 1978.

[3] C. Moore and J. M. Robson. Hard tiling problems with simple tiles. *Discrete & Computational Geometry*, 26(4):573–590, January 2001.

[4] E. Rémila. Tiling a simply connected figure with bars of length 2 or 3. *Discrete Mathematics*, 160(1-3):189–198, 1996.

[5] R. M. Robinson. Undecidability and nonperiodicity for tilings of the plane. *Inventiones Mathematicae*, 12(3):177–209, September 1971.

[6] G. Fredericks. Polyomino Tiler. `https://gfredericks.com/things/polyominoes`. Retrieved May 5, 2022.

[7] C. Meadors. Polyomino Solver. `https://cemulate.github.io/polyomino-solver`. Retrieved May 5, 2022.

[8] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.4)*, 2021. `https://www.sagemath.org`.

[9] S. Labbé. Tiling Solver. Retrieved May 5, 2022, documentation available at `https://doc.sagemath.org/html/en/reference/combinat/sage/combinat/tiling.html`.

[10] J. Grahl. POLYOMINO – Python package for polyomino tiling problems. `https://github.com/jwg4/polyomino`. Retrieved May 5, 2022.

[11] D. Knuth. Dancing links. *Millenial Perspectives in Computer Science*, pages 187–214, 2000.

[12] S. W. Golomb. *Polyominoes: Puzzles, patterns, problems, and packings*. Princeton University Press, 2nd edition, 1994.

[13] I. Jensen and A. J. Guttmann. Statistics of lattice animals (polyominoes) and polygons. *Journal of Physics A: Mathematical and General*, 33(29):257–263, July 2000.

[14] D. H. Redelmeier. Counting polyominoes: Yet another attack. *Discrete Mathematics*, 36(2):191–203, 1981.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] D. Knuth. *The art of computer programming, volume 4, fascicle 5*. Addison-Wesley, 2019.

[17] D. Knuth. DLX1 – Algorithm 7.2.2.1X for exact cover via dancing links. `https://www-cs-faculty.stanford.edu/~knuth/programs.html`. Retrieved May 5, 2022.

[18] B. Lynn. The DLX Library. `https://github.com/blynn/dlx`. Retrieved May 5, 2022.

[19] G. Karanikas. Dancing Links and Sudoku. `https://github.com/gkaranikas/dancing-links`. Retrieved May 5, 2022.

[20] A. Biere, K. Fazekas, M. Fleury, and M. Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, and M. Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[21] A. Biere. CaDiCaL Simplified Satisfiability Solver. `https://github.com/arminbiere/cadical`. Retrieved May 5, 2022.

[22] M. Soos, K. Nohl, and C. Castelluccia. Extending SAT solvers to cryptographic problems. In O. Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.

[23] M. Soos. CryptoMiniSat SAT solver. `https://github.com/msoos/cryptominisat`. Retrieved May 5, 2022.

[24] K. Claessen, N. Een, M. Sheeran, and N. Sorensson. SAT-solving in practice. In *2008 9th International Workshop on Discrete Event Systems*, pages 61–67, 2008.

[25] A. Biere, V. Ganesh, M. Grohe, J. Nordström, and R. Williams. Theory and Practice of SAT Solving (Dagstuhl Seminar 15171). *Dagstuhl Reports*, 5(4):98–122, 2015.

[26] C. Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming - CP 2005*, pages 827–831, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[27] W. Klieber and G. Kwon. Efficient CNF Encoding for Selecting 1 from N Objects. In *Proceedings of the Fourth Workshop on Constraint in Formal Verification*, 2007.

[28] S. Hölldobler and V. Nguyen. An Efficient Encoding of the at-most-one Constraint. In *The Twelfth International Workshop on Constraint Modelling and Reformulation*, 2013.

[29] J. Chen. A New SAT Encoding of the At-Most-One Constraint. In *The 9th International Workshop on Constraint Modelling and Reformulation*, 2010.

[30] A. M. Frisch, T. J. Peugniez, A. J. Doggett, and P. Nightingale. Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. *Journal of Automated Reasoning*, 35(1-3):143–179, 2005.

[31] T. Philipp and P. Steinke. PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 9–16. Springer International Publishing, 2015.

[32] P. Steinke, R. Černoch, and M. Hořeňovský. pblib: Encoding Pseudo-Boolean Constraints into CNF. `https://github.com/master-keying/pblib`. Retrieved May 5, 2022.

[33] M. Björk. Successful SAT Encoding Techniques. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:189–201, 07 2009.

[34] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. BDDs for Pseudo-Boolean Constraints — Revisited. In *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing*, volume 6695 of *Lecture Notes in Computer Science*, pages 61–75. Springer International Publishing, 2011.

[35] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In *19th International Conference on Principles and Practice of Constraint Programming*, CP'13, 2013.

[36] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. Improved static symmetry breaking for SAT. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 104–122. Springer International Publishing, 2016.

[37] J. Devriendt and B. Bogaerts. BreakID 2.5. `https://bitbucket.org/krr/breakid`. Retrieved May 5, 2022.

[38] J. Devriendt, B. Bogaerts, and M. Soos. BreakID 3.0. `https://github.com/meelgroup/breakid`. Retrieved May 5, 2022.

[39] G. B. Dantzig and M. N. Thapa. *Linear programming*. Springer series in operations research. Springer, 1997.

[40] J. Forrest, T. Ralphs, H. G. Santos, S. Vigerske, et al. coin-or/Cbc: Version 2.10.5. `https://doi.org/10.5281/zenodo.3700700`, March 2020. Retrieved May 5, 2022, source code available at `https://github.com/coin-or/Cbc`.

[41] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. `https://www.gurobi.com`, 2022. Retrieved May 5, 2022.

[42] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In C. Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543. Springer Berlin Heidelberg, 2007.

[43] G. Tack, P. J. Stuckey, et al. MiniZinc. `https://github.com/MiniZinc/libminizinc`. Retrieved May 5, 2022.

[44] Gecode Team. Gecode: Generic Constraint Development Environment. `https://github.com/Gecode/gecode`. Retrieved May 5, 2022.

[45] G. Chu, P. J. Stuckey, A. Schut, T. Ehlers, G. Gange, and K. Francis. Chuffed, a lazy clause generation solver. `https://github.com/chuffed/chuffed`. Retrieved May 5, 2022.

[46] N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A Generic Geometrical Constraint Kernel in Space and Time for Handling Polymorphic k-Dimensional Objects. In C. Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 180–194. Springer Berlin Heidelberg, 2007.

[47] M. Z. Lagerkvist and G. Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints BPCC'08*, 2008.

[48] M. Z. Lagerkvist. State Representation and Polyomino Placement for the Game Patchwork. arXiv e-prints, 2020.

[49] G. Tack. `libmzn`: A modular CP infrastructure based on MiniZinc. In *MZN'11, first international MiniZinc workshop*, 2011.

[50] G. E. Martin. *Polyominoes: A Guide to Puzzles and Problems in Tiling*. Mathematical Association of America, 1991.

[51] Google Open Source. Benchmark. `https://github.com/google/benchmark`. Retrieved May 5, 2022.

Appendix A

Tiler source code

The electronic attachment contains the source code of the Tiler polyomino tiling solver and a collection of problem instances. Tiler's dependencies are not included; the steps how to obtain them and build Tiler are provided in `README.md`.

The same code is also published on `https://github.com/davidmisiak/tiler`.

Appendix B

Benchmarking results

The electronic attachment contains the benchmarking measurements in the JSON format produced by Google's Benchmark library. We ran all solver configurations (except for the CSP approach, see section 3.2.4 for explanation) against all problem instances collected in appendix A. Furthermore, we visualized the measured execution times (displaying a related group of solvers at a time) and attached the plots as PDF files.