Solving Hitori puzzles using Satisfiablity Modulo Theories

Robin Rietdijk[1]

Supervisor: Dr. Anna L.D. Latour [1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 25, 2026

Name of the student: Robin Rietdijk
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L.D. Latour, Dr. Tim Coopmans

Abstract

Modelling-and-solving paradigms are widely used to solve combinatorial problems, but comparisons of these paradigms on identical problem instances remain scarce and are often incomplete or biased. As a results, finding the correct paradigm for a given logic problem can be difficult. In this study, we evaluate the performance of the *Satisfiability Modulo Theories* (SMT) paradigm for solving Hitori puzzles, as part of a larger effort to compare modelling-and-solving paradigms. During the evaluation, the running time of the solver will be used as the primary performance statistic. Across all evaluated encodings, a straightforward encoding using linear integer arithmetic shows the best and most stable performance. Alternative encodings and redundant constraints generally do not improve performance, often increasing encoding size or adding unnecessary complexity. Results show that solver runtime is dominated by puzzle size and encoding size, with puzzle structure having little influence. These findings show that SMT is a suitable paradigm for solving Hitori puzzles, as the rules naturally translate into a compact encoding without the need for added complexity or redundancy.

1 Introduction

Logic puzzles are more than just a relaxing pastime or a way to keep your mind occupied, they provide structured challenges highly suitable for studying modelling-and-solving paradigms. Many logic puzzles are at least NP-hard [14], which means that even small changes in how the problem is modelled can greatly affect the computational complexity of the problem. For this reason, logic puzzles serve as valuable benchmarks for analysing individual paradigms and how modelling choices can influence solver performance, as well as to provide a common basis for comparing paradigms against each other.

Hitori is a logic puzzle played on an $n \times n$ grid of numbers. The goal of the game is to mark tiles as black such that no number repeats in any row or column, no two black tiles are adjacent and all remaining white tiles remain connected. This puzzle is proven to be NP-complete [14], which makes it a suitable benchmark for evaluating modelling-and-solving paradigms and the impact of different modelling choices. Hitori is one of many puzzles that can be modelled as a constraint satisfaction problem.

The most well-known constraint satisfaction problem is the *Boolean satisfiability problem*, or SAT. A SAT-solver aims to decide whether a formula over Boolean variables can evaluate to True by choosing True or False for its variables [6]. However, some problems require determining the satisfiability of formulas in more expressive logic, with respect to some background theory [4]. The research field concerned with the satisfiability of such formulas is called *Satisfiability Modulo Theories* (SMT). Because of its support for richer constraints compared to SAT, SMT has become a powerful tool for modelling problems such as theorem proving, scheduling, planning and more [11; 10; 4].

Even though SMT's performance has been evaluated in previous studies, direct comparisons between different modelling-and-solving paradigms on the same problem are uncommon and can be influenced by the modeller's expertise in one specific paradigm. To address this issue, a broader effort is set up with the aim of creating proper comparisons between modelling-and-solving paradigms.

As part of this effort, this study evaluates the SMT paradigm by modelling and solving the Hitori puzzle. Our goal is to understand how modelling choices affect the performance of the solver and to characterize the strengths and weaknesses of the SMT paradigm when applied to this problem.

To reach this goal, we build an SMT solver to solve the Hitori puzzle and use it to evaluate the runtime performance, characteristics and usability of the paradigm. We aim to answer the following questions:

- How does the size of the SMT encoding and solving performance scale with the increasing Hitori puzzle size?
- Which redundant constraints, including known Hitori patterns and human-inspired strategies, improve solving performance?
- How does the differing structure of puzzle instances influence the solving performance of SMT?

Z3 is an SMT solver developed by Microsoft Research, targeted at solving problems arising from software verification and analysis [9]. Z3 offers support for all theories required for our solver, is well documented and supports many programming languages, which makes it a suitable choice for experimenting with different modelling approaches.

This report is structured as follows. We begin by reviewing background knowledge relevant to our work. We discuss the Hitori puzzle and its constraints and introduce the SMT paradigm with a high-level overview. We continue with presenting several SMT encodings, redundant constraints and method for filtering difficulty. Finally, we discuss our experimental results and give an answer to our research questions.

2 Related work

In *Solving, Generating and Classifying Hitori* [24], Wensveen takes a closer look at human solving strategies for the puzzle and introduces several solving techniques. These techniques were used to create a fully rule-based solver and a solver based on 2-SAT. Whilst the thesis does not go experiment much with solver performance and its scalability, it does offer useful insights into different kinds of structural constraints that show up in Hitori instances and provides patterns and strategies that can be implemented as redundant constraints.

A more implementation-oriented study is the thesis *Hitori Solver* by Gander and Hofer [12]. In this work, the authors combine a collection of patterns into a single multi-stage algorithm. In addition to this algorithm, the thesis also explores a logical approach by translating Hitori into a satisfiability problem and solving it using a SAT-solver. Their experiments show that the size of the SAT formula grows rapidly as puzzle size grows, making this approach unsuitable for larger puzzles. This shows the effect of the size of the encoding and the importance of carefully modelling expensive constraints like connectivity.

Solving and generating puzzles with a connectivity constraint [23] by Van der Knijff, describes a more general view on solving puzzles with connectivity constraints. The author shows several SMT encodings for various logic puzzles, including Hitori, and shows how connectivity can be solved using a ranking-based approach. Whilst this work confirms that Hitori can be modelled and solved using SMT, it does not present any experimental results or performance statistics and does not reflect on the practical use or modelling choices for the SMT encoding.

Zanteman and Joosten also investigate related modelling choices for graph properties in *Latin Squares with Graph Properties* [26]. Although this work does not describe Hitori directly, it explores SAT and SMT based encodings for Latin squares with graph properties such as connectivity, trees and Hamiltonicity. Latin squares share the row and column uniqueness constraint and the graph structure with Hitori, making several encodings transferable. The study compares several SAT and SMT based encodings for various graph properties and concludes that performance depends strongly on the modelling choices and problem size.

Even though these studies demonstrate useful SMT encodings, creating a full evaluation of SMT for a comparative study remains difficult. Existing literature is sparse, often too specific without exploring alternative encodings or lacks a detailed experimental evaluation. As a result, important aspects such as the impact of modelling approaches on solver performance remain unclear. This paper aims to bridge this gap by exploring multiple SMT encoding variants and evaluates the effect of modelling choices on solving performance.

3 Background

In this section, we give relevant background on important topics discussed during this study. We begin with a more detailed look at the Hitori puzzle and then move on to a high-level overview of SMT.

3.1 The Hitori puzzle

The Hitori puzzle is a Japanese logic puzzle played on an $n \times n$ grid in which each tile initially contains an integer between 1 and n. Each tile in the grid can either be *white* and show its corresponding number, or be marked *black* and eliminate its number from the grid. The task is to create a grid where each number is unique in its row and column by eliminating numbers and satisfying three rules [21; 24]. Figure 1 shows an example of a Hitori puzzle instance before and after solving.

Hitori rules

Hitori puzzles have three rules that enforce both local and global structure in the grid. The rules control rows, columns, neighbouring tiles and the overall connectivity of the solution, making Hitori a suitable puzzle for comparing different modelling approaches. The rules are as follows:

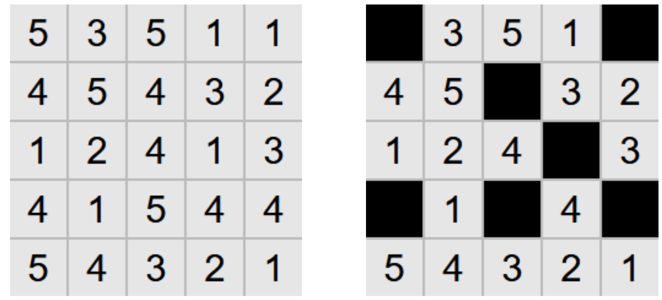


Figure 1: Example Hitori puzzle before (left) and after (right) solving, from Simon Tatham's Portable Puzzle Collection [22].

- **Uniqueness**: All numbers are unique in both their row and column.
- **Adjacency**: Black tiles are not allowed to be orthogonally adjacent.
- **Connectivity**: White tiles must form a single orthogonally connected component.

NP-completeness

Hearn and Demaine show that deciding whether a Hitori instance has a valid solution is NP-complete using a reduction from Constraint Graph Satisfiability based on planar AND/OR constraint graphs [14]. This result further motivates the use of Hitori as a benchmark for this study, since NP-complete problems are known to be heavily influenced by modelling and encoding choices, especially as the problem size increases.

3.2 Satisfiability Modulo Theories

Propositional satisfiability (SAT) is the decision problem of determining whether a formula has a satisfying assignment or not. However, many applications require determining satisfiability of formulas in more expressive logics [4]. *Satisfiability Modulo Theories* (SMT) addresses this by checking the satisfiability of first-order formulas with respect to some background theory [4; 11]. SMT extends SAT by allowing constraints to be expressed directly using theories such as arithmetic, arrays, bitvectors, and others, instead of encoding everything as propositional variables [3; 10]. Modern SMT solvers typically follow a structure in which a SAT solver handles the Boolean structure while a dedicated theory solver reasons about the theory-specific parts of the problem [4].

Theories in SMT

In SMT, a *theory* constrains the interpretation of the symbols in a given vocabulary by determining the domain and operations available [4; 5]. For example, the theory of linear integer arithmetic includes integer variables together with operations such as $+$, $-$, and relations such as $\leq$ or $\geq$ [10]. For an SMT formula to be consistent with a theory, the interpretation of the symbols must satisfy all constraints imposed by that theory [10]. SMT formulas are always interpreted with respect to a background theory and satisfiability depends on both the Boolean structure of the formula and the underlying theory [4].

SMT solving strategies

The two most widely used approaches to SMT solving are the *eager* and the *lazy* approaches. In the *eager* approach, an SMT formula is translated into an equivalent Boolean formula, encoding all theory specific constraints upfront [4]. This can be very efficient for theories that can be translated into a compact Boolean formula. However, for more expressive theories, this translation may grow exponentially, leading to formula *blowups* [18; 4]. Although fully eager solvers are less common, modern SMT solvers do apply eager techniques selectively. For example, bitvector constraints are often handled by a technique called *bit-blasting*, where they are reduced to Boolean formulas over individual bits.

Modern SMT solvers most often use the *lazy* approach, which avoids upfront translations by combining a SAT solver with theory-specific decision procedures [4]. In the simplest form, known as the *offline* approach, each atom in the input formula φ is simply considered as a propositional variable, forgetting about the theory and creating the propositional formula φ^P [18]. This formula is fed to a SAT solver, which either decides if φ^P is unsatisfiable, which means it is also theory-unsatisfiable, or it returns a satisfying assignment μ^P [4]. The corresponding set of literals μ is then checked by a theory solver. If μ is found to be theory-inconsistent, the clause $\neg\mu^P$ is added to φ^P and the SAT solver is restarted on the resulting formula. This approach avoids creating large SAT formulas and lets the solver check constraints only when they become relevant, allowing them to scale better for a wide range of theories [4; 18].

The DPLL(T) architecture

Modern SMT solvers extend the offline approach with several additional mechanisms, allowing further cooperation between the SAT solver and the theory solvers [10]. These mechanisms define the *online* variant of the lazy SMT approach and is often referred to as the *DPLL(T)* architecture, which includes:

Early pruning [4; 18]. The theory-consistency can be checked incrementally, while the assignment is still being built by the SAT solver through a sequence of Boolean decisions. If the assignment is theory-inconsistent, then all extensions are also inconsistent. This allows the solver to backtrack and avoid large amounts of unnecessary searches.

T-backjumping & T-learning [4]. If a theory conflict is detected, the solver can backtrack directly to a point where the assignment was still theory-consistent, rather then completely restarting the search [18]. The conflicting clause is then added either temporarily or permanently to the SAT solver.

T-propagation [4; 18]. Theory solvers can introduce new literals that must hold based on the current assignment. These literals are then propagated by the SAT solver in the same way as Boolean implications.

4 Approach

This section provides an overview of the encodings and strategies evaluated during this study. We describe the individual encodings, the redundant constraints and the strategy of finding difficult puzzles. We also give a brief overview of the generator used to generate the evaluated puzzles.

4.1 Puzzle generator

All puzzles used during our evaluation have been generated using a manually developed puzzle generator. To generate new puzzles of any size n, we start by constructing a solution grid that satisfies all Hitori rules. Once a valid solution is generated, we fill up the white tiles with numbers using recursive backtracking, making sure that all numbers are unique in their row or column. Numbers for black tiles are generated afterwards, making sure to always copy a number already used in its row or column. The completed puzzle is handed over to a solver which checks for a unique solution. If the puzzle is not solvable or the solution is not unique, the entire process starts again. A proof of completeness for this generator can be found in Appendix C.

4.2 SMT modelling Strategy

This section describes seven encodings that have been developed during this study. We start by giving an overview of the initial encoding that was developed and describe how each of the Hitori rules was modelled for this encoding. We then introduce four alternative encodings derived from the initial encoding, each developed by reformulating a single Hitori constraint. Finally, we describe two encodings that make use of alternative theories from the initial encoding.

In these encodings, we use the symbols $\text{black}_{i,j}$ and $\text{white}_{i,j}$ to denote whether the tile (i, j) is marked black or white respectively. Consequently:

$$\neg\text{black}_{i,j} = \text{white}_{i,j} \qquad and \qquad \neg\text{white}_{i,j} = \text{black}_{i,j}$$

We further use the symbol $\text{neighbours}_{i,j}$ to represent the set of orthogonally adjacent neighbours for the tile (i, j) and $\text{num}_{i,j}$ to represent the number of that tile.

Initial encoding

The initial encoding involved encoding the Hitori rules using linear integer arithmetic. This version takes inspiration from the approach used by van der Knijff [23] and the Sudoku solver implementation as described in *Z3Py Guide* [15]. *Latin Squares with Graph Properties* [26] also provided valuable insights on a ranking-based approach to graph connectivity.

Uniqueness rule: A constraint for the uniqueness rule is easily implemented as it is similar to that of Sudoku or Latin square solvers. However, since we are not placing numbers on the grid, simply checking for distinct numbers as is done in the *Z3Py Guide* [15] or *SAT/SMT by Example* [25] is not appropriate. Instead, we create a pairwise encoding by imposing the following constraints on each row i:

$$\begin{aligned} \forall\, i \in \{0, \ldots, n-1\}, \\ \forall\, j \in \{0, \ldots, n-2\}, \\ \forall\, k \in \{1, \ldots, n-1\},\ j < k : \\ \text{num}_{i,j} = \text{num}_{i,k} \;\Rightarrow\; \text{black}_{i,j} \vee \text{black}_{i,k}. \end{aligned}$$

The same constraints are added to each column by swapping the indices accordingly.

Neighbour rule: To ensure that no orthogonally adjacent tiles are both marked black, we add a constraint for each pair of neighbouring tiles. To avoid adding redundant constraints, we only consider each pair once for the forward tiles to the right and below each tile:

$$\forall\, i, j \in \{0, \dots, n-1\} :$$
$$i + 1 < n \;\Rightarrow\; \neg(\text{black}_{i,j} \wedge \text{black}_{i+1,j}),$$
$$j + 1 < n \;\Rightarrow\; \neg(\text{black}_{i,j} \wedge \text{black}_{i,j+1})$$

Connectivity rule: The connectivity rule is the most difficult rule to model. We follow the ranking-based connectivity encoding as described by Van der Knijff and by Zantema and Joosten [23; 26]. In this approach, each tile (i, j) is assigned an integer rank $\text{rk}_{i,j}$ according to the following rules:

- Exactly one white tile is marked as the root and is assigned rank 0.
- All tiles that are marked black have rank -1.
- All non-root white tiles must have a rank > 0.

Then, for all $i, j \in \{0, \dots, n-1\}$:

$$\text{white}_{i,j} \wedge \neg\text{root}_{i,j} \;\Rightarrow\; \exists\,(i', j') \in \text{neighbours}_{i,j} \mid \text{white}_{i',j'} \;\wedge\; \text{rk}_{i',j'} < \text{rk}_{i,j}$$

Where $\text{root}_{i,j}$ denotes whether tile (i, j) is the chosen root or not.

This implementation creates a path such that for any white tile, we will always be able to move to a neighbouring tile with a lower rank until we end up at the root. A path like this exists for all white tiles if and only if all white tiles are connected in a single connected component, thus enforcing the connectivity rule.

Alternative encodings

As an alternative to the pairwise encoding for the equality constraint, we implemented a counting-based variant using the $AtMost$ assertion, named `qf_ia_alt_u`. For each row i we add the following constraint:

$$\forall\, i \in \{0, \dots, n-1\},\ \forall\, v \in \{1, \dots, n\} :$$
$$\sum_{\{j \,\mid\, \text{num}_{i,j} \,=\, v\}} \text{white}_{i,j} \;\leq\; 1$$

The same constraint is added to each column by swapping the indices accordingly.

We further experimented with a small rewrite of how we keep track of the root tile in the ranking connectivity approach. In this encoding, named `qf_ia_alt_c`, we introduce a grid of Boolean variables with $\text{root}_{i,j}$ indicating whether a tile (i, j) is the root. This allows us to refer to the root directly in the constraints, rather than comparing indices.

An alternative to the connectivity approach was implemented using a tree-based method inspired by Zantema and Joosten [26], named `qf_ia_tree_c`. By creating a spanning tree, we create a single connected component that satisfies the connectivity constraint. We assign a depth $\text{dep}_{i,j}$ and orthogonally adjacent parents $\text{prnt}_{i,j}$ to each tile (i, j) and enforce the following rules:

- Exactly one tile is marked as root.
- All tiles that are marked black have depth -1, the root has depth 0 and no parent.
- Every non-root white tile (i, j) has exactly one orthogonally adjacent white parent with smaller depth:

$$\forall\,(i, j)\ \exists_{=1}\,(i', j') \in \text{prnt}_{i,j} : \text{dep}_{i',j'} < \text{dep}_{i,j}$$

Lastly we implemented an encoding named `qf_ia_external`, in which we only use the uniqueness and adjacency constraints to find a candidate solution. This candidate is checked for connectivity outside of the solver using Breadth-First Search (BFS) and if connectivity is not satisfied, a new constraint is added to block the current candidate. The solver retries and repeats this process until a valid solution is found.

Alternate theories[1]

To compare the effect of the underlaying theory on solving performance, we implemented a bitvector-based variant called `qf_bv`. The neighbour and uniqueness constraints are purely Boolean and therefore translate directly without modification. However, The connectivity constraint makes use of inequality operations such as $<$ and must be rewritten in terms of bitvector comparisons. Replacing the integer ranks for bitvector ranks removes the use of the linear integer arithmetic theory in our solver and enables the use of bit-blasting, potentially improving performance due to the eager SAT-based solving properties of the bitvector theory [10; 4].

We also implemented a variant that uses only Boolean logic, without any additional theories, called `qf_bool`. A Boolean encoding is already been given in *Hitori Solver* [12], where the authors show that this approach quickly becomes inefficient for large puzzle sizes. For this reason, we implemented an alternative Boolean encoding, only rewriting our connectivity constraint.

With the `qf_bool` encoding, we aim to visit every white tile within n^2+1 steps using BFS, starting from a chosen root. We mark a tile (i, j) as visited in step k by setting $\text{visited}_{k,i,j}$ to True. The encoding is formulated as follows:

$$\forall\, k \leq n^2 + 1,\ \forall\, i, j\ \in \{0, \dots, n-1\} :$$
$$\text{visited}_{k,i,j} \;\Leftrightarrow\; \neg\text{black}_{i,j} \;\wedge$$
$$(\text{visited}_{k-1,i,j} \;\vee\; \exists\,(p, q) \in \text{neighbours}_{i,j} : \text{visited}_{k-1,p,q})$$

The connectivity constraint is satisfied if all white tiles are visited at $k = n^2 + 1$.

4.3 Redundant constraints

In an attempt to improve previous encodings, we implemented a number of redundant constraints. These constraints do not restrict the solver in any way, but aim to reduce the

[1] ChatGPT was used as a supportive tool for reflecting on alternate theories and developing the bitvector encoding. More information on the use of ChatGPT can be found in Section 6.2

Table 1: Overview of all redundant constraints implemented and evaluated in the SMT encoding.

Short	Name	Description
WN	*White Neighbours*	Every white tile must have at least one orthogonally adjacent white neighbour.
CC	*Corner Close* [13]	A corner white tile may not be isolated by surrounding black tiles.
ST	*Sandwich Triple* [13; 12]	Three equal orthogonally adjacent tiles in a row or column force the middle tile to be white and the outer two tiles to be marked black.
SP	*Sandwich Pair* [13]	Two equal tiles with exactly one tile between them force the middle tile to be white.
TC	*Triple Corner* [13; 12]	Corner pattern with three equal tiles that forces a specific white/black assignment to prevent isolation.
QC	*Quad Corner* [13; 12]	A 2×2 block of equal tiles in a corner forces a specific white/black assignment to prevent isolation.
TEP	*Triple Edge Pair* [13; 12]	Border pattern on three stacked equal pairs that determine a number of forced white tiles.
DEP	*Double Edge Pair* [13; 12]	Border pattern on two stacked equal pairs that determine a number of forced white tiles.
CE	*Close Edge* [13]	Prevents a white tile at the border from being isolated by surrounding black tiles.
FDE	*Force Double Edge* [13]	Border pattern that forces a tile to be white to avoid isolation.
BC	*Border Close* [12]	Border Pattern that forces a black tile in order to avoid isolation of a white tile.
LW	*Least Whites*	Every row and every column must have at least $\lfloor \frac{n}{2} \rfloor$ white tiles.
MB	*Most Blacks*	Every row and every column must have at most $\lceil \frac{n}{2} \rceil$ black tiles.
PI	*Pair Isolation* [13; 12]	An orthogonally adjacent equal pair forces all other occurrences of that value in the same row or column to be black.
CI	*Close Isolation* [12]	An equal pair threatening to isolate a tile forces all other occurrences of that value in the same row or column to be black.
WB	*White Bridges*	Each pair of adjacent rows has at least one pair of vertically adjacent white tiles (and similarly for columns).

size of the search space explored by the solver. Most constraints are based on common Hitori patterns and human solving strategies as described in *Hitori solver* [12] and on *Menneske.no*, whilst some constraints aim to strengthen the connectivity constraint.

Table 1 shows a summary of the names, abbreviations and descriptions of all redundant constraints that were evaluated for this study. All constraints were encoded directly in Z3 as additional constraints using only Boolean assertions, meaning that all constraints can be added to any encoding without changing the underlying theory. A full overview of the redundant constraints is found in Appendix D.

4.4 Puzzle difficulty

Instead of generating a large number of puzzles with different structures, we measure the influence of puzzle structure by filtering puzzles that are difficult for our solver to handle. However, determining puzzle difficulty is a complicated task. We used two methods to classify difficult puzzles. The classifications resulting from these two methods are then used to analyse whether and how puzzle structure differs between difficult and non-difficult puzzle instances.

Our first filtering method relies on solver running time. Runtime can be a unreliable statistic due to environmental noise, but we reduce this by solving each puzzle several times and combining the runtimes into a single value using the median runtime. We filter puzzles based on a threshold relative to the median runtime and the interquartile range (IQR), such that only puzzles with substantially longer runtimes are classified as difficult.

Our second approach relies on Z3's internal statistics to calculate an *effort-score*[2] and filter using an IQR-based threshold. These statistics are not specifically documented in the Z3 documentation and thus must be handled with care. However, prior community discussions[3] in combination with the source code can be used to create a better understanding of several statistics.

For each puzzle instance we compute a effort-score based on a weighted sum of statistics. The statistics used are branching *decisions* with weight 1.0, the number of *conflicts* found in branches with a weight of 0.7 and the number of *propagations* with weight 0.5. These three statistics all relate to the DPLL(T) algorithm used by Z3 and show an indication of the solving effort required to explore the search space.

The statistics and weights used in the effort-score have been selected through analysing and exploring various options. We evaluated several statistics and analysed how these differed in puzzles labelled as hard versus non-hard. In addition, we analysed several ratios between statistics, such as the ratio of propagations per conflict. These ratios provided additional insights on the differences between hard and non-hard puzzles and were used as a reference to adjust weights.

[2] ChatGPT was used as a supportive tool for suggesting candidate statistics, weights and analysis approaches. All evaluations were performed manually and the final weight distribution was decided by us. More information on the use of ChatGPT can be found in Section 6.2

[3] Community discussions on interpreting Z3 statistics:
https://stackoverflow.com/questions/18491922/,
https://stackoverflow.com/questions/45457131/,
https://stackoverflow.com/questions/17856574/

5 Experimental Setup and Results

All encodings described in section 4.2 have been implemented and evaluated using Python `3.13.2` and Z3's Python implementation using the `z3-solver` (`4.15.4.0`) package. Test sets were generated using the collaborative generator and consisted of 150 puzzles for each $n \in \{5, \ldots, 25\}$ for Sections 5.1 and 5.2, 5000 puzzles of $n = 8$ for Section 5.3 and 50 puzzles for each $n \in \{5, 10, 15, \ldots, 50\}$ for Section 5.4.

Several encoding variants as described in Section 4.2 have been implemented and experimented with. We will be referencing the redundant constraints by their abbreviations shown in table 1. We further evaluate encodings implemented in several SMT theories, which are in the theory of linear integer arithmetic (`qf_ia`), the theory of bitvectors (`qf_bv`) and using only Boolean constraints (`qf_bool`). The evaluated alternative encodings are referenced by their name as described in Section 4.2, which are `qf_ia_alt_u`, `qf_ia_alt_c`, `qf_ia_tree_c` and `qf_ia_external`. The `qf_ia` encoding is used as the baseline encoding for the experiments.

Runtime is the primary performance statistic used in the experiments. Runtime is measured from solver initialization up until a result is returned. To reduce the effect of environmental noise, each encoding is run three times and the resulting median is used as its runtime value. To prevent extreme runtime spikes, a timeout of 10 seconds was enforced on all solvers. If a solver instance times out, we follow the *PAR2* method, where a penalized average runtime of twice the timeout value is used for the runtime of this instance.

5.1 Encoding size and performance

RQ1: How does the size of the SMT encoding and solving performance scale with the increasing Hitori puzzle size?

For this question we experiment with all encodings and the evaluated runtime and encoding size. To compare all encodings equally, regardless of theory used, we create a consistent theory-independent indicator for the encoding size by counting the number of SMT variables and assertions created during the initialization of the constraints. These statistics are useable across theories and define the formula that Z3 must parse, thus forming a useful representation of the encoding size. Figure 2 shows the runtime and encoding size by puzzle size for all encodings evaluated for this experiment.

For all encodings, runtime shows strong correlation with encoding size, with *Spearman's correlation coefficients* of $\rho \geq 0.98$ for all solvers except `qf_ia_external`, which shows a slightly lower ρ of 0.906. Across theories, `qf_ia` shows the best runtime performance. At $n = 25$, `qf_bv` shows a slow down of $8.72\times$ compared to `qf_ia`, whilst `qf_bool` is roughly $38.20\times$ slower at $n = 13$ before it starts timing out.

The `qf_ia`-based variants compare similarly or worse compared to the `qf_ia` encoding. `qf_ia_external` shows a large increase in runtime at lower puzzle sizes of up to $5.12\times$, but rapidly slows down before consistently timing out at $n = 21$.

Variability is calculated using a relative IQR $((q3 - q1)/median)$ per puzzle size and averaged over all available sizes. For encodings that are not able to solve all sizes within the timeout limit, the variability is calculated only over sizes that it is able to solve. This means that variability for `qf_bool` is calculated up to $n = 13$ and for `qf_ia_external` up to $n = 21$.

Even though `qf_bool` times out early, it shows the lowest relative IQR of 0.0383 within its viable range. This is followed by `qf_ia` and its variants, ranging from 0.0667 to 0.131. `qf_bv` shows a much larger variability of 0.304, while `qf_ia_external` is the least stable with a relative IQR of 1.79.

To estimate growth by puzzle size, we fit a linear model to the logarithm of the median runtimes as a function of puzzle size: $\log(median\ runtime(n)) \approx a \cdot n + b$. All log median runtimes show a strong linear fit with $R^2 \geq 0.96$.

For LIA-based encodings, the baseline encoding shows the slowest growth with a slope of $a = 0.1469$, closely followed by `qf_ia_alt_u` ($a = 0.1496$). The variants `qf_ia_alt_c` (0.1771) and `qf_ia_tree_c` (0.1736) show slightly steeper growths with `qf_ia_external` showing the largest growth with a slope of $a = 0.4177$. The encodings based on alternative theories show the worst scaling overall, with slopes of $a = 0.2294$ for `qf_bv` and $a = 0.4595$ for `qf_bool`.

Discussion

The results indicate that runtime strongly correlates with encoding size, with only `qf_bv` and `qf_ia_external` showing notable additional influences as well. The `qf_bv` encoding likely suffers from additional overhead from the bitvector theory, while `qf_ia_external`'s guessing-based approach scales badly as search space grows. The `qf_ia` encoding shows the best and most stable performance across all evaluated puzzle sizes, with the alternative variants showing slightly worse scaling.

Overall, these results suggest that creating a compact encoding and aligning the underlying theory with the Hitori rule structure provides the best solver performance.

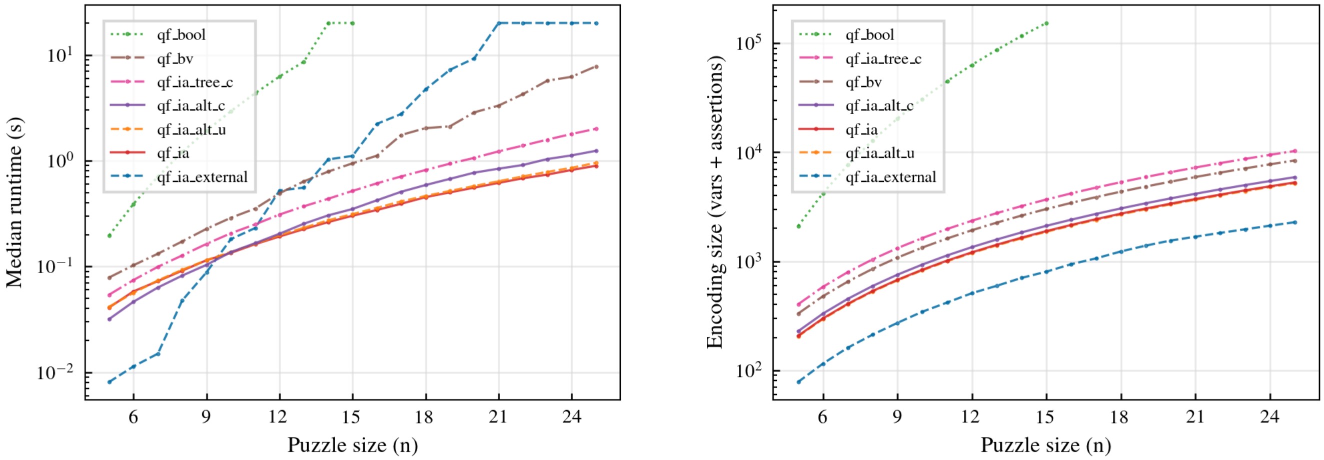
5.2 Redundant constraints

RQ2: Which redundant constraints, including known Hitori patterns and human-inspired strategies, improve solving performance?

For this experiment we use 16 redundant constraints added individually on top of the `qf_ia` encoding. We use the *Wilcoxon signed-rank test*[4] to calculate p-values between each constraint and the baseline for each tested puzzle size. We then apply the *Holm-Bonferroni*[4] method with $\alpha = 0.05$ for each constraint to identify which puzzle sizes show statistically significant changes.

Using these methods, `CE`, `WB`, `WN` and `LW` are revealed to significantly slow down the encoding for all evaluated puzzle sizes. `PI` and `SP` are the only constraints showing significant

[4]ChatGPT was used as a supportive tool for evaluating statistical methods to use in the experiments. All methods are manually verified to ensure suitability for the data and goals of this study. More information on the use of ChatGPT can be found in Section 6.2



(a) Median runtimes by puzzle size, shown on a logarithmic y-axis

(b) Encoding size by puzzle size, shown on a logarithmic y-axis

Figure 2: Median runtimes and encoding sizes by puzzle size for all evaluated encodings

speed-ups compared to the baseline for sizes $n \geq 18$. All other constraints show a mixture of either no significance or a significant slowdown, but never a significant speed-up.

`CE`, `WB`, `WN` and `LW` also show the largest average increase in encoding size by up to 12%. Most other constraint encodings only show a slight increase in encoding size.

Even though statistically significant, the actual speed-ups for `PI` and `SP` are minimal, with a median increase of $1.03\times$ and $1.02\times$ at $n = 25$.

When applied to the `qf_ia_external` encoding, `MB`, `LW`, `WN`, `CE` and `CC` show significant speed-ups for almost all evaluated puzzle sizes. For `LW`, `MB`, `WN`, the median speed-up at $n = 25$ are $15.2\times$, $14.7\times$ and $34.7\times$ respectively. The `CE` and `CC` constraints show smaller, but still significant improvements, with maximum speed-ups of $2.53\times$ at $n = 20$ and $1.78\times$ at $n = 12$. Other redundant constraints show either no statistically significant effects or small but significant slowdowns.

Discussion

Redundant constraints did not add much in terms of performance for a complete encoding of the Hitori rules. While these constraints aim to strengthen the existing encoding, such constraints are already implied by the full rule set. As a result, redundant constraints mainly increase encoding size and generally decrease performance. However, when a specific Hitori rule is omitted, constraints targeting this missing rule become highly effective. This indicates that truly redundant constraints offer little benefit for strengthening encodings, but such constraints can be used to compensate for missing structural information, in which case they are no longer redundant.

Overall, these results show that SMT can effectively handle the full Hitori rule set without requiring additional modelling to increase performance.

5.3 Puzzle structure

RQ3: How does the differing structure of puzzle instances influence the solving performance of SMT?

We classify difficult puzzles using two criteria according to the approach described in section 4.4. Puzzles are identified as outliers using an IQR-based threshold of $Q3 + 1.5 \cdot IQR$.

Runtime-based filtering classifies 79 puzzles as hard outliers, whereas the effort-based method identified 140 hard puzzles. Only 8 puzzles are classified as hard by both runtime- and effort-based criteria.

The structural properties of puzzles in the runtime-hard and effort-hard sets are compared against non-hard puzzles using the *Mann-Whitney U test*[4] and resulting p-values are corrected using the *Benjamini-Hochberg*[4] method. Puzzle properties that are analysed are the number of black tiles in the solution (BT), adjacent duplicates (AD), triple adjacent duplicates (TAD), non-adjacent duplicates (NAD) and duplicates in both row and column (RCD).

The runtime-hard set shows no significant differences against the baseline set. The effort-hard set however, shows significant differences for the NAD, RCD and BT properties. The direction of these differences were all negative, indicating lower values for each of these properties. Table 2 shows a summary of corresponding mean values and the correlation between solver effort and runtime.

For all puzzles, Spearman's correlation coefficient shows weak to moderate correlation between solver effort and the properties that differ significantly in the effort-based analysis. Solver runtime shows no meaningful correlation with any of the evaluated structural properties.

Table 2: Structural properties for baseline, effort-hard and runtime-hard 8×8 puzzles, displaying the average occurrences of structural properties and their correlation coefficients with solver effort and runtime. Significant differences are highlighted in bold

Feature	Mean occurrences			Spearman's ρ	
	baseline	effort	runtime	effort	runtime
BT	18.2	**17.7**	18.3	-0.328	0.001
AD	8.3	8.1	8.4	-0.007	0.018
TAD	0.2	0.3	0.4	-0.003	0.016
NAD	19.2	**18.2**	19.1	-0.126	0.005
RCD	15.0	**14.2**	14.8	-0.093	0.027

Discussion

Puzzle structure shows little correlation with solver runtime. Although effort-based filtering is able to identify puzzles that differ significantly in certain puzzle properties, the correlation between these properties and the effort-score is weak. Furthermore, puzzles identified using runtime-based filtering show no meaningful correlation with any structural properties. This suggests that structural puzzle features are insufficient indicators of solving difficulty and have little impact on solver performance compared to puzzle size and encoding size.

5.4 Paradigm comparison

A benchmark was run to compare the runtime performance of solvers based on several modelling-and-solving paradigms: Logic Programming (Prolog), Integer Linear Programming (Gurobi), Satisfiability Modulo Theories (Z3), Constraint Satisfaction Problem (Pumpkin) and Answer Set Programming (ASP). Figure 3 shows the median runtimes per puzzle size for each paradigm. Pumpkin and ASP achieve the lowest runtime and scale well with large puzzle sizes. Z3 displays moderate runtime scaling, with runtime increasing more rapidly with larger puzzle sizes. Prolog and Gurobi show poor scalability, reaching the timeout limit for relatively small puzzle sizes.

Discussion

While this benchmark provides a useful comparison, it should be handled with care. A fully fair comparison between paradigms is difficult, as the solvers not only differ in modelling strategies but also in implementation languages and specific functionalities. As a result, this benchmark should be seen as an indication rather than a definitive ranking of paradigms.

6 Responsible Research

This study does not involve human participants, personal data or data collected from external sources as all puzzles and data used in this study is generated by us. Because of this, ethical questions related to privacy, consent or data collection are not applicable. Instead, responsible research relevant to this study are the reproducibility and the use of Large Language Models (LLMs).

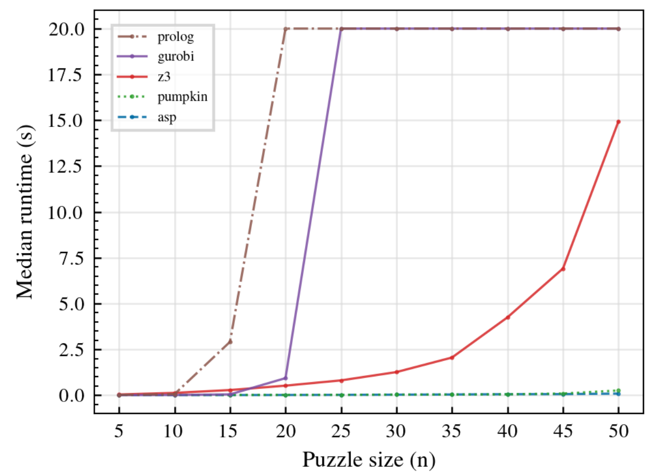


Figure 3: Median runtimes by puzzle size for all solvers

6.1 Reproducibility

To ensure full reproducibility of our work, all source code is publicly available in a GitHub repository[5]. All group-related work, such as the puzzle generator, can be found in separate repository[6]. The main repository contains a `requirements.txt` file specifying all packages and version used during experimentation, allowing the creation of a precise copy of the experimental environment.

All scripts used in the experiments are included in the source code. Solvers are initialised using a seed based on the input command. This allows exact replicas of experiments to be made by using the same command to run it.

6.2 LLM usage

LLMs were used during certain parts of this project as a supporting tool. We used a paid version of ChatGPT, which allowed access of GPT-5.1/5.2 Thinking.

ChatGPT was used as a sparring partner to obtain feedback and reflect on ideas, to clarify concepts and suggest potential alternative directions. All responses generated by ChatGPT were carefully fact-checked using existing literature and external sources. Suggestions made by ChatGPT were not adopted without careful consideration and were evaluated using our own understanding before being explored further or rejected.

ChatGPT was used to assist in writing parts of the text. A draft text was used in the prompt to request feedback and suggestions. No responses generated by ChatGPT were copied into the main text. Instead, responses were carefully examined and used to iteratively improve the draft.

At no point was ChatGPT used to replace our own critical thinking or decision-making. To make sure of this, we made sure to formulate and experiment with our own ideas first, before consulting ChatGPT. Appendix E describes the LLM usage and prompts used during the study in more detail.

[5]https://github.com/RobinRietdijk/CSE3000Q2_SMT

[6]https://github.com/sappho3/Thesis-Hitori-shared/tree/a2eb9006ef758783665ef2287a0dbf1f0d555641

7 Conclusion and Future Work

This study investigated the suitability of SMT for solving Hitori puzzles by evaluating several encodings using Z3. The goal was to evaluate how modelling choices, redundant constraints and puzzle structure impacted solver performance as well as to analyse the strengths and weaknesses of SMT for solving Hitori.

Overall, our findings show that SMT is a viable method for solving Hitori puzzles of all evaluated sizes (up to $n = 25$). The Hitori rules can be naturally expressed using their underlying theories in SMT and a complete translation of these rules requires no additional modelling to achieve stable performance. Across all evaluated encodings, solving performance was primarily influenced by encoding size and puzzle size, with puzzle structure having little impact. The results highlight the effectiveness of SMT in handling the base set of Hitori rules and suggest that compact encodings implementing these rules are most effective, without needing additional modelling complexity or specific solving strategies. While other solvers may achieve better performance, SMT offers a strong balance between expressiveness, simplicity and solver stability.

During this study, Z3 and its theory-specific configuration options have not been explored due to scope limitations. Future work could investigate the influence of these options on the performance of the encodings used in this study. Additionally, evaluating our encodings using other SMT solvers could provide further insight into the general SMT paradigm and its behaviour.

Finally, since this work is part of a larger effort towards making proper comparisons between constraint-solving paradigms, future work includes studying and evaluating other paradigms on the same problem. These studies can then be combined to make a systematic comparison between various modelling-and-solving paradigms.

Acknowledgements

I would like to thank Dr. Anna L.D. Latour for their supervision throughout this study. I also thank the research group consisting of Sophieke van Luenen, Sappho de Nooij, Tom Friederich and Lesley Smits, for their contributions to the collaborative parts of this study, as well as for their general support.

References

[1] Krzysztof R. Apt. The logic programming paradigm and prolog. *CoRR*, cs.PL/0107013, 2001.

[2] Chitta Baral and Michael Gelfond. Logic programming and knowledge representation. *J. Log. Program.*, 19/20:73–148, 1994.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). www.SMT-LIB.org, 2016.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.

[5] Clark W. Barrett, Cesare Tinelli, Haniel Barbosa, Aina Niemetz, Mathias Preiner, Andrew Reynolds, and Yoni Zohar. Satisfiability modulo theories: A beginner's tutorial. In André Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi, editors, *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part II*, volume 14934 of *Lecture Notes in Computer Science*, pages 571–596. Springer, 2024.

[6] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

[7] Nikolaj S. Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. Programming Z3. In Jonathan P. Bowen, Zhiming Liu, and Zili Zhang, editors, *Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018, Tutorial Lectures*, volume 11430 of *Lecture Notes in Computer Science*, pages 148–201. Springer, 2018.

[8] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006.

[9] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[10] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: An appetizer. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2009.

[11] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[12] Matthias Gander and Christian Hofer. *Hitori Solver*. Bachelor, Universität Innsbruck, April 2006.

[13] Vegard Hanssen. Hitori puzzles. https://menneske.no/hitori/eng/. Accessed: 2025-12-22.

[14] Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation*. A K Peters, 2009.

[15] Eric Hong, Rob Gardner, and Elias Bachaalany. Z3 api in python. https://ericpony.github.io/z3py-tutorial/guide-examples.htm. Accessed: 2025-11-16.

[16] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, José F. Morales, Jan Wielemaker, Daniel Diaz, and Salvador Abreu. Fifty years of prolog and beyond. *Theory Pract. Log. Program.*, 22(6):776–858, 2022.

[17] Robert A. Kowalski. The early years of logic programming. *Commun. ACM*, 31(1):38–43, 1988.

[18] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[19] Ulf Nilsson and Jan Maluszynski. *Logic, programming and Prolog*. Wiley, 1990.

[20] Leon Sterling and Ehud Shapiro. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[21] Akira Suzuki, Masashi Kiyomi, Yota Otachi, Kei Uchizawa, and Takeaki Uno. Hitori numbers. *J. Inf. Process.*, 25:695–707, 2017.

[22] Simon Tatham. Simon tatham's portable puzzle collection. https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/singles.html, October 2025. Accessed: 2025-11-21.

[23] Gerhard van der Knijff. *Solving and Generating Puzzles with a Connectivity Constraint*. PhD thesis, Radboud University, 2021.

[24] Roos Wensveen. *Solving, Generating and Classifying Hitori*. PhD thesis, Leiden University, 2024.

[25] Dennis Yurichev. SAT/SMT by Example. https://smt.st/, January 2021.

[26] H. Zantema and S.J.C. Joosten. Latin Squares with Graph Properties. *Technical Report*, October 2015.

A Collaboration

Several parts of this study have been done in collaboration with our research group. This collaboration included the design of a shared file format as well as the development of a solution checker and puzzle generator for Hitori puzzles. In addition, we constructed a proof of completeness for the generator and created a benchmark script to evaluate and compare the performance of our respective solvers. During the literature study, we also made use of a Zotero group to share and manage literature.

Following the CredIT contribution roles taxonomy, the author's contribution to the collaborative parts of the study are as follows:

Lesley Smits: software (equal); formal analysis (equal). **Robin Rietdijk:** software (equal). **Sappho de Nooij:** software (equal); formal analysis (equal). **Sophieke van Luenen:** software (equal); formal analysis (equal); project administration; and visualisation. **Tom Friederich:** software (equal); formal analysis (equal)

B Literature review

These are the unedited notes taken during the literature review done at the start of the project. Two paradigms were chosen to review and the aim was to find a number of useful sources, create a high-level understanding of the paradigm, find related research, compare the paradigms and think of problems to explore for both paradigms.

B.1 Logic Programming with Prolog

Logic programming works on the principle of using logic to represent knowledge and using deduction to solve problems by deriving logical consequences. Instead of writing explicit algorithms, we create a database filled with facts and rules about the problem, the knowledge base [17; 2; 20].

Relational Programming

Logic programs define relations, not functions like typical programming. This means that we can query a statement in any direction. Let's say we have the relation Siblings(X, Y) :- Mother(X, Z), Mother(Y, Z). We can query this relation to solve for either X, Y or Z based on the given facts. This principle is one of the core things that separates logical programming from other paradigms. Because predicates define relations instead of functions, the same Prolog rule can be used to generate, search or validate a query based on the way the variables are instantiated [1; 8; 20].

To derive an answer from a given query, logic programming uses resolution through unification, a core concept in LP [1; 19].

Unification

The problem of deciding whether an equation between terms has a solution is called the unification problem. Unification is used on resolution to match the selected goal with the head of a clause and to determine the variable bindings needed to further the derivation. In typical implementations, unification runtime is near linear in terms of input size [1; 19].

Resolution

In order to infer an answer from a query, resolution is used. There are different variations of resolution, but for Prolog, the SLD-resolution principle is used. Resolution works by negating the query and making it into a goal. During each step of resolution, the program will try to unify the current goal to either a fact, or the head of a rule in our knowledgebase. If the goal is matched to a rule, a new (set of) subgoal(s) is created and resolution will continue. Resolution finished if either the set of goals is empty, proving the query, or if no rules apply to the current subgoal, thus failing the query [19].

Resolution-based proof procedures can be implemented in different ways: backwards reasoning approaches such as SLD-resolution, used in Prolog, or forward reasoning approaches such as hyper-resolution.

SLD-Resolution is a complete proof procedure in theory, but Prolog's implementation has some major downsides. Prolog implements SLD-resolution using a leftmost selection rule and depth-first search with backtracking. This choice of implementation means that the solver can slow down runtime by proceeding into a long path or even completely loop in an infinite path.

Ordering of rules matters, making optimizing a solver a tedious job as the knowledge base grows in size.

Closed World Assumption & Negation as failure

The Closed World Assumption is the principle that assumes that every known possibility is listed and anything not probable is false. This allows us to draw negative conclusions based on the lack of evidence. However, this does not always work. We cannot always decide if we can use this rule, since a ruleset might loop. This is why a somewhat weaker version of CWA is often implemented. This weaker version only assumes the opposite is derivable from a query iff there exists a finite failed resolution tree. This is called negation as failure and is one of the most important features of modern day Prolog, as it makes it nonmonotonic. Negation as failure extends the expressive power of logic programming beyond Horn-clause monotonic reasoning [2; 19].

Prolog

Prolog is the first logic programming language, made by Kowalski and Colmerauer. Modern day prolog uses a straightforward case of logic programming where information is expressed by means of Hornclauses and deduction is performed by backwards reasoning embedded in resolution. Prolog programs are simple just knowledge bases, collections of rules and facts which describe some collection of relations that we find interesting [17; 2; 8].

Prolog operates with the following execution strategy:

- Goal: The program evaluates the leftmost goal first using SLD-resolution.
- Rule selection: Top to bottom
- Search Strategy: Depth-first search, backtracking

This means that Prolog is considered to be operationally incomplete

The execution strategy means that the order of your facts and rules matters. Using a different order can speed up or slow down your runtime on specific problems, or make it loop altogether. It is a good idea to experiment with this and to experiment with different types of problem instances to check the performance.

SWI-Prolog

SWI-Prolog is a well maintained open source implementation of Prolog. SWI-Prolog has a decent documentation available online, as well as a book to get you started on learning SWI-Prolog. Because of its niche use and the unusual programming style, the community is not as big as other popular programming languages on forums like StackOverflow. This could create some problems when trying to program in specific scenarios. However, due to its limited use space, you are still able to find a decent amount of helpful sources for your problem. unique features of other Prolog implementations can be found in 4.

Table 1. Unique Features and Foci of Prolog Systems.

System	Uniqueness
B-Prolog	action rules, efficient CLP supporting many data structures
Ciao	multi-paradigm, module-level feature toggle, extensible language, static+dynamic verification of assertions (types, modes), performance/scalabilty, language interfaces, parallelism
ECL^{i}PS^e	focus on CLP, integration of MiniZinc and solvers, backward-compatible language evolution of Prolog
GNU Prolog	extensible CLP($\mathcal{FD}$) solver, lightweight compiled programs
JIProlog	semantic intelligence / NLP applications
Scryer	new Prolog in development, aims at full ISO conformance
SICStus	commercial Prolog, focus on performance and stability, sophisticated constraint system, advanced libraries, JIT compilation
SWI-Prolog	general-purpose, focus on multi-threaded programming and support of protocols (e.g., HTTP) and data formats (e.g., RDF, XML, JSON, etc.), slight divergence from ISO, compatibility with YAP, ECL^{i}PS^e and XSB
tuProlog	bi-directional multi-platform interoperability (JVM, .NET, Android, iOS), logic programming as a library
XSB	commercial interests, tabled resolution, additional concepts (e.g., SLG resolution, HiLog programming)
YAP	focus on scalability, advanced indexing, language integrations (Python, R), integration of databases

Figure 4: A table showing the unique properties of the different Prolog implementations [16]

Pros and Cons of Prolog

Figure 5 shows a strengths, weaknesses, opportunities and threat analysis of Prolog.

Pros

- Since Prolog uses declarative semantics, it is quite easy to read and understand the individual rules and facts. It requires a different way of programming that for some can be easier than the usual programming languages.
- It allows for very compact and expressive models.
- The relational programming model can be extremely powerful for modeling constraints.
- Due to Prolog's declarative nature, it can be much easier to directly translate otherwise complex constraints into the knowledge base. A complex constraint can simply become a single rule in the knowledge base.
- Prolog allows for fast prototyping and debugging due to their easy REPL.

Cons

- Ordering of your knowledge base matters, which makes optimizing a program tedious. This can also cause a program to run worse on a different problem instance, meaning the ordering can be different for each problem.
- Performance can change drastically between different instances of the same problem. This is because the search order is fixed (leftmost goal, depth-first-search) instead of guided by heuristics.
- Prolog can loop indefinitely even though a correct solution could be available.
- Because of the execution strategy of Prolog, large search spaces are bad for performance

Table 3. SWOT Analysis

Strengths (Section 4.1)	**Weaknesses** (Section 4.3)
• clean, simple language syntax and semantics • immutable persistent data structures, with "declarative" pointers (logic variables) • arbitrary precision arithmetic • safety (garbage collection, no NullPointer exceptions, ...) • tail-recursion and last-call optimization • efficient inference, pattern matching, and unification, DCGs • meta-programming, programs as data • constraint solving (3.3.2), independence of the selection rule (coroutines (3.3.3)) • indexing (3.3.3), efficient tabling (3.3.3) • fast development, REPL (Read, Execute, Print, Loop), debugging (3.3.4) • commercial (2.10.1) and open-source systems • active developer community with constant new implementations, features, etc. • sophisticated tools: analyzers, partial evaluators, parallelizers, ... • successful applications — program analysis, — domain-specific languages — heterogeneous data integration — natural language processing — efficient inference (expert systems, theorem provers), symbolic AI • many books, courses and learning materials	• syntactically different from "traditional" programming languages, not a mainstream language • learning curve, beginners can easily write programs that loop or consume a huge amount of resources • static typing (see, however, 3.3.3) • data hiding (see, however, 3.3.1) • object orientation (see, however, 4.5.4) • limited portability (see 4.5.1) • packages: availability and management • IDEs and development tools: limited capabilities in some areas (e.g., refactoring; 4.5.2) • UI development (usually conducted in a foreign language via FLI (3.3.1)) • limited support for embedded or app development
Opportunities (Section 4.2)	**Threats** (Section 4.4)
• new application areas, addressing societal challenges 4.2: — neuro-Symbolic AI — explainable AI, verifiable AI — Big Data • new features and developments — probabilistic reasoning (3.5.1) — embedding ASP (3.5.1) and SAT or SMT solving — parallelism (2.7, 3.3.3) (resurrecting 80s, 90s research) — full-fledged JIT compiler	• comparatively small user base • fragmented community with limited interactions (e.g., on StackOverflow, reddit), see 4.4.1 • active developer community with constant new implementations, features, etc. • further fragmentation of Prolog implementations, see 4.4.1 • new programming languages • post-desktop world of JavaScript web-applications • the perception that it is an "old" language • wrong image due to "shallow" teaching of the language

Figure 5: SWOT analysis of Prolog [16]

Problems to explore

- Search spaces, it is likely that large puzzles will slow performance, but how much is the impact and can we reduce the amount of rules in order to increase performance, without affecting accuracy.
- What will the effect of different types of puzzle instances be? Will reordering the knowledgebase for an instance increase its performance and how will this in turn affect the performance on other instances.
- Prolog likely will not have a problem with puzzles with multiple answers, but what will it actually be?
- Can we **add** constraints in order to help Prolog find the correct answer faster or will this simply decrease performance?

Puzzles in Prolog

Logic puzzles with Prolog
This article gives some examples of different puzzles solved with Prolog. The puzzles don't directly correlate with the Singles puzzle, but the article gives nice examples of how the modeling constraints into rules with appropriate examples. It shows us how easy it can sometimes be to directly translate the constraints of the puzzle into Prolog.

Encodings using just Prolog, no CLP

Using Prolog to solve logic puzzles
This article is not great to read, but it shows us how the Zebra problem was encoded into pure Prolog, a nice change from all the CLP. The same author also wrote an article about solving 3 puzzles using CLP libraries.

Two fun ways to solve a logic puzzle
This is a very basic article that shows an encoding for a

puzzle using both baseline Prolog and Z3. It is nice to see the encodings using different solvers side-to-side. Unfortunately, the author mentions he is not too experienced with Prolog, so it's not likely to be a very usable source for making the encoding.

Solving a Zebra Puzzle using Prolog
This article shows a very nice way of directly translating puzzle constraints into basic Prolog facts and rules. Whilst the puzzle is again not very relatable to our puzzle, I do like to have a visual overview of how to do this process and how a set of constraints and the corresponding knowledgebase could look. It shows the thought process and allows me to attempt a similar technique for the Singles puzzle.

The Lion and the Unicorn Meet PROLOG
This paper shows the use of Prolog to make a solution for a puzzle that, in another puzzle, was solved using automatic theory-proving techniques.

Grid placement puzzles
Sudoku with Prolog
This shows a short Sudoku encoding using CLP(FD) constraints. Sudoku is much more closely related to the Singles puzzle for our assignment. Unfortunately there is not a lot of information about why or how it works, but it shows us what a puzzle solver in prolog using CLP can look like. More solvers using CLP are available on the author's github.

Luckily, a video by the same autor is available on youtube, which goes much more in depth on the problem.

N-Queens in Prolog
The N-Queens puzzle is another puzzle that is very closely related to Singles. Here, we must place a number of queens on a grid such that none of the queens is under attack. This encoding is again implemented using CLP(FD) constraints.

In the video, the author explains some optimization techniques to keep in mind when making encodings for optimizing the search space, which can get quite big for grid puzzles. This will help us to optimize our encoding for the Singles puzzle as well, as we are also dealing with a grid puzzle and placing objects with constraints.

Other useful encodings
A Pearl on SAT Solving in Prolog
This paper shows a construction of an DPLL algorithm in prolog. Whilst I wont use this for making my encoding for the Singles problem most likely, it does nicely show us some of the strengths and weaknesses of Prolog compared to a regular SAT solver. I might use these facts to reason about the usability of my own encoding, such as why we would use Prolog instead of SAT or SMT solvers.

Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system
This paper again does not correlate with our puzzle program, but it shows implementations and arguments for puzzles and why Logic Programming could have some advantages in some use cases. This could be used in my paper to properly support the benefits for using LP.

Constraint Logic Programming approach to protein structure prediction
This paper shows us that Prolog has wide use cases in fields where it has not generally been applied to yet. It can show us some more of the hidden strengths of using Prolog and maybe even help us with our problem, by using those strengths to our advantage.

A Prolog application for reasoning on maths puzzles with diagrams
Shows another example of a problem where Prolog and Logic Programming is a perfect fit where other solutions are not. Could be nice to help establish the fact that Prolog has its use. The paper is about employing a logic programming to perform spatial reasoning on a puzzle diagram in order to integrate the deriving knowledge into the solving process.

B.2 Satisfiability Modulo Theories with Z3

The research field concerned with the satisfiability of formulas with respect to some background theory is called Satisfiability Modulo Theories, or SMT, for short. SMT extends boolean SAT solving by allowing the use of meaningful constraints from arithmetic, arrays, bitvectors, and other domains, instead of encoding everything as propositional variables. The defining characteristic of SMT is that satisfiability is determined by implicitly or explicitly translating to SAT. For this reason, early SMT solvers were often called SAT-based decision procedures [3; 4].

The advantage SMT solvers are usually considered to have over pure SAT solvers, which are also often used as verification backends (e.g., for bounded model checking), is the higher level of abstraction at which they can operate. By implementing theories like arithmetic, arrays, and uninterpreted functions directly, SMT solvers have the promise to provide higher performance than SAT solvers working on encodings of such structures to the bit level. [5]

Theories in SMT
In SMT, a theory is a set of first order formulas that define the meaning of operations and relations. We do not write these theories ourselves, but they describe mathematical concepts such as the theory of linear integer arithmetic:

- Symbols:
  - Integers
  - $+, -, <, \leq, =$
- Rules:
  - Transitivity: $x < y \land y < z \Rightarrow x < z$
  - Linearity: $x + (y + z) = (x + y) + z$
  - etc.

Lazy and Eager SMT
Two of the most majorly successful approaches to SMT are lazy and eager approaches.

Eager The eager approach translates the entire input statement into a SAT instance up front. After encoding, a SAT solver handles everything. The main allure of this approach is that the translation imposes upfront all theory-specific constraints on the SAT solver's search space, potentially solving the input formula quickly; in addition, the translated formula can be given to any off-the-shelf SAT solver. However, for more expressive theories, the SAT encoding may become extremely large and inefficient.

- Can be extremely fast for certain problems.
- Easy to plug into high-performance SAT engines
- A single SAT solver does all the work solving.

But

- Certain structures can increase the SAT encoding drastically
- It is not well suited for mixed theories.

Lazy The lazy approach uses the DPLL(T) approach which rewrites the input formula into 2 layers, a boolean layer and a theory layer. For the boolean layer, all theory constraints are replaced by boolean variables. The boolean skeleton is converted to CNF and solved using a SAT solver. The SAT solver proposes truth assignments, which are then in turn checked by the theory solver for consistency in the underlying theoretic constraints. If the truth assignment is theory-inconsistent, the theory solver generates a conflict set, which the SAT solver uses to prune the search [18; 5].

Some lazy solvers use eager techniques in specific use cases, such as for dealing with bitvectors, as eager solving techniques are fast in that subject.

Important features of a lazy SMT solver:

- *Model generation* - If the theory solver is invoked on a theory-consistent set, it is able to produce a full model giving concrete values for all variables. This can be useful for, for instance, verification or debugging.
- *Conflict set generation* - When a theory solver detects inconsistencies, it produces a theory lemma (a clause explaining the conflict). This lemma is added to the SAT solver and used to prune the search indefinitely. This mechanism links semantic reasoning with syntactic pruning.
- *Incrementality* - The theory solver remembers its computation status from one assignment to another, If a new assignment extends a previously checked consistent set of constraints, the solver does not recompute from scratch.
- *Backtrackability* - It is possible for a theory solver to undo steps and return to a previous state in an efficient manner.
- *Deduction of unassigned literals (Theory propagation)* - Theory solvers can actively deduce new consequences from the current set of constraints.
- *Deduction of interface equalities* - When multiple theories are used simultaneously, solvers need to share equalities between them. Interface equalities communicate shared constraints across theory solvers, enabling consistent reasoning in combined theories.

Combining Theories

Often SMT solvers will need to reason over problems that use multiple theories. This requires a theory combination, which integrates separate theory solvers so that they can jointly decide satisfiability. Not all theories combine well together and may result in an undecidable theory, even if both theories are decidable. The Nelson-Oppen framework allows for combining theories under certain conditions. Z3 works around this problem by using custom integrations, partial encodings or restricting themselves to decidable fragments. In theory, this problem can cause issues, though you are unlikely to encounter it when using Z3.

Z3

The Z3 SMT solver is a solver using the lazy SMT approach developed by Microsoft. Z3 is an efficient SMT solver with specialized algorithms for solving background theories. It offers a wide range of built-in theories and supports the latest SMT-LIB language. SMT-LIB also holds a yearly competition called the SMT-COMP [9; 7].

Z3 is available in Python, making it very easy to install using pip and use. It also has an online variant available. There is documentation for the Python version available by Microsoft, which also makes available some example programs with explanations about the problem.

Pros and Cons of SMT

Pros

- **Highly expressive** - Constraints can be written in a natural mathematical form. Compared to SAT this creates a much easier developing cycle and makes them much easier to understand, debug and verify.
- SMT combines both SAT and Theory solvers. The hybrid approach is what makes it very efficient in multiple domains.
- SMT can generate fully satisfied concrete models. Whilst SAT solvers can also do this, models generated by SMT solvers can fill models with actual values, not just boolean values.
- Because of its high expressiveness, many models map easily to SMT.

Cons

- For pure boolean problems, SAT would be faster.
- Depending on the problem, performance might differ greatly. Combining theories, large SAT assignments, or generally hard theories can differ between problems and cause large differences between problems.
- Some theories are undecidable.
- Encoding is very important and bad encodings perform badly

SMT generally works better for problems with structured constraints and is not ideal if the problem is purely boolean, if theories are huge, if there are many SAT combinations or if nonlinear arithmetic is needed.

Problems to explore

- Puzzle size will be an obvious question to explore, as input size seems to be quite a factor with SMT. Can we change the encoding to minimize runtime?
- I'm interested in looking at puzzles with multiple possible solutions. Does a puzzle with multiple solutions run faster or not?
- What is the effect of certain patterns of puzzles? Does my encoding pick up on it? Does it run faster or slower?
- I want to experiment with different encodings without redundancy. What would the difference be if we can make a more compact encoding?
- I want to look into adding redundant constraints. If we add some 'helping' constraints to a working encoding, will it help the runtime by helping the SAT pruning? What if the constraint adds an extra theory into the mix? How many and which constraints work best?

Puzzles with SMT

SAT/SMT by Example
A collection of puzzles and problems solved using SAT and SMT solvers. This can create a nice baseline in case we ever need to better understand how to translate a set of constraints into a working solver. It is a collection of problems and not so much a tutorial/article, so referencing it might be more difficult outside of our own development process.

Two fun ways to solve a logic puzzle
This is a very basic article that shows an encoding for a puzzle using both baseline Prolog and Z3. It is nice to see the encodings using different solvers side-to-side. Unfortunately, the problem is not similar to our puzzle, but at least it gives a view on the syntax and what an encoding could look like.

Jorian Woltjer z3 scripts
A collection of z3 solvers for solving various puzzles. This includes the 8 queens and sudoku puzzles, which are closely related to the Singles puzzle. Unfortunately there is no explanation and reasoning for choices given at all, so this cannot effectively be used as a source in our research except for some inspiration and examples on how to solve other puzzles.

Solving SAT and SMT problems using Z3
This source gives some puzzles, but more importantly it also shows us some of the practical and theoretical limitations of SMT solvers.

Grid like or constraint puzzles

Coding Interview – Solve Sudokus Using Python and Z3
This article shows a nice tutorial of how to build a Z3 solver for solving the Sudoku puzzle. It does unfortunately not go into the details about why or how this works and is really just a top-level tutorial, but it might still be useful for building our own solver for the Singles puzzle.

Solving and generating puzzles with a connectivity constraint
This source just straight up gives an SMT solver for Hitori, which is the Singles puzzle. Connectivity constraints are the major part of our constraint set, so this source will be extremely useful. We can also use this source to compare our solver to that of the Hatori solver, check if there is different design choices and reason as to why. Lastly, this gives an explanation source on Hatori and a proof of NP completeness.

Latin squares with graph properties
This paper investigates how to encode graph properties such as connectedness, tree structure or Hamiltonian as requirements for a SAT/SMT solver. This paper talks about adding graph properties to Latin squares, which is a partially related to our Singles squares.

Other useful encodings and sources

An Algorithmic and SMT-Based Approach
This study shows a nice comparison between an algorithmic approach to solve a puzzle and the SMT approach. Whilst the SMT solver itself might not be that useful, having a study on the actual advantage and comparison is very handy to have and we could use this to expand our own reasoning on the usability of SMT in problem solving.

Evaluating SAT and SMT Solvers on Large-Scale Sudoku Puzzles
This article compares the performance of SMT solvers such as Z3 and CVC5 against an older SMT solver in DPLL(T) and a SAT solver in DPLL on a 25x25 Sudoku puzzle. Such comparisons add more insights into the strengths and reasons for using SMT solvers. On top of this, this article gives a nice summary of some of the concepts and gives a small explanation of the Z3 implementation of a Sudoku solver.

Modeling constraint satisfaction problem with model checker
Whilst this paper is not necessarily about Z3, it is used as the comparison tool. This paper shows some more insights into SAT and a comparison in performance, which can again be used in our own study.

Reformulation of constraint models into SMT
This source aims to reformulate constraint models in SMT instances in order to be solved by SMT solvers. The study shows that SMT is competitive with state-of-the-art CSP and WCSP solvers. Another comparison study, can be useful in the paper, unfortunately not useful for making our encoding.

B.3 Comparing LP vs SMT

Since the foundations of Logic Programming and Satisfiability Modulo Theories are vastly different, it can be difficult to compare the two paradigms. The paradigms differ completely in their operational details, but we can try to compare their core philosophies. Below are a couple of fields of comparison between the two.

Commonalities
Logic Programming and SMT are both declarative modelling-and-solving paradigms. the user describes a problem using formulas or rules, and the system automatically searches for assignments that satisfy these descriptions.

Both are rooted in fragments of first-order logic, both aim to find satisfying assignments, and both are commonly applied to constraint satisfaction problems.

Differences

LP finds the solution to a query using proof search. a program is a set of logical rules, and queries are answered by deriving them from the program using a proof procedure such as resolution. LP can naturally express a recursive and relational structure and can be useful for describing concepts like reachability or connectedness.

SMT treats computation as model finding. The user writes a formula using background theories (e.g., linear arithmetic, bit-vectors, arrays), and the solver decides satisfiability using a combination of SAT solving and theory solving (DPLL(T)). SMT provides strong support for numeric and structural theories and offers robust global propagation.

LP focusses symbolic reasoning, relational definitions, and inference, whereas SMT emphasises constraint specification over theories and automatic model construction. Both are declarative, but they differ in the direction of computation (goal-directed vs model-directed), the naturalness of expressing certain types of constraints, and the underlying foundations.

Expressiveness

The expressiveness or how easy it is to transform a constraint into a solver format. This is important for the ease of use of the paradigm, if it is difficult to translate the constraints, programming and debugging a solver might take an incredible amount of time and simple puzzles will be very difficult to solve. Luckily, both LP and SMT are quite expressive. LP is literally built on building relations between variables instead of writing functions. This often makes transforming a constraint, which is a type of relation, quite easy. It is also not that big of a problem using SMT. Since SMT uses theories on top of a SAT solver, we do not have to rewrite the constraints into a long boolean formula. Instead, we are still able to use the appropriate theories to build the constraints. It's not like LP, but also not difficult.

Solver behaviour (Prolog vs Z3)

The behaviour of the solver and the way it resolves the problems is very important. Certain techniques are much better suited for specific problem types and thus this often decides whether we should use one solver over another. SMT solvers often make use of a variant of DPLL; this algorithm has the potential to be very inefficient on certain problems and large datasets. LP uses resolution with unification, Prolog uses resolution with depth-first-search, which can be very inefficient in certain encodings.

Understanding the underlying mechanisms of the different paradigms and solvers allows us to make better decisions of when to use a certain paradigm over the others.

C Generator Proof

This proof was initially constructed by Sappho de Nooij, Tom Friederich and Lesley Smits, rewritten by Sophieke van Luenen and reviewed by Robin Rietdijk

Below, we prove that our generator can generate any single-solution Hitori puzzle, and nothing else. We do so in two steps. In the first step we prove that we can generate any valid solution topology. In the second step we prove that, given a valid solution topology S, we can generate any valid Hitori puzzle that has that solution topology, and only that solution topology. In the last step we put everything together to prove the below theorem:

Theorem 1: *Algorithm 1 is complete. That is, our generator can generate exactly only every uniquely-solvable puzzle H.*

Algorithm 1 Algorithm that exactly any valid Hitori instance H

```
function GENERATEHITORIINSTANCE
    Let S = GENERATESOLUTIONTOPOLOGY
    return GENERATEHITORIINSTANCE(S)
end function
```

C.1 Generating any Solution Topology

A solution topology S is an $n \times n$ grid where each element $S_{i,j}$ (with $i, j \in [1..n]$) is *marked* or *unmarked*. Given a Hitori instance H with S as its solution instance, having $S_{i,j}$ be *marked* means the solution of H has tile $H_{i,j}$ marked. Similarly, if $S_{i,j}$ is *unmarked*, the solution of H has tile $H_{i,j}$ unmarked. A solution topology S is valid if it adheres to the three constraints defined by the Hitori rules. On top of this, since we only want uniquely-solvable Hitori instances, the topologies we generate must be able to create uniquely-solvable Hitori instances.

Lemma 1.1: *A solution topology S with at least one uniquely-solvable problem does not have an unmarked tile that can be marked without violating the* adjacency *and* connectivity *constraint.*

Proof. We will use a proof by contradiction. Take an arbitrary uniquely-solvable puzzle P with solution topology S, such that S has an unmarked cell c that can be marked without violating the adjacency and connectivity constraint. We now construct a solution topology S' which is identical to S except that cell c is marked. Marking c does not violate the adjacency or connectivity constraint (by the definition of P and S). The uniqueness constraint is also not violated since we do not change the numbers of unmarked cells or add an unmarked cell. Therefore S' is another valid solution topology for P. This creates a contradiction as P was set to be uniquely solvable.

Since the assumption leads to a contradiction, it must be false. Therefore S cannot contain an unmarked cell that can be marked without violating the adjacency and connectivity constraint. □

Given Lemma C.1 our generator may not generate solution topologies with unmarked tiles that can be marked without violating the adjacency or connectivity constraints.

Algorithm 2 is a pseudo-code representation of the algorithm with which we generate our solution topologies.

Algorithm 2 Algorithm that generates a solution topology S.

```
function GENERATESOLUTIONTOPOLOGY
    Let S[1, . . . , n][1, . . . , n] be the two-dimensional array of tiles, all unmarked
    Let C be the collection of all coordinates in S
    ((1, 1), (1, 2), . . . (1, n), (2, 1), . . . , (n, n)) in random order
    for i = C[1] to C[n^2] do
        if no orthogonally adjacent tile is black then
            S[i] = marked
            if the unmarked tiles of S are disconnected then
                S[i] = unmarked
            end if
        end if
    end for
    return S
end function
```

Lemma 1.2: *Algorithm 2 only generates valid solution topologies.*

Proof. For any marked tile that the algorithm places it checks whether the adjacency or connectivity constraint are met. If this is not the case, it rolls back the decision and moves on. Since no numbers are generated in Algorithm 2 it cannot break the uniqueness constraint.

Since our generator loops over every tile on the board and checks whether it can be marked, and only leaves the tile unmarked if it were to break the adjacency or connectivity constraints, it cannot generate any solution topology with unmarked tiles that could be marked without violating the adjacency or connectivity constraints. □

Lemma 1.3: *Algorithm 2 can generate exactly only any valid solution topology.*

Proof. Algorithm 2 generates solution topologies by iterating over the tiles in a random order. We will use this to show that it can generate any valid solution topology.

Take any valid solution topology S with marked tiles m and unmarked tiles u. Since the solution topology is valid, none of the tiles in m violate the adjacency or connectivity constraints. Since Algorithm 2 visits tiles in a random order, there is a non-zero chance that it will first visit all the tiles in m before visiting any tile in u. Since none of the tiles in m violate the adjacency or connectivity constraints, all will be marked by the algorithm.

Since S is a valid solution topology, no tiles in u could be marked without breaking the adjacency or connectivity constraints, thus when the algorithm visits the tiles in u, it will mark none of them. After having visited the last tile in u, the algorithm will return solution topology S.

Now given that Lemma C.1 proves that Algorithm 2 can only generate valid solution topologies, we have now proven that the algorithm can generate exactly only any valid solution topology. □

C.2 Generating any valid Hitori puzzle from S

A puzzle instance of Hitori H is an $n \times n$ grid of numbers where each element $H_{i,j} \in [1..n]$ with $i, j \in [1..n]$. We only consider H a valid instance if it has a single valid solution topology S.

Algorithm 3 is a pseudo-code representation of our algorithm for generating an instance H from a given solution topology S. It consists of two subsequent algorithms, Algorithm 4 which generates numbers for the tiles in H which correspond to unmarked tiles in S, and Algorithm 5 which generates numbers for the tiles in H which correspond to marked tiles in S.

Algorithm 3 Algorithm that generates a Hitori instance H from a solution topology S.

```
function GENERATEHITORIINSTANCEFROMS(S)
    Let H[1,...,n][1,...,n] be a grid of 0s
    FILLUNMARKEDCELLS(H, S, n, 1)
    FILLMARKEDCELLS(H, S, n, 1)
    return H
end function
```

Algorithm 4 Algorithm that fills in the unmarked cells given a partial Hitori instance H and a solution topology S.

```
function FILLUNMARKEDCELLS(H, S, n, k)
    Let i = ⌈k/n⌉
    Let j = ((k − 1) mod n) + 1
    if k > n^2 then
        return true
    else if S[i][j] == marked then return FILLUNMARKEDCELLS(H, S, n, k + 1)
    else
        Let row be the numbers used in the row of H[i][j]
        Let col be the numbers used in the column of H[i][j]
        C = {1,...,n} \ row \ col
        if C = ∅ then
                                    ▷ We check if a conflict occurred
            return false
                      ▷ this is optimized by analyzing the conflict and returning to the conflict's cause
        else
            shuffle C
            H[i][j] = C[1]
                                    ▷ Assign H[i][j] the first element in C
        end if
    end if
     return FILLUNMARKEDCELLS(H, S, n, k + 1)
end function
```

Algorithm 5 Algorithm that fills in the marked tiles of a partial Hitori instance H.

```
function FILLMARKEDCELLS(H, S, n, k)
    Let i = ⌈k/n⌉
    Let j = ((k − 1) mod n) + 1
    if k > n^2 then
        return true
    else if S[i][j] == unmarked then return FILLMARKEDCELLS(H, S, n, k + 1)
    else
        Let row be the numbers used in the row of H[i][j]
        Let col be the numbers used in the column of H[i][j]
        C = row ∪ col
        shuffle C
        H[i][j] = c[1]
                                    ▷ Assign H[i][j] the first element in C
    end if
     return FILLMARKEDCELLS(H, S, n, k + 1)
end function
```

Lemma 1.4: *Given a valid solution topology S, Algorithm 4 can generate all valid combinations of numbers in unmarked tiles.*

Proof. Take any valid partial Hitori instance H corresponding to solution topology S, which has numbers assigned to all its unmarked tiles such that all of the assigned numbers are unique in their row and column. We will now show that our generator can create this partial Hitori instance.

Our generator iterates over all tiles in order, moving from left to right, top to bottom. At each unmarked tile the generator will select a list C of valid numbers to put in this tile. This list consists of the numbers $1, 2, \ldots, n$ excluding any number that is already present in the row or column.

If a number is not in C, putting it in the given tile would not result in a valid partial Hitori instance corresponding to the solution topology S, as it would either break the uniqueness constraint if it remains unmarked in the solution, or it would break the adjacency or connectivity constraints if it is marked (by the definition of S).

Since C contains all valid numbers that the tile could receive, and Algorithm 4 selects a number at random, each possible valid number has a non-zero chance of being chosen, including the corresponding value in H. Since this holds for every unmarked tile that the algorithm visits, it can generate H. As such, given a valid solution topology S, Algorithm 4 can generate all valid combinations of numbers in unmarked tiles. □

Lemma 1.5: *Given a valid solution topology S, Algorithm 4 can only generate valid combinations of numbers in unmarked tiles.*

Proof. Any invalid combination of numbers in unmarked tiles has to contain two of the same numbers on a given row or column. Since Algorithm 4 selects a number to give to a tile from a list C that contains every number from 1 to n excluding any number that is already present in the row or column, the generator cannot create an invalid combination of numbers in unmarked tiles. □

Lemma 1.6: *Given a valid solution topology S, and a valid partial Hitori instance H with numbers assigned to each unmarked tile, Algorithm 5 can generate any valid combination l of numbers for in the marked tiles.*

Proof. For a combination of numbers for in the marked tiles to be valid, each number in l must already be present in the row or column that l will be assigned to. When assigning numbers to tiles, Algorithm 5 will create a list C which consists of all numbers in the row and column of the given tile.

Furthermore, since all numbers in l must be covered, assigning multiple tiles in l with a new number that is not present in their row and column is not a valid move: at least one of those tiles will not have to be covered.

Algorithm 5 then randomly selects a number from C and assigns it to the given tile. Given that C contains all valid options for in the tile, and given that the number is chosen at random from C, each number has a non-zero chance of being selected for the tile. As such, Algorithm 5 can generate any valid combination l of numbers for in the marked tiles. □

Lemma 1.7: *Given a valid solution topology S, and a valid partial Hitori instance H with numbers assigned to each unmarked tile, Algorithm 5 can generate only any valid combinations l of numbers for in the marked tiles.*

Proof. For a combination of numbers for in the marked tiles to be invalid, at least one number in l must not already be present in the row or column that l will be assigned to. Since we pick a number at random from C, and C only contains numbers from the tiles' row and column, it is not possible for the generator to pick an invalid number. As such, Algorithm 5 cannot generate an invalid combination of numbers for in the marked tiles. □

Lemma 1.8: *Given a valid solution topology S, Algorithm 3 can generate any valid puzzle instance H.*

Proof. Lemma C.2 proves that, given any valid solution topology S, we can generate all valid combinations of numbers for the unmarked tiles of a valid corresponding partial Hitori instance H. Lemma C.2 proves that we can generate nothing but valid combinations of numbers.

Lemma C.2 then proves that given a valid solution topology S, and a valid partial Hitori instance H, we can generate any valid combination of numbers for the marked tiles in H. Lemma C.2 proves that we can only generate valid combinations of numbers for the marked tiles in H.

Since we can generate only exactly any valid combination of unmarked tiles, and given any valid combination of unmarked tiles we can generate only exactly any valid combination of marked tiles, we can generate any valid combination of tiles to create a valid Hitori instance given a valid solution topology S. □

C.3 Proving Theorem C

Proof. Lemma C.1 has proven that our algorithm can generate exactly any valid solution topology S, and Lemma C.2 has proven that, given any valid solution topology S we can generate exactly only any valid single-solution puzzle instance H. These lemmas together prove that Algorithm 1 can generate exactly only every uniquely-solvable puzzle H. □

D Redundant constraint descriptions

In this appendix:

- $\text{black}_{i,j}$ denotes the cell (i, j) being colored black.
- $\text{white}_{i,j}$ denotes the cell (i, j) being colored white.
- $\text{symbol}_{i,j}$ denotes the value of the cell (i, j).

D.1 White Neighbours (WN)

This constraint is meant to reinforce the connectivity constraint. For every puzzle with $n > 1$:

$$\forall i, j : \quad \text{white}_{i,j} \Rightarrow$$
$$(j < n \wedge \text{white}_{i,j+1}) \vee (i < n \wedge \text{white}_{i+1,j}) \vee$$
$$(j > 1 \wedge \text{white}_{i,j-1}) \vee (i > 1 \wedge \text{white}_{i-1,j})$$

D.2 Corner Close (CC)

[24] For every puzzle with $n > 1$, we cannot block the connecticity of the corner cell. Formally, for every corner:
(Top-left corner)

$$\text{black}_{0,1} \Rightarrow \neg\text{black}_{1,0},$$
$$\text{black}_{1,0} \Rightarrow \neg\text{black}_{0,1},$$

(Top-right corner)

$$\text{black}_{0,n-2} \Rightarrow \neg\text{black}_{1,n-1},$$
$$\text{black}_{1,n-1} \Rightarrow \neg\text{black}_{0,n-2},$$

(Bottom-left corner)

$$\text{black}_{n-2,0} \Rightarrow \neg\text{black}_{n-1,1},$$
$$\text{black}_{n-1,1} \Rightarrow \neg\text{black}_{n-2,0},$$

(Bottom-right corner)

$$\text{black}_{n-1,n-2} \Rightarrow \neg\text{black}_{n-2,n-1},$$
$$\text{black}_{n-2,n-1} \Rightarrow \neg\text{black}_{n-1,n-2},$$

D.3 Sandwich Triple (ST)

One of the human solving patterns decribed by Gander and Hofer [12], and Wensveen [24] visualized in 6. This pattern is applicable everywhere in puzzles of size $n > 2$ in both columns and rows.

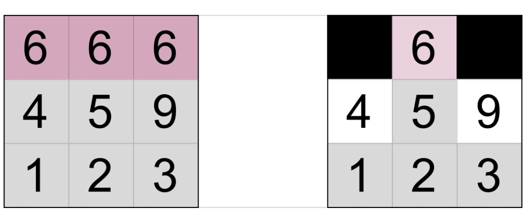


Figure 6: Pattern and implication

D.4 Sandwich Pair (SP)

[12; 24] Pattern that is similar to the Sandwich Triple, but now the center tile has a different symbol than the sandwiching tiles, visualized in 7. the sandwiched tile is always white.

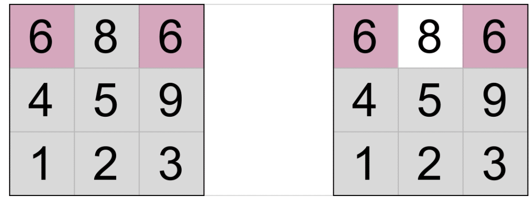


Figure 7: Pattern and implication

D.5 Triple Corner (TC)

Another human solving pattern described by Gander and Hofer [12], visualized in figure 8. This pattern is only applicable in corners of puzzles of size $n > 1$.

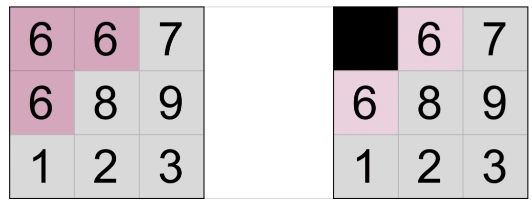


Figure 8: Pattern and implication

D.6 Quad Corner (QC)

Human solving pattern described by Gander and Hofer [12], visualized in figure 9. This pattern is only applicable in corners of puzzles of size $n > 2$.

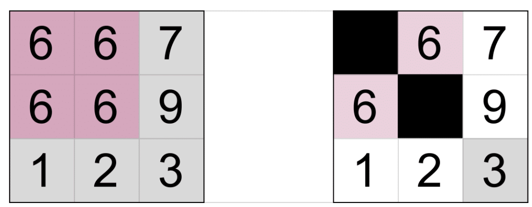


Figure 9: Pattern and implication

D.7 Triple Edge Pair (TEP)

Human solving pattern described by Gander and Hofer [12], visualized in figure 10. This pattern is only applicable at borders of puzzles of size $n > 3$.

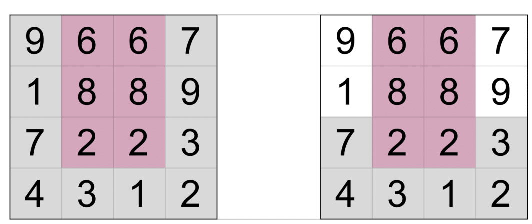


Figure 10: Pattern and implication

D.8 Double Edge Pair (DEP)

Human solving pattern described by Gander and Hofer [12], visualized in figure 11. This pattern is only applicable at borders of puzzles of size $n > 3$.

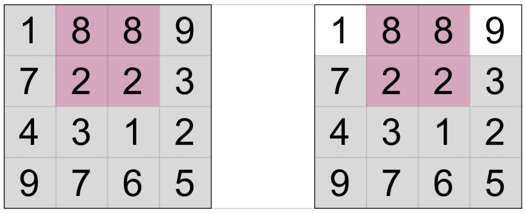


Figure 11: Pattern and implication

[13] also describes Triple Edge Pair, where the same pattern is applied but for a 2x3 part. This can be generalized "N-Edge Pair".

D.9 Close Edge (CE)

If a white cell is threatened to be closed off, make sure it won't be. using [13]

D.10 Force Double Edge (FDE)

If a white cell is threatened to be closed off, make sure it won't be. using [13]

D.11 Border Close (BC)

Human solving pattern described by Gander and Hofer [12], visualized in figure 12. This pattern is applicable only at borders in a puzzle of size $n > 3$.

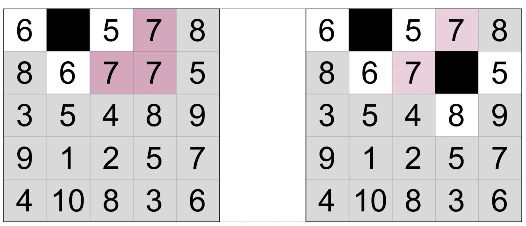


Figure 12: Pattern and implication

D.12 Least Whites (LW)

Constraint to help the connectivity constraint by restricting the placement of black and white cells in the grid. We enforce the number of white cells in rows and columns by imposing the following rules:

$$\forall i \in \{0, \dots, n-1\} : \sum_{j \in \{0,\dots,n-1\}} \text{white}_{i,j} \geq \lfloor \frac{n}{2} \rfloor$$

$$\forall j \in \{0, \dots, n-1\} : \sum_{i \in \{0,\dots,n-1\}} \text{white}_{i,j} \geq \lfloor \frac{n}{2} \rfloor$$

D.13 Most Blacks (MB)

Constraint to help the connectivity constraint by restricting the placement of black and white cells in the grid. We restrict the number of black cells in rows and columns by imposing the following rules:

$$\forall i \in \{0, \dots, n-1\} : \sum_{j \in \{0,\dots,n-1\}} \text{black}_{i,j} \leq \lfloor \frac{n}{2} \rfloor + 1$$

$$\forall j \in \{0, \dots, n-1\} : \sum_{i \in \{0,\dots,n-1\}} \text{black}_{i,j} \leq \lfloor \frac{n}{2} \rfloor + 1$$

D.14 Pair Isolation (PI)

Human solving pattern described by Gander and Hofer [12] and Wesnveen [24], visualized in figure 13. This pattern is applicable anywhere in a puzzle of size $n > 3$ for both rows and columns.

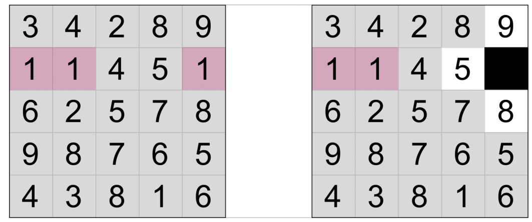


Figure 13: Pattern and implication

D.15 Close Isolation (CI)

Human solving pattern described by Gander and Hofer [12], visualized in figure 14. This pattern is applicable anywhere in a puzzle of size $n > 4$ for both rows and columns.

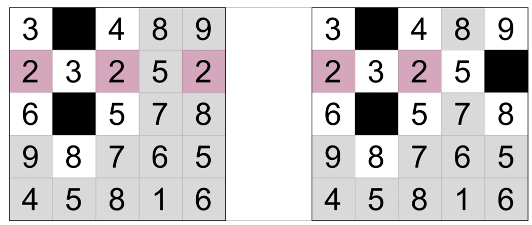


Figure 14: Pattern and implication

D.16 White Bridges (WB)

This constraint is meant to reinforce the connectivity constraint by making sure white cells are connected in all columns and rows. We introduce the following rules:

$$\forall i \in \{0, \dots, n-2\} : (\exists j : \text{white}_{i,j}) \wedge (\exists j : \text{white}_{i+1,j}) \Rightarrow \exists j : (\text{white}_{i,j} \wedge \text{white}_{i+1,j})$$

$$\forall i \in \{0, \dots, n-2\} : (\exists j : \text{white}_{j,i}) \wedge (\exists j : \text{white}_{j,i+1}) \Rightarrow \exists j : (\text{white}_{j,i} \wedge \text{white}_{j,i+1})$$

E Usage of Large Language Models

During this study, ChatGPT was used for feedback, reflect on ideas, clarify concepts or to get suggestions for new directions to explore. I used the paid model if ChatGPT, allowing the use of GPT-5.1/5.2 Thinking. ChatGPT base style and tone were used, with no changes to its characteristics or custom instructions.

All chats were conducted within a single project, allowing ChatGPT to use information from other chats within that same project. The project was given a small instruction to set the context of the project as follows: "In this project, we discuss parts of my thesis in solving Singles using Satisfiability Modulo Theories. The Singles puzzle is also known as Hitori. I am going to be using Python to implement my encodings for solving Hitori in SMT. I will be creating SMT encodings and explore with alternatives such as alternate theories or other versions of the basic Hitori rules. My goal is to evaluate SMT for solving Hitori solvers and find its usability, strengths and weaknesses."

Below is an overview of the conversations that were created and used during this project. For each conversation, we describe how it was used and what was concluded or used in the project.

As ChatGPT tends to give long and repeating answers which can be difficult to format, even to small prompts, some prompts that were not relevant for the project, repeated information or that implemented code have been left out. In all of these cases a description of the left-out conversation is given instead.

E.1 Conversation about alternative theories

In this conversation, our goal was to get inspiration and feedback on other SMT theories to explore for the project. Before conculting ChatGPT, an initial encoding was already constructed and a possible encoding using bitvectors was being looked at.

prompt I have implemented an SMT encoding into linear integer arithmetic and am looking for alternative theories. I was thinking of bitvectors because of bitblasting. What do you think which theories would be best suited for this puzzle? For reference, here is my current implementation of LIA: [Code implementation]

answer Short answer: for Singles/Hitori-style puzzles, LIA is already the "natural" SMT theory. Anything else is either equivalent-but-more-work or strictly worse in practice. If you had to use something else, the only genuinely interesting alternative is:

Equality with uninterpreted symbols (and maybe arrays): Treat the cell values as an enumerated / uninterpreted sort and keep everything else Boolean.

Let me unpack that in puzzle terms and compare options.

What structure your encoding actually needs Your Singles encoding mainly uses:

- A small finite domain for each cell value: 1,. . . ,n
- Equality/inequality on those values (for "no duplicates in row/column")
- Local Boolean constraints for black/white adjacency
- A global connectivity constraint (often encoded via integer labels or flows)

Crucially: you're not doing real number arithmetic, multiplication of decision variables, or anything non-linear. It's "nice" linear-ish stuff. That's exactly what QF_LIA / UFLIA is tailored for.

Candidate alternative theories and how well they fit 1. Bitvectors (QF_BV)

- Internally, QF_BV is bit-blasted to SAT. You lose all the high-level arithmetic reasoning that the LIA engine has (like cutting planes, propagating bounds, etc.).
- Your constraints (equality checks, ordered ranges) are very regular and arithmetic-y, which the LIA engine exploits. In BV, they become low-level bit constraints.

So: BV is semantically fine but not "better suited" for this problem than LIA – usually the opposite.

2. Equality + Uninterpreted Functions / Enumerated Sorts (QF_UF / ADTs) This is the one alternative that's genuinely interesting from a modelling perspective:

- Represent each symbol not as an Int but as an element of a custom sort, e.g.: (declare-datatypes () ((Value v1 v2 v3 ... vn))) Now each grid cell is a Value instead of an Int.
- The "no duplicates in a row/column" constraints only need equality, so they carry over directly: grid[i][j] = grid[i][k] =¿ one of them must be black.
- You can then encode connectivity purely with Booleans + a reachability style encoding (e.g., BFS levels or spanning-tree parent pointers where parents are positions, not values).

This pushes you towards a QF_UF + Bool + maybe arrays setting instead of arithmetic. It doesn't usually perform better than LIA, but it's:

- A clean fit if you want to emphasise "symbolic" reasoning rather than numeric.
- A nice way to contrast "numbers as arithmetic objects" vs "numbers as opaque symbols" in your paper/poster.

Downside: you have to re-do your connectivity constraints without integer labels / distances, which is nontrivial.

3. Arrays (QF_AUFLIA / QF_AUF) Using Array (Int, Int) -¿ Value/Int to represent the grid:

- This is mostly a representation change, not a theory change.
- Arrays shine when you have updates or unknown positions. In Singles, positions are fixed and all cells are variables anyway.
- You'll still be in LIA (or UF) for indices/values, so it's not really "other than LIA" in spirit.

So: arrays don't buy you much here.

4. Finite sets / multisets You could, in theory, encode each row's multiset of values as a finite set and assert no duplicates by set cardinality tricks. In practice:

- Z3's finite sets are cool but relatively heavy.
- You'd still need LIA for cardinals or a bunch of auxiliary Booleans.

This is more of a curiosity than "best suited".

So what's actually "best" if not LIA?

If you're asking performance-wise: LIA (what you already have) is the best suited SMT theory. If you want a conceptually different SMT encoding to talk about in your report/-poster:

- Enumerated / uninterpreted values + Booleans
  - Values: ADT or UF sort instead of Int.
  - Constraints: only equality (no arithmetic), plus Boolean constraints for adjacency & connectivity.
  - Theory label: something like QF_UF (possibly with arrays for the grid).
- Optionally, pure propositional (SAT)
  - Not SMT anymore, but: encode each "cell has symbol k" as a Boolean and push everything into CNF.
  - Then you can compare your SMT LIA encoding to a SAT version solved with e.g. kissat (if that fits the project scope).

Practical suggestion for your project

Given where you are now, I'd frame it like this:

- Primary encoding: QF_LIA (integers + Booleans) – main experiments.
- Secondary / exploratory encoding:
  - Either a QF_BV attempt,
  - Or a QF_UF / enumerated sort variant to show a qualitatively different modelling style.

If you'd like, I can help sketch what the "values-as-enumerated-sort" version would look like in Z3Py, including how you'd do connectivity without integer labels.

description This prompt was followed by a discussion on my implementation of the Bitvector encoding, cleaning up and correcting some of the mistakes in the code. We used small prompts such as "This is how i've implemented bitvectors currrently, is this the correct way to do this?" followed by a small code snipped of my implementation to get feedback on the current implementation. This conversation resulted in a complete Bitvector encoding implemented in code.

The next conversation continues relecting on alternate theories to implement.

prompt I have implemented the basic constraints using qf_lia and qf_bv. I am looking for alternate solvers. Should I try more theories or maybe start looking for redundant constraints?

answer Short answer: I'd stop hunting for more theories and start playing with redundant constraints + alternative encodings inside QF_LIA.

A bit more structured: 1. Alternate "theories" vs. better encodings For Singles, your natural ingredients are:

- Integers in [1..n] for the grid
- Booleans for "cell is black/white"
- Linear constraints for uniqueness, adjacency, connectivity

That already fits QF_LIA very well. QF_BV is essentially giving you fixed-width ints instead of unbounded ints; it doesn't magically give you a smarter solver, so seeing it get slower on larger instances isn't shocking – bitvector reasoning is often more low-level and less optimized for this kind of "combinatorial integer + graph" structure.

Other Z3 theories (arrays, uninterpreted functions, datatypes) don't obviously match new structure in the puzzle, so adding them usually just makes the logic more complex for no gain.

So:

- Stick with QF_LIA as your main workhorse.
- Keep QF_BV as a comparison point.
- Don't chase exotic mixed-theory encodings unless you have a concrete idea for connectivity or something similar that truly needs them.

If you want "alternate solvers", that's more like feeding the same SMT-LIB encoding to cvc5, Yices, Boolector, etc., which is nice but optional.

2. Where the real wins usually are: redundant constraints Some concrete ideas you can try now:

a) Stronger uniqueness constraints

Right now, you probably have pairwise implications like:

if grid[i][j] == grid[i][k] then at least one of (i,j), (i,k) is black.

This is correct, but weak. You could add cardinality-style constraints: For each row r and value v, let S_r,v be the set of cells with that value. Add: "At most one white" in S_r,v:

$$\sum_{(i,j)\in S_{r,v}} \neg\text{black}_{i,j} \leq 1$$

Same for columns.

These are logically implied by your pairwise uniqueness constraints but they prune search much earlier.

b) Simple bounds and counting constraints

Things that are obviously true but not explicitly encoded yet, for example:

- Each row must have at least one white cell (otherwise numbers can't all be unique).
- Each column must have at least one white cell.
- Upper bounds on number of black cells in a row/column (e.g. it can't be all black, can't be all but one black if that would break connectivity).

You can experiment here: some bounds might not help, others might give big speedups.

c) Symmetry-breaking

Singles puzzles usually don't have huge symmetry, but your encoding might:

- If your puzzle is symmetric under flipping/rotation, you can fix one or a few cells: e.g. "top-left cell is white" if both choices are symmetric.
- Or a lexicographic ordering constraint on the first row/-column of black/white bits.

Symmetry-breaking is classic “redundant but helpful” stuff.

d) Redundant connectivity information

If you encoded connectivity via some reachability / flow / spanning-tree style constraints, you can often add redundant things like:

- “The number of white cells is $\geq$ size of any connected component candidate” (component labelling).
- Local constraints that forbid small disconnected islands that must be impossible given other rules.

Even simple extra constraints like “every white cell must have at least one white neighbour (unless it’s the only white cell)” can help tighten the connectivity model.

3. A practical plan from here If I were in your shoes, I’d do:

- Freeze your baseline QF_LIA model (version-controlled).
- Implement one redundant-constraint idea at a time:
  - Start with cardinality-style uniqueness constraints (they’re easy and often powerful).
  - Then add a simple symmetry-break (if applicable).
  - Then experiment with one or two connectivity-strengthening constraints.
- For each variant, run your benchmark set (e.g. 5×5, 10×10, 15×15, 20×20; maybe multiple instances per size) and log:
  - solve time
  - number of conflicts / decisions (if you can query them)
  - maybe encoding size (clauses, assertions)

Only after you’ve squeezed a bit of juice out of redundant constraints would I invest more time in new theories or external solvers.

prompt So there is no point in looking at other theories? Was it even useful to try bitvectors?

answer Short, honest answer:
No — there was no strong modelling reason to try bitvectors for Singles.
Yes — it was still useful that you tried them.

Now the longer, more helpful breakdown:

Was there a point in trying bitvectors (QF_BV)?

Yes, from a research perspective. Even though BV isn’t the “right” theory for this puzzle, comparing it to QF_LIA still gives you:

1. A meaningful baseline comparison

Your report can now say something like:
“Although bitvectors are often efficient for low-level reasoning, our puzzle constraints require high-level arithmetic and Boolean structure. BV encoding therefore produced larger/less structured constraints and scaled worse than linear integer arithmetic.”
That is a valid and publishable experimental observation.

2. Empirical insight

You’ve experimentally confirmed something important:

- BV $\rightarrow$ fixed-width integer arithmetic
- LIA $\rightarrow$ high-level symbolic integers + linear constraints

Since Singles constraints never require bit-level reasoning, the BV solver essentially “bit-blasts” everything into SAT, which explodes combinatorially as puzzle size increases.

That’s good insight and can be discussed in your methodology and results.

3. It clarifies what’s not worth pursuing

You now know:

- The puzzle doesn’t need modular arithmetic
- It doesn’t need overflow semantics
- It doesn’t benefit from bit-blasting

This is exactly what a research methodology aims to show: You tested plausible alternatives and justified why you don’t continue with them.

So:
Even if it didn’t help performance, it was still useful academically.

Is there a point in trying other theories?
For this puzzle: Not really.

Let’s check the likely candidates:

Arrays theory

Useful if:

- You want indexed reads/writes
- You have variable indices
- You want functional constraints

But your grid is fixed-sized and fully unrolled; arrays give no real advantage.

Uninterpreted Functions (UF)

Useful if:

- You want abstraction
- You want to eliminate arithmetic detail

Not useful here, since Singles uses plain integers and Booleans.

Datatypes / Algebraic data types
Overkill. Your grid isn’t an algebraic structure.
Non-linear arithmetic
Singles does not require multiplication of variables — so don’t go there.
Quantifiers
These explode and are unnecessary — you already expand constraints for every cell.
Strings
Obviously irrelevant.

Conclusion

Bitvectors:

- Not the right theory for this puzzle
- But still valuable to test
- You can use the results in your analysis section
- Now you can justify why QF_LIA is your main model

Other theories:

- No useful alternative theories for Singles
- All important rules fit naturally into integer linear arithmetic + Booleans

Where you should go next:

- Better QF_LIA modelling
- Add redundant constraints
- Try alternative encodings of connectivity
- Try symmetry-breaking
- Possibly benchmark a second solver (like cvc5) using SMT-LIB output

These are the things that actually produce performance differences.

If you want, I can also:

- Suggest concrete redundant constraints for your current encoding
- Look at your QF_BV code to see if it can be improved
- Help you structure the "Why QF_LIA? Why not BV?" section for your paper

prompt A colleague mentioned possibly trying a fully Boolean version, I think i might try that. Would something like this even make sense in Z3?

answer Yes — a fully Boolean / SAT-style encoding can absolutely make sense in Z3, and it's a legit alternative model to compare against QF_LIA/QF_BV.

In practice it often ends up being: QF_BOOL (pure propositional) or "mostly Boolean + pseudo-Boolean constraints" (which Z3 still handles well).

When a fully Boolean encoding makes sense

It makes sense if you replace "cell value is an Int in [1..n]" with Boolean indicator variables, e.g.:

- isBlack[r,c] : Bool
- hasVal[r,c,v] : Bool meaning "cell (r,c) is value v (and white)"

Then you add constraints like:

- Exactly-one value per white cell
- Row/column uniqueness for each v
- No adjacent blacks
- Connectivity of whites (the hardest part)

This is basically the classic CSP $\rightarrow$ SAT compilation.

Pros

- All constraints become simple clauses / cardinality constraints.
- Uniqueness becomes very natural with "at most one" constraints.
- Can scale surprisingly well because SAT engines are extremely optimized.

Cons

- Variable blow-up: $O(n^3)$ booleans for hasVal ($n \times n$ cells $\times n$ values).
- Connectivity is non-trivial and can dominate the formula size.

For typical puzzle sizes, it can still be very workable.

What about connectivity?

This is the make-or-break point.

Common Boolean-friendly ways:

1. Layered reachability (BFS layers)
   - Boolean reach[k][r][c] = cell is reachable in $\leq k$ steps from a chosen root.
   - Add propagation constraints from layer k to k+1 via neighbors.
   - Enforce every white cell is reachable by some layer.
   - This is pure Boolean but can be large: k up to n^2.
2. Spanning tree / parent pointers
   - Give each white cell (except root) exactly one parent among its neighbors.
   - Add acyclicity / ranking constraints (often needs integers or bitvectors for ranks unless you do more Boolean machinery).
   - Without ranks, cycles are hard to forbid purely propositionally.
3. Flow encoding
   - Send flow from root through edges to cover all white cells.
   - Usually needs integers (even small bounded) or pseudo-Boolean sums.
   - If you allow a tiny bit of arithmetic, this can be very strong.

If you insist on pure Bool, the BFS-layer approach is the most straightforward.

"Fully Boolean" in Z3 specifically

Z3 is perfectly happy as a SAT solver when you stay in Bool land. Two practical notes:

- If you use lots of "exactly-one / at-most-one" constraints, encode them smartly (sequential counters / cardinality networks) instead of naïve pairwise (a or b) for every pair. Pairwise is easy but can explode in clauses.

- Z3's native handling of pseudo-Boolean (PbLe, PbEq) can be quite decent. So even if you're "fully Boolean", letting Z3 manage cardinalities through PB constraints is often a win and still feels SAT-like.

What you'll learn (and why it's worth trying)
Trying this gives you a clean story in your results:

- QF_LIA: compact model, relies on arithmetic reasoning.
- SAT/QF_BOOL: larger model, but potentially faster due to SAT efficiency.
- The key differentiator becomes connectivity encoding and cardinality encoding.

Even if it's slower, it's still informative (like your BV experiment): it tells you whether arithmetic reasoning or SAT-style search is better suited for this puzzle family.

Recommendation

If you do this, don't try to go "perfectly pure" at all costs. A very reasonable "Boolean" variant is:

- Booleans for values/blackness
- Pseudo-Boolean cardinality constraints for "exactly one" and row/col uniqueness
- Connectivity via either BFS layers (pure Bool) or a small bounded-int flow (still "mostly Boolean", but often much stronger)

That will likely give you the best comparison.

If you paste how you currently encode connectivity in QF_LIA, I can suggest the closest Boolean/SAT-style equivalent that won't explode unnecessarily.

conclusion As a result of this second conversation, we decided to not look for more theories to explore aside from bitvectors and booleans. Overal, ChatGPT helped us implement the bitvector encoding. It also provided possible theories to further look into, but also warned us to not expect great results.

E.2 Conversation about puzzle difficulty

We consulted ChatGPT during the process of formulating a method to determine difficulty using the Z3 solver. We had already experimented and researched Z3 statistics at this point, but were not able to find a way to use these for filtering puzzles. The goal of this conversation was to find a more robust measure and experiment with different approaches of filtering puzzles using statistics.

prompt am looking for puzzles that are hard to solve for my Z3 solver. Currently i am using runtime and conflicts as my statistics to find these. I chose conflicts as i thought that it would be a good representation of the size of the search space, and because there just isnt too much documentation on what the z3 statsitics mean. What do you think? And what would you think is the best approach to tackle this? below is the code i currently have to gather the statistics of the solver. I also have a function to gather some properties such as number of pairs or triple pairs from the puzzles directly, i could use this to examine puzzles that have been marked as an outlier by this function. [code snippet]

answer Conflicts can be a useful "difficulty signal", but in Z3 it's also a bit of a blunt instrument:

- "conflicts" is not a pure search-space size proxy. It's mainly counting how often the solver derives a contradiction and learns from it. That number depends heavily on heuristics, preprocessing, restart policy, learned clause quality, and which engine is active (SAT vs theory reasoning). Two puzzles with the same "combinatorial hardness" can show very different conflicts just because the solver took different routes.
- Also, depending on your encoding, the top-level "conflicts" may not be the one you think you're measuring (Z3 exposes several SAT-specific counters).

What I'd use instead (and alongside conflicts) If you're trying to find "hard for my encoding + Z3" instances, measure a small bundle of stats and treat them as a profile:
SAT/CDCL-ish effort

- decisions (or sat_decisions if available): branching steps
- conflicts / sat_conflicts: contradiction+learning events
- propagations / sat_propagations: unit propagation volume
- restarts (if exposed): how turbulent the search was

Theory / arithmetic load (if you're in QF_LIA etc.)

- anything like theory-specific counters if present (varies a lot by version/build)
- wall time + decisions often tells you if the theory solver is the bottleneck (high time with relatively low decisions can indicate expensive theory reasoning)

Resource-style metrics

- rlimit-count (if you use it): "work done" in Z3's internal resource units (often more stable than time)
- memory (sometimes exposed), or at least track Python RSS externally if you can

Practical "best approach" to find genuinely hard puzzles 1) Make runs comparable (otherwise outliers are noisy)

- Fix a seed: s.set("random_seed", k) (and keep k recorded per run).
- Disable/standardize parallelism if applicable.
- Run $\geq$ 5–10 repeats per puzzle (especially if you're going to use z-scores).

Then use median time/conflicts (or trimmed mean) instead of mean; it's much more robust.

2) Don't z-score raw time/conflicts; use robust or log scaling Time and conflicts are usually heavy-tailed. Your current z-score approach is fine as a first pass, but you'll get fewer false "hard" flags if you do either:

- z-score on log(elapsed) and log(conflicts+1), or
- use a robust outlier score like MAD (median absolute deviation)

3) Normalize within (solver, size) like you do — but also rank For "hard benchmark selection", I'd honestly do something simple and stable:

- For each (solver, size), compute ranks for time and conflicts
- Define difficulty_score = rank_time + rank_conflicts (or weighted) This avoids assuming normality and makes your "hard set" stable.

4) Use puzzle features after you've found hard ones (exactly what you suggested) Your idea to compute puzzle properties (pairs, triplets, etc.) is the right next step:

- First: identify "hard" puzzles by runtime/effort stats
- Second: correlate hardness with features (pairs/triples, duplicate density per row/
- col, symmetry, number of forced blacks, etc.) Third: generate more by mutating puzzles toward those features

That will give you a story like: "hard instances tend to have X structure; adding constraint Y reduces decisions/conflicts particularly on those."

Small improvements to your current function (without changing the spirit)

- Use median instead of mean for elapsed and conflicts aggregation across runs.
- If runs < 3, don't label anything an outlier (too unstable).
- Consider printing which statistic keys exist for a run, because Z3 stats differ a lot across settings; e.g. dump sorted(stats.keys()) once per solver/config so you can decide whether you should switch to sat_conflicts/sat_decisions etc.

If your goal is "hard puzzles for my solver", the strongest method Once you have a difficulty score, you can actively search for hard instances:

- start from random valid puzzles
- apply small mutations (swap numbers, tweak a row/col, regenerate with constraints)
- keep the mutation if difficulty_score increases
- repeat (hill-climb), or do a small genetic algorithm

That tends to find nastier instances than passive sampling, and it gives you a reproducible benchmark generator.

If you paste a sample of rs["statistics"] (just one dict) from a run, I can tell you which exact counters you should prioritize for your Z3 build/encoding (e.g., whether you're actually seeing SAT-level sat_* stats or mainly theory ones).

prompt lest move on to the next part of extracting the appropriate statistics and using them properly. Ive done run_time and conflicts, but you didnt really agree. Id like your opinion on what you think the statistics mean but id also like to know on what sources you are basing this

answer Great, this is exactly the right moment in your project to be critical about what the numbers actually mean. I'll structure this in three layers:

- What Z3 statistics conceptually measure (with sources)
- Which statistics are meaningful for "difficulty" and why
- How to use them properly in your analysis

I'll be explicit about where conflicts do and do not make sense.

1. What Z3 statistics actually mean (with sources) Conflicts What they are In Z3 (and CDCL-based solvers generally), a conflict is detected when the current partial assignment violates a clause or theory constraint, triggering:

- backtracking
- clause learning
- possibly a restart

What conflicts measure

- How often the solver reaches an inconsistent state that requires learning.
- Roughly: how turbulent the search is, not how large the search space is.

What conflicts do not measure

- They do not directly measure:
  - number of explored assignments
  - theoretical search-space size
  - puzzle difficulty independent of solver heuristics

Sources

- Eén & Sörensson, "An Extensible SAT-solver", SAT 2003 (MiniSAT) → canonical explanation of conflicts and clause learning.
- Biere et al., Handbook of Satisfiability, 2009 → Chapter on CDCL solvers.
- de Moura & Bjørner, "Z3: An Efficient SMT Solver", TACAS 2008 → explains how SAT conflicts interact with theory reasoning in Z3.

Decisions What they are

- A decision is a branching choice: the solver assigns a value to a variable not forced by propagation.

Why decisions matter

- Decisions are the closest analogue to: "how many branches did the solver actively choose to explore?"

Interpretation

- High decisions → wide or deep search
- Low decisions + high runtime → expensive theory reasoning

Sources

- Same as above; decisions are a core CDCL concept.
- Pipatsrisawat & Darwiche, "On the Power of Clause Learning", SAT 2009

Propagations What they are

- Forced assignments derived by unit clauses or theory propagation.

Interpretation

- High propagations usually correlate with:
  - dense constraint interaction
  - strong constraint propagation
- Propagations alone are not a difficulty measure, but:
  - propagations / decision is very informative

Sources

- Handbook of Satisfiability, CDCL chapters
- Z3 internal documentation (limited, but consistent with SAT literature)

Restarts What they are

- Solver abandons the current search path and restarts with learned clauses.

Interpretation

- Many restarts → solver struggling to find a productive search direction
- Often correlated with hardness for that encoding

Caveat

- Restart policies vary; absolute numbers are less meaningful than relative ones.

rlimit-count (very important but underrated) What it is

- Z3's internal resource counter, incremented for many low-level operations.

Why it's useful

- Much more stable than wall-clock time
- Often correlates better with "actual work" than conflicts alone

Source

- de Moura & Bjørner, TACAS 2008 (mentioned briefly)
- Z3 codebase comments (this is common knowledge among Z3 users, but poorly documented)

2. Which statistics actually make sense for your goal Your stated goal is: "Find puzzles that are hard to solve for my Z3 encoding." That's important — you're not doing theoretical hardness, you're doing solver-relative hardness.

My opinion, clearly stated

- Conflicts alone are not sufficient
- Conflicts + decisions + runtime form a meaningful core
- rlimit-count (if available) is often better than conflicts
- Runtime alone is too noisy without control

A good minimal set If I had to pick four stats only:

1. elapsed time → user-visible difficulty
2. decisions → branching effort
3. conflicts → learning turbulence
4. rlimit-count → total solver effort

If you can only pick three: time + decisions + conflicts

3. How to use these statistics properly 3.1 Don't interpret raw values in isolation

Bad: "Puzzle A has more conflicts, therefore it's harder."

Better: "Puzzle A has significantly more decisions and conflicts than the median puzzle of the same size."

Best: "Puzzle A lies in the 95th percentile for decisions and conflicts within its size class."

3.2 Use ratios (this is important and often missed) These derived metrics are much more informative:

- conflicts / decisions
- propagations / decision
- time / decision

Interpretation examples:

- High decisions, low conflicts → wide but consistent search
- Low decisions, high time → theory-heavy reasoning
- High conflicts / decision → poor heuristic guidance

These ratios let you explain why a puzzle is hard, not just that it is.

3.3 Difficulty is multi-dimensional — embrace that

Instead of asking: "Which puzzle is hardest?"

Ask:

- hardest by time?
- hardest by search effort?
- hardest by theory reasoning?

In your context, that's a strength, not a weakness.

3.4 How I'd define "hard puzzle" in your project Very defensible definition: A puzzle is considered hard for a solver if it is an outlier with respect to solver effort (runtime, decisions, and conflicts) within its size class, averaged over multiple seeded runs. That sentence is thesis-ready.

4. How this ties back to your puzzle-structure analysis Once you identify hard puzzles:

- correlate difficulty with:
  - number of duplicate pairs
  - number of triplets
  - symmetry
  - forced black regions
- explain why certain patterns cause:
  - more branching
  - more conflicts
  - slower propagation

This is where your work becomes interesting, not just empirical.

5. Bottom line (very explicit)

- You were right to look at conflicts.
- You were wrong to trust them alone.
- Your current setup is very close to something publishable.

- Add decisions + (optionally) rlimit-count, and treat difficulty as a profile, not a scalar.

If you want, next we can:

- inspect one real statistics() dict and decide exactly which keys to rely on, or
- design a single composite difficulty score that you can justify academically.

prompt I can use all 5, time, decisisoin, conflicts, propagations and rmlimit, that no issue

answer Perfect — that's actually an excellent position to be in. With those five you can do something that's both methodologically sound and easy to justify in writing.

I'll now be very concrete and give you:

- How to interpret each of the 5 (precisely, in your context)
- Which combinations matter most
- How to turn them into a usable "difficulty" analysis
- How to present this cleanly in a thesis/poster

No hand-waving, and everything stays within SMT/SAT literature norms.

[Another repeat of interpretations]

Which combinations actually matter Core triplet (must-have)

- decisions
- conflicts
- rlimit-count

These tell you:

- how wide the search was,
- how turbulent it was,
- how expensive it was overall.

Diagnostic ratios (extremely useful) These are where real insight comes from:

- conflicts / decisions → heuristic quality
- propagations / decisions → propagation strength
- rlimit / decisions → cost per branch
- time / rlimit → machine-level overhead

You do not need all of them in the paper — but use them in analysis.

3. How to define and use "difficulty" properly Step 1: Normalize within (solver, size) You already do this — good. For each metric m: $z\_m = (m - mean(m))/std(m)$ (or median/MAD if you want to be robust).

Step 2: Treat difficulty as a vector Instead of collapsing immediately: $D(puzzle) = (z\_time, z\_decisions, z\_conflicts, z\_prop, z\_rlimit)$ This lets you say things like:

- "hard due to search"
- "hard due to propagation"
- "hard due to theory overhead"

This is far more defensible than a single scalar.

Step 3 (optional but powerful): Composite score If you do want one number, use something like: $difficulty = z_decisions + z_conflicts + z_rlimit$ Why this works:

- decisions → search size
- conflicts → learning effort
- rlimit → total work

I would not include time in the composite (use it separately).

description ChatGPT followed by giving a code implementation that provided outliers based on the z-scores of the five statistics. We did not implement this code yet, as we found a mistake when analysing the implementation.

prompt This is nice, but still uses the old logic of if any individual z > 2, we mark it. You said that looking at individual stats was not worth it and instead we should look at combined statistics to find the actual difficulty our solver had

answer Yep — you're right to call that out. "any z > 2" finds spiky puzzles, not necessarily "overall hard". What we want is a combined effort score that only flags puzzles that are consistently heavy across multiple effort dimensions.

Here are two solid ways to combine the five stats, and then I'll show you how to implement it cleanly in your function.

A good combined difficulty score Option A (recommended): weighted Euclidean norm of positive z-scores

This treats "hardness" as "far out in effort-space", but ignores "easier-than-average" negatives.

$$D = \sqrt{\sum_i w_i \cdot max(0, z_i)^2}$$

Why this works:

- A single spiky metric doesn't dominate as much as with any z ¿ threshold
- Being moderately high on several metrics pushes you over the threshold
- Using only positive z avoids "hard in one metric but weirdly low in another" cancelling out

My suggested weights:

- rlimit: 1.0 (best proxy for total work)
- decisions: 1.0 (search effort)
- conflicts: 0.7 (learning turbulence; helpful but less stable)
- propagations: 0.5 (often correlates with encoding density)
- time: 0.5 (noisy; keep it but don't let it dominate)

You can tweak these, but this default is defensible.

Option B: sum of positive z-scores (simpler)

$$D = \sum_i w \cdot max(0, z_i)$$

Simpler, but more sensitive to "many slightly-above-average" metrics.

What threshold to use? For the Euclidean version (Option A), a good starting point is:

- difficulty ≥ 3.0 → “strong outlier”
- difficulty ≥ 2.5 → “possible hard”

Reason: if two metrics are at z=2, norm ≈ sqrt(4+4)=2.83. So 3.0 roughly means “at least two big signals, or three moderate ones”.

[ChatGPT followed by giving an implementation into our code]

discussion This conversation was followed by an iterative process of implementing the hard score using 5 statistics. After this, we evaluated the weights and statistics used in this score by manually altering them and reviewing the resulting filtered sets. We discussed some of the results with ChatGPT in order to get more clarity on how to interpret the results as well as to find better methods for finding the correct set of weights to use.

prompt ive noticed that the ratios involving rlimits are not useful, since rlimit is almost always incredibly high. Also, the ratios dont seem to be hat crazy for the most difficult puzzles, for instance, the puzzle with hard_score 6.85 has the ratios: z_C/D: 1.03, z_P/D: -1.52, z_R/D: -1.35 z_T/R= -0.27

answer This is actually a very good observation — and it tells us something real about your encoding and Z3’s behavior, not that the analysis is “wrong”.

Let me unpack what this means, why it’s expected, and how I’d adjust the pipeline accordingly.

1. Why rlimit ratios are useless in your case You’re seeing:

- rlimit is huge for almost everything
- rlimit/decision barely discriminates
- z_R/D and z_T/R cluster near 0 or negative

This typically means:

Interpretation

- Most of Z3’s internal work is front-loaded and global:
  - simplification
  - preprocessing
  - theory bookkeeping
- That cost dominates and scales roughly with puzzle size, not search behavior.

So: rlimit is acting like a baseline tax, not a search signal.

This is very common for SMT encodings with:

- heavy arithmetic constraints,
- large numbers of static constraints,
- connectivity constraints (which Hitori has).

Conclusion

- Dropping rlimit ratios is the right call
- Keeping raw rlimit as part of the effort score is still fine

2. Why ratios aren’t “crazy” for the hardest puzzles Your example actually paints a very coherent picture: What this says about that puzzle

- High hard_score (6.85) → very high absolute effort (decisions + conflicts + time)
- conflicts/decision only mildly high (1.03) → search is not chaotic
- propagations/decision very low (-1.52) → propagation is weak
- rlimit/decision low → per-branch work is not expensive

Interpretation (important) This puzzle is hard because the solver has to explore a very large search space, not because it’s making bad decisions.

In other words:

- many decisions,
- each decision is relatively “clean”,
- but constraints don’t propagate much,
- so the solver must branch deeply.

That’s exactly the kind of instance that defeats SMT encodings of Hitori.

So the ratios are not supposed to be extreme for hard puzzles — they explain why the puzzle is hard.

3. This tells you something interesting for your thesis You now have evidence for a strong qualitative claim: The hardest puzzles for the SMT encoding are not characterized by high conflict density or expensive propagation, but by weak propagation that forces extensive branching.

That’s a much stronger insight than: “some puzzles take longer”.

You could even contrast:

- hard puzzles: high decisions, low prop/dec
- easy puzzles: low decisions, high prop/dec

4. How I’d refine the analysis pipeline (based on your observation) Drop these ratios entirely

- rlimit / decision
- time / rlimit

They’re not informative for your encoding.

Keep these two

- conflicts / decisions
- propagations / decisions

And use them only diagnostically, not for hardness scoring.

5. How I’d present this in code (small adjustment)

When printing outliers, emphasize interpretation, e.g.:

```
hard_score=6.85
z_decisions=3.12  z_conflicts=2.94  z_propagations=-0.88

conf/dec z=+1.03   → mildly turbulent search
prop/dec z=-1.52   → weak propagation (branch-heavy)
```

This makes the result explainable, not just numeric.

6. What you should not worry about

- Ratios not being extreme for the hardest puzzles → that’s expected when hardness comes from breadth, not chaos
- rlimit ratios being useless → very common; you tested and validated that empirically

That's good experimental practice.

7. One last conceptual takeaway (worth remembering) Hardness can come from three very different causes:

1. Bad guidance (high conflicts/decision)
2. Expensive reasoning (high rlimit/decision)
3. Weak propagation (high decisions, low prop/decision)

Your solver is hitting (3).

That's a clean, defensible conclusion.

The conversation continues with evaluating results and I attempt to add the `bool_vars` statistic to the hard-score evaluation, after a few prompts, this attempt eventually concludes with the following prompt and answer.

prompt Since we dont know much about bool_vars and given that it might not be that useful after all, should we really take it into account for our score?

answer Short answer: no — you should not include bool_vars directly in the hard score. Long answer: you use bool_vars to explain hardness, not to define it.

Let me explain why this distinction matters, and what the clean design is.

1. What the hard score is supposed to mean Your hard score currently measures: How much work the solver actually did on this instance.

That's why it's built from:

- decisions (branching effort)
- conflicts (learning effort)
- propagations (interaction volume)
- rlimit / log-time (total internal cost)

These are dynamic effort measures.

2. What bool_vars measures (and does not) bool_vars measures:

- the static size of the Boolean abstraction

It says nothing about:

- how many branches were taken,
- how many conflicts occurred,
- how long the solver ran.

So if you include bool_vars in the hard score, you'd be mixing:

- "how big the model is" with
- "how hard the solver worked"

That muddies the interpretation.

3. Why including bool_vars in the hard score is wrong Let's look at the failure mode.

Imagine two puzzles of the same size:

Puzzle	bool_vars	decisions	hard?
A	100k	5k	medium
B	200k	5k	same

If you include bool_vars:

- Puzzle B looks "harder"
- But the solver did the same work

That's misleading.

The solver did not struggle more on B — the model was just larger.

4. Where bool_vars does belong As a normalizer or explanatory axis Use bool_vars in ratios like:

- decisions / bool_vars
- propagations / bool_vars
- conflicts / bool_vars

These answer: "How much effort was required per unit of model size?" That's exactly what you want for interpretation.

5. How to talk about this in your report

A clean separation:

- Hard score → solver effort
- Ratios involving bool_vars → constraint effectiveness

You can phrase it like this: The combined difficulty score captures solver effort independent of encoding size. Structural measures such as the number of Boolean variables are used only in normalized ratios to interpret solver behavior and constraint effectiveness. That's a very defensible methodological choice.

6. When would it make sense to include bool_vars? Only in a different kind of score, e.g.:

- an encoding complexity score, or
- a model size comparison across encodings.

But that's not what your hard score is for.

Bottom line

- Do not include bool_vars in the hard score
- Do include decisions / bool_vars and similar ratios as diagnostics

Your current separation is exactly right.

prompt We currently use the following weights in our function. I am going to experiment a bit with these to find the best fit. Can you suggest some ways to do this?

answer Good idea. At this point you've earned the right to tune the weights, because you now understand what each metric is actually measuring and what your data is telling you.

I'll structure this as:

1. What the weights are supposed to encode
2. What your observations imply for re-weighting
3. Two concrete alternative weight sets (with justification)
4. How to validate that the weights make sense (without overfitting)

No hand-waving.

[More repeat on what the weights mean and my observersations]

3. Recommended weight sets Option A: "Solver-effort–centric" (my top recommendation) This reflects exactly what your results show.

- "decisions": 1.2, # primary hardness signal
- "rlimit_count": 1.0, # total effort
- "conflicts": 0.6, # turbulence, secondary

- ”propagations”: 0.4, # interaction volume, noisy
- ”log_time”: 0.3 # visibility only

Why this is good

- Hard puzzles remain hard
- Branch-heavy puzzles are emphasized
- Time anomalies don’t dominate
- Ratios stay interpretable

If you use this set, your hard score becomes: “How much branch-heavy effort the solver expended.” Which fits your conclusions perfectly.

Option B: “Time-aligned” (only if you care about wall-clock) Use this only if you want hard score ≈ runtime.

- ”decisions”: 1.0,
- ”rlimit_count”: 0.8,
- ”conflicts”: 0.6,
- ”propagations”: 0.4,
- ”log_time”: 0.8

This will:

- pull time-hard puzzles upward,
- blur the distinction between effort-hard and time-hard.

I would not recommend this for your thesis, but it’s a useful comparison.

Option C: Minimalist (very defensible) If you want to be extremely conservative:

- ”decisions”: 1.0,
- ”rlimit_count”: 0.8,
- ”conflicts”: 0.5,
- ”propagations”: 0.3
- # no time

Time is reported separately. This is hard to criticize academically.

4. How to validate the weights (without cheating) You should not “optimize” weights numerically (that’s overfitting). Instead, do sanity validation: Check 1: Stability

- Do the same puzzles appear in the top 5–10% across:
  - Option A
  - Option C
- If yes → weights are robust

Check 2: Interpretability For a few top puzzles, ask:

- Is hardness explained by:
  - high decisions?
  - weak propagation?
- Or is it driven by time noise?

If a weight set makes “nonsense-hard” puzzles appear, reject it.

Check 3: Correlation table (very easy, very strong) Compute Spearman correlation between:

- hard_score
- decisions
- runtime

You want:

- high corr with decisions
- moderate corr with runtime
- not perfect corr with either

That shows your score captures effort, not just time.

5. How to justify weights in writing (important) You do not need to claim optimality.

Just say something like: Weights were chosen to emphasize stable solver effort indicators (decisions and rlimit-count), while down-weighting metrics that are either noisy (runtime) or ambiguous in interpretation (propagations). Alternative weightings were evaluated and produced qualitatively similar outlier sets. That’s exactly the right level of rigor.

6. One last important rule (keep this) If changing a weight slightly changes which puzzles are hard, your metric is unstable. If changing a weight slightly only changes ordering within the hard set, your metric is good.

My clear recommendation Use Option A, and keep time separate in plots/tables.

prompt Im going to add memory usage as well, since time doesnt really shows up in difficulty, maybe that will.

answer For Z3’s two memory-related statistics fields:

- memory = current allocated heap size (roughly “how much Z3 thinks it has allocated right now”), reported in MiB. This comes from memory::get_allocation_size() and is then scaled by 1024*1024.
- max-memory = peak (maximum) heap usage seen so far during the run, also in MiB, from memory::get_max_used_memory().

And separately (easy to confuse with the above): max_memory is also a parameter you can set to put a (soft) memory cap on Z3, expressed in MB/MiB depending on docs phrasing.

Which one should you use for “difficulty”? Use max-memory (peak) as your main memory signal.

Reason: “difficulty” is usually about how bad it got at any point. Peak memory captures clause database blow-ups / big intermediate states; end-of-run memory can be lower (because some things are freed) or just not representative.

So:

- Keep collecting both if you want.
- Use max-memory in your hard/easy score, and either ignore memory or treat it as a secondary diagnostic.

[Implemntation of using the statistics in my code]

One more practical note These memory counters are approximations (not updated on every allocation), so don’t overfit weights to tiny differences.

If you want, paste one sample s.statistics() dump (just the printed stats dict) and I’ll tell you exactly which key spellings your build is using so you don’t chase another “always the same” bug.

prompt this also makes me think about including time or log_time in the weights. Is a hard puzzle purely effort or is it runtime based, currently we are looking a bit at both, but mostly effort. I think i want to lean on effort the most and then take a look at the impact in time afterwards, same goes for memory

answer That's a clean and defensible framing, and it matches what your data has been telling you.

Define "hard" as effort-first In your project, "hard for my solver" is best defined as:

high internal solver effort (branching + learning + propagation volume), and then you analyze runtime and memory as outcomes of that effort.

That avoids the trap where noise in wall-clock time (or OS-/machine effects) changes what counts as "hard".

Why this is methodologically strong

- Effort stats (decisions/conflicts/propagations/rlimit) are mostly solver-internal and more reproducible.
- Runtime is a dependent variable: it's affected by hardware, Python overhead, scheduling, etc.
- Peak memory is also an outcome: it can explode due to clause learning or preprocessing even if time stays modest (and vice versa).

So: effort → (time, memory) as downstream effects is the right causal story.

What I'd do with your weights 1) Remove time from the hard score (or keep it tiny) If you want "effort-hard", don't let time define hardness.

Option A (pure effort):

- "decisions": 1.0,
- "conflicts": 0.8,
- "propagations": 0.6,
- "rlimit_count": 0.3,

Option B (still acknowledge time, but not define hardness):

- "decisions": 1.0,
- "conflicts": 0.8,
- "propagations": 0.6,
- "rlimit_count": 0.3,
- "log_time": 0.1, # "tie-breaker" level

Given your observation that time doesn't discriminate much, I'd pick Option A or B with 0.1.

2) Keep memory OUT of the score; analyze it afterwards Same logic as time. Make it a reported field, not part of the definition.

How to "look at impact afterwards" (simple + strong) Once you have the effort-hard set (say top K per size), do:

- Compare median runtime and median peak memory of:
  - effort-hard puzzles
  - "normal" puzzles (middle 50
  - effort-easy puzzles (bottom K)
- Or compute correlations per size bucket:
  - corr(hard_score_effort, log_time)
  - corr(hard_score_effort, log_max_mem)

This gives you a solid results narrative:

"Effort-hard puzzles generally correlate with higher runtime and peak memory, but the relationship is imperfect; some puzzles are effort-hard but time-light."

That's exactly the kind of insight you were already seeing.

Implementation suggestion (tiny change, big clarity)

Store three scores instead of one:

- effort_score (your hard_score with no time/mem)
- z_log_time (reported, not scored)
- z_log_max_mem (reported, not scored)

Then you can filter by effort score and sort within that set by time or memory depending on what you want to inspect.

Thesis-ready wording you can use Difficulty was defined primarily in terms of solver effort (decisions, conflicts, propagations, and resource-limit steps). Runtime and peak memory were analyzed separately as outcomes of solver effort, rather than as part of the difficulty definition, due to their sensitivity to external factors and implementation overhead.

That's rigorous and hard to argue with.

prompt Ive analyzed runtime, memory, conflicts, decisions, propagations and rlimit for the set of difficult puzzles and an equally sized set of non-hard puzzles. It seems that only the conflicts, decisions and propagations are noticable higher for our 'harder' puzzles. rlimit, memory and solving time is pretty much equal (¡ 2% difference). This is not what i expected and makes me doubt on the method we use to determine hard puzzles. What do i do? do i spent more time trying to make a new way of finding hard puzzles or do i just report this failure and move on?

answer This is a very important result, and the fact that it makes you uncomfortable is actually a good sign. Short answer first, then the reasoning:

You should not throw this away or redesign everything. You should report this result clearly, explain why it happens, and move on.

This is not a failure of your method. It's a finding about the solver and the problem.

1. Why this result is not actually contradictory You found:

- Hard puzzles (by your metric) have:
  - much higher decisions
  - much higher conflicts
  - much higher propagations
- But almost identical:
  - runtime
  - peak memory
  - rlimit

This means: The solver is doing more search work, but that extra work is cheap.

That is exactly the pattern you were already seeing earlier with:

- low or negative z(rlimit / decisions),
- low z(time),

- branch-heavy but fast reasoning.

So this result is consistent with everything you observed before — it just becomes clearer when you aggregate.

2. What this tells you about Hitori + your encoding This strongly suggests:

- The hard instances are combinatorially harder, not computationally heavier.
- Z3 handles this kind of Boolean-heavy branching very efficiently.
- Extra decisions and conflicts do not translate into:
  - expensive theory reasoning,
  - large clause databases,
  - or measurable time/memory growth.

In other words: Harder puzzles require more branching, but each branch is cheap and quickly pruned.

That is a property of the encoding and solver, not a flaw in your metric.

3. Why redefining "hard" in terms of time or memory would be worse If you now switch to:

- runtime-based hardness, or
- memory-based hardness,

you would actually be measuring noise:

- OS scheduling,
- Python overhead,
- cache effects,
- machine variability.

You would lose the solver-internal signal that you do see clearly (decisions/conflicts/propagations).

So changing the definition now would weaken the methodology, not strengthen it.

4. How to frame this correctly (this is the key) The right interpretation is:

"Difficulty" as measured by solver effort does not necessarily translate to higher runtime or memory usage for this problem class.

That is a valid and interesting conclusion.

You're not claiming:

- that your metric predicts runtime,
- or that "hard" means "slow".

You are claiming:

- that some puzzles induce significantly more solver-internal work,
- even when this does not manifest in wall-clock performance.

That's a subtle but solid point.

5. What not to do Don't:

- spend weeks inventing a new hardness metric,
- force time/memory into the definition,
- treat this as an error unless you find a clear bug.

You already validated:

- the stats,
- the z-scoring,
- the grouping,
- the comparisons.

Nothing is obviously broken.

6. What you should do instead (concrete steps) 1. Report this explicitly

In your results section, say something like: While hard puzzles exhibit substantially higher numbers of decisions, conflicts, and propagations, no significant differences were observed in runtime, peak memory usage, or rlimit-count.

2. Explain why this happens You already have the explanation:

- CDCL with non-chronological backtracking
- efficient pruning
- cheap Boolean reasoning
- little theory work

3. Turn doubt into insight This becomes a discussion point, not a failure:

- "solver effort $\neq$ runtime"
- "branch-heavy $\neq$ slow"
- "difficulty is multi-dimensional"

That's exactly the kind of nuance examiners like.

7. If you want one small sanity check (optional) Do one extra check, then stop:

- Plot or compute correlation:
  - effort score vs runtime
  - effort score vs memory

If correlation ≈ 0, that confirms your conclusion quantitatively.

Do not redesign the method unless this check contradicts your observations.

8. How this actually strengthens your project Many student projects end with: "Hard puzzles take longer."

Yours can end with: "Harder puzzles induce more solver effort, but modern SMT solvers can absorb this effort without measurable runtime or memory impact for this problem class."

That's more interesting.

Bottom line (very important)

- You did not fail to find hard puzzles.
- You found that hardness manifests internally, not externally.

That's a real result. Report it, explain it, and move on.

conclusion This conversation helped us refine our method of defining hard puzzles using an effort-based score and helped us implement this method in code. Using the responses from ChatGPT and additional sources from StackOverflow we were able to give meaning to statistics and determine the weights needed to form this score. The weights were evaluated repeatedly using the methods as suggested by ChatGPT and with further discussion and manual evaluation we were able to find a suitable set to use for our experiment.

E.3 Conversations about data uniformality and statistics

During our experimental evaluation, we required statistical methods in order to measure significance of our results. Due to the limited knowledge on statistics by us, we consulted ChatGPT to help find the correct methods to use.

prompt I am going to working on my results. The first research question I will be tackling is this one: How does the size of the SMT encoding and Z3's solving performance scale with the increasing Hitori puzzle size? I have multiple encodings that I want to use as a base, such as a Bitvector variant, a LIA variant and some variants with small variations. In total these are 7 variantions. 3 are actual different theories used, 3 are variants of the LIA encoding and the final one is a 'lazy' encoding that uses a completely different strategy. I have the runtime available, as well as a bunch of stats from Z3 which i will likely not use. I also have the encoding size in terms of SMT variables and assertions, but i still need a proper explanation as to why this is sufficient as encoding size. I want to use proper statistical methods to answer this question, but dont really know where to start

answer You're actually in a very good position already — what you're missing is not data, but structure. Below is a concrete, defensible way to answer this research question using standard empirical methods without overclaiming, and in a way that examiners in constraint solving will recognize as sound.

1. Precisely restate the research question (this matters) You are not asking whether one encoding is "better" in absolute terms. You are asking:

How does encoding size and solver performance scale as a function of puzzle size, and how do modelling choices affect that scaling?

This framing immediately justifies:

- asymptotic analysis
- regression on log–log plots
- relative comparison between encodings

You are not required to explain Z3 internals.

2. What you should measure (keep it minimal) Independent variable

- Puzzle size n (grid is $n \times n$)

Dependent variables

You only need two, both defensible:

1. Encoding size
   - Number of SMT variables
   - Number of assertions
2. Solver performance
   - Runtime (elapsed wall-clock time)

Everything else (conflicts, propagations, decisions, rlimit) can be mentioned as available but unused. That restraint is a strength, not a weakness.

3. Why "#variables + #assertions" is a valid encoding size measure This is the key explanation you said you're missing. You can argue this cleanly:

Core argument (this is what you write)

- SMT solvers operate on a finite formula composed of:
  - declared symbols (variables, functions)
  - asserted constraints
- Regardless of the underlying theory (LIA, BV, Bool), all solving effort originates from these two sources
- Z3 internally normalises theory constraints into Boolean structure via its SAT layer
- Therefore:
  - more variables $\rightarrow$ larger search space
  - more assertions $\rightarrow$ more clauses after preprocessing

You are not claiming this is a perfect measure — only that it is:

- theory-agnostic
- solver-independent
- monotonic with problem complexity

That's exactly the right level.

One-sentence justification you can reuse We therefore use the number of declared SMT variables and assertions as a proxy for encoding size, as these quantities directly determine the size of the logical formula presented to the solver, independent of the theory used.

That's enough. You don't need to cite Z3 internals.

4. Statistical method: what to actually do Step 1: Aggregate properly For each encoding variant and puzzle size:

- run multiple instances (or puzzles)
- report median runtime
  - median is preferable due to heavy-tailed runtimes
- optionally show IQR (interquartile range)

Avoid means unless you justify them.

Step 2: Scaling analysis (this is the core)

You expect polynomial or exponential growth. To detect that:

Use log–log regression For each encoding:

- plot:
  - $log(n)$ vs $log(encodingsize)$
  - $log(n)$ vs $log(runtime)$

Then fit:

$$log(y) = a\ log(n) + b$$

Interpretation:

- slope a = scaling exponent
- higher a = worse scalability

This avoids overfitting and is standard in solver evaluation papers.

Step 3: Compare encodings by slope, not raw time This is crucial:

- Raw runtime comparisons are noisy and hardware-dependent
- Scaling slope is structural

So you say things like:

- "The BV encoding exhibits a steeper scaling slope than LIA"
- "LIA variants differ mainly in constant factors, not asymptotic behaviour"

This is exactly aligned with SMT literature.

5. How to handle the 7 encodings cleanly Group them explicitly:

1. Different theories
   - LIA
   - Bitvectors
   - Boolean / lazy
2. Same theory, modelling variations
   - LIA baseline
   - LIA + variants

This allows you to make two separate claims:

- Theory choice affects scaling behaviour
- Small modelling variations mainly affect constants

That distinction is very strong academically.

6. What not to do (important) Do not:

- claim causality ("this constraint causes slowdown")
- overinterpret Z3 statistics
- do hypothesis tests (t-tests, ANOVA) — unnecessary and fragile
- average runtimes across different sizes

Scaling analysis is enough.

7. Suggested subsection structure (you can almost copy this) Results – RQ1: Encoding size and solver scalability

1. Experimental setup
   - encodings
   - puzzle sizes
   - runtime measurement
2. Encoding size growth
   - variables vs size
   - assertions vs size
   - log–log analysis
3. Solver performance scaling
   - runtime vs size
   - regression slopes
4. Comparison across encodings
   - theory-level differences
   - modelling-level differences
5. Discussion
   - implications for SMT modelling
   - limitations (single solver, single hardware)

8. One final reassuring note You do not need "advanced statistics" to make this rigorous. In solver research, clean plots + scaling arguments beat p-values every time.

If you want, next we can:

- draft the exact text for the encoding-size justification
- decide whether to show linear or log plots
- sanity-check your regression interpretation before you write it up

prompt Why use median runtime and not the mean? I was thinking of using the stadard deviation and such to show stability and such

answer Good question — this is exactly the kind of thing examiners do look for. Short answer: solver runtimes are almost never well-behaved random variables, so the mean + standard deviation often misrepresents what is going on.

Let's break this down cleanly and then I'll show you how you can still talk about stability without using the mean.

1. The core issue: solver runtimes are not "normal" Mean and standard deviation implicitly assume something like:

- a symmetric distribution
- light tails
- no extreme outliers

SMT solvers violate all three.

Typical runtime behaviour:

- many "easy" instances → very fast
- a few "hard" instances → orders of magnitude slower
- sometimes timeouts

This produces:

- right-skewed distributions
- heavy tails
- occasional extreme outliers

In that situation:

- the mean is dominated by rare slow runs
- the standard deviation explodes
- neither reflects "typical" solver behaviour

This is well-known in SAT/SMT benchmarking.

2. What the median gives you (and the mean does not) Median runtime answers: "How long does the solver typically take?"

It is:

- robust to outliers
- stable across repetitions
- interpretable per puzzle size

This is why:

- SAT competitions
- SMT-COMP
- MiniZinc Challenge

all rely heavily on median-based metrics or quantiles.

Using the median is not "less statistical" — it's more appropriate.

3. Why standard deviation is especially misleading here Suppose for one encoding and size you get:

Run	Runtime (s)
1	0.8
2	0.9
3	0.8
4	0.9
5	12.3

- Mean $\approx$ 3.14 s
- Std dev $\approx$ very large
- Median = 0.9 s

Which reflects reality better?

- The solver usually takes under a second
- Occasionally it blows up

Mean + std would suggest "unstable and slow", which is misleading.

4. How to still talk about stability (this is the key) You're right that you want to show stability — you just shouldn't use standard deviation of the mean to do it.

Instead, use robust dispersion measures: Recommended options (pick one or two)

1. Interquartile Range (IQR)
   - IQR = Q3 - Q1
   - Shows spread of the middle 50
   - Immune to outliers

   You can say: "Runtime variability is limited, as indicated by a small interquartile range."
2. Median Absolute Deviation (MAD)
   - Less common, but very robust
   - Excellent if you want to be fancy and precise
3. Min–max bands or error bars (carefully)
   - Only if you explicitly say they include outliers
   - Good visually, weaker analytically

5. What about repeated runs on the same instance? This is an important distinction.

Case A: different puzzle instances (most common)

- Runtime variability reflects instance hardness
- Median is the right statistic

Case B: repeated runs on the same instance

- Variability reflects solver nondeterminism
- Here you can talk about stability more explicitly

Even then:

- median + IQR is still preferable
- std dev is only acceptable if you show the distribution is tight

6. How to justify this in one paragraph (you can reuse this) Solver runtimes typically exhibit heavy-tailed distributions, where a small number of hard instances dominate the total runtime. In such settings, the mean and standard deviation are highly sensitive to outliers and do not accurately reflect typical solver behaviour. We therefore report median runtimes, which provide a robust estimate of central tendency, and use dispersion measures such as the interquartile range to assess variability and stability.

That is a textbook-correct justification.

7. If you really want to include standard deviation You can, but do it carefully:

- Only as a secondary statistic
- Explicitly acknowledge its limitations
- Never base conclusions on it

For example: "Although we also report standard deviation for completeness, we base our analysis primarily on median-based statistics due to the skewed nature of runtime distributions."

Bottom line

- Median $\rightarrow$ typical performance
- IQR / MAD $\rightarrow$ stability
- Mean + std $\rightarrow$ misleading for solver runtimes

Using median does not weaken your results — it makes them more credible in the SMT community.

prompt So you assume my data is not uniformally distributed, how do i make sure this is the case? Does visually looking at a QQ graph suffice?

answer This is a very good methodological question — and you're thinking about it in exactly the right way.

Short answer: Yes, visually inspecting distributions (including QQ plots) is sufficient — and in your context, even preferable — but you should be clear about what you're checking and why.

Let's be precise.

1. First correction: it's not "uniform", it's "normal" What we actually care about is whether runtimes are approximately normally distributed, not uniformly distributed.

- Uniform: flat distribution $\rightarrow$ almost never realistic for runtimes
- Normal (Gaussian): symmetric, light tails $\rightarrow$ required for mean + std to be meaningful

So the real question is: Are solver runtimes approximately normal?

2. Do you need to prove non-normality? No. You only need to justify your choice of robust statistics.

In empirical CS and solver research:

- formal normality tests are rare
- visual diagnostics are standard
- prior knowledge of solver behaviour is explicitly acknowledged

Trying to "prove" non-normality with p-values often weakens papers rather than strengthening them.

3. Is a QQ plot sufficient? Yes — if you use it correctly and modestly A QQ plot against a normal distribution is one of the best tools here.

What you're looking for:

- straight line $\rightarrow$ roughly normal
- curvature $\rightarrow$ skewness

- strong deviations in upper tail → heavy-tailed behaviour (very common!)

Solver runtimes almost always show:

- right-tail deviation
- clustering near small values
- a few very large points

That alone is enough to justify median-based statistics. You do not need to show the QQ plot in the paper unless you want to.

4. What about formal tests (Shapiro–Wilk, KS, etc.)? You can mention them, but I would not recommend relying on them. Why:

- with large samples, everything becomes "significantly non-normal"
- with small samples, they lack power
- they produce binary outcomes that add little insight

If you use one:

- treat it as supporting evidence
- never as the main justification

5. The strongest justification: empirical + domain knowledge The best argument combines three elements, not just one.

1. Visual inspection
   - histogram or boxplot
   - QQ plot showing tail deviations
2. Empirical observation
   - large gap between median and max
   - occasional extreme runtimes
   - presence of timeouts (if any)
3. Domain knowledge
   - SMT solvers are known to exhibit heavy-tailed runtime distributions
   - instance hardness varies sharply even at fixed size

This triangulation is far stronger than a single statistical test.

6. What you can safely write (example paragraph) You can write something like: Preliminary inspection of runtime distributions revealed strong right-skewness and heavy-tailed behaviour, with occasional runs taking orders of magnitude longer than the median. QQ plots against a normal distribution showed substantial deviations in the upper tail. This behaviour is consistent with known characteristics of SMT solver runtimes. As a result, mean-based statistics were deemed inappropriate, and median-based measures were used instead.

This is:

- honest
- standard
- methodologically sound

7. Do you need to include the QQ plot in the paper? Not necessary, unless:

- you expect a very statistically-minded reviewer
- or you want to put it in an appendix

Most solver papers do not include distribution diagnostics unless the paper is specifically about runtime variability.

8. Bottom line

- You are not making an assumption out of thin air
- Visual inspection (including QQ plots) is sufficient
- Formal normality tests are optional and often unhelpful
- Combining visuals + empirical behaviour + domain knowledge is best practice

If you want, next we can:

- decide where to put this justification (methods vs results)
- draft a concise "statistical methodology" subsection
- decide whether one illustrative plot in an appendix is worth it

discussion This conversation helped us visually inspect our data using QQ-graphs and inspired several experiments to answer our first research question. Based on the QQ-graphs and prior reasoning, we were able to conclude that our data was not uniformly distributed.

The following conversation reflected on statistical methods to use for the second experiment.

prompt For RQ2, i am comparing several redundant constraints. I plan on testing them individually by applying them on top the default qf_ia encoding. How can i best do this in a statistical approach? I know my data (runtimes) is not uniformly distributed. You also mentioned some wilcoxon test in an earlier conversation, can we use this here?.

answer This is actually a very clean setup for a proper statistical comparison RQ2 is much easier to treat rigorously than RQ1 because you are doing paired comparisons.

Below is a concrete, defensible statistical approach that fits your situation and that you can explain well in the paper.

1. Experimental design (this matters more than the test) You already made the right design choice:

- Fix one baseline encoding: qf_ia
- Add one redundant constraint at a time
- Evaluate the same set of puzzles with:
  - baseline
  - baseline + constraint C_i

This gives you paired data:

$$(t_{baseline}(p),\ t_{constraint}(p)) \quad \text{for each puzzle } p$$

That pairing is the key to everything that follows.

2. What not to do (and why) Because your runtimes are:

- non-normal
- heavy-tailed
- sometimes dominated by a few very hard instances

You should not:

- use a t-test
- compare means

- rely on standard deviation as your main argument

Even log-transforming runtime doesn't fully fix this in solver data.

3. Use differences, not absolute runtimes For each puzzle p, compute either:

Option A: Absolute difference

$$\Delta_p = t_{constaint}(p) - t_{baseline}(p)$$

Option B (better): Relative change

$$\Delta_p = \frac{t_{constraint}(p)}{t_{baseline}(p)}$$

or equivalently:

$$log\Delta_p = log\ t_{constraint}(p) - log\ t_{baseline}(p)$$

Why this helps:

- Normalizes across easy vs hard puzzles
- Makes "speedup vs slowdown" interpretable
- Is standard in solver evaluations

I strongly recommend relative change or log-differences.

4. Wilcoxon signed-rank test — yes, this is exactly the right tool

When to use it The Wilcoxon signed-rank test is appropriate when:

- data is paired
- differences are not normally distributed
- you want to test systematic improvement or degradation

That is precisely your situation.

What hypothesis it tests

For each constraint C_i:

- H_0 (null): median difference = 0 $\rightarrow$ the constraint has no systematic effect
- H_1: median difference $\neq 0 \rightarrow$ the constraint systematically helps or hurts

You can even use a one-sided test if you want to test: "Does this constraint improve performance?"

5. What to report (this is crucial) Do not just report a p-value.

For each constraint, report:

1. Median relative change Example: Median runtime ratio = 1.18 (meaning 18% slower)
2. Wilcoxon test result Example: Wilcoxon signed-rank test, $p < 0.01$
3. Directionality Also report:
    - % of puzzles that got faster
    - % that got slower

   This prevents "statistically significant but meaningless" results.

6. Visualization (very important) Use paired visualizations, not boxplots of raw runtime.

Best options:

- Boxplot or violin plot of log runtime ratios
- Scatter plot:
    - x = baseline runtime
    - y = constraint runtime
    - with diagonal $y = x$

These visually reinforce:

- skew
- pairing
- outliers

7. How to phrase this in your paper (example text)

You can say something like: Since solver runtimes are highly skewed and non-normally distributed, we evaluate the effect of redundant constraints using paired non-parametric tests. For each puzzle, we compare the runtime of the baseline qf_ia encoding with the runtime obtained by adding a single redundant constraint. We analyse relative runtime changes and apply the Wilcoxon signed-rank test to determine whether a constraint introduces a systematic performance difference across instances.

That's exactly what reviewers expect to see.

8. Interpretation guidance (important for your conclusions) Be very explicit:

- Statistically significant slowdown $\rightarrow$ constraint is harmful
- No significant difference $\rightarrow$ constraint provides no consistent benefit
- Small but significant improvement $\rightarrow$ limited usefulness, likely instance-dependent

9. Sanity checks (optional but nice) You do not need to prove non-normality again here, but you can:

- mention earlier QQ-plots
- or simply state that solver runtime distributions are well-known to be heavy-tailed

That's widely accepted in the SAT/SMT community.

Bottom line Yes — Wilcoxon signed-rank is the correct test, and your setup is ideal for it. Focus on paired differences, medians, and directionality, not averages.

If you want, next we can:

- draft a results table template
- write a ready-to-drop-in LaTeX paragraph
- or decide whether to separate results by puzzle size or difficulty

prompt Are there other statistical test that might be better applied to our situation? My goal is to show which constraints show a statistically significant increase or decrease per puzzle size

answer Yes — there are other tests you can use, and some are arguably better than Wilcoxon depending on what exactly you want to claim. Since your goal is: to show which constraints show a statistically significant increase or decrease per puzzle size we should be very precise about what is being tested. Below is a decision-oriented overview, then a recommended setup that fits your paper well.

1. Wilcoxon signed-rank (baseline choice) You already know this one, but let's anchor it.

What it tests

- Median paired difference $\neq 0$
- Assumes symmetry of differences (often approximately true after log-ratio)

Strengths

- Simple
- Well-known
- Very defensible for solver runtimes
- Works well per puzzle size

Weakness

- Only tests location (median), not magnitude
- Less informative when distributions are very skewed or multimodal

Conclusion: Still a very solid baseline, especially if reviewers are not statistics-heavy.

2. Sign test (simpler, weaker, but very robust) What it tests

- Do more puzzles get faster or slower?
- Ignores magnitude entirely

When it's useful

- If runtime variance is extreme
- If a few huge outliers dominate Wilcoxon
- As a sanity check alongside Wilcoxon

Example interpretation "For size 10×10, constraint C slowed down 73% of instances (sign test, p ¡ 0.01)."

Weakness

- Much lower statistical power
- Does not capture how much slower

Use this only as a supporting test, not your main one.

3. Permutation (randomization) test — arguably the best fit If you want the strongest methodological argument, this is it.

What it tests

- Whether the observed median (or mean of log-ratios) is larger than expected by chance
- No distributional assumptions at all

Why it fits your case perfectly

- Paired data
- Non-normal
- Heavy-tailed
- Small-to-medium sample sizes per puzzle size

How it works (conceptually) For each puzzle:

- Randomly swap "baseline" and "constraint" labels
- Recompute the statistic
- Repeat many times
- Compare observed value to null distribution

What you test Usually:

- Median log runtime ratio
- Mean log runtime ratio (this is OK here!)

Weakness

- Harder to explain in text
- Slightly more work to implement

Verdict Statistically superior to Wilcoxon, but slightly heavier cognitively for readers.

4. Cliff's delta + confidence intervals (effect size!) This is not a test, but it is extremely valuable.

What it measures Probability that a randomly chosen instance is slower with the constraint than without

Values:

- +1 → always slower
- 0 → no effect
- -1 → always faster

Why reviewers like this

- Scale-free
- Intuitive
- Not tied to distribution assumptions

Best use Report together with Wilcoxon or permutation test.

Example: Cliff's δ = 0.62 (large negative effect) This strengthens your claims a lot.

5. Mixed-effects models? (Probably overkill — and risky) You could model:

- puzzle size as fixed effect
- puzzle instance as random effect
- constraint as fixed effect

But:

- Assumptions are harder to justify
- Overkill for RP2
- Easy to get wrong
- Reviewers may question correctness

I do not recommend this for your project.

6. Recommended setup (clean, strong, defensible) Given your goals, I recommend:

Per puzzle size n, per constraint C:

1. Compute log runtime ratio

$$d_p = log\frac{t_C(p)}{t_{baseline}(p)}$$

2. Report:
   - Median d_p
   - % instances faster / slower
3. Apply one of:
   - Wilcoxon signed-rank test (default)
   - Permutation test (if you want to be extra solid)
4. Optionally include:

- Cliff's delta (effect size)

7. How to justify this choice in writing You can say: Since runtimes are paired and exhibit heavy-tailed, non-normal distributions, we analyse relative runtime changes per puzzle size. To test whether a redundant constraint introduces a systematic performance difference, we apply a paired non-parametric test on the per-instance log runtime ratios.

This wording works for Wilcoxon or permutation tests.

8. Short answer to your question

- Yes, there are other tests
- Permutation tests are statistically the strongest
- Wilcoxon signed-rank is perfectly acceptable and expected
- Effect sizes (Cliff's δ) are more important than p-values alone

If you want, next I can:

- help you choose Wilcoxon vs permutation definitively
- write a methods subsection
- design a results table that scales across puzzle sizes
- or help you handle multiple-testing correction (which will come up)

prompt I think ill just stick to the Wilcoxon method. Can we then just say that if the p-value for n=15 ¡ 0 the constraint is significantly worse? What if the p-value for n=16 ¿ 0? How do we deal with these issues? Do i report by size of do we use some other method? The Holm method which seemed interesting and involved sorting p-values per constraint, does this method fit here?

answer Great questions — this is exactly the point where many papers quietly get things wrong, so it's good you're thinking about this carefully.

I'll answer in layers:

1. What you are allowed to conclude per size
2. Why "p < 0.05 at n=15 but not at n=16" is not a contradiction
3. Why multiple-testing correction is required
4. Why Holm correction fits your setup very well
5. What I would actually recommend you report

1. Can you say "for n=15 the constraint is significantly worse"? Yes — but only locally.

If you do:

- one Wilcoxon test
- at puzzle size n=15
- comparing baseline vs baseline + constraint

then: If $p < \alpha$, you may conclude that for size 15, the constraint has a statistically significant effect. That statement is correct and defensible.

But it does not mean:

- the constraint is globally bad
- the effect generalises to all sizes

This distinction is important in how you phrase results.

2. What if n=15 is significant but n=16 is not? This is normal, not a problem.

Reasons:

- sample size per n changes
- runtime variance grows with size
- solver behaviour becomes more chaotic
- effect size may shrink relative to noise

A non-significant result at $n = 16$ means: we cannot reject the null hypothesis at this size It does not mean:

- the constraint stops being harmful
- the effect reverses

This is why effect sizes and medians must always be shown alongside p-values.

3. The real problem: multiple testing Here's the actual statistical issue you must address:

You are doing:

- many constraints
- $\times$ many puzzle sizes
- $\times$ possibly multiple variants

That's dozens of hypothesis tests.

Without correction: false positives are guaranteed So no, you cannot just say: "$p < 0.05 \Rightarrow significant$" globally.

4. Does Holm correction fit your situation? Yes — extremely well. In fact, Holm–Bonferroni is exactly the right correction here.

Why Holm is a good choice

- Controls family-wise error rate
- Less conservative than Bonferroni
- Standard in empirical CS / solver papers
- Easy to explain and justify

What you sort You were right to notice this: "sorting p-values per constraint" This is the correct intuition.

5. How to apply Holm correctly in your case The key question is: What is a "family" of tests?

You have two defensible options.

Option A (recommended): Holm per constraint across sizes For each constraint C:

- Perform Wilcoxon tests for:

$$n = 10, 11, \ldots, 18$$

- Collect p-values:

$$p_{C,10}, p_{C,11}, \ldots$$

- Apply Holm correction within this set

Interpretation You may then say: Constraint C shows a statistically significant slowdown for sizes $n = 14$ and $n = 15$ after Holm correction. This directly answers: for which sizes does this constraint matter? This is the cleanest answer to your research question.

Option B: Holm per size across constraints (less aligned)

You could instead:

- Fix size n
- Compare all constraints at that size
- Apply Holm across constraints

This answers: Which constraint is worst at size n? But that's not your main RQ, so I wouldn't prioritise this.

6. What NOT to do Avoid:

- pooling sizes into one test (loses structure)
- declaring global significance from one size
- reporting only corrected p-values without effect sizes
- hiding non-significant sizes

7. How to phrase conclusions correctly Bad (overclaiming) Constraint C is significantly worse. Good Constraint C introduces a statistically significant slowdown for puzzle sizes $n \geq 14$, with median runtime increases of 20–35%. Very good While constraint C shows a consistent slowdown across all sizes, this effect is only statistically significant for sizes $n = 14$ and $n = 15$ after Holm correction.

8. Suggested reporting structure For each constraint:

- Table or plot:
  - size n
  - median log ratio
  - % slower
  - significance marker (after Holm)

Visually, this reads beautifully.

9. Short answer to your core question

- Yes, Wilcoxon is fine
- No, you cannot treat sizes independently without correction
- Yes, Holm correction per constraint across sizes is appropriate
- This is a strong, reviewer-proof choice

If you want, next I can:

- help you define the exact null/alternative hypotheses
- write a precise LaTeX methods paragraph
- help design a significance annotation scheme for plots
- or sanity-check whether Holm is too conservative given your sample sizes

discussion This conversation ended with some small discussion on our implementation of these methods in code. This conversation resulted in the use of the Wilcoxon method in combination with the Holm-Bonferroni method for our second experiment. The final conversation explored methods to be used for the final experiment.

prompt For my third question, I am analysing puzzles that have been filtered using different methods. I used two methods to filter hard puzzles, one using runtime and another using effort. I ended up with 4 sets, a set containing only effort-filtered puzzles, a set containing only runtime-filtered puzzles, a set containing puzzles filtered by both and a set containing puzzles filtered by none. For each of these puzzle sets, i can analyze the puzzles for a couple of properties. I want to figure out if one of these two methods is able to filter out puzzles that have a significantly different number for any of the properties against the non-filtered puzzles. What method could i best use for this?

answer You're basically asking a comparative group analysis question: Do puzzles selected by method A (runtime) or method B (effort) differ significantly in certain properties compared to unfiltered puzzles? And which method is "better" at isolating different puzzles? That's a very standard and clean statistical setup. Here's a method that fits your situation well and is defensible in a research report.

1. Reframe the structure (this helps choose the test) You have four groups:

1. None – not filtered (baseline / control group)
2. Runtime-only
3. Effort-only
4. Both (runtime $\cap$ effort)

For each puzzle, you measure one or more numeric properties (e.g. conflicts, propagations, encoding size, etc.).

Your core comparison is always: Filtered group vs non-filtered group Not really filtered-vs-filtered.

2. Recommended primary test: Non-parametric group comparison Given everything you've said earlier:

- solver metrics are skewed
- distributions are not normal
- sample sizes may differ

Non-parametric tests are the right choice.

Option A (most straightforward): Mann–Whitney U test Use this when comparing two groups at a time.

For each property:

- Runtime-only vs None
- Effort-only vs None
- Both vs None

This directly answers: "Does this filtering method select puzzles with different values for property X?" Why this is good

- No normality assumption
- Works with skewed solver data
- Very common in solver-performance papers

3. Multiple properties $\rightarrow$ multiple tests $\rightarrow$ correction Because you'll test:

- multiple properties
- for multiple filtered groups

you must correct for multiple testing.

You already implemented Holm–Bonferroni, which is perfect here (And more defensible than plain Bonferroni.)

So your pipeline becomes:

1. For a fixed group comparison (e.g. effort-only vs none):
   - run Mann–Whitney for each property
2. Collect p-values
3. Apply Holm correction

4. Mark which properties differ significantly

4. Effect size (important!) Significance alone isn't enough. You also want to say how they differ.

For Mann–Whitney, report:

- Cliff's delta or
- Rank-biserial correlation

Then you can say things like: Effort-filtered puzzles show significantly higher conflicts ($p < 0.01$, large effect), whereas runtime-filtered puzzles only show a small increase. This is exactly what your research question is really about.

5. How to compare "runtime vs effort" as methods You don't need a direct statistical test between the two methods.

Instead, argue comparatively:

- How many properties differ from baseline?
- How strong are the effect sizes?
- Are the directions consistent?
- Does the "both" group amplify effects?

Example interpretation (very strong in a paper): Effort-based filtering isolates puzzles that differ significantly from the baseline in conflicts, propagations, and decisions, while runtime-based filtering only shows a significant difference in runtime itself. That answers the question qualitatively but rigorously.

6. Optional global test (nice but not required) If you want a single "overall" test per property:

- Kruskal–Wallis test across all four groups → followed by post-hoc Dunn tests vs None

This is clean, but:

- more complex
- harder to explain
- not strictly necessary

Your pairwise approach is perfectly acceptable.

7. How to phrase this in your methodology section You can say something like: To evaluate whether runtime-based or effort-based filtering selects puzzles with distinct characteristics, we compared each filtered puzzle set against the unfiltered baseline. For each puzzle property, we used the Mann–Whitney U test due to non-normal and skewed distributions. To control for multiple comparisons across properties, Holm–Bonferroni correction was applied. Effect sizes were computed to quantify the magnitude and direction of observed differences. That's textbook-correct.

TL;DR (recommended method) Use:

- Mann–Whitney U test
- Filtered group vs non-filtered group
- Holm–Bonferroni correction
- Report effect sizes

This directly answers:

- Does runtime filtering work?
- Does effort filtering work better?
- What kinds of puzzles do they actually select?

If you want, I can:

- sanity-check your current code against this setup
- help you choose and interpret effect size thresholds
- help phrase the results section so it doesn't sound like fishing

prompt I dont think Holm-bonferroni correction is applicable here, are there other corrections i can use?

answer Good instinct to question this — and yes, there are good alternatives, some of which arguably fit your setup better than Holm–Bonferroni.

Let's be precise about why Holm–Bonferroni may feel wrong, and what you can use instead.

1. Why Holm–Bonferroni may indeed be questionable here Holm–Bonferroni controls the family-wise error rate (FWER). That means: The probability of making any false positive across all tests is bounded. This is appropriate when:

- each false positive is costly, or
- you want very strong claims ("we are sure none of these are spurious").

But in your case:

- you're exploring properties, not testing a single confirmatory hypothesis
- properties are correlated (conflicts ↔ propagations ↔ decisions)
- you care more about patterns than absolute guarantees

So Holm can be:

- too conservative
- artificially suppress weak-but-consistent signals
- misleadingly suggest "nothing differs" when things clearly do

That's a perfectly defensible reason to not use it.

2. Better alternatives for your situation Benjamini–Hochberg (FDR control) — best overall fit This is the method I'd recommend first.

What it controls

- False Discovery Rate (FDR): Among the results you call significant, how many are expected to be false?

Why it fits your research

- You expect some differences
- You're comparing many related properties
- You want to identify which properties differ, not prove none do

How to justify it Because we test multiple correlated puzzle properties in an exploratory analysis, we control the false discovery rate using the Benjamini–Hochberg procedure. This is extremely common in empirical CS and solver-analysis papers.

Benjamini–Yekutieli (more conservative FDR) If you want to explicitly acknowledge correlation between properties:

- Benjamini–Yekutieli controls FDR under arbitrary dependence

- More conservative than BH, less than Holm

You can say: To account for potential dependence between solver statistics, we applied the Benjamini–Yekutieli correction. Use this if you want to be extra safe.

Bonferroni (classic) — not recommended here

- Even stricter than Holm
- Almost guaranteed to wipe out subtle effects
- Hard to justify for exploratory performance analysis

You'd only use this if your supervisor explicitly demands it.

3. Another option: avoid correction entirely (yes, really) This is valid if done carefully.

You can:

- treat each property as a separate analysis
- explicitly label results as exploratory
- emphasize effect sizes over p-values

Example justification: Since puzzle properties are analyzed independently and interpreted jointly via effect sizes and directionality, no global correction was applied. Reported p-values are descriptive rather than confirmatory. This is acceptable if:

- you do not overclaim
- you don't say "statistically proven"
- you show consistent patterns across properties

That said, this is riskier with some reviewers.

4. Strong alternative framing (often overlooked) Instead of correcting across properties, correct across groups per property.

For a single property:

- runtime-only vs none
- effort-only vs none
- both vs none

That's three tests, not dozens.

Then:

- apply Holm or Bonferroni per property
- not globally

This is much more defensible and less conservative.

5. What I'd recommend for your paper Best balance of rigor and signal:

- Mann–Whitney U tests
- Benjamini–Hochberg correction per comparison family
  - e.g. "effort vs none across properties"
- Report:
  - corrected p-values
  - effect sizes
  - direction of change

Suggested sentence you can basically copy To account for multiple comparisons across puzzle properties while maintaining sensitivity to systematic effects, we controlled the false discovery rate using the Benjamini–Hochberg procedure rather than family-wise error rate corrections. That sentence alone justifies not using Holm.

TL;DR If Holm feels wrong, you're probably right.

Use instead:

- Benjamini–Hochberg (recommended)
- Benjamini–Yekutieli (if you want to mention dependence)
- Or no correction + effect sizes (only if framed carefully)

If you want, I can:

- help you rewrite your existing Holm code to BH in 5 lines
- help you decide whether to correct per property or per method
- sanity-check the final statistical claims wording

conclusion This conversation once again ended with feedback on our implementation of these methods in code. This conversation helped us find the two methods to use in our final experiment. All methods suggested by ChatGPT were manually verified to ensure suitability and to expand our own understanding of these methods.

E.4 Conversations about rewriting paragraphs

ChatGPT has been used to help with rewriting several paragraphs for readability, wording and text flow. All of these conversations followed a similar structure by first prompting a draft of the paragraph, asking for feedback. The draft was iteratively improved by analysing the structure of the response text from ChatGPT. This process made sure to only improve on the flow, wording and structure of the text, without changing any of the contents. All changes were implemented manually and verified to make sure no new information was added by ChatGPT.

One conversation using this process is shown below. Other conversations that were used to improve parts of the text followed the exact same process and have been left out to conserve space. All current parts that were revised using this process are the following:

- Section 1, SMT and Z3 parts
- Section 3.2
- Section 4.2
- Section 4.3
- Section 5.1

prompt I need some help making this paragraph more readable, i especially dislike the second part:

We also implemented a variant that uses only Boolean logic, without any additional theories. Such an encoding is already given in [12], however the authors also show that this approach is very inefficient for large puzzle sizes. Therefore we decided on a different approach where we rewrite the connectivity constraint such that each white cell is assigned a Boolean value for each step, indicating whether the cell has been visited in that step, with only the root cell marked as visited for the initial step. For $n \cdot n + 1$ steps and for each cell (i, j), the cell is visited if either the cell itself or any of its neighbours was visited in the previous step. The connectivity constraint is satisfied if all white cells are visited after the final step.

answer Here's a tightened and more academic-sounding rewrite that keeps your technical content intact, improves flow, and removes a bit of redundancy:

We also implemented a variant that relies exclusively on Boolean logic, without the use of additional theories. A Boolean encoding for Hitori has previously been proposed by [12], where the authors show that such an approach becomes highly inefficient for larger puzzle sizes. Motivated by this limitation, we adopt a different Boolean formulation of the connectivity constraint.

In this encoding, each white cell is associated with a Boolean variable for each discrete step, indicating whether the cell is visited at that step. Initially, only a designated root cell is marked as visited. For each of the subsequent $n \cdot n + 1$ steps and for each cell (i, j), a cell is marked as visited if either the cell itself or at least one of its neighbours was visited in the previous step. The connectivity constraint is satisfied if all white cells are visited after the final step.

If you want it slightly more compact or less repetitive (e.g. fewer "visited" mentions), I can do a more aggressive compression version as well.

prompt I've changed it to this now, what do you think? We also implemented a variant that uses only Boolean logic, without any additional theories. A Boolean encoding is already been given in [12], where the authors show that this approach quickly becomes very inefficient for large puzzle sizes. Because of this, we attempted a different Boolean approach, only rewriting our connectivity constraint. In this encoding, each white cell is assigned a Boolean value for each step, indicating whether the cell has been visited in that step, with only the root cell marked as visited for the initial step. For $n \cdot n + 1$ steps and for each cell (i, j), the cell is visited if either the cell itself or any of its neighbours was visited in the previous step. The connectivity constraint is satisfied if all white cells are visited after the final step.

answer It's definitely better in terms of clarity — there are just a few small language issues and some phrasing that can be made more academic and smooth. Here's a lightly polished version that stays very close to what you wrote:

We also implemented a variant that uses only Boolean logic, without any additional theories. A Boolean encoding has already been proposed in [12], where the authors show that this approach quickly becomes inefficient for large puzzle sizes. For this reason, we instead explore a different Boolean formulation by rewriting only the connectivity constraint.

In this encoding, each white cell is assigned a Boolean variable for each step, indicating whether the cell has been visited at that step, with only the root cell marked as visited in the initial step. For each of the $n \cdot n + 1$ steps and for each cell (i, j), the cell is considered visited if either the cell itself or any of its neighbours was visited in the previous step. The connectivity constraint is satisfied if all white cells are visited after the final step.

What changed (briefly):

- Fixed grammar: "is already been given" → "has already been proposed"
- Slightly more formal phrasing ("because of this" → "for this reason")
- "attempted" → "explore" (sounds more deliberate and academic)
- Minor flow improvements and reduced repetition

If you want, we can also:

- Explicitly contrast this with the original encoding in one sharper sentence, or
- Add a one-line remark about the encoding size / step count implication (if you discuss that elsewhere).

prompt I've changed it quite a bit now by adding an equation as well. I think ill be keeping as this, as im quite happy with it: We also implemented a variant that uses only Boolean logic, without any additional theories. A Boolean encoding is already been given in [12], where the authors show that this approach quickly becomes inefficient for large puzzle sizes. For this reason, we implemented an alternative Boolean encoding, only rewriting our connectivity constraint. With the `boolean` encoding, we aim to visit every white cell within $n^2 + 1$ steps using BFS, starting from a chosen root. We mark a cell (i, j) as visited in step k by setting $\text{visited}_{k,i,j}$ to True. The encoding is formulated as follows: [equation] The connectivity constraint is satisfied if all white cells are visited at $k = n^2 + 1$.

answer You're right that this version is solid conceptually, and the equation helps a lot

I'd keep it too — there are just two small but important language issues and one clarity tweak I'd strongly recommend fixing before final submission.

Here is a minimally corrected version that preserves everything you intend: We also implemented a variant that uses only Boolean logic, without any additional theories. A Boolean encoding has already been given in [12], where the authors show that this approach quickly becomes inefficient for large puzzle sizes. For this reason, we implemented an alternative Boolean encoding, rewriting only the connectivity constraint.

With the `boolean` encoding, we aim to visit every white cell within $n^2 + 1$ steps using a breadth-first search (BFS), starting from a chosen root. We mark a cell (i, j) as visited at step k by setting visitedk, i, j to `True`. The encoding is formulated as follows: [equation] The connectivity constraint is satisfied if all white cells are visited at step $k = n^2 + 1$.

What I fixed (and why)

1. Grammar bug (important)
   - "is already been given"
   - "has already been given"
2. Academic phrasing
   - "only rewriting our connectivity constraint" → "rewriting only the connectivity constraint" (clearer scope, standard phrasing)
3. Minor precision

- Added “a” before BFS (standard in academic writing)
- “at $k = n^2 + 1$” $\rightarrow$ “at step $k = n^2 + 1$” (reads cleaner)

Nothing else needs changing — the structure, explanation, and formalization are all very reasonable and well-justified. This paragraph will read well to reviewers.