Universiteit Leiden
The Netherlands

Opleiding Informatica

Comparing different encodings for the continuity rule of the logic puzzle Context

Jente de Waart

Supervisors:
Hendrik Jan Hoogeboom, Rudy van Vliet & Mark van den Bergh

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

3/08/2025

Abstract

This thesis investigates the application of Boolean Satisfiability (SAT) solvers to a grid-based logic puzzle called *Context* [AJ25]. Specifically, it focuses on the critical 'continuity' constraint, which means that a subset of tiles, in our case the white tiles, must form a single connected region. We conduct a comparative analysis of two distinct SAT encodings for this rule: and show that the *Double Tree* encoding is significantly more efficient than the *Topological Order* encoding in terms of variables, clauses and solver cycles. We show that for larger *Context* puzzles such as a 10x10 grid we find a geometric mean speed-up of 8.25 times by using the *Double Tree* encoding over the *Topological Order* encoding.

Contents

1 Introduction

Boolean Satisfiability (SAT) solvers have become increasingly important tools for solving combinatorial problems, including grid-based logic puzzles. This thesis focuses on applying SAT-based approaches to the *Context* puzzle, with particular attention to how different encoding strategies affect solver performance.

Logic puzzles on grids, such as the *Context* puzzle studied in this thesis, can be modeled and solved using SAT solvers. After encoding the puzzle's rules as a Boolean formula in Conjunctive Normal Form (CNF), a SAT solver can efficiently search for valid solutions or prove that none exist. Central to a SAT-based approach is the encoding of the puzzle's constraints, which transforms the problem into a set of logical statements that can be processed by the solver. The encoding strategy determines how much memory the encoding takes up and also affects the speed of the solver.

A common and often crucial constraint in many grid puzzles is the 'continuity' rule, ensuring that a designated set of tiles (e.g., all white tiles) forms a single, unbroken region. As this rule is the most complex, traditionally requiring more than just local data to determine the validity of a solution, it is often the most computationally intensive part of solving such puzzles. Therefore the efficiency of the SAT encoding for this continuity constraint can significantly impact the overall performance of the SAT solver, especially as puzzle size increases.

This bachelor thesis, conducted at the Leiden Institute of Advanced Computer Science (LIACS) under the supervision of Hendrik Jan Hoogeboom, Rudy van Vliet and Mark van den Bergh, makes a comparative analysis of two distinct SAT encodings for the white tile continuity constraint within the *Context* puzzle: a *Topological Order* based encoding and a *Double Tree* based encoding. The core research question is: *Does the novel Double Tree encoding outperform the Topological Order encoding in terms of variables, clauses and CPU cycles to solve?* To answer this, we will investigate their performance characteristics, specifically examining differences in SAT solver runtimes in CPU cycles, the number of Boolean variables generated, and the total number of clauses required for typical puzzle instances. The ultimate goal is to determine which encoding, or under what conditions an encoding, is preferable for this class of puzzles.

The remainder of this document is structured as follows: Section 2 formally defines the *Context* puzzle, the basic variable representation, and relevant SAT terminology. Section 3 discusses existing literature. Section 4 details the specific SAT encodings for the continuity constraint, explaining the Topological order encoding and Double Tree encoding in detail. Section 5 presents the complete CNF encodings for all *Context* puzzle rules. Sections 6 and 7 describe our puzzle generation methodology and validation. Section 8 describes the experimental setup and presents the results. Finally, Section 9 summarizes the findings and suggests avenues for future research.

2 Definitions

This section formally defines the *Context* puzzle, the fundamental Boolean variables used for its representation in a SAT context, and key concepts related to Boolean Satisfiability.

2.1 The *Context* Puzzle: Rules and Objective

The *Context* puzzle is presented on a grid consisting of individual tiles. The objective is to determine a valid coloring of these tiles, where each tile is assigned either a black or white color according to the following constraints:

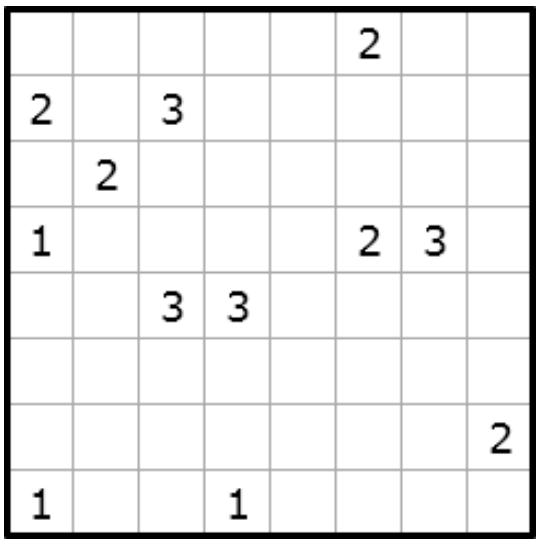


Figure 1: An example of an unsolved *Context* grid with numerical clues.

1. **Tile Coloring:** Each tile may be either colored black or left white. This applies to all tiles, including those containing numerical clues.
2. **No Black Tile Adjacency:** No two black tiles may be orthogonally adjacent; that is, black tiles cannot share a common edge.
3. **White Tile Clues:** A white tile containing a numerical value specifies the exact number of orthogonally adjacent black tiles. Orthogonal adjacency refers to tiles located directly above, below, to the left, or to the right, as illustrated in Figure 2.

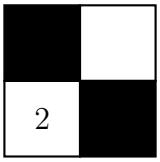


Figure 2: Rule 3: A white tile with clue '2' must have exactly two black orthogonal neighbours.

4. **Black Tile Clues:** A black tile containing a numerical value specifies the exact number of diagonally adjacent black tiles. Diagonal adjacency refers to tiles located at the four corners surrounding the tile, as illustrated in Figure 3.

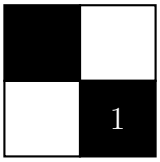


Figure 3: Rule 4: A black tile with clue '1' must have exactly one black diagonal neighbour.

5. **White tile Continuity:** All white tiles must form a single connected region under orthogonal adjacency. In other words, it must be possible to traverse from any white tile to any other white tile by moving only through white tiles that share an edge, as illustrated in Figure 4.

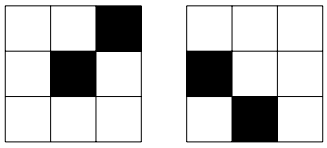


Figure 4: Examples of valid (left) and invalid (right) white tile connectivity in a *Context* puzzle.

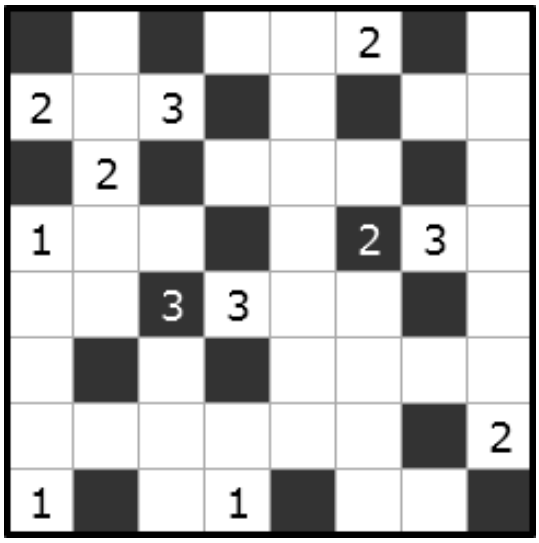


Figure 5: A valid solution to a *Context* puzzle, satisfying all rules.

2.2 Boolean Satisfiability (SAT)

The Boolean Satisfiability problem (SAT) is the problem of determining if there exists an assignment of truth values (true or false) to variables that makes a given Boolean formula true. SAT is a fundamental problem in computer science and is useful for various applications, including artificial intelligence, hardware verification, and combinatorial optimization.

- A **literal** is a Boolean variable (e.g., x) or its negation (e.g., $\neg x$).

- A **clause** is a disjunction (OR) of one or more literals (e.g., $x \vee \neg y \vee z$). A clause is satisfied if at least one of its literals is true.
- **Conjunctive Normal Form (CNF)**: A Boolean formula is in CNF if it is a conjunction (AND) of one or more clauses (e.g., $(x \vee \neg y) \wedge (\neg x \vee z)$). A CNF formula is satisfied if all of its clauses are satisfied.

CNF is particularly important because most modern SAT solvers are designed to work specifically with formulas in CNF. The choice of CNF as the standard representation is motivated by several factors: it provides a uniform structure that enables efficient algorithms, supports effective conflict analysis in modern solvers, and allows for systematic unit propagation—a key inference technique where clauses with only one unassigned literal immediately determine that literal's value.
SAT solvers are algorithms designed to solve the SAT problem where the goal is to find if a satisfying assignment exists for a boolean formula. The process of using a SAT solver for puzzles involves modeling the puzzle's rules and instance as a CNF formula, where a satisfying assignment corresponds to a valid solution.

2.3 CaDiCaL SAT Solver

For this research, we employ CaDiCaL [Bie24], a powerful and modern SAT solver developed by Armin Biere. CaDiCaL is based on the Conflict-Driven Clause Learning (CDCL) architecture, which combines systematic search with powerful learning mechanisms to prune the search space effectively. It is well documented and provides a native c++ interface allowing us to run everything in the same program. This makes it well suited for our experiments.

Key features of CaDiCaL relevant to our work include:

- **Incremental Solving**: Supports adding new clauses to an existing formula and re-solving, which we utilize for uniqueness checking.
- **Robust Performance**: Consistently performs well across diverse problem instances in SAT competitions.

2.4 Incremental Solving and Uniqueness Verification

An important aspect of puzzle solving is determining whether a solution is unique. We leverage CaDiCaL's incremental solving capability for this purpose. After finding an initial solution, we add clauses that forbid that specific solution and attempt to find another. If the solver reports unsatisfiability, we can conclude the original solution is unique.
Formally, if ϕ is our puzzle formula and σ is a satisfying assignment, we add the clause $\bigvee_{x_i \in \sigma} \neg x_i$ to exclude σ and check if $\phi \wedge \bigvee_{x_i \in \sigma} \neg x_i$ is satisfiable. This incremental approach is much more efficient than solving the problem from scratch, as the solver retains learned clauses and other internal state from the initial solve.

2.5 Boolean Variables for Puzzle State

To model the *Context* puzzle for a SAT solver, we associate a Boolean variable with the state of each tile. For a tile at row i and column j in an $R \times C$ grid, we define a variable $w_{i,j}$.

- $w_{i,j}$ is true if the tile (i, j) is white.
- $w_{i,j}$ is false if the tile (i, j) is black.

For non-square grids of size $R \times C$, we use $0 \leq i < R, 0 \leq j < C$ where $N = R \times C$ represents the total number of tiles. We consider the grid to be indexed from the top-left corner, with the first tile at $(0, 0)$. When encoding neighbours in clauses we require that the neighbours are within the bounds of the grid, if the neighbour would fall outside the grid we encode it as a placeholder tile that is always white. For the clues on the tiles, we only need tile variables $w_{i,j}$. We require additional variables for the continuity encodings, which will be discussed in Section 4.

3 Related Work

Connectivity (or continuity) constraints are fundamental in many grid-based logic puzzles, such as Slitherlink, Masyu, Nurikabe, Hitori, and Hashi, where parts of the solution must form a single connected component. A common strategy is to translate these connectivity requirements into graph reachability constraints within a Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) framework, thereby allowing general-purpose solvers to tackle these puzzles.

3.1 Topological Order Encoding

The Topological Order encoding is an adaptation of the Directed Acyclic Graph (DAG) approach described by Bofill et al. [BBEV23], which has general applications for SAT based graph reachability problems including those on a grid. This encoding makes use of the properties of a spanning tree to enforce connectivity. The paper proposes two ways to prevent cycles from forming, to ensure a valid spanning tree structure. The methods discussed in the paper make use of an ordering using distance variables to prevent cycles or by ordering the tiles in the path using transitivity. We chose to implement the distance based method for this thesis. As we will compare two encoding strategies that make use of a Directed Acyclic Graph, we chose to name this encoding *Topological Order* for the cycle prevention method used. Though we do not strictly enforce a topological order in our implementation, as that would needlessly constrain the solution space.

3.2 Double Tree Encoding

This thesis builds on the work by Hannah Straathof [Str25], who proposed a novel Double Tree encoding for the continuity constraint in grid-based puzzles. Straathof's work demonstrated that this encoding is significantly more efficient in terms of both the number of clauses and variables compared to traditional encodings, such as the Topological Order encoding we will be comparing it against. The Double Tree encoding uses a clever construction that leverages the grid's structure to enforce connectivity without requiring distance propagation, which is used in the Topological Order encoding.

3.3 Encoding Size vs. Solver Performance

While more compact SAT encodings can reduce memory demands, their effect on solver runtime is not always straightforward. It is a well-observed phenomenon that smaller encodings do not invariably lead to faster solution times; the interplay between the encodings structure and the heuristics employed by modern SAT solvers (like conflict-driven clause learning, CDCL) is critical. For example, Wynn [Wyn18], in a comparison of SAT encodings for cardinality constraints, found that encodings generating more clauses, such as the sequential counter, could outperform more compact encodings like some binary representations. This was attributed to more effective unit propagation, which can guide the solver more efficiently. Wynn explicitly states that "clause count and other measures of encoding size are not reliable indicators of the difficulty of a SAT problem" [Wyn18].
These findings underscore the motivation for comparing SAT encodings that may differ significantly in size and in their underlying structural approaches, such as the Topological-Order and Double-Tree strategies we investigate, as their impact on solver performance can be substantial and not predicted by size alone.

4 Continuity Constraint Encodings

The continuity constraint—ensuring that all white tiles form a single connected region—is the most complex rule in the *Context* puzzle and the primary focus of our comparison. It traditionally requires an encoding that scales quadratically with the size of the grid. This section details two distinct SAT encodings for this constraint: the *Topological Order* encoding and the *Double Tree* encoding. We explain how each encoding works and how it successfully enforces connectivity by forming a spanning tree.

4.1 Mapping the problem from Graph to Grid

Many existing SAT encodings for connectivity constraints are based on graph-theoretic principles, where the grid is treated as a directed graph with white tiles as nodes and orthogonal adjacency as possible edges. We then want to make use of the spanning tree, as it is a minimal connected subgraph. If we can form a spanning tree over the white tiles, we can guarantee that all white tiles are connected. We do this by firstly selecting a root tile which can be any tile as long as it is part of the subgraph of white tiles. We then require that all white tiles have an outgoing arrow pointing to one of their orthogonal white neighbours, this arrow must point to another white tile that also has an outgoing arrow that does not point back to the original tile. We can do all this using only local information, meaning each tile only needs to consider its immediate neighbours. However, some white tiles may form a directed cycle that is not part of the spanning tree and is hard to detect without some form of information propagation. We discuss two ways to prevent cycles from forming in the following sections.

4.2 Topological Order Encoding

The Topological Order encoding, enforces continuity by assigning a distance value to each white tile, representing the distance along some path to a designated root tile.

4.2.1 How the Encoding Works

Each white tile (i, j) is associated with a Boolean variable $d_{i,j,k}$ for every possible distance value $k \in [0, \text{maxDist}]$. The meaning of these variables is as follows:

- A white tile can only have distance $k > 0$ if at least one of its orthogonal neighbours has distance $k - 1$.
- Multiple $d_{i,j,k}$ may be true for a white tile (i, j); none may be true for a black tile.
- The root tile has $d_{i,j,0}$ true while all other tiles may not have distance 0 true.

This enforces a local propagation of distance from the root, ensuring that all white tiles are reachable from the root through a valid path. This can be considered a directed graph where each white tile is a node with one or multiple outgoing arrows pointing to one or multiple orthogonal neighbours with a distance to the root. If all white tiles form a connected graph, this means that all white tiles are reachable from the root tile.

4.2.2 Root Selection Strategy

We select the top-left tile $(0, 0)$ as the root. If it is black (e.g., constrained to be so by other puzzle rules), we choose $(0, 1)$ instead, which then must be white due to the puzzle's no adjacent black tiles rule. This guarantees a white root tile.

4.2.3 How Cycles Are Prevented

The encoding guarantees acyclicity by the following logic:

1. Distances increase strictly along any path from the root.
2. No tile can point 'backwards' to a neighbour of equal or higher distance.
3. Cycles are structurally impossible: a closed path would require distance to both increase and decrease (see Figure 6).

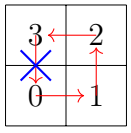


Figure 6: A cycle in the Topological Order encoding, showing how the constraints are violated in the case of a cycle.

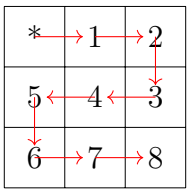


Figure 7: A long snake-like path in a 3x3 grid, illustrating the maximum path length $K_{\max}$ for a 3×3 grid of 8.

4.2.4 Complexity Challenge

This encoding introduces $O(N \cdot K_{\max})$ (see 5.2.1) variables and clauses, where N is the total number of tiles and $K_{\max}$ is the maximum possible distance of a path. For worst-case configurations in a grid problem where paths between tiles may be restricted can approach $N - 1$ as shown in Figure 7, leading to a substantial encoding size of $O(N^2)$.
The value of $K_{\max}$ should be at least the length of the longest shortest path between any two tiles in the grid. To establish a lower bound for $K_{\max}$, we make use of a scalable pattern that can be applied to any square grid of size N. As we can not have straight walls of black tiles due to the no adjacent black tiles rule, traditional long paths on grids are not possible. However one way to create a long path is to place zigzagging walls every three tiles, as shown in Figure 8. On small grids without walls, the longest shortest path is one that walks the distance once horizontally and once vertically, which yields distance $2n - 2$ for a grid of size $n \times n$. Every added wall increases the maximum distance by one horizontal walk. Because walls require an empty row on either side, walls are added at intervals of three tiles starting from 4×4 grids (4,7,10,13...).

This can be expressed as:

$$L(n) = 2n - 2 + (n - 1) \left\lfloor \frac{n-1}{3} \right\rfloor, \tag{1}$$

where $L(n)$ is the longest path length in a grid of size $n \times n$ using such zigzagging walls.
We can resolve formula (1) to find a lower bound for the longest path length:

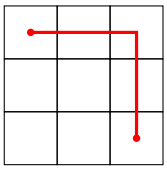
$$\begin{aligned} L(n) &= 2n - 2 + (n - 1) \left\lfloor \frac{n-1}{3} \right\rfloor \\ &\geq 2n - 2 + (n - 1) \left(\frac{n-1}{3} - \frac{2}{3} \right) \\ &= 2n - 2 + \frac{(n-1)(n-3)}{3} \\ &= \frac{n^2 + 2n - 3}{3} \\ &= \frac{(n-1)(n+3)}{3} \end{aligned} \tag{2}$$

From this, we can conclude that the lower bound for the maximum distance is in the order of

$$L(n) = O(n^2) = O(N), \quad \text{where } N = n \times n \text{ is the total number of tiles in the grid.}$$

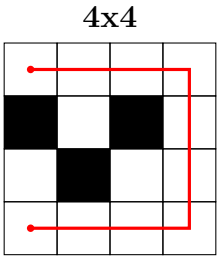
Consequently, since each tile may require variables representing its position along the longest path, the total number of variables in the Topological Order encoding scales as

$$O(N \cdot L(n)) = O(N^2).$$



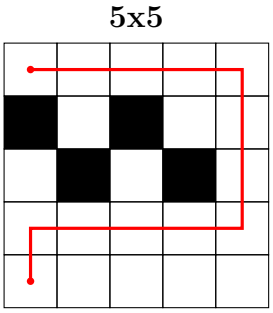
(a) No walls, one horizontal walk

$$L(3) = 2 \cdot 3 - 2 + 2 \cdot \left\lfloor \frac{2}{3} \right\rfloor = 4$$



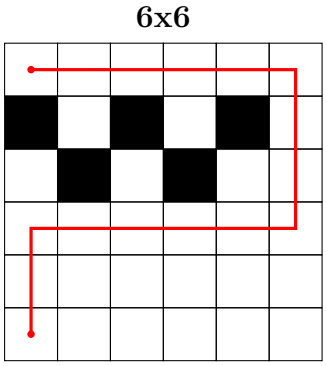
(b) One wall, two horizontal walks

$$L(4) = 2 \cdot 4 - 2 + 3 \cdot \left\lfloor \frac{3}{3} \right\rfloor = 9$$

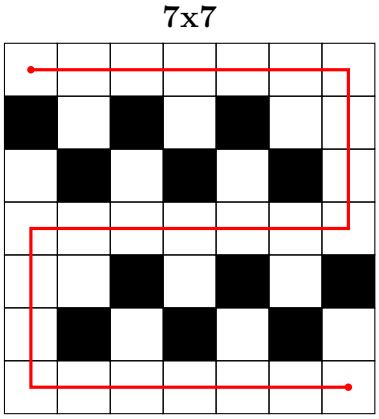


(c) One wall, two horizontal walks

$$L(5) = 2 \cdot 5 - 2 + 4 \cdot \left\lfloor \frac{4}{3} \right\rfloor = 12$$



(d) One wall, two horizontal walks



(e) Two walls, three horizontal walks

$$L(6) = 2 \cdot 6 - 2 + 5 \cdot \left\lfloor \frac{5}{3} \right\rfloor = 15$$

$$L(7) = 2 \cdot 7 - 2 + 6 \cdot \left\lfloor \frac{6}{3} \right\rfloor = 24$$

Figure 8: Zigzagging walls force the path to wind through the grid, increasing its length according to $L(n) = 2n - 2 + (n-1) \cdot \left\lfloor \frac{n-1}{3} \right\rfloor$.

4.3 Double Tree Encoding

The Double Tree encoding enforces connectivity through a constructive layering of constraints, progressively eliminating invalid configurations until only a single connected white region remains.

This approach, inspired by a technique from Klaus Reinhardt [Rei98], adapted for continuity in SAT encodings in a 2D grid by Straathof [Str25], uses two non-intersecting spanning trees: one on the grid tiles and one on the grid vertices. The final structure guarantees global connectivity using only local constraints.

4.3.1 Constructing the Encoding Step by Step

Step 1: The Root Tile We start by selecting a root tile, this will serve as the root of the spanning tree and may be placed on any white tile.

Step 2: Enforcing One Outgoing Arrow per Tile We then require that each tile (except for the root tile) have exactly one outgoing arrow pointing to one of its orthogonal white neighbours. By enforcing these rules the white tiles on the grid decompose into multiple disjoint directed graphs, each of which is either a separate directed cycle, a tree rooted by a directed cycle or a tree starting from the root such as is shown in Figure 9.

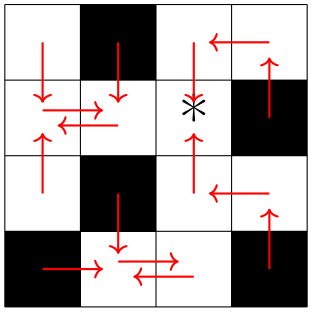


Figure 9: A 4x4 grid with a root tile at (2,1). Arrows assigned such that each (non-root) tile has exactly one outgoing arrow to a white orthogonal neighbour.

Step 3: Avoiding Cycles with Local Constraints To prevent immediate local violations, we disallow any pair of tiles from pointing into each other, eliminating 2-cycles. However, longer cycles remain possible. These cycles isolate white regions without local indication that they are not part of the spanning tree.

Step 4: Introducing the Vertex Tree We will now introduce a second tree, this tree has nodes at the corners of the grid tiles, we will refer to this as the *vertex tree*. Each vertex must also have one outgoing arrow to a neighbouring vertex, except the vertex root which we will further discuss later, forming a second tree. Crucially, we impose a non-intersection constraint: tile-tree arrows and vertex-tree arrows may not share the same grid edge as in Figure 10. For the vertex tree we also introduce a rule that forbids 2-cycles as described in the previous step.

By enforcing these constraints any cycle in either tree would mean splitting the other tree into two disconnected components. Each of these components must either form a cycle or a tree in order to be valid. However forming a cycle inside the other cycle would increasingly restrict the space inside the cycle until only a subset of nodes that can't form a cycle is left in either tree. This

subset of nodes requires an outgoing arrow unless one of them is the root node. Thus, the only remaining valid configurations according to the rules thus-far that do not require the white tiles to be connected, is one with the root in the middle as shown in Figure 11.

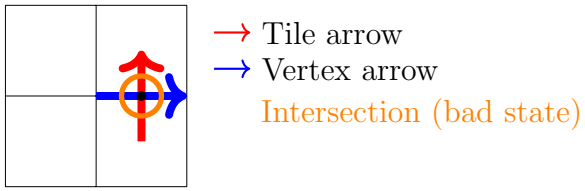


Figure 10: A 2x2 grid showing a tile arrow (red) and a vertex arrow (blue) intersecting at the same edge, which is forbidden in the Double Tree encoding.

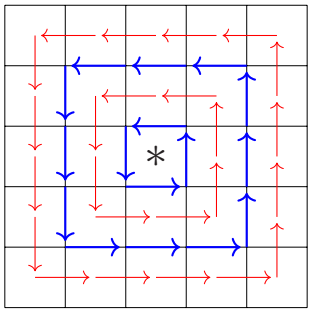


Figure 11: A white tile is enclosed by a cycle. It requires an outgoing edge, but every possible direction intersects an edge in the other tree, violating the non-intersection constraint unless the middle tile is the root.

Step 5: Root Cycle Loophole A remaining loophole allows a cycle to form around either of the roots. To close this gap, we place the root tile at the boundary of the grid. A cycle enclosing it would have to wrap around the edge of the grid, which is not possible. For the vertex root we consider the entire outer boundary of the grid as root. This simplifies solutions and works because a valid path between two vertices on the outer edge of the grid always exists. With this, both the tile and vertex graphs form spanning trees rooted at the grid's boundary. This enforces that a valid configuration can only exist if all white tiles form a single continuous area.

4.3.2 Final Result: Two Spanning Trees, One White Region

By layering these constraints, the encoding can only be satisfied if all white tiles form a single continuous area, we visualize what a valid solution to the encoding looks like in Figure 12. This means we have now encoded the continuity constraint entirely through local constraints and structural properties of the grid.

4.3.3 Complexity Advantage

This encoding remains efficient. Each tile and vertex has only a constant number of directions to choose from, and each constraint affects a small local neighbourhood. The total number of

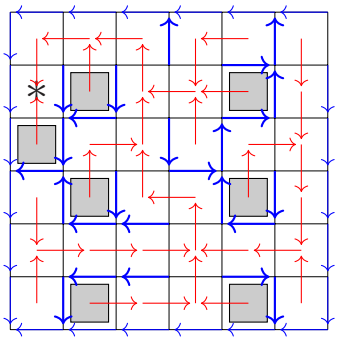


Figure 12: Valid solution to our Double Tree encoding. White tiles form a single spanning tree; black tiles are leaves. The blue arrows form a single spanning tree connecting all vertices, the outer edge of arrows is redundant and for our implementation we treat the whole outer edge as root for the vertex tree. (Figure obtained from [Hoo25])

variables and clauses is $O(N)$, as we will explain in 5.2.2, making the Double Tree encoding highly scalable for large puzzles [Str25].

4.4 Comparison Summary

Both encodings enforce global connectivity but rely on fundamentally different strategies: one on distance propagation and hierarchical structure this requires a number of variables and clauses quadratic in the number of tiles (See Table 1), the other on dual spanning trees using local constraints scaling linearly with the number of tiles (See Table 1).

Property	**Topological Order**	**Double Tree**
Variables	$O(N \cdot K_{\max})$	$O(N)$
Clauses	$O(N \cdot K_{\max})$	$O(N)$
Cycle Prevention	Distance hierarchy	Tree structure + boundary roots
Key Mechanism	Explicit path distances	Dual spanning trees
Root Strategy	Single white tile	Tile tree + boundary vertices

Table 1: Comparison of the two continuity encodings, where N is the total number of tiles and $K_{\max}$ is the maximum distance of a path to the root.

5 CNF Encodings for Context

This section details the translation of the *Context* puzzle rules into Conjunctive Normal Form (CNF). We first address the constraints that only apply to a local neighbourhood of tiles, such as adjacency and clue constraints. In subsection 5.2 we provide the CNF for the continuity encodings discussed in the previous section.

5.1 Encoding Local Puzzle Constraints

In the case where a tile is outside the grid, we consider it to be white by default, any clauses that become arbitrarily true due to a tile outside the grid is removed.

5.1.1 Encoding the Adjacency Constraint

The rule that "no two black tiles may be orthogonally adjacent" is a constraint that goes for the whole grid. For any tile (i, j) if it is black ($\neg w_{i,j}$), its neighbours may not be black. To encode this we only have to consider tiles to the right and bottom of each tile (i, j) due to symmetry. This can be expressed for a tile (i, j) as:

$$\neg w_{i,j} \rightarrow (w_{i,j+1} \wedge w_{i+1,j})$$

By resolving the implication and applying the distributive law, we obtain the CNF representation, which consists of two clauses per tile (where applicable for boundary conditions):

$$(w_{i,j} \vee w_{i,j+1}) \wedge (w_{i,j} \vee w_{i+1,j})$$

5.1.2 Encoding a 0-Clue

A tile (i, j) with a 0-clue imposes constraints based on its color.

- If the tile is white ($w_{i,j}$), all four of its orthogonal neighbours must also be white.
- If the tile is black ($\neg w_{i,j}$), all four of its diagonal neighbours must be white.

A key observation is that if tile (i, j) contains a 0-clue, its orthogonal neighbours must be white regardless of the color of (i, j). This is because if (i, j) is white, the clue forces them to be white; if (i, j) is black, the general adjacency constraint forces them to be white. Therefore, we can assert:

$$w_{i-1,j} \wedge w_{i,j+1} \wedge w_{i+1,j} \wedge w_{i,j-1}$$

Additionally, for the case where the 0-clue tile is black, we add constraints for the diagonal neighbours:

$$\neg w_{i,j} \rightarrow (w_{i-1,j-1} \wedge w_{i-1,j+1} \wedge w_{i+1,j-1} \wedge w_{i+1,j+1})$$

This translates to up to four clauses (considering boundary conditions) :

$$(w_{i,j} \vee w_{i-1,j-1}) \wedge (w_{i,j} \vee w_{i-1,j+1}) \wedge (w_{i,j} \vee w_{i+1,j-1}) \wedge (w_{i,j} \vee w_{i+1,j+1})$$

5.1.3 Encoding a 1-Clue

A 1-clue on tile (i, j) requires that if the tile is white, exactly one of its orthogonal neighbours is black. This is an "exactly-one" constraint. A standard CNF encoding for this involves two components: "at-least-one" and "at-most-one".

- **At-most-one** black neighbour: For every pair of neighbours, at least one must be white.
- **At-least-one** black neighbour: Not all neighbours can be white.

This leads to the following implication for a white 1-clue tile, where again $N_o(i,j)$ is the set of four orthogonal neighbours:

$$w_{i,j} \rightarrow \left(\bigwedge_{\{u,v\} \subseteq N_o(i,j), u \neq v} (w_u \vee w_v) \right) \wedge \left(\bigvee_{u \in N_o(i,j)} \neg w_u \right)$$

When removing the implications, this results in 7 clauses for the white case:

For all 6 distinct pairs: $u, v \in N_o(i,j)$:	$(\neg w_{i,j} \vee w_u \vee w_v)$
At least one black neighbor:	$(\neg w_{i,j} \vee \neg w_{u_1} \vee \neg w_{u_2} \vee \neg w_{u_3} \vee \neg w_{u_4})$

A similar set of clauses is generated for the case where the tile is black ($\neg w_{i,j}$), constraining the diagonal neighbours instead. This results in 7 clauses for the white case and 7 for the black case, totaling 14 clauses per 1-clue.

5.1.4 Encoding a 2-Clue

A 2-clue on tile (i,j) requires that if the tile is white, exactly two of its orthogonal neighbours are black. This is an "exactly-two" constraint, which can be encoded by enforcing both:

- **At-most-two** black neighbours: Any three neighbours cannot all be black.
- **At-least-two** black neighbours: Any three neighbours cannot all be white.

This leads to the following implication for a white 2-clue tile, where $N_o(i,j)$ is the set of four orthogonal neighbours:

$$w_{i,j} \rightarrow \left(\bigwedge_{\{u,v,z\} \subseteq N_o(i,j)} (w_u \vee w_v \vee w_z) \right) \wedge \left(\bigwedge_{\{u,v,z\} \subseteq N_o(i,j)} (\neg w_u \vee \neg w_v \vee \neg w_z) \right)$$

When removing the implication, this results in 8 clauses for the white case:

For all 4 distinct triples $u, v, z \in N_o(i,j)$:	$(\neg w_{i,j} \vee w_u \vee w_v \vee w_z)$
For all 4 distinct triples $u, v, z \in N_o(i,j)$:	$(\neg w_{i,j} \vee \neg w_u \vee \neg w_v \vee \neg w_z)$

An analogous set of clauses is generated for the case where the tile is black ($\neg w_{i,j}$), constraining the diagonal neighbours instead. This results in 8 clauses for the white case and 8 for the black case, totaling 16 clauses per 2-clue.

5.1.5 Encoding a 3-Clue

A 3-clue on a white tile implies that exactly three of its four orthogonal neighbours are black, which is equivalent to saying exactly one is white. This allows us to reuse the logic for the 1-clue encoding, but with the values of the neighbour variables flipped. This results in 14 clauses per 3-clue, covering both the white and black cases for the clue tile.

5.1.6 Encoding a 4-Clue

A 4-clue on a tile (i, j) has only one possible interpretation: the tile itself must be black, and all four of its diagonal neighbours must also be black. This is because if the tile were white, all four orthogonal neighbours would have to be black, violating the continuity rule. Therefore, we can directly assert the state of these five tiles as unit clauses:

$$\neg w_{i,j} \land \neg w_{i-1,j-1} \land \neg w_{i-1,j+1} \land \neg w_{i+1,j-1} \land \neg w_{i+1,j+1}$$

5.2 Encoding the Continuity Constraints

5.2.1 Topological Order Encoding

As described in Section 4.2, $T_{i,j,k}$ is a boolean variable that is true if tile (i, j) is white and has a neighbour with value $T_{i,j,k-1}$, where k is the distance along some path from the root tile.
The main constraints are:

1. **Distance Propagation:** Each tile can only have distance $k > 0$ if at least one of its orthogonal neighbours has distance $k - 1$.

   $$T_{i,j,k} \rightarrow (T_{i-1,j,k-1} \lor T_{i,j-1,k-1} \lor T_{i+1,j,k-1} \lor T_{i,j+1,k-1})$$

   Resolving the implication yields a clause for each tile (i, j) and distance $k \in \{1, \ldots, K_{\max}\}$, where $K_{\max} = N - 1$ is the maximum possible distance a tile can be from the root:

   $$(\neg T_{i,j,k} \lor T_{i-1,j,k-1} \lor T_{i,j-1,k-1} \lor T_{i+1,j,k-1} \lor T_{i,j+1,k-1})$$

   This results in $N \times K_{\max}$ clauses.

2. **Black Tile Constraints:** A black tile (i, j) cannot have a distance value for any k.

   $$\neg w_{i,j} \rightarrow \bigwedge_{k=0}^{K_{\max}} \neg T_{i,j,k} \quad \text{or} \quad w_{i,j} \lor (\neg T_{i,j,k} \text{ for all } k \in \{0, \ldots, K_{\max}\})$$

   This adds $N \times (K_{\max} + 1)$ clauses.

3. **White Tile Requirements:** All white tiles (i, j) must have at least one distance value $T_{i,j,k}$ for some k.

   $$w_{i,j} \rightarrow \bigvee_{k=0}^{K_{\max}} T_{i,j,k} \quad \text{or} \quad \neg w_{i,j} \lor (\bigvee_{k=0}^{K_{\max}} T_{i,j,k})$$

   This adds one clause per tile or N clauses.

4. **Root Assignment:** We conditionally assign the root based on the color of $(0, 0)$.

$$\neg w_{0,0} \vee T_{0,0,0} \quad \text{and} \quad w_{0,0} \vee T_{0,1,0}$$

We explicitly disallow any other tile to be the root by adding:

$$\neg T_{i,j,0} \quad \text{for all } (i, j) \neq (0, 0) \text{ and } (i, j) \neq (0, 1)$$

Finally we need to enforce that one of the two possible root tiles is actually the root:

$$w_{0,0} \vee \neg T_{0,0,0} \quad \text{and} \quad \neg w_{0,0} \vee \neg T_{0,1,0}$$

This adds 4 clauses for root assignment plus $N - 2$ unit clauses.

The total number of clauses for the Topological Order encoding scales quadratically with the number of tiles. The total number of clauses is approximately: $N \times K_{\max} + N \times (K_{\max} + 1) + N + N = N \times (2K_{\max} + 3) = O(N \times K_{\max})$, where $K_{\max} = N - 1$, resulting in $O(N^2)$ complexity.

5.2.2 Double Tree Encoding

This encoding uses arrow variables for tiles and vertices. Let $G_{i,j,D}$ be true if tile (i, j) has an arrow in direction $D \in \{N, E, S, W\}$, and $V_{i',j',D}$ be true if vertex (i', j') has an arrow. the tiles and vertices are numbered as follows:

- Tiles are numbered from $(0, 0)$ to $(R - 1, C - 1)$.
- Vertices are numbered from $(0, 0)$ to (R, C), where the vertex at (i', j') is the top left corner of the tile (i, j).

The main constraints are:

1. **Exactly-One-Arrow:** Every tile and every vertex must have exactly one outgoing arrow (except the roots, see 5). This is encoded using the same "at-least-one" and "at-most-one" logic as the 1-clue. For each tile (i, j) (except the roots, see 5):

$$\left(\bigvee_{D} G_{i,j,D} \right) \wedge \left(\bigwedge_{D_1 \neq D_2} (\neg G_{i,j,D_1} \vee \neg G_{i,j,D_2}) \right)$$

An analogous set of 7 clauses is added for each vertex totalling approximately 14 clauses per tile.

2. **Arrow to White Tile:** An arrow from a tile (i, j) pointing North must lead to a white tile $(i - 1, j)$.

$$\neg G_{i,j,N} \vee w_{i-1,j}$$

An analogous set of clauses is added for each of the other directions that do not point outside the grid, for those pointing outside of the grid we add the clause:

$$\neg G_{i,j,D} \quad \text{for } D \in \{N, E, S, W\} \text{ where direction is outside grid}$$

In total we add 4 clauses per tile.

3. **Non-Intersection:** A tile arrow and a vertex arrow cannot cross. For a tile arrow $G_{i,j,N}$, the corresponding edge is between vertices (i, j) and $(i, j+1)$. This edge cannot be used by vertex arrows crossing it.

$$\neg G_{i,j,N} \vee \neg V_{i,j,E} \quad \text{and} \quad \neg G_{i,j,N} \vee \neg V_{i,j+1,W}$$

We add another analogous set of clauses for each of the other directions that do not point outside the grid, for those pointing outside of the grid we skip adding the clause entirely. This results in approximately 8 clauses per tile.

4. **No 2-Cycles:** Opposing arrows between adjacent tiles are forbidden.

$$\neg G_{i,j,N} \vee \neg G_{i-1,j,S}$$

This is added for each pair of adjacent tiles and each pair of adjacent vertices, resulting in 2 clauses per tile and 2 per vertex as each has 4 neighbours and each clause is shared by 2 tiles or vertices. This yields approximately 4 clauses per tile.

5. **Root Handling:** We assign the root conditionally based on the color of $w_{0,0}$. If $(0, 0)$ is white, it is the root tile and has no outgoing arrow. If $(0, 0)$ is black, the root tile shifts to $(0, 1)$, which must be white by adjacency. We encode this by prefixing the Exactly-One-Arrow constraint for $(0, 0)$ in case $(0, 0)$ is black:

$$w_{0,0} \vee \left(\bigvee_{D} G_{0,0,D} \right) \wedge \left(\bigwedge_{D_1 \neq D_2} (\neg G_{0,0,D_1} \vee \neg G_{0,0,D_2}) \right)$$

Which can easily be transformed into CNF by distributing the $w_{0,0}$ over the clauses. We then also encode the case where $(0, 0)$ is white:

$$\neg w_{0,0} \vee \left(\bigvee_{D} G_{0,1,D} \right) \wedge \left(\bigwedge_{D_1 \neq D_2} (\neg G_{0,1,D_1} \vee \neg G_{0,1,D_2}) \right)$$

This does not introduce any additional clauses, as these replace the clauses for $(0, 0)$ and $(0, 1)$ described earlier. For the vertex root we can take the entire outer edge of the grid as the root, as a valid path between two vertices on the outer edge of the grid always exists. This allows us to remove all Exactly-One-Arrow constraints for the outer edge of the vertex grid.

The total number of clauses for the Double Tree encoding scales linearly with the number of tiles, as each tile contributes a constant number of clauses regardless of the grid size. The total number of clauses is approximately: $14N + 4N + 8N + 4N = 30N = O(N)$

5.2.3 Clause and Variable Comparison

Overall, the Double Tree encoding is significantly more compact than the Topological Order encoding for larger grids, scaling linearly with the number of tiles, while the Topological Order encoding scales quadratically.

6 Puzzle Generation

As *Context* puzzles are a niche puzzle type with limited availability, we developed a generation process to create valid *Context* puzzle instances. This section outlines the motivation for generating puzzles, the methodology used, and the characteristics of the generated instances.

6.1 Motivation for Puzzle Generation

The availability of human-designed *Context* puzzles is severely limited, with only 40 puzzles accessible from Janko.at [AJ25] and a few dozen across other sources we did not include in our test set. This small dataset is insufficient for robust statistical analysis and comprehensive evaluation of different SAT encodings. To address this limitation, we developed a generation process that produces valid *Context* puzzle instances. While we attempted to incorporate some characteristics observed in human-designed puzzles, these generated instances should primarily be understood as random uniquely solvable puzzle configurations rather than close approximations of human-designed puzzles.

6.2 Generation Methodology

Our puzzle generation process consists of five distinct phases, designed to produce valid *Context* puzzles that satisfy the basic structural requirements while maintaining unique solvability.

6.2.1 Phase 1: Determining Optimal Black Tile Density

The first step involves determining the optimal number of black tiles for a given grid size. We analyzed existing 6×6 *Context* puzzles and found that most puzzles have the same or similar black tile count, we therefore decided to adopt the most common value of 10 as the amount of black tiles for generated puzzles.
The number of black tiles is crucial for puzzle solvability and uniqueness. If there are too many black tiles, the continuity constraint (all white tiles must form a single connected region) becomes impossible to satisfy, as excessive black tiles can split the white area into disjoint regions. Conversely, if there are too few black tiles, the clues can often be satisfied in multiple ways, since each clue may allow several possible arrangements of black neighbours. To ensure a unique solution, the concentration of black tiles must be such that their positions restrict the possible arrangements due to the no-adjacent-black or white connectivity rule. Mimicking the most common black tile count in human-designed puzzles should provide a good balance to ensure unique solvability.

6.2.2 Phase 2: Initial Grid Generation

Using the predetermined optimal black tiles count, we generate random 6×6 grids by randomly placing the appropriate number of black tiles. During this phase, we enforce only one constraint: no two black tiles may be orthogonally adjacent, as required by the *Context* puzzle rules. This ensures that all generated grids satisfy the fundamental adjacency constraint.
The random placement process continues until a configuration with 10 non-orthogonally-adjacent black tiles is found. While this approach is straightforward, it effectively explores the space of possible grid layouts without introducing unintended biases towards particular patterns or structures.

6.2.3 Phase 3: White Tile Connectivity Verification

After generating a candidate grid layout, we verify that all white tiles form a single connected region. For a simple comparison and because we had already implemented it, we used a SAT solver to determine if the grid satisfies the white tile connectivity constraint. In a very basic comparison between the two encodings, we found that the Double Tree encoding performed significantly better than the Topological Order encoding for these random and potentially non-connected grids. We therefore decided on using Double Tree to determine white connectivity for generating large amounts of puzzle instances.

6.2.4 Phase 4: Clue Assignment

We repeat Phase 2 and 3 until a grid with proper white tile connectivity is confirmed, we assign numerical clues to each tile according to the *Context* puzzle rules:

- **White tiles**: Receive clues indicating the exact number of orthogonally adjacent black tiles
- **Black tiles**: Receive clues indicating the exact number of diagonally adjacent black tiles

At this stage, every tile contains it's clue value, creating a fully filled grid that is uniquely solvable.

6.2.5 Phase 5: Clue Minimization

The final phase involves systematically removing clues while preserving the puzzle's unique solvability. This process employs a greedy reduction algorithm:

1. Randomly select a clue to remove
2. Verify that the resulting puzzle maintains a unique solution using the SAT solver (see Section 2.4)
3. If uniqueness is preserved, permanently remove the clue
4. If uniqueness is lost, restore the clue
5. Repeat until no more clues can be removed

As a result, the puzzle reaches a minimal state where removing any remaining clue would allow for multiple solutions.

6.3 Characteristics of Generated Puzzles

The resulting puzzle instances exhibit the following characteristics:

- **Unique Solvability**: Each puzzle has exactly one valid solution, verified through exhaustive SAT solving
- **Minimality**: No clue can be removed without introducing multiple solutions

- **Structural Validity**: All puzzles satisfy the fundamental *Context* rules
- **Varied Difficulty**: The random generation and clue reduction process naturally produces puzzles of varying computational complexity, however the proportions of difficulties are unknown

6.4 Limitations and Considerations

It is important to acknowledge the significant limitations of our generation approach:

- **Limited Resemblance to Human Design**: The generated puzzles are essentially random configurations that happen to be uniquely solvable, rather than carefully crafted instances with intentional design patterns
- **Absence of Human Logic**: These puzzles may not be solvable using traditional human reasoning techniques and may require thinking many steps ahead

While these generated puzzles may not closely resemble human-designed *Context* puzzles in terms of solving techniques required in human solving, design intent or computational complexity, they serve the essential purpose of providing a substantially larger dataset for comparative analysis of SAT encodings. Determining similarity between generated and authentic puzzles is evaluated using a statistical comparison in Section 7.

7 Validating the Use of Generated Puzzle Data

As described in the previous section, a large number of puzzles were generated algorithmically to support the experiments in this thesis. However, in order to use these generated puzzles as valid data points, we must first determine whether they are comparable to human-designed puzzles in terms of solver difficulty. If their characteristics deviate significantly, conclusions based on generated data may not generalize to human-designed puzzle design.
To that end, we compare the generated puzzles to a set of 40 human-designed puzzles obtained from `Janko.at` [AJ25]. Since puzzle size strongly influences solving time, we restrict our comparison to only the 6x6 puzzles, of which 9 are included in the Janko set. We have a generated puzzle set of 6,485 puzzles that are also of size 6x6. Additionally, to determine if the generated puzzles are representative we make a comparison of the two encodings: *DoubleTree* and *Topological Order*. For each encoding, we do statistical tests to determine if there is a significant difference in solver cycles between the human-designed and generated puzzles. As our null hypothesis, we assume that the generated puzzles are representative of the human-designed puzzles in terms of solver cycles. We reject the null hypothesis if the p-value is below a significance level of 0.05. In that case, we conclude that the generated puzzles showcase significantly different characteristics compared to the human-designed puzzles.

7.1 Statistical Methodology

To assess similarity between the human-designed and generated puzzles, we used two statistical approaches.

- **Bootstrap Simulation:** We repeatedly draw 9-puzzle samples from the generated dataset and compute their mean solving time. By repeating this process 100,000 times, we obtain a distribution of sample means, from which we calculate a 95% confidence interval and a p-value indicating how often a sample as extreme as the human-designed puzzles occurs by chance.
- **One-Sample Z-Test:** As a classical alternative, we compute a Z-score to quantify how many standard errors away the human-designed mean is from the generated mean. This test also provides a p-value under the assumption of normality.

For our small human-designed sample of 9 puzzles, many statistical tests are unreliable. For Z-tests we have to assume that the generated puzzles are normally distributed, this is hard to determine for a small sample. Bootstrap is more robust for small datasets as it does not make strong assumptions about the underlying distribution. We therefore use the bootstrap method as our primary analysis, while the Z-test may provide additional insights assuming normality. Both tests are however limited by the small sample size of human-designed puzzles, which affects the reliability of the results.

7.2 Results

Table 2: Comparison of Human-Designed and Generated 6x6 Puzzles (SolverCycles)

Encoding	Test	Human-Designed Mean	P-value	95% CI (Generated)
DoubleTree	Bootstrap	191,266.70	0.3762	[150,116.30, 690,475.10]
	Z-Test	191,266.70	0.5767	–
Topological	Bootstrap	2,018,524.90	0.0129	[688,422.73, 1,803,219.73]
	Z-Test	2,018,524.90	0.0080	–

As shown in Table 2, the representativeness of the generated puzzles differs significantly by encoding.

For the **DoubleTree encoding**, the results show no statistically significant difference between the human-designed and generated puzzles. Although the generated puzzles are on average somewhat harder to solve than the human-designed ones, both the bootstrap ($p = 0.3762$) and Z-test ($p = 0.5767$) fail to reject the null hypothesis. The human-designed mean lies comfortably within the 95% confidence interval of generated sample means, suggesting a meaningful similarity.

In contrast, for the **Topological encoding**, both the bootstrap test ($p = 0.0129$) and Z-test ($p = 0.0080$) reject the null hypothesis at a significance level of 0.05. The average solving time of the human-designed puzzles is substantially higher than that of the generated ones, and the human-designed mean falls outside the 95% confidence interval. This suggests that the generated 6x6 puzzles are significantly easier to solve than their human-designed counterparts when using the Topological encoding. As for the DoubleTree encoding, the human-designed puzzles fall on the lower end of the confidence interval implying they are slightly easier to solve, though significance is not reached at the 0.05 level.

These findings indicate that our generated puzzles are only representative for the DoubleTree encoding. For the Topological encoding, they are significantly easier than their human-designed counterparts. This discrepancy means that conclusions drawn from the generated set for the Topological encoding may not generalize perfectly to human-designed puzzles. Despite this limitation, we will proceed with comparing both encodings on the full puzzle set. The primary aim is to observe the performance differences across a large dataset, acknowledging that the results for the Topological encoding are not indicative of the encodings' ability to solve human-designed *Context* puzzles. The results rather show more generally how well the encodings perform when solving valid instances of the *Context* puzzle. The core comparison remains valuable for understanding the efficiency of each approach for solving valid *Context* puzzles, and both samples show an improvement in performance when using the DoubleTree encoding.

8 Experiments

8.1 Experimental Setup

We conducted experiments to compare the performance of the Topological Order and Double Tree encodings for the *Context* puzzle. The goal was to evaluate how each encoding affects the computational efficiency of solving the puzzle using a SAT solver. One obvious way the encodings can impact the runtime is by reaching the memory limit or more likely causing cache misses on lower levels of cache. This would be due to one encoding having a significantly higher number of variables and clauses than the other. Our evaluation will not include puzzles large enough to reach the memory limit and is almost exclusively influenced by cache behaviour and the interaction between the solver and the encoding.

8.1.1 Solver and Hardware Configuration

We used CadiCal version 2.1.3 for our experiments [Bie24]. The tests were run on a Ryzen 3900X chip with 32GB of RAM. The human-designed puzzle set is sourced from `https://www.janko.at` [AJ25] and consist of 40 puzzles. We also generated a large set of 6,485 6x6 puzzles using the generation process described in Section 6. For these puzzles we calculate how many CPU cycles the solver took for each encoding to determine satisfiability by measuring CPU cycles using the C++ rdtsc interface.

During work on this thesis we collected results using CadiCal and Kissat [Bie21]. We noted significant performance differences between the two solvers, which were not consistent as some puzzles performed better with CadiCal while others performed better with Kissat. For our experiments we ended up using CadiCal as it allowed us to make use of incremental solving which is not supported by Kissat. This was used to ensure a puzzle is uniquely solvable, which is a requirement for *Context* puzzles. Future work could explore the performance differences between different solvers and encodings in more detail.

8.1.2 Statistical Analysis

To compare the performance of the two encodings, we made use of a paired t-test to determine if the differences in solver cycles are statistically significant. The paired t-test is appropriate here as we compare the same set of puzzles under two different encodings, allowing us to control for individual puzzle characteristics. Our null hypothesis is that there is no significant difference in the mean solver cycles between the two encodings. We calculate the mean and standard deviation of the solver cycles for each encoding and then apply the t-test to assess whether the observed differences are statistically significant at a 95% confidence level.

8.2 Results

To evaluate the performance of the Double Tree encoding relative to the Topological Order encoding, we analyze three key metrics: solver cycle speed-up, variable reduction, and clause reduction. We first present the results from the original dataset of 40 human-designed puzzles, grouped by puzzle size. We then use a large set of 6,485 generated 6x6 puzzles to validate these findings and provide a more robust statistical comparison.

8.2.1 Performance on Human-Designed Puzzles

The original dataset contains puzzles of varying dimensions. The performance improvements of the Double Tree encoding become more pronounced as the puzzle size increases. Table 3 summarizes the speed-up and reduction ratios for different puzzle size groups. The number of variables used is identical for puzzles of the same size. The number of clauses show small variations due to differences in clues in the puzzle, however they show very low standard deviations and ranges as most of the clauses are determined by the grid size.

Table 3: Performance Ratios of Double Tree vs. Topological Order on Human-Designed Puzzles

Metric	6x6 Puzzles (9)	7x7 Puzzles (4)	8x8 Puzzles (8)	10x10 Puzzles (17)
Solver Cycle Speed-up				
Geometric Mean	2.94x	3.11x	5.80x	8.25x
Median	2.97x	3.18x	5.58x	6.85x
Geometric Std Dev	1.68	1.19	1.88	2.21
Range	1.37x - 6.75x	2.38x - 3.86x	2.41x - 14.34x	1.55x - 33.95x
Variable Reduction				
Geometric Mean	2.43x	3.36x	4.45x	7.07x
Geometric Std Dev	1.00	1.00	1.00	1.00
Clause Reduction				
Geometric Mean	4.93x	6.62x	8.74x	13.57x
Geometric Std Dev	1.02	1.01	1.02	1.02

These findings are reinforced by the solver speed-up histograms for human-designed puzzles shown in Figure 13. In particular, for the 6x6 human-designed puzzles in Figure 13 we see that only 2 out of 9 data points fall below a speed up of 2.4x which is towards the right end of the distribution for the generated puzzles in Figure 14. This visual separation supports the earlier statistical

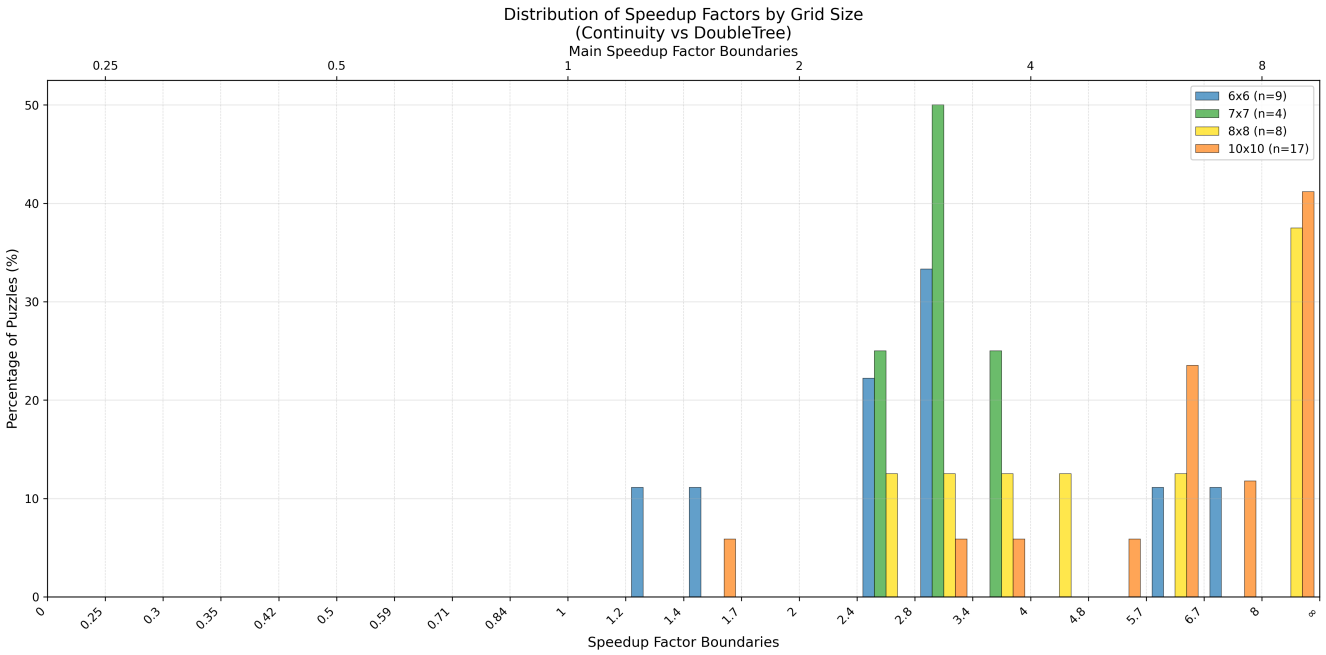


Figure 13: Solver speed-up distribution by grid size for human-designed puzzles, values to the right of the 1.0 line indicate a speed-up when using the Double Tree encoding. Bar height shows percentage of puzzles in the respective bin for that grid size.

observation that human-designed puzzles benefit more from the Double Tree encoding than the generated ones. Notably, there are no instances among the 40 human-designed puzzles where the Topological Order encoding outperformed the Double Tree encoding.

The solver speed-up consistently increases with puzzle size: 2.94x for 6x6, 3.11x for 7x7, 5.80x for 8x8, and 8.25x for 10x10 puzzles. This trend is clearly visible in the histogram plots in Figure 13, where the distributions for larger grid sizes progressively shift to the right. The results confirm that the performance advantage of the Double Tree encoding is not only consistent but becomes increasingly pronounced on more complex puzzles, both in terms of solver cycles and encoding compactness.

8.2.2 Validation with Generated Puzzles

To confirm the trends observed in the smaller dataset, we analyzed the performance on 6,485 generated 6x6 puzzles. While these puzzles are not representative of human design, as discussed in Section 7, they provide a large sample for assessing the baseline performance difference between the encodings when solving valid instances of *Context* puzzles. The results are summarized in Table 4.

The results from the generated dataset strongly support our initial findings. The mean variable reduction (2.43x) and clause reduction (4.79x) are nearly identical to those observed for the human-designed 6x6 puzzles (2.43x and 4.93x, respectively), with extremely low standard deviations, confirming the consistency of the encoding size reduction. The average solver speed-up of 1.80x, while lower than the 2.94x for human-designed puzzles, still demonstrates a significant and consistent performance gain, as indicated by the relatively low coefficient of variation (34.75%). A

Table 4: Performance Ratios on 6,485 Generated 6x6 Puzzles

Metric	6x6 Performance
Solver Cycle Speed-up	
Geometric Mean	1.80x
Median	1.78x
Geometric Std Dev	1.39
Range	0.21x - 6.46x
Variable Reduction	
Geometric Mean	2.43x
Geometric Std Dev	1.00
Clause Reduction	
Geometric Mean	4.79x
Geometric Std Dev	1.01

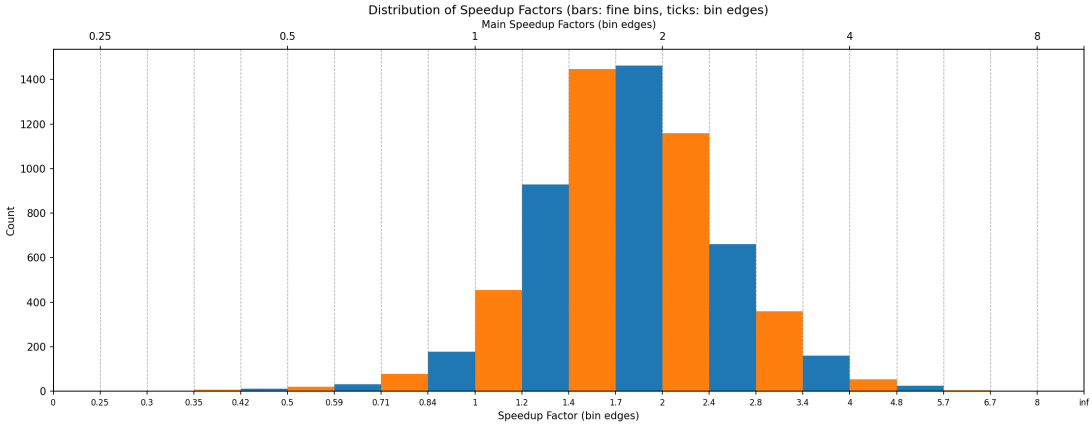


Figure 14: Histogram of Solver Speed-up Ratios for 6,485 Generated 6x6 Puzzles

histogram of solver speed-up ratios (Figure 14) reveals that the generated puzzles exhibit a roughly normal distribution centered around a mean speed-up of 1.80x. Notably, only 173 puzzles (2.71%) were solved faster using the Topological Order encoding. This further highlights the robustness of the Double Tree approach in the majority of cases.

Overall, both datasets confirm that the Double Tree encoding provides substantial and consistent improvements in encoding size and solver efficiency, making it the preferable choice for solving *Context* puzzles.

8.3 Statistical Significance of Results

To evaluate the performance difference between the Topological Order and DoubleTree encodings, a paired t-test was conducted on solver cycle counts from 6,485 puzzles. Table 5 summarizes the key statistics and test results.

As shown in Table 5, the mean difference in solver cycles between the two encodings is approxi-

Table 5: Summary statistics and paired t-test results for SolverCycles (Topological Order vs DoubleTree).

Metric	**Value**
Number of puzzles	6,485
Mean difference (Topological Order - DoubleTree)	5,598,832.93
Standard deviation of differences	4,520,563.05
Cohen's d (effect size)	1.2385
t-statistic	99.7378
p-value	< 0.0001

mately 5.6 million cycles, with Topological Order consistently requiring more cycles than DoubleTree. The paired t-test yields a highly significant result ($t(6484) = 99.74$, $p < 0.0001$), indicating that this difference is unlikely to be due to chance. Moreover, the large effect size (Cohen's $d = 1.24$) confirms that the observed speedup with DoubleTree encoding is statistically significant and represents a substantial improvement in solver efficiency.

9 Conclusions and Further Research

This thesis investigated two distinct SAT encodings for the continuity constraint in *Context* puzzles: the Topological Order encoding and the Double Tree encoding. Through our experimental evaluation across both human-designed and generated puzzle instances, we found clear evidence that the Double Tree encoding significantly outperforms the Topological Order encoding across all measured metrics.

9.1 Key Findings

Our experimental results demonstrate that the Double Tree encoding offers substantial advantages:

- **Computational Efficiency**: The Double Tree encoding consistently required fewer CPU cycles to solve puzzles. The speed-up factor for human-designed puzzles increases with puzzle size, from an average of 2.94x for 6x6 puzzles to 8.25x for 10x10 puzzles, demonstrating enhanced performance on larger instances. For the large set of generated 6x6 puzzles, we observed a mean speed-up of 1.80x, with only 2.71% of puzzles being solved faster using the Topological Order encoding. The paired t-test confirmed that this performance difference is statistically significant ($p < 0.0001$) with a large effect size (Cohen's $d = 1.24$).

- **Encoding Size**: As predicted by our theoretical analysis, the Double Tree encoding generated significantly fewer variables and clauses. As these values are directly related to the size of the grid they have very low variability with clauses being slightly affected by the clues of a puzzle. Theoretically, the Topological Order encoding scales as $O(N \cdot D)$ where D is at least $N/3$ as established in subsection 4.2.4, the Double Tree encoding maintains $O(N)$ complexity.

9.2 Theoretical Contributions

Our work shows that CadiCal benefits from the Double Tree encoding as compared to the Topological Order encoding, for larger puzzles, the Double Tree encodings performance advantage grows, making it a more scalable solution for larger *Context* puzzles and likely other grid-based problems with similar connectivity constraints.

9.3 Future Work

While our findings strongly favor the Double Tree encoding, further research would have to be done to verify its performance on grids that are arranged differently, as the data we tested on is fairly uniform in terms of how many black tiles are present and how they are arranged.
Additionally, our validation revealed that generated puzzles are not fully representative of human-designed puzzles for the Topological encoding ($p = 0.0129$), which means the effectiveness on human-designed puzzles is not completely understood. Future work should focus on developing better puzzle generation methods that more closely mimic human designed puzzles, or alternatively obtaining larger datasets of authentic puzzles for more robust statistical analysis.
Our work also showed that the choice of SAT solver can significantly impact performance. Future research could explore how different solvers handle these encodings.
Finally, while our focus was on *Context* puzzles, the Double Tree encoding may have broader applications in other types of grid-based puzzles or problems requiring connectivity constraints. Future work could explore its applicability in these domains.

References

[AJ25] Otto Janko Angela Janko. Website von Angela und Otto Janko. https://www.janko.at, 2025. Accessed: 2025-05-24.

[BBEV23] Miquel Bofill, Cristina Borralleras, Joan Espasa, and Mateu Villaret. On grid graph reachability and puzzle games. *arXiv preprint arXiv:2310.01378*, 2023.

[Bie21] Armin Biere. Kissat SAT Solver. https://github.com/arminbiere/kissat, 2021. Accessed: 2025-05-29.

[Bie24] Armin Biere. CaDiCaL SAT Solver (version 2.1.3). https://github.com/arminbiere/cadical/releases/tag/rel-2.1.3, 2024. Accessed: 2025-05-29.

[Hoo25] Hendrik Jan Hoogeboom. Sat encoding connectedness in a planar grid. Unpublished manuscript, February 2025.

[Rei98] Klaus Reinhardt. On some recognizable picture-languages. In Luboš Brim, Jozef Gruska, and Jiří Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, pages 760–770, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[Str25] Hanna Straathof. Solving and generating Kuroshuto puzzles. Bachelor thesis, LIACS, Leiden University, January 2025.

[Wyn18] Ed Wynn. A comparison of encodings for cardinality constraints in a SAT solver. *arXiv preprint arXiv:1810.12975*, 2018.