UNIVERSITÄT BASEL

SAT Encodings vs. Dancing Links for Polyomino Tiling

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Computer Science
Research Group Artificial Intelligence
ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Tanja Schindler

Jan Karol Jungfer
jan.jungfer@stud.unibas.ch
22-813-828

30.10.2025

Abstract

This work investigates the application of Boolean satisfiability (SAT) solvers to the exact cover problem, with a particular focus on polyomino tiling. While the Dancing Links (DLX) algorithm is a well-established method for solving exact cover instances efficiently, modern SAT solvers provide an alternative framework. We present several SAT encodings tailored to polyomino tiling, including direct, Bimander, and cell-based encodings. The performance of these encodings is evaluated and compared against DLX.

Table of Contents

1 Introduction

The exact cover problem is a classical combinatorial problem with numerous applications in areas such as scheduling [1], set partitioning, and tiling. While there are many unique ways of solving an exact cover problem, even such as biocomputation [2]. The most prominent algorithm remains Donald Knuth's Dancing Links (DLX)[3], which systematically explores all possible coverings.

An alternative approach is to reduce instances of exact cover problems to Boolean satisfiability (SAT) formulations, thereby leveraging the efficiency of modern SAT solvers. Polyomino tiling, illustrated in Figure 1.1, involves covering a given surface completely with polyominoes without overlaps. This can be naturally expressed as an exact cover problem. Focusing specifically on polyomino tiling allows for the development of tailored SAT encodings that exploit structural characteristics of the problem, potentially improving solver performance. This work investigates several SAT encodings for polyomino tiling, including direct, Bimander, and cell-based encodings, and evaluates their performance relative to DLX. Chapter 2 introduces the necessary background on the exact cover problem, polyomino tiling, and SAT. Chapter 3 describes the SAT encodings and their implementation. Chapter 4 presents the experimental evaluation and a comparison of the different approaches in which we find that DLX outperforms all three examined encodings.

Similar findings have been brought by [4].

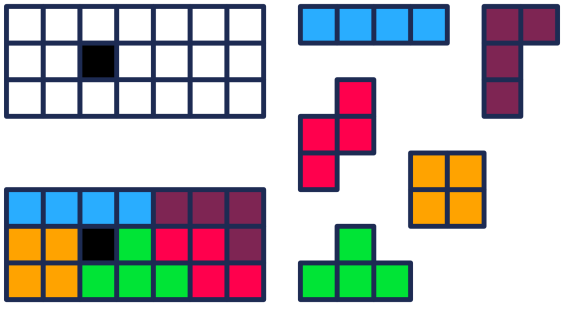

Figure 1.1: Polyomino tiling

2

Background

2.1 Exact Cover Problem

The exact cover problem is an NP-complete problem [5] that will be defined here.

Definition 1 (Cover)**.** *Let X be a finite set and let $S \subseteq \mathcal{P}(X)$. A* cover *of X is a subset $S^* \subseteq S$ such that*

$$\bigcup_{E \in S^*} E = X.$$

Definition 2 (Exact Cover)**.** *An* exact cover *of X is a cover $S^* \subseteq S$ such that the sets in S^* are pairwise disjoint, i.e.,*

$$E_i \cap E_j = \emptyset \quad \textit{for all distinct } E_i, E_j \in S^*.$$

Example: Let $X = \{a, b, c, d\}$ and $S = \{\{a, b\}, \{a, b, c\}, \{c, d\}\}$. Then $\{\{a, b, c\}, \{c, d\}\}$ is a cover but not an exact cover of X because c is covered twice. On the other hand, $\{\{a, b\}, \{c, d\}\}$ is an exact cover of X.

Definition 3 (Exact Cover Problem)**.** *The* exact cover problem *is the problem of finding an exact cover $S^* \subseteq S$ for a given set X and a collection of subsets S.*

2.2 Polyomino tiling

Polyomino tiling is a classic application of the exact cover problem. A polyomino is a shape constructed out of connected equal-sized squares. We will call these squares cells. An n-omino consists of n cells. A 1-omino is called a monomino; the 2-omino is called a domino; 3-omino is called a tromino etc. The most recognisable of those are the dominoes, from which all polyominoes get their names, and the tetrominoes which are featured in the game Tetris. Polyominoes can be:

- Free: The polyomino can be rotated and mirrored. All these variations are considered to be the same polyomino.

- One-sided: The polyomino can only be rotated. As above, all rotation are considered to be the same polyomino. The mirrored polyomino is considered to be distinct.
- Fixed: The polyomino can neither be rotated nor mirrored. All variations are considered to be distinct polyominoes.

Each additional degree of freedom reduces the total amount of polyominoes as more and more variations are considered to be equivalent. In polyomino tiling, the goal is to cover the entirety of a given surface with a given group of polyominoes without overlap. This can be represented as an exact cover problem.

As an exact cover: The set X consists of every cell of the surface (to ensure every cell is covered exactly once) and of every given polyomino (to ensure every polyomino is placed exactly once). The set S consists of every possible placement for every polyomino. Such a placement consists of the polyomino in question and the cells it would cover in this placement.

Example: 2.1 Consider a two by six surface and two fixed polyominoes: a horizontal domino d and a square tetromino t. The domino has three possible placements and the tetromino only two. If we number the cells 1 through six we have: $X = \{d, t, 1, 2, 3, 4, 5, 6\}$ and $S = \{\{d, 1, 2\}, \{d, 3, 4\}, \{d, 5, 6\}, \{t, 1, 2, 3, 4\}, \{t, 3, 4, 5, 6\}\}$ The polyomino is included to avoid placing the same polyomino twice. A cover such as 2.2 is represented as $S* = \{\{t, 1, 2, 3, 4\}, \{d, 5, 6\}\}$.

Figure 2.1: Example of cover problem

Figure 2.2: Example of cover

Binary Matrix: The problem can be encoded in a binary matrix M where each row represents an element s of S, therefore a placement, and each column represents an element x of X. The matrix $M_{i,j} = 1$ if $x \in s$ otherwise 0 This encodes which cells are covered by which polyominoes.

Example: The matrix representation of the previous example is

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The first two columns represent the polyominos, the remaining represent the cells. The same previous cover can be represented as:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

consisting of the third and fourth rows.

2.3 Knuth's Algorithm X

One approach to solving an exact cover problem, such as polyomino tiling, is to employ Knuth's Algorithm X. This algorithm operates by recursively and deterministically selecting a subset, which in the context of tiling corresponds to placing a tile on the board, and then eliminating all subsets that are incompatible with that selection. The process continues recursively, exploring further placements, and if at any stage no valid selections remain, the algorithm backtracks to previous decisions to explore alternative options. In essence, this means that Algorithm X systematically explores all partial solutions and is therefore guaranteed to find all full solutions.
An efficient implementation of Algorithm X can be achieved through the use of a doubly linked list, which enhances memory efficiency (by only storing the non-zero entries) and accelerates the search process. When Algorithm X is implemented in this way, the technique is referred to as Dancing Links (DLX). The structure of the doubly linked list allows for rapid covering (or uncovering) of rows and columns by removing them from (or returning them to) the link chain that binds them. The method is very efficient as it removes almost all overhead for adding and removing from the list. This is where the name Dancing Links comes from as the links are ever-changing, back and forth. The X stands simply for exact cover. Both Algorithm X and the Dancing Links implementation are described in detail in [3].

2.4 SAT

Another approach to solving the exact cover problem, and the one we will explore, is by reducing it to the Boolean satisfiability problem (SAT). This can be done in polynomial time as both problems are NP-complete. Modern SAT solvers [6] are highly optimized and can handle large instances efficiently.

Definition 4 (Boolean Formula)**.** *A* Boolean formula *is a formula constructed using Boolean variables, the logical operators $\neg$ (not), $\wedge$ (and), and $\vee$ (or). A Boolean variable can take the value* `true` *or* `false`*.*

Definition 5 (Assignment)**.** *An* assignment *is a mapping from the set of Boolean variables in a formula to the set* $\{$`true`, `false`$\}$*. An assignment specifies a truth value for each variable in the formula.*

Definition 6 (Evaluation)**.** *Given a Boolean formula φ and an assignment α, the* evaluation *of φ under α, denoted $\varphi(\alpha)$, is the result of recursively substituting the truth values of variables according to α and applying the logical operators. The evaluation yields either* `true` *or* `false`*.*

Definition 7 (Conjunctive Normal Form)**.** *A Boolean formula is in* conjunctive normal form *(CNF) if it is a conjunction of clauses, where each clause is a disjunction of literals. A* literal *is either a Boolean variable or its negation.*

Definition 8 (Boolean Satisfiability Problem)**.** *The* Boolean satisfiability problem (SAT) *is the problem of determining whether a given Boolean formula has an assignment of truth values to its variables that makes the formula evaluate to* `true`*.*

Modern SAT solvers employ a combination of search and reasoning techniques to determine the satisfiability of Boolean formulas. The most common approach is based on the *Conflict-Driven Clause Learning* (CDCL) algorithm. The solver incrementally constructs a partial assignment of variables and performs the following key operations:

- Decisions: The solver chooses an unassigned variable and assigns it a truth value, effectively making a *decision* that extends the current partial assignment.
- Propagation: Logical implications of the current assignment are deduced using *unit propagation*, assigning additional variables without making further decisions.
- Conflicts: If propagation leads to a clause that cannot be satisfied, a *conflict* occurs. The solver analyses the conflict to determine which assignments caused it.
- Backtracking: Based on the conflict analysis, the solver *backtracks* to a previous decision level, reversing some assignments, and learns a new *conflict clause* that prevents the same conflict from reoccurring.

In practice, SAT solvers generally expect the input in CNF. One commonly used format for representing CNF formulas is the *DIMACS* format, which encodes each clause as a line of integers corresponding to variables (positive for the variable itself, negative for its negation) and ends each clause with a 0. It also includes the number of variables and the number of clauses at the very front. This format allows standardized input to most modern SAT solvers.

3 Encodings

There are many different ways to reduce an exact cover problem to SAT. In this chapter, we will explore several possible encodings in CNF and their translation into the DIMACS format for use with SAT solvers. The implementation used in the experimental evaluation was written in C++, but will be described using pseudocode.

3.1 Direct encoding

For this straightforward encoding, we use the matrix representation as a starting point. Recall that the rows of M 2.2 correspond to the possible placements and that the columns correspond to the cells that are covered by these placements. The columns also include the polyominoes themselves, but they will be treated the same as the cells when not mentioned otherwise.

3.1.1 Variables

A solution to the polyomino tiling problem consists of a set of placements for the polyominoes. Since each row of M represents a possible placement, we associate one Boolean variable x_i with each row i. The variable is assigned true if the corresponding placement is included in the solution, and false otherwise.

3.1.2 Constraints

For an assignment to these variables to represent a solution, every cell must be covered exactly once. This is commonly referred to as the exactly-one constraint. We decompose it into two parts: at-least-one and at-most-one. For every column j, at least one placement covering that column must be selected. This is expressed as a disjunction over all placements that include the column:

$$\bigvee_{i:M_{i,j}=1} x_i$$

Each such clause ensures that each cell is covered by at least one placement. The at-most-one constraint ensures that no two incompatible placements are selected simultaneously. Two

placements a and b are incompatible if they overlap, i.e., if they both cover the same cell. For each such pair, we introduce the clause:

$$\neg x_a \vee \neg x_b \text{ if } a < b$$

We make sure to avoid duplicates by restricting a to be smaller than b. This form, known as the pairwise encoding of the at-most-one constraint, guarantees that at most one placement covering a given cell is active. Its primary drawback is that the number of clauses grows quadratically with the number of incompatible placements.

3.1.3 Full encoding

Together, these variables and constraints encode the polyomino exact cover problem. Each polyomino must be placed exactly once, and each cell must be covered exactly once. The resulting CNF can then be translated into the DIMACS format for use with standard SAT solvers.

3.1.4 Implementation

As mentioned above, the input is the matrix. For the DIMACS format, we need the number of variables, the number of clauses, and the clauses themselves. The number of variables is simply the number of rows, and the number of clauses can be determined incrementally as the clauses are generated.

Algorithm 1 At-least-one Clause Generation

```
for each column j in M do
    clause ← {}
    for each row i in M do
        if M_{i,j} = 1 then
            add literal x_i to clause
        end if
    end for
    add clause to clauses
    increment clause count n
end for
```

Algorithm 2 Pairwise At-most-one Clause Generation

```
for each row a in M do
    for each row b in M where b > a do
        for each column j in M do
            if M_{a,j} = M_{b,j} = 1 then
                add clause (¬x_a, ¬x_b) to clauses
                increment clause count n
                break loop over j
            end if
        end for
    end for
end for
```

3.2 Bimander Encoding

The Bimander encoding [7] for the at-most-one constraint reduces the number of required clauses down to $nlog_2n$ per n incompatible variables while introducing only $\lceil log_2n \rceil$ new auxiliary variables. It combines the ideas of the Commander encoding [8] and the Binary encoding [9]. Similar to the Commander encoding, the incompatible variables are partitioned into m equally sized groups, each containing g variables. The at-most-one constraint is then enforced both within each group and across the groups. Within a group, the constraint can be enforced using a simple method, such as pairwise encoding. Across groups, the constraint is enforced via a binary encoding on auxiliary variables representing the groups. This ensures that no two variables from different groups can be simultaneously true.

3.2.1 Variables

In addition to the variables x_i from the previous section, the Bimander encoding requires $\lceil \log_2(m) \rceil$ auxiliary variables b_k, which represent the binary code identifying the selected group.

3.2.2 Constraints

To enforce the at-most-one constraint within each group, we use the pairwise method as previously described. To enforce the constraint across groups, each variable within a group is allowed to be true only if the binary code of the auxiliary variables matches the code of that group. Formally, for group G_h with binary code $C_h \in \{0,1\}^{\lceil \log_2(m) \rceil}$ ($C_h[k]$ being the kth bit of C_h) and variable $x_i \in G_h$, the following clauses are added:

$$\bigwedge_{k=0}^{\lceil \log_2(m) \rceil - 1} \begin{cases} (\neg x_i \vee b_k), & \text{if } C_h[k] = 1 \\ (\neg x_i \vee \neg b_k), & \text{if } C_h[k] = 0 \end{cases}$$

As every variable b_k has to correspond to the code C_h for x_i to be true, no other code can be encoded at the same time. It follows that only variables of G_h and no other group can be true. As the at-most-one constraint is enforced both within and between groups, the at-most-one constraint is fully enforced.

3.2.3 Implementation

Algorithm 3 is implemented as an alternative to Algorithm 2. Algorithm 1 remains unchanged.

3.3 Cell-based Encoding

The previous two encodings can be applied to any exact cover problem. Since, we focus on polyomino covers , we can use some of the problem's properties to our advantage. Notably, the columns can be separated into polyominoes and cells. We also know for every placement which polyomino is used. With this information, we can create a unique new encoding for the cells. Instead of enforcing the at-most-one constraint using overlapping placements, we can directly relate every cell to a polyomino.

Algorithm 3 Bimander At-Most-One Clause Generation

```
for each column j in M do
    incompatible_variables ← {}
    for each row i in M do
        if M_{i,j} = 1 then
            add variable x_i to incompatible_variables
        end if
    end for
    variable_groups ← split incompatible_variables into m equal-sized groups
    for each group in variable_groups do
        generate pairwise at-most-one clauses over group
    end for
    for bit index k = 0 to ⌈log_2(m)⌉ − 1 do
        for each group in variable_groups do
            if (the k-th bit of the binary code for group) = 1 then
                for each variable x_i in group do
                    add clause (¬x_i ∨ b_k) to clauses
                    increment clause count n
                end for
            else
                for each variable x_i in group do
                    add clause (¬x_i ∨ ¬b_k) to clauses
                    increment clause count n
                end for
            end if
        end for
    end for
end for
```

An initial approach was not to encode the polyomino directly for each cell, but to create a conjunction of all possibilities for every cell. Converting to CNF, however, led to an excessively large number of clauses, so this idea was dropped.
Instead, for each polyomino, we assign a unique binary code, which can be encoded by any of the cells. Overlap is avoided as every cell can only encode one polyomino at a time. Additionally, we need to ensure that for every cell encoding a polyomino, the neighboring cells encode polyominoes in such a way that the polyomino preserves the correct shape.

3.3.1 Variables

Let there be p polyominoes. For each cell c, we introduce $\lceil \log_2 p \rceil$ auxiliary variables $b_{c,k}$, representing the binary code C of the polyomino that covers the cell. Each variable encodes one bit of the polyomino's code.

3.3.2 Constraints

For each polyomino P represented by code C_P and each variable x that is a placement of P, the encoding ensures that every cell covered by this placement encodes C_P. Formally, for a placement covering m cells $c_1, c_2, ...c_j...c_m$, we enforce:

$$x \implies \bigwedge_{k=0}^{\lceil \log_2 p \rceil - 1} \bigwedge_{j=1}^{m} \begin{cases} (b_{c_j,k}), & \text{if the } k\text{th bit of } C_P \text{ is 1} \\ (\neg b_{c_j,k}), & \text{if the } k\text{th bit of } C_P \text{ is 0} \end{cases}$$

This ensures that all cells in a placement encode the same polyomino. Conflicts between overlapping placements are avoided because no two different polyomino codes can occupy the same cell. However, this does not enforce that each polyomino can be placed only once or that each cell is covered by a polyomino that can actually cover it. The at-least-one constraint of Section 2 will again need to be enforced separately as described there. The at-most-one constraint can be limited to only the columns representing polyominoes as at-most-one is already enforced for the cells.

3.3.3 Implementation

Algorithm 1 remains unchanged once again.

Algorithm 4 Cell-based Clause Generation

generate pairwise at-most-one clauses over columns that represent polyominoes
for each row i in M **do**
  for each column j in M where j represents a cell **do**
    $clause \leftarrow \{x_i\}$
    for bit index $k = 0$ **to** $\lceil \log_2(p) \rceil - 1$ **do**
      if (the k-th bit of the binary code for the polyomino of row i) $= 1$ **then**
        add literal $\neg b_{c_j,k}$ to *clause*
      else
        add literal $b_{c_j,k}$ to *clause*
      end if
    end for
  end for
  add *clause* to *clauses*
end for

4 Evaluation

4.1 Setup

All algorithms used in this study, including the SAT-based encodings and the DLX algorithm, were implemented in C++. The DLX implementation followed Knuth's specification [3], while the SAT-based approaches were implemented as described in Chapter 3.
Experiments were executed on an Intel(R) Core(TM) i5-8250U CPU @ 1.60 GHz with a 10-minute time limit per instance. The problem instances were chosen as varied yet classical examples of polyomino tiling problems. For each instance, all free, one-sided, or fixed n-ominoes were used exactly once.
For the Bimander encoding m was set to $n/2$. As this was reported to be most efficient by [1, 10]

4.2 Metrics

4.2.1 Problem Characteristics

The characteristics of the exact cover problem instances that were recorded are:

- **Board surface:** The dimensions of the target area to be covered by polyominoes.
- **Polyomino number:** The total number of distinct polyominoes provided for tiling.
- **Possible placement number:** The total number of possible placements of all polyominoes on the board.
- **Satisfiability:** Whether the instance admits a valid tiling configuration .

4.2.2 SAT Solver Metrics

The performance of SAT solvers was evaluated using the following metrics:

1. **CPU runtime:** The total CPU time required to find a satisfying assignment.
2. **Memory usage:** The peak memory consumption during SAT solving.

3. **Decisions:** The total number of variable assignments made by the solver during the search.
4. **Propagations:** The number of propagations performed to deduce variable assignments.
5. **Conflicts:** The total number of conflicts encountered.

These metrics are provided by MiniSat and can be compared to analogous metrics for DLX.

4.2.3 DLX Metrics

The performance of the DLX algorithm was evaluated according to the following metrics:

1. **CPU runtime:** The total CPU time required to find a solution or to determine that none exist.
2. **Memory usage:** The peak memory consumption during execution.
3. **Decisions:** The total number of placements chosen during the search, equivalent to assigning a placement variable true in the SAT approach.
4. **Propagations:** The number of times a row was temporarily covered during the search, analogous to setting incompatible variables to false.
5. **Conflicts:** The total number of backtracking steps performed to explore alternative solutions, performed whenever the search encounters a dead end.

4.2.4 Comparison of Metrics

The first two metrics, CPU runtime and memory usage, are directly comparable between DLX and SAT solvers. The remaining metrics are not directly comparable due to differences in search space structure, but they provide valuable insight into the behaviour of each algorithm.

4.3 Problem Set

The problem instances on which tests were performed include classical tiling configurations. Instances are identified by board dimensions and satisfiability. The subtracted number indicates holes in the surface, i.e., cells that must not be covered.

- **Pentominoes**
  - *Free*: 60 cells, 12 pentominoes
    - 3x20: satisfiable
    - 4x15: satisfiable
    - 5x12: satisfiable
    - 6x10: satisfiable
    - 8x8-4: satisfiable
  - *One-sided*: 90 cells, 18 pentominoes

        * 10x10-10: satisfiable
        * 3x30: satisfiable
        * 5x18: satisfiable
        * 9x10: satisfiable

- **Tetrominoes**
    - *Free*: 20 cells, 5 tetrominoes
        * 3x7-1: satisfiable
        * 3x7-1: unsatisfiable
    - *One-sided*: 28 cells, 7 tetrominoes
        * 3x10-2: satisfiable
        * 3x10-2: unsatisfiable
        * 5x6-2: satisfiable
        * 5x6-2: unsatisfiable
    - *Fixed*: 76 cells, 19 tetrominoes
        * 7x11-1: satisfiable
        * 7x11-1: unsatisfiable

For the tetrominoes there always are both a satisfiable instance and an unsatisfiable counterpart. The satisfiability depends on the placement of the holes. This is quite convenient as it allows us to compare very similar instances differing almost only in satisfiability.

4.4 Results

4.4.1 DLX

As we can see in Table 4.1, DLX was able to complete all tests in under 10 minutes. In fact, all but one test were completed in fractions of a second. Only 7x11-1 unsat took almost two minutes to complete. The memory usage was very similar throughout and quite minor as well. Conflicts, decisions, propagations, and CPU time all seem very strongly correlated with each other. Unsatisfiable instances took longer than their satisfiable counterparts, as all partial solutions had to be traversed before unsatisfiability was confirmed.
In terms of scalability, DLX demonstrates a relatively consistent growth in runtime and memory usage with increasing instance size. Smaller boards or fewer polyominoes are solved extremely quickly, while larger boards see a manageable increase in computational resources, indicating good scalability for moderately larger instances.

4.4.2 Direct Encoding

The direct encoding shown on Table 4.2 allowed all but the 7x11-1 unsat instance to be completed in under 10 minutes. Most were also completed in under one minute, but the completion time rises sharply for both encoding and the search itself with larger instances. Memory usage for the search also rises considerably with larger instances while memory usage for encoding rises more gradually.

4.4.3 Bimander Encoding

The Bimander encoding of Table 4.3 shows its efficiency during the encoding. Both CPU time and memory use rise only slightly with larger instances thanks to its compact encoding of the at-most-one constraint. It did not, however, translate to an improvement during search time.

4.4.4 Cell-Based Encoding

The cell-based Encoding performed poorly. It was not able to complete any of the pentomino instances and was slow for the small tertomino instances as well. This is likely due to the large number of auxiliary variables this encoding introduces. Scalability is particularly poor for Cell-Based Encoding. The rapid growth in the number of variables and clauses with increasing board size renders even moderately sized instances difficult to solve.

Table 4.1: DLX results

Instance	Conflicts	Decisions	Propagations	CPU time [s]	Memory [MB]
3x20 sat	1326	234	17352	0.009589	3.75
4x15 sat	1122	200	13682	0.011058	3.875
5x12 sat	5238	886	56449	0.014583	4
6x10 sat	4596	779	57459	0.015001	4
8x8-4 sat	6804	1147	67764	0.016817	4.125
10x10-10 sat	40224	6723	467223	0.07187	4.75
3x30 sat	64698	10802	727849	0.063681	4
5x18 sat	138948	23177	1685540	0.144209	4.375
9x10 sat	44856	7495	497206	0.061751	4.5
3x10-2 sat	460	100	2288	0.00108	3.375
3x10-2 unsat	1009	203	5239	0.00133	3.375
3x7-1 sat	5	7	147	0.000452	3.375
3x7-1 unsat	303	62	1313	0.000598	3.375
5x6-2 sat	180	44	1219	0.001307	3.375
5x6-2 unsat	1399	281	7045	0.001513	3.375
7x11-1 sat	1550	330	8887	0.006365	3.625
7x11-1 unsat	275101933	55020388	1448687742	116.949	3.375

4.5 Comparison

4.5.1 DLX and SAT

DLX consistently outperformed all three SAT-based both in terms of CPU time and memory usage. As can be seen in Tables 4.5 and 4.6, it performed substantially more propagations and encountered more conflicts per decision than the SAT encodings. This seems to indicate that not only was it implemented with less overhead, but that its heuristic selection algorithm also outperformed the SAT solver's. Its search time was also more consistent in relation to the number of possible placements of the test instance, as can be seen on Figure 4.1.

Table 4.2: Direct Encoding Results

Instance	Encoding		Search				
	CPU time [s]	Memory [MB]	Conflicts	Decisions	Propagations	CPU time [s]	Memory [MB]
3x20 s.	2.31265	7.171	11147	60737	652571	0.64443	25.29
4x15 s.	3.99141	10.992	38042	200155	3048119	6.2518	79.86
5x12 s.	4.92379	18.492	8710	69385	801152	0.582327	36.22
6x10 s.	5.4918	18.503	13944	112460	1529388	1.27028	48.37
8x8-4 s.	7.30714	18.492	24094	132569	1868586	2.60752	52.21
10x10-10 s.	37.0314	33.578	3046	33303	279499	0.277846	28.8
3x30 s.	7.58587	11.003	242731	943392	15723338	255.322	443.4
5x18 s.	19.4779	18.355	54038	263776	4427916	13.4272	176.95
9x10 s.	25.5386	33.53	46909	293838	4125333	10.9999	210.35
3x10-2 s.	0.046258	3.625	44	194	1024	0.00439	7.33
3x10-2 u.	0.044943	3.625	377	798	11123	0.002688	7.33
3x7-1 s.	0.010733	3.5	1	20	141	0.001729	7.07
3x7-1 u.	0.011638	3.5	18	62	790	0.004033	7.07
5x6-2 s.	0.065172	3.875	93	226	2564	0.002656	7.21
5x6-2 u.	0.069431	3.875	464	985	14368	0.003119	7.33
7x11-1 s.	1.81318	5.332	140	1048	7602	0.010265	9.01
7x11-1 u.	1.80924	5.328				>600	

Table 4.3: Bimander Encoding Results

Instance	Encoding		Search				
	CPU time [s]	Memory [MB]	Conflicts	Decisions	Propagations	CPU time [s]	Memory [MB]
3x20 s.	0.046287	4.472	55099	212397	6409672	17.6958	85.44
4x15 s.	0.06737	4.468	3166	18137	367808	0.212215	14.46
5x12 s.	0.074659	5.464	49416	239880	8084070	17.2472	108.09
6x10 s.	0.079619	5.480	31333	141235	4400889	6.37287	77.14
8x8-4 s.	0.088903	5.464	131395	578411	18560493	86.2891	177.29
10x10-10 s.	0.228952	7.449	24407	116898	13886933	4.61235	70.45
3x30 s.	0.107787	5.476	602872	2358756	75332674	563.478	314.6
5x18 s.	0.181954	5.519	44799	205599	6597884	16.4028	144.48
9x10 s.	0.192946	7.375	145434	711812	21955936	129.049	544.99
3x10-2 s.	0.006126	3.625	749	1711	56556	0.010541	7.46
3x10-2 u.	0.006092	3.625	485	1692	30423	0.00733	7.34
3x7-1 s.	0.00358	3.5	56	118	2651	0.00188	7.07
3x7-1 u.	0.00357	3.5	117	171	6125	0.004332	7.07
5x6-2 s.	0.007108	3.625	132	383	7273	0.002939	7.2
5x6-2 u.	0.007489	3.375	408	778	27420	0.005759	7.21
7x11-1 s.	0.039523	4	226	691	18492	0.005949	7.77
7x11-1 u.	0.044513	3.875				>600	

Table 4.4: Cell-Based Encoding Results

Instance	Encoding		Search				
	CPU time [s]	Memory [MB]	Conflicts	Decisions	Propagations	CPU time [s]	Memory [MB]
3x10-2 s.	0.023281	3.625	387611	545667	9958760	14.0702	19.2
3x10-2 u.	0.019283	3.625	1543883	2088415	34514864	57.6698	26.58
3x7-1 s.	0.008489	3.5	2186	3667	48861	0.019501	7.71
3x7-1 u.	0.006708	3.5	3829	5848	88787	0.041433	7.86
5x6-2 s.	0.024854	3.625	264268	405221	9399234	11.3036	21.44
5x6-2 u.	0.027779	3.375	1278494	1858781	35967381	72.7083	36.12
7x11-1 s.	0.471616	4				>600	
7x11-1 u.	0.480008	4				>600	

4.5.2 SAT

Surprisingly, the Bimander Encoding performed worse overall than the Direct Encoding. It seems the large reduction in clauses did not offset the increase in variables. Perhaps, it even worsened the result. The Cell-Based Encoding performed worst of all overall yet still managed to slightly outperform the Direct Encoding during the encoding process itself while the Bimander encoding performed best during the encoding process.

Table 4.5: Average propagations per decision

Approach	Average Propagations per Decision
DLX	47.76
Direct Encoding	12.00
Bimander Encoding	34.46
Cell-Based Encoding	17.64

Table 4.6: Average conflicts per decision

Approach	Average Conflicts per Decision
DLX	5.11
Direct Encoding	0.22
Bimander Encoding	0.32
Cell-Based Encoding	0.67

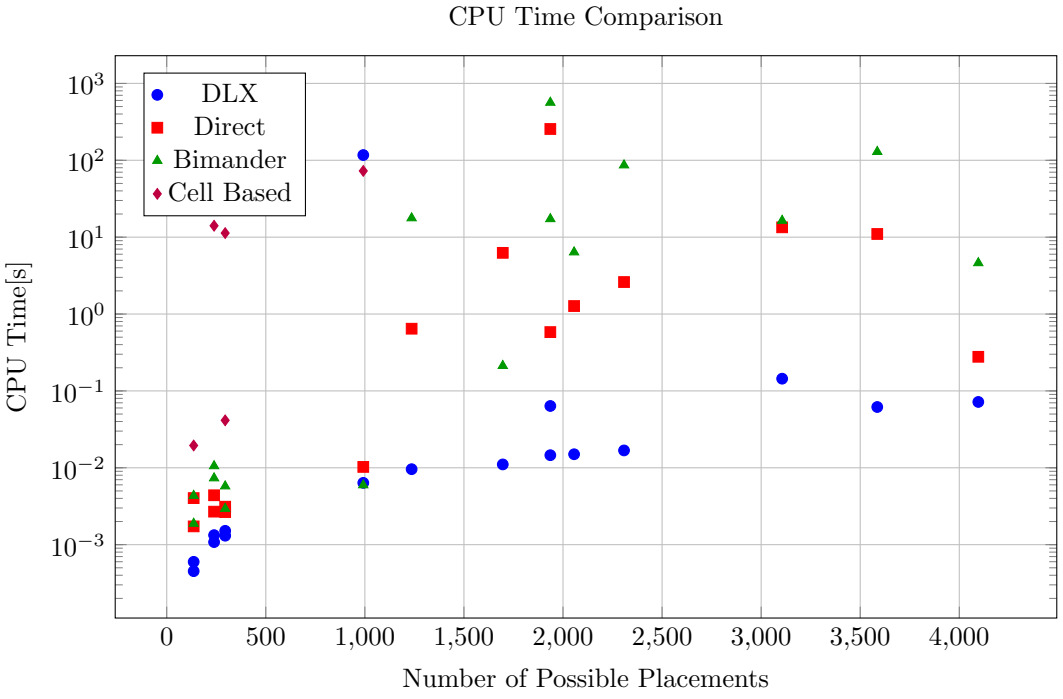

Figure 4.1: CPU Search Time Comparison

4.5.3 Discussion

DLX clearly outperformed all SAT-based approaches in the context of polyomino tiling. Its advantage stems mainly from representing the exact cover problem directly, without the overhead of CNF encoding or auxiliary variables. Each operation in DLX corresponds directly to a logical step in the search, making it both faster and more memory efficient.
The data structure itself allows immediate propagation and efficient backtracking. The heuristic used by DLX, always selecting the column with the fewest options, also proved more effective than the variable selection heuristics of the SAT solvers.

5 Conclusion

In this work, we implemented three encodings, Direct, Bimander, and Cell-Based, to investigate the performance of SAT-based approaches on the polyomino tiling problem. Our experiments demonstrate that although SAT solvers are capable of solving these instances, their performance remains inferior to Knuth's DLX algorithm. Among the SAT encodings, the most straightforward Direct Encoding achieved the best results, while the newly developed Cell-Based Encoding performed the worst.

While further exploration of alternative encodings and SAT solvers, perhaps across a broader range of problem instances, could yield additional insights, it seems unlikely that these approaches will surpass the efficiency of DLX for this class of problems.

AI use: Writefull was used as a spellchecker. GPT-5 Mini was used to revise text style. No new content was generated.

Bibliography

[1] Nguyen, V., Mak-Hau, V., Moran, B., and Novak, A. An efficient and exact algorithm for military timetabling and trainee assignment problems. *Computers & Industrial Engineering*, 169:108192 (2022).

[2] Surendiran, P., Meinecke, C. R., Salhotra, A., Heldt, G., Zhu, J., Månsson, A., Diez, S., Reuter, D., Kugler, H., Linke, H., et al. Solving exact cover instances with molecular-motor-powered network-based biocomputation. *ACS nanoscience Au*, 2(5):396–403 (2022).

[3] Knuth, D. Dancing Links. *Millennial Perspectives in Computer Science*, 1 (2000).

[4] Junttila, T. and Kaski, P. Exact cover via satisfiability: An empirical study. In *International Conference on Principles and Practice of Constraint Programming*, pages 297–304. Springer (2010).

[5] Karp, R. M. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*, pages 219–241. Springer (2009).

[6] Eén, N. and Sörensson, N. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer (2003).

[7] Nguyen, V.-H. and Mai, S. T. A new method to encode the at-most-one constraint into SAT. In *Proceedings of the 6th International Symposium on Information and Communication Technology*, pages 46–53 (2015).

[8] Klieber, W. and Kwon, G. Efficient CNF encoding for selecting 1 from n objects. In *Proc. International Workshop on Constraints in Formal Verification*, page 14 (2007).

[9] Frisch, A. M., Peugniez, T. J., Doggett, A. J., and Nightingale, P. W. Solving non-boolean satisfiability problems with stochastic local search: A comparison of encodings. *Journal of Automated Reasoning*, 35(1):143–179 (2005).

[10] Nguyen, V.-H., Nguyen, V.-Q., Kim, K., and Barahona, P. Empirical study on SAT-encodings of the at-most-one constraint. In *The 9th International Conference on Smart Media and Applications*, pages 470–475 (2020).