Smashing Hitori

An analysis of the strengths and weaknesses of constraint programming paradigm Pumpkin

Lesley Smits[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: Lesley Smits
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L. D. Latour, Dr. T.J. Coopmans

Abstract

Logic-based puzzle solving serves as a standard benchmark for evaluating computational paradigms; however, comparisons are frequently biased by the researcher's familiarity with specific tools. To ensure an objective evaluation, we contribute to a collaborative benchmarking effort by analysing the suitability of the Constraint Satisfaction Problem (CSP) paradigm for solving the puzzle Hitori. Specifically, we use PUMPKIN [5]. Our results show that redundant constraints can have a positive impact on solve time, particularly when using a Lazy constraint strategy where dynamic constraint generation is employed. Additionally, our analysis of puzzle characteristics reveals that the Lazy strategy benefits significantly from high counts of non-adjacent duplicates, while the Distance strategy proves sensitive to deviations in the total black cell count. We demonstrate that this Lazy strategy significantly outperforms the Distance strategy, offering superior scalability with respect to puzzle size. Furthermore, in a comparative analysis against other computational paradigms, our solver proved more efficient than LP (GUROBI), Prolog, and SMT (Z3) implementations, ultimately ranking second, surpassed only by ASP (CLASP). This establishes CSP, specifically with the Lazy strategy, as a competitive approach for modelling and solving Hitori.

1 Introduction

Logic puzzles are frequently used in computer science because they transform complex computational challenges into a format that is intuitive for humans to understand and solve, rather than relying on abstract or highly specialized mathematical topics. This accessibility makes them an ideal medium for evaluating various computational problem-solving paradigms. A common challenge, however, is that comparisons are often biased: researchers tend to be more familiar with certain paradigms, giving those approaches an inherent advantage. To achieve fair and objective comparisons, our group has organized a collaborative effort in which each member focuses on a single paradigm, ensuring that each solver is developed by an expert in that paradigm. To allow for fair benchmarking results, the puzzle instances will be generated randomly, so that the problem instances do not favour certain paradigms over others.

The puzzle covered in this research is Hitori. Hitori presents a combination of global connectivity constraints and local structural rules. A Hitori puzzle consists of an $n \times n$ grid with numbers ranging from 1 to n. To solve it, certain numbers must be marked so that the resulting grid satisfies the following conditions:

- *Uniqueness*: every number only appears once in the white tiles of a row or column.
- *Adjacency*: black tiles are never orthogonally adjacent.
- *Connectivity*: all white tiles are connected to all other white tiles.

2	2	3	3
3	2	2	1
4	3	1	3
1	3	4	2

(a) Puzzle

2	2	3	3
3	2	2	1
4	3	1	3
1	3	4	2

(b) Solution

Figure 1: A Hitori puzzle and its solution

Puzzles also differ by specific characteristics, such as the count of marked tiles required in the solution. As Hitori is proven to be NP-complete [9], it presents significant challenges that scale with puzzle size. This makes it a solid candidate for evaluating the efficiency of different computational paradigms and modelling choices.

This research investigates how well the CSP implementation PUMPKIN is suited for solving the Hitori puzzle.

The research questions we answer are:

- What is the impact of adding redundant constraints to our base encoding on solving time?
- What impact do the puzzle size and puzzle characteristics have on encoding size and solve time?
- How does Pumpkin compare to different paradigms at solving Hitori puzzles in solving capability and time?

In this paper, we begin by showing relevant work done, then show the differences between candidate paradigms, and the relevant background information. We will then model Hitori for PUMPKIN and create a solver. Using this solver, we explore the impact of redundant constraints, evaluate solve time and encoding size scalability across sizes and puzzle characteristics and compare our solver to other solvers created by the research group.

2 Related Work

In this section, we show what related work has already been done into logic puzzle solving using CSP and Hitori.

Wensveen conducted a comprehensive study on the Hitori puzzle, focusing on the interplay between solving, generation, and difficulty classification [12]. Wensveen's work provides a robust framework for generating valid Hitori instances and analysing their complexity using metrics derived from standard SAT solvers. However, while Wensveen uses existing solving technologies to facilitate generation and classification, the primary scope of that research lies in the properties of the puzzle itself.

M. Gander and C. Hofer used an SAT solver to solve Hitori [6]. They used methods to encode the constraints that lead to exponential encodings, which caused slow solve times for large puzzles. They also identified and described

many puzzle patterns, but they did not analyse their impact.

Simonis provides an analysis of Sudoku viewed through the lens of Constraint Programming [11]. Simonis establishes the critical importance of global constraints; he shows that applying *Generalized Arc Consistency* (GAC) on these global constraints allows the solver to perform much stronger propagation, pruning values that binary constraints would miss, often allowing the solver to deduce the solution without requiring any backtracking search. This work underscores the necessity of expressive constraint modelling for logic puzzles, an approach we use for solving the Hitori puzzle by utilizing the advanced global constraint capabilities of the PUMPKIN solver.

A specific challenge in Hitori is ensuring the connectivity constraint holds. Dumas et al. demonstrated that reasoning about connectivity as a global constraint allows for powerful pruning of the search space [3]. While their work represents the theoretical state-of-the-art for connectivity propagation, our implementation adopts a standard constraint modelling approach. This allows us to evaluate the performance of modern solvers like PUMPKIN on logic puzzles without the need for domain-specific algorithmic augmentations.

3 Background

Logic puzzles can be solved using a variety of paradigms, each offering their own strengths and weaknesses. To highlight these differences, this literature review examines two specific paradigms, discussing their inner workings as well as their respective strengths and weaknesses. We then evaluate these paradigms in the context of a Hitori implementation so that a comparison can be made for these approaches. This is followed by discussing related work on Hitori and other logic puzzles, showing their relevance to this research.

3.1 Constraint Satisfaction Programming (CSP)

A constraint satisfaction problem (CSP) involves finding a valid assignment for a set of variables. This assignment must select values from a finite domain so that all defined constraints are satisfied [1]. An important concept CSP solvers use to reduce variable domains is *arc consistency*. By utilizing the problem's constraints, the solver identifies specific values that can never result in a satisfiable solution and removes them from the domain. More specifically, if a binary constraint C involves variables x and y with domains D_x and D_y, the domains are reduced to include only values for which a valid combination exists. A value a is kept in D_x if:

$$\exists b \in D_y \text{ such that } (a, b) \text{ satisfies } C$$

If no such supporting value b exists, a is pruned from the domain [1]. CSP solvers are particularly effective for highly constrained problems, which is the case for the Hitori puzzle. In these scenarios, the restrictive constraints actively guide the search, allowing propagation techniques to prune large portions of the search space efficiently. Crucially, this local pruning allows CSPs to solve problems without generating the entire logical search space upfront [1]. The CSP solver that has been chosen for this research is PUMPKIN [5]. It is a solver that can be used as a solving engine for MiniZinc, a high-level modelling language, as well as directly as a package for Rust. PUMPKIN is a CSP solver that implements the *Lazy Clause Generation* (LCG) paradigm. LCG enhances standard constraint propagation by incorporating SAT solving techniques, giving the solver the ability to learn from failures, nogoods, and add them as constraints [4].

3.2 Answer Set Programming (ASP)

Answer Set Programming (ASP) is a declarative paradigm where problems are modelled as logic programs consisting of rules. Unlike CSP, which is defined by variables and domains, an ASP problem is defined by a set of logical rules, and the solutions correspond to the stable models (or answer sets) of that program [7]. For this research, the chosen solver is CLASP. CLASP uses Conflict-Driven ASP, this improves ASP by allowing back-jumping and Conflict-Driven-Learning (CDL) [7].

In terms of problem modelling, an ASP program consists of a finite set of logic rules. As defined in the preliminaries of [7], a rule r is typically expressed in the form:

$$a_0 \leftarrow a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n$$

Here, a_0 is the **head** of the rule, and the remaining elements form the **body**. The comma represents a logical conjunction (AND), while "not" denotes default negation (negation as failure). Intuitively, this rule states: "if a_1 through a_m are true, and there is no evidence that a_{m+1} through a_n are true, then a_0 must be true."

Constraints, which are vital for puzzles like Hitori, are often modelled as rules with an empty head:

$$\leftarrow a_1, \ldots, a_n$$

This structure forbids any solution where the body is satisfied, effectively acting as a nogood that prunes invalid states from the search space [7]. An advantage of ASP for the Hitori puzzle is its native support for recursive definitions, allowing for the direct modelling of the connectivity constraint. While standard propositional logic struggles with such structures, ASP solvers handle them through the *stable model semantics*. Specifically, Gebser et al. describe how modern solvers utilize *unfounded set propagation* to verify the support of atoms within recursive loops (referred to as "non-tight" programs), ensuring that cyclic dependencies are resolved correctly [7]. However, ASP also retains the same issue CSP has, where it was designed for satisfaction. Consequently, identifying a unique solution may be computationally expensive.

3.3 Comparison

Having analysed the inner workings of both CSP (the LCG approach of PUMPKIN) and ASP (the CDL approach of CLASP), we now compare their suitability for the Hitori implementation.

In terms of pure expressiveness, ASP offers a distinct advantage for the Hitori puzzle due to its native support for recursion. However, while CSP solvers lack this native recursive syntax, the PUMPKIN solver's architecture allows for

tight integration with the host language (Rust). Instead of relying on a solver to pre-calculate all connectivity paths, we can implement some additional constraints that remove most of these invalid solutions, and add a connectivity check in Rust which dynamically adds constraints to cut off disconnected solutions. This approach grants finer control over the performance trade-offs than the black-box nature of standard ASP solvers.

Furthermore, computational requirements must be considered. ASP solvers like CLASP require a full *grounding* phase. For large Hitori grids, this could lead to an explosion in memory usage and startup time, as the solver must generate rules for every possible tile connection upfront.

In contrast, the CSP paradigm avoids this bottleneck by relying on domain propagation (arc consistency) and dynamic clause generation. By generating SAT clauses dynamically during the search and only when necessary, PUMPKIN can potentially identify solutions for large instances significantly faster while taking up less computational resources.

Given the need for scalable performance and the desire for fine-grained control over constraint application via Rust, the CSP paradigm (using PUMPKIN) is selected for this research. The LCG architecture offers an effective balance between memory efficiency and search speed for the specific constraints of Hitori.

4 Approach

To answer our research questions, this research adopts a hybrid approach. It combines a collaboratively developed benchmarking infrastructure to ensure fair comparison across paradigms, with an individual implementation focus on puzzle size, characteristics and redundant constraints.

4.1 Collaborative Benchmarking Framework

To ensure a robust evaluation, we use a shared infrastructure developed by our research group. Using these tools, we create our puzzle instances. This approach mitigates the selection bias inherent in static problem libraries. As J.N. Hooker describes [10], rigorous empirical science requires moving beyond 'competitive testing on limited datasets' to instead analysing how algorithmic performance scales across varying problem characteristics.

The shared infrastructure contains:

1. *Puzzle Generator*
   We developed a *Puzzle Generator* capable of producing any valid, *unique* Hitori instance of size $n \times n$. By unique, we mean that the puzzle admits exactly one valid solution. A proof that the generator is able to generate all unique puzzle instances can be found in Appendix B.
2. *Solution Checker*
   A *Solution Checker* was developed to allow verification of solver solutions.

4.2 Constraint Modelling in Pumpkin

The modelling process transforms the logical rules of Hitori into a Constraint Satisfaction Problem (CSP) compatible with the PUMPKIN solver. This process is divided into three stages: variable definition, constraint formulation, and the implementation of global connectivity strategies.

Variable Representation
The grid is mapped to a 2D array of boolean decision variables, $B_{r,c}$, where:

$$B_{r,c} = \begin{cases} \text{true} & \text{if tile } (r,c) \text{ is marked (Black)} \\ \text{false} & \text{if tile } (r,c) \text{ remains unmarked (White)} \end{cases}$$

Inside PUMPKIN, these variables are mapped to numerical values, true being 1 and false being 0, so that numerical constraints can also be applied.

Constraint formulation
The local rules of Hitori are directly encoded as propositional logic clauses.

- **Adjacency Constraint:** To satisfy the rule that no two painted tiles may be orthogonally adjacent, binary clauses are added for every tile (r,c) and its neighbours $(n_r, n_c) \in ((r+1,c),(r-1,c),(r,c+1),(r,c-1))$:
$$\neg B_{r,c} \vee \neg B_{n_r,n_c}$$
- **Uniqueness Constraint:** For any two different tiles (r,c) and (r',c') in the same row or column that contain the same initial numerical value, at least one of them must be marked:
$$B_{r,c} \vee B_{r',c'}$$

Connectivity Strategies
The condition that all unpainted tiles must form a single connected component is a global constraint that poses a significant modelling challenge in CSP. To evaluate performance trade-offs, two distinct strategies are implemented

1. Lazy Connectivity In this approach, the solver initially ignores the connectivity constraint. It generates a candidate solution satisfying only the local constraints. An external Breadth-First Search (BFS) algorithm then verifies the connectivity of the white tiles. If disconnected, a nogood clause is learned, preventing that specific configuration, and the solver searches for a new solution until a valid solution is found or no other options are possible.

2. Distance-Based Connectivity This strategy, used by Wensveen in their solver [12], encodes connectivity directly into the CSP using a distance matrix. The implementation of this requires introducing auxiliary variables:

- $Root_{r,c}$: A boolean variable indicating if tile (r,c) is the root of the connected component.
- $Distance_{r,c}$: An integer variable representing the distance of a white tile from the root.

The constraints enforce a single-source valid path for every white tile:

- **Single Root:** Exactly one white tile is designated as the root ($\sum Root_{r,c} = 1$), with the root being the only tile with $Distance_{r,c} = 0$.
- **Parent Validation:** Every non-root White tile (r,c) must have at least one valid "parent" neighbour: $(n_r, n_c) \in ((r+1,c),(r-1,c),(r,c+1),(r,c-1))$, such that the neighbour is white and closer to the root ($Distance_{n_r,n_c} < Distance_{r,c}$).

Redundant Constraints
Finally, a set of logically redundant constraints implied by the constraints of Hitori but useful for pruning the search space are implemented. These constraints can be toggled on or off, allowing for an experimental analysis of their impact on our solver's solving time. These are based on common Hitori patterns and human solving strategies. A summary of the constraints can be found in Table 1. A full explanation can be found in Appendix C.

Table 1: Summary of Redundant Constraints.

Abbr.	Name	Summary
WN	White Neighbours	Every white tile must have ≥ 1 white neighbour.
CCH	Corner Check	A 2×2 block of two pairs next to a corner force white assignment to prevent isolation.
SP	Sandwich Pair	If a number is between a pair, it must be white.
EP	Edge Pair	Prevents border tiles from being marked if it has a pair neighbour and a pair diagonal neighbour in the same direction.
CC	Corner Close	Prevents black tiles from isolating a corner tile.
FI	Flanked Isolation	A tile flanked by two adjacent pairs must be black.
LW	Least White	Sets a minimum for the number of white tiles in a row/col.
UC	Unique Cell	Symbols unique in their row/col are set to white.
PI	Pair Isolation	Prevents isolation when identical symbols flank a tile.
CI	Close Isolation	In a 9 tile square, if the 4 tile "diamond" is black, the other are white.
WB	White Bridges	Adjacent rows must share ≥ 1 pair of orthogonal white tiles.
DW	Diagonal Wall	Ensures no full diagonal is entirely black tiles.

4.3 Puzzle Characteristics

In order to identify structural differences in puzzle instances, we categorize them based on the following characteristics.

- *Black Count* (BC): The total number of marked tiles in the solution.
- *Adjacent Duplicates* (AD): The number of duplicate pairs that are orthogonally adjacent.
- *Non-Adjacent Duplicates* (NAD): The number of duplicate pairs in the same row or column that are separated by at least one other tile.
- *Triple Adjacent Duplicates* (TAD): The frequency of three identical numbers appearing consecutively (e.g., "5 5 5").

4.4 Paradigm Benchmarking

As this paper is part of a larger collaboration, a critical step is the comparative evaluation of the CSP paradigm against other paradigms.

To ensure an objective evaluation, we benchmark our solver against those developed by other members of the research group, specifically covering the paradigms of Answer Set Programming (CLASP), Linear Programming (GUROBI), Prolog, and Satisfiability Modulo Theories (Z3). This collaborative framework mitigates the implementation bias often present when a single researcher attempts to implement multiple paradigms, ensuring that each solver is optimized by a dedicated researcher.

5 Experimental Setup

In this section, we will discuss the setup used to run our experiments, as well as how our measurements are taken, and the Metrics we will use.

5.1 Puzzle Generation

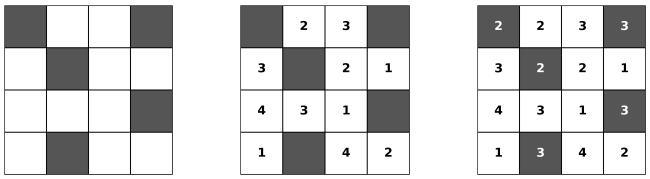
The *Puzzle Generator*, which generates $n \times n$ puzzle grids, employs a "reverse-generation" strategy: it first constructs a valid solution state (topology), a grid that shows what tiles need to be marked when solving the puzzle, and then populates the grid with numbers that force that specific solution.

The generation consists of three distinct phases:

1. **Solution Topology Generation**
   The first phase generates a valid solution mask (a binary grid denoting black and white tiles) without yet assigning numerical values. This process ensures the structural rules of Hitori are met from the start.

   The algorithm initializes an $n \times n$ grid of white tiles and iterates through a randomized list of coordinates. For each tile, it attempts to toggle the state to black. This change is accepted only if it violates neither the adjacency constraint nor the connectivity constraint.

   To ensure that our generator is capable of producing the full spectrum of possible solution topologies, we provide a theoretical proof of coverage in Appendix B.1.
2. **Constraint-Based Number Population**
   Once the solution topology is established, the generator populates the grid with integers $v \in \{1, \ldots, n\}$. This is done in two steps using a recursive backtracking algorithm:
   - **White Tile Assignment:** The algorithm first assigns numbers to the white tiles. To satisfy Hitori rules, these numbers are placed such that no number appears more than once in any row or column.
   - **Black Tile Assignment:** The black tiles are populated subsequently. To ensure the puzzle is solvable, a black tile must contain a number that is equal to a number of a white tile in its respective row or column.



(a) Solution topology

(b) White cell values

(c) Black cell values

Figure 2: A visualisation of the Puzzle Instance Generation

We demonstrate that this two-step population method can generate valid numerical configurations for any given valid topology in Appendix B.2.

3. **Uniqueness Check**
   Generated grids are sent to a *Base Solver*. This is a Hitori solver that is distinct from our experimental models, it is written in *MiniZinc-Python* using the solving engine *Chuffed*. It verifies that the puzzle yields exactly one unique solution. If the solver finds multiple solutions or no solution, the grid is discarded, and the process restarts.

5.2 Measurement in Rust

To ensure accurate performance profiling, we strictly define the boundaries of the measured solving time, as well as the encoding size. File I/O operations are excluded from the performance metrics. These operations are heavily dependent on disk speed and serialization libraries, which are not relevant to the algorithmic efficiency of the solver. The measured *Solving Time* is the sum of two measurements: *Encoding Time*, the time required to declare variables and define constraints within the solver instance, and *Search Time*, the time spent by the solver's internal engine to propagate constraints and backtrack to a valid solution. These durations are captured using the high-precision `std::time::Instant` monotonic clock in Rust, measuring elapsed wall-clock time.

The measured *Encoding Size* is recorded immediately after the solver finds a solution and is a combination of: *Variables*, the total number of decision variables, and *Clauses*, the number of propositional logic clauses.

5.3 Experimental Procedure

To answer our research questions, we designed three separate experiments to evaluate our solver. We first analyse the impact of redundant constraints, then evaluate the solver's behaviour regarding scaling and puzzle characteristics, and finally benchmark against solvers from other paradigms.

Throughout these experiments, when reporting quantitative changes in solving time or encoding size, we use the arithmetic mean. We prioritize the mean over the median to ensure that the computational impact of outliers is fully reflected in our performance assessment.

All internal experiments are conducted on a Desktop with an AMD Ryzen 7 5800X and 32 GB 3000-MHz RAM. The operating environment is Ubuntu 22.04 running via Windows Subsystem for Linux (WSL) on Windows 11. To ensure consistent benchmarking, the solver is restricted to a maximum memory allowance of 8 GB and a solving timeout of 10 seconds per instance. For the Collaborative benchmarking, we use the same Desktop, but inside of a Docker container running *python:3.11-slim* with a hard resource limit of 8GB. This allows for more accurate reproducibility of benchmarking, as everyone is able to simulate the same environment.

Redundant Constraint Analysis

To answer RQ1, we evaluate the baseline performance of each connectivity encoding strategy. We use the *redundant test set*, comprising 40 puzzle instances per grid size, with sizes ranging from 5×5 to 25×25. Subsequently, using the same instances, we analyse the change in solving performance when redundant constraints are enabled. To identify statistically significant performance changes, we employ the *Wilcoxon Signed-Rank test*, where we consider $p < 0.05$ to be significant. We use this test because our data consists of paired measurements on identical puzzle instances, allowing us to isolate performance differences from instance hardness while accounting for the non-normal distribution of runtimes.

Solver Behaviour Analysis

To answer RQ2, this experiment measures how puzzle size and structural puzzle characteristics impact solving time and encoding size.

- **Size Analysis:** To isolate the impact of grid dimensions, we use the *size analysis set*, consisting of 100 random puzzle instances per grid size (5×5 to 25×25). We report the growth in solving time and encoding size as dimensions increase.
- **Characteristics Analysis:** To isolate structural effects from size scaling, we use the *characteristics set*, comprising 1000 puzzles at a fixed grid size of 10×10.

For each puzzle characteristic defined in Section 4.3, we partition the instances into three groups based on their quartile distribution: *Low* ($x < Q_1$), *Medium* ($Q_1 \leq x \leq Q_3$), and *High* ($x > Q_3$). We then apply the *Mann-Whitney U test* to compare the *Low* and *High* extremes against the *Medium* baseline to determine if structural deviations significantly impact performance. We consider $p < 0.05$ to be significant. This test is used here because the groups consist of independent, unpaired samples (distinct puzzle instances) and the test is robust to both non-normal distributions and the unequal sample sizes resulting from our quartile-based partitioning.

Paradigm Benchmarking

To answer RQ3, we benchmark our solver against the other solvers developed by the research group for alternative paradigms. For this comparison, we select the best-performing strategy by analysing our previous results.

For testing, we use a python script that tracks the CPU time elapsed between the start and end of the solver. This is done to get accurate measurements between different code languages

For the benchmarking, we use the *stress-test set*, a set of 50 randomly generated puzzle instances per grid size, for sizes 5×5 to 50×50 with a step size of 5. The benchmark comparison focuses on *Scalability*, a comparison of solve time growth as grid size increases, and *Robustness*, evaluating the consistency of the solver across varying puzzle instances. For

any instance that reaches the timeout without a solution, a 20-second penalty is recorded as the completion time.

To analyse the scalability, we use the mean solving time per grid size, where a mean solve time of 20 seconds indicates the solver was unable to solve any puzzles of that grid size within the time limit. To measure robustness, we use *Performance Profiles* as proposed by Dolan and Moré [2]. These profiles visualize the robustness of each paradigm by plotting the probability P that a solver's performance is within a factor τ of the best-performing solver for any given instance, the *performance ratio*.

6 Results

In this section, we present the experimental results to address our three research questions. We first analyse the impact of redundant constraints across different connectivity strategies. Next, we examine the solver's scaling behaviour and the impact of specific puzzle characteristics. Finally, we benchmark our solver against external paradigms.

6.1 Impact of Redundant Constraints (RQ1)

This section provides results to answer RQ1 "*What is the impact of adding redundant constraints to our base encoding on solving time?*". We compared our two implementations of the connectivity constraint against versions with a redundant constraint added to it across the *redundant test set*.

Lazy Connectivity Strategy

The impact of redundant constraints on the Lazy strategy is detailed in Table 2. Although the baseline and most redundant constraints failed to solve all the puzzles within the time limit, the White Neighbours (WN) constraint significantly improved the solving capability, enabling a 100% success rate. This gain is likely due to the pruning of the solution space by the WN constraint. By enforcing local white-tile connectivity, the solver generates fewer nogood solutions, reducing the amount of BFS connectivity checks needed. Consequently, we adopted this configuration as the updated baseline for the Lazy Connectivity strategy, hereafter referred to as the LazyWN solver.

When evaluating additional redundant constraints on the LazyWN solver, we found that the impacts of LW, EP, WB, DW and CC on solving time were not statistically significant. PI and SP showed a significant slowdown for every puzzle size. FI, CI, CCH showed a significant improvement in mean solving time, but the actual impact is minimal, with mean decreases of 0.4%, 1.7% and 1% at all grid sizes. However, UC, which showed a significant improvement in solving time, has a mean solve time decrease of 16.4% over all grid sizes, and a 93.7% mean decrease for $n = 25$. This likely has the same reason as the impact of WN, UC forces a lot of tiles to be white, making the solution space smaller, and thus the solver comes up with fewer nogood solutions.

Distance Connectivity Strategy

During the experiments on the Distance solver, we found that LW and FI both did not show a statistically significant impact on solving time. While CCH yielded a significant result, it actually introduced a 2.82% slowdown in performance over all grid sizes. Among the constraints that significantly improve solving time, the actual mean impact was modest. Several constraints, including CI, CC, DW, PI and SP, clustered around a 4.8% to 4.9% mean decrease in solving time for all grid sizes. WB offered the smallest significant decrease at 1.78%. The Distance strategy benefits less from the constraints than the Lazy strategy does; this is likely because while the redundant constraints prune the search space for black-tile selection, the solver must still assign roots and calculate distances for all white tiles. This suggests the Distance solver maintains a high computational baseline regardless of the pruned black-tile options.

6.2 Solver Behaviour Analysis (RQ2)

This section provides results to answer RQ2: " *What impact do the puzzle size and puzzle characteristics have on encoding size and solve time?*".

Impact of Puzzle Size

Using the *size analysis set*, we measured the solving time and encoding size as the grid dimension n increases from 5 to 25.

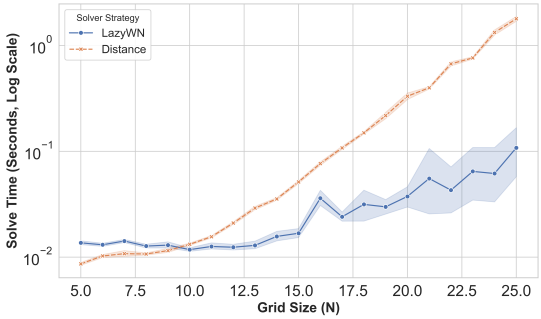


Figure 3: Mean solving time in seconds (log) per grid size for size analysis set. Shaded regions represent the standard deviation.

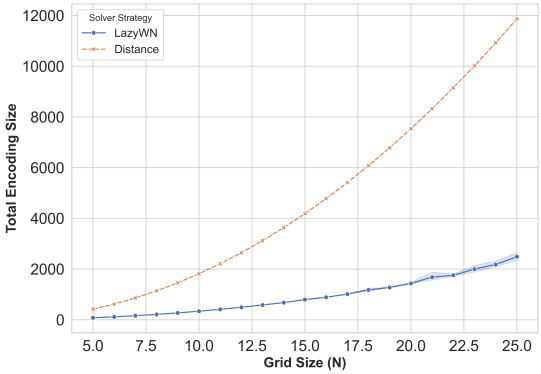


Figure 4: Encoding size (Variables + Clauses) per grid size for size analysis set. Shaded regions represent the standard deviation.

Figures 3 and 4 demonstrate a strong correlation between encoding size and solving time. While the Distance strategy provides predictable solve times with low variance, it suffers from an encoding overhead that scales aggressively with grid

Table 2: Impact of redundant constraints on Lazy strategy solving capability for puzzles from redundant tests set, ordered by Max solved grid size.

Metric	**WN**	UC	WB	PI	LW	CC	CI	*Base*	EP	FI.	DW	CCH	PS
Max Solved Grid Size (N)	**25**	23	21	17	17	16	16	*16*	15	15	15	15	15
Solved (%)	**100.0**	65.1	42.1	37.7	37.1	39.6	37.0	*36.8*	38.5	38.2	37.7	37.6	37.5

Table 3: Distribution of puzzle characteristics across characteristic set. Groups are defined by quartiles: Low ($< Q1$), Medium ($Q1 - Q3$), and High ($> Q3$).

	Thresholds		**Group Size [N (%)]**		
Char.	**Q1**	**Q3**	**Low**	**Medium**	**High**
BCC	121	124	208 (20.8%)	599 (59.9%)	193 (19.3%)
AD	21	27	209 (20.9%)	561 (56.1%)	230 (23.0%)
NAD	184	196	224 (22.4%)	528 (52.8%)	248 (24.8%)
TAD	0	0	0 (0.0%)	790 (79.0%)	210 (21.0%)

size. This is explained by the rapid growth in variables required to represent the root and distance values; for each size increase, the number of decision variables added expands the solution space significantly. In contrast, LazyWN maintains an encoding size that is 80% smaller at $n = 25$, which appears to facilitate better scalability in solve times for larger instances, despite showing higher variance in performance across individual puzzles.

Impact of Puzzle Characteristics

To isolate structural difficulty from size, we analysed the performance of our solver strategies on the *characteristics set*. The distribution for each of the puzzle characteristics in the set can be found in Table 3. For most characteristics, we can see a distribution of approximately 50-60% of puzzles falling into the Middle category. However, TAD shows a norm of 0, meaning that there is no low group to analyse.

Analysis of the LazyWN solver identified NAD as the only characteristic statistically significantly impacting solve time, with the high-count group achieving a 47.38% mean decrease in solving time over the medium group. NAD also significantly altered the encoding size with a 9.61% mean increase for Low and a 6.06% mean decrease for High. This is likely because high NAD counts spread out duplicate values, meaning that decisions on one part of the grid can have an impact on distant tiles, allowing the solver to propagate constraints more effectively.

The Distance solver proved to be more robust but less reactive. For BC, we saw a statistically significant increase in solving time for High and Low groups when compared to the medium group, with a 1.26% and 6.39% mean increase respectively. This indicates that the Distance strategy is sensitive to the stability of the black-tile count. For low counts, an explanation could be that because if there are more white tiles, there are more roots to choose, increasing solve time for root and distance assignment. With a higher black count a reason could be that, because there will likely be many black tiles diagonal of each other, it is easier for the solver to create islands, which would require more backtracking. The overall influence of the characteristics on encoding size remained minimal, with BC, AD and NAD showing significant impact, but the mean changes being below 1%.

6.3 Paradigm Benchmarking (RQ3)

To answer RQ3 "*How does Pumpkin compare to different paradigms at solving Hitori puzzles in solving capability and time?*". We compare the best configuration of our solver against the solvers from the other paradigms. When analysing the results from our previous sections, we can see that LazyWN, although less stable, is the fastest of the two solvers. As UC had the biggest impact on solve time out of the redundant constraints applied to LazyWN, we add this to the solver used for our benchmarking.

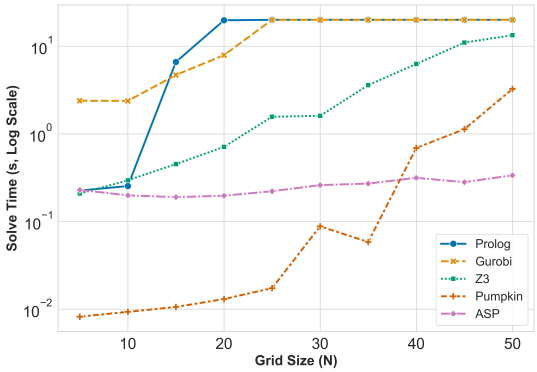


Figure 5: Mean solving time in seconds (log) per grid size for Stress-Test Set using script reported solve time.

Comparing Figures 5 and 6 reveals that the script-reported solve time introduces significant bias. Our solver, as compiled binary, incurs negligible startup time. In contrast, the other solvers face substantial overhead from Python initialization and license verification. To ensure a fair comparison, we will instead use the internal solve time reported by each solver. For our solver, this means the measured time as explained in Section 5.2. For the other solvers, this corresponds to the measured CPU time, excluding file I/O operations. While this provides a more accurate reflection of the solve times, it is important to note that the difference in internal clocks introduces a degree of measurement variance. However, we believe the removal of the overhead outweighs the potential skew of internal timing differences.

Figures 6 and 7 illustrate solver performance across grid sizes ranging from 5 to 50.

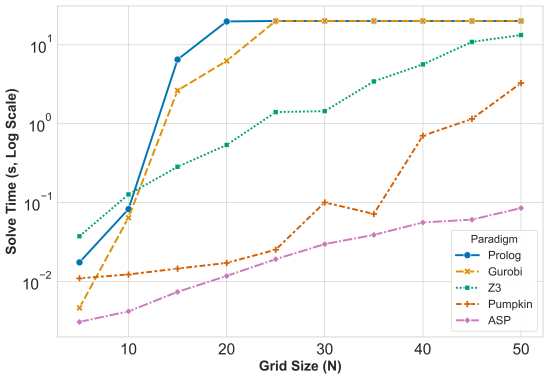


Figure 6: Mean solving time in seconds (log) per grid size for Stress-Test Set using solver reported solve time.

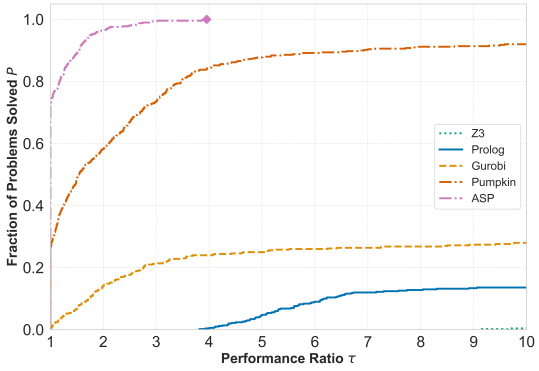


Figure 7: Performance profile of the internally reported solve times, showing the probability (vertical axis) that a solver is within a certain ratio of the fastest solve time (horizontal axis).

The ASP solver consistently outperforms all other solvers, with a mean solving time for $n = 50$ of less than 0.1 seconds, and a maximum performance ratio of $\tau = 4$; this indicates that its solving time is, at most, four times slower than the fastest solve time of a puzzle. The dominance of ASP can be attributed to its efficient handling of connectivity rules during its grounding phase, avoiding the iterative overhead of our Lazy approach.

Our LazyWN solver also demonstrates competitive performance, having a mean solving time for $n = 50$ of 3.3 seconds. It also achieves the fastest solve time in approximately 30% of puzzle instances, and 80% of its solving times fall within a performance ratio of $\tau = 4$. However, the LazyWN performance curve plateaus at 90%, indicating that for the remaining 10% of puzzle instances it either timed out or exceeded a performance ratio of $\tau = 10$.

The remaining solvers struggle with performance. Prolog and Gurobi display poor scalability, failing to solve any instances for grid sizes $n \geq 20$ and $n \geq 25$ respectively. Consequently, their performance profiles flatline early, with Gurobi exceeding a performance ratio $\tau = 10$ at around 30% and Prolog around 10%. In contrast, while Z3 is able to solve puzzles up to $n = 50$, it is consistently orders of magnitude slower than the leaders, its performance profile only appears at $\tau > 9$, showing it is never within competitive range of the other solvers.

7 Conclusion and Future Work

In this research, we explored the suitability of the Constraint Satisfaction Problem (CSP) paradigm for solving Hitori puzzles. By implementing a solver using PUMPKIN, we evaluated the paradigm's performance across varying puzzle sizes and characteristics, ultimately finding CSP to be a competitive approach. To facilitate this, we developed a generator proven to produce any unique Hitori puzzle instance, which served as the foundation for our evaluation and cross-paradigm benchmarking.

We investigated the impact of redundant constraints on solving performance, demonstrating that they yield a substantial positive impact on the Lazy connectivity strategy and a minor but consistent improvement for the Distance strategy. Furthermore, our analysis of scaling behaviour revealed a distinct trade-off: while the Lazy strategy scales efficiently with puzzle size, the Distance strategy's performance degrades rapidly as grid size increases. Regarding puzzle characteristics, we found that a high frequency of non-adjacent duplicates significantly accelerates the LazyWN solver, whereas the Distance strategy is sensitive to deviations in the count of black tiles.

Finally, our benchmarking establishes PUMPKIN as a top-tier contender, securing the second-fastest solving times and outperforming LP, Prolog, and SMT implementations. While ASP remains dominant, our results prove that a well-optimized CSP approach is a robust and viable alternative for modelling and solving Hitori.

Future research into CSP could focus on optimizing connectivity, either by refining the Lazy and Distance strategies presented here or by exploring entirely new strategies. Additionally, further investigation is required to identify other structural puzzle characteristics that may govern solving difficulty. Finally, replicating these experiments with CSP solvers other than PUMPKIN would help verify the generalizability of our findings and isolate solver-specific performance.

Future research could expand the scope of this study by evaluating additional paradigms on Hitori and comparing them to our established benchmarks. In addition, a more in-depth analysis of current paradigms is required to pinpoint the specific structural areas where each approach excels.

Future research should also aim to establish a standardized, balanced benchmarking library that comprehensively covers the full spectrum of Hitori's structural characteristics. Furthermore, the benchmarking methodology requires refinement to mitigate artifacts such as solver initialization overhead, ensuring a more equitable comparison between compiled and interpreted solving environments.

8 Responsible Research

In alignment with the ethical guidelines of our course and the broader research community, we have taken specific steps to

ensure the transparency, reproducibility, and integrity of our work.

8.1 Reproducibility

To ensure that our findings are reproducible, all artifacts required to replicate our experiments are publicly available. The source code for the solver and puzzle instances used in this research can be found in our repository.[1] The source code for the puzzle generator, benchmarking setup and solution checker can be found in our shared research group repository [2]. Both of these repositories contain an MIT license so that our code can be used in future experiments.

We have included a `README` file that details:

- Instructions on how to run the solver, as well as examples.
- Instructions on how to run a benchmark.
- A list of all third-party libraries used, all containing licences (MIT, Apache 2.0), ensuring our right to use and analyse these tools.

All of the used repositories accessed were open source, and used both MIT and Apache 2.0 Licenses. These licenses allow the source code to be freely distributed, as long as the copyright notices in the code are retained.

- Rust-Language Repository [3]
- Pumpkin Repository [4]

8.2 Generative AI Usage

In this research, we utilized Generative AI (Google's Gemini 3 Pro model [5]) as a supportive tool to enhance the quality and efficiency of our work. Our usage was strictly limited to the following areas:

- **Syntax Engine:** Since Rust's naming convention differs from other coding languages, we used AI as a descriptive search engine for the documentation. (e.g., "How do I parse an array from a comma separated string").
- **Technical Troubleshooting:** We utilized the tool to assist in debugging software errors.
- **Scripting:** We let AI generate boiler plate code for files that we needed (e.g., "Generate a rust file that accesses a file and calls a function on every line. It has a 2D array as output.").
- **Writing Assistance:** We used AI-assisted tools to improve the grammar, flow, and readability of the final text (e.g., "How do I make the sentence 'test the strengths and weaknesses of lazy and distance strategies and the strengths and weaknesses of redundant constraints' flow nicely without repeating the same sentence").

We never copied AI-generated answers without critical review, nor did we present AI-hallucinated facts as our own. All claims, definitions, and code logic suggested by the model were cross-referenced with official documentation and primary literature to ensure accuracy. We retain full responsibility for the content of this paper.

[1] https://github.com/LesleySmits/Pumpkin-Hitori-Solver/tree/50cd4dd1897003970b9adc02dab8cb4f553d8c80

[2] https://github.com/sappho3/Thesis-Hitori-shared/tree/2a5b61ce05c3ff93de09e61e64e8952a4556940f

[3] https://github.com/rust-lang/rust

[4] https://github.com/ConSol-Lab/Pumpkin

[5] https://aistudio.google.com/models/gemini-3

Acknowledgments

I would like to thank the research team for their collaboration in developing the shared repository and the theoretical frameworks utilized in this paper. Specific collaborative outputs, including the proof of generator completeness (Appendix B) and the shared redundant constraints (Appendix C), are detailed in the Appendices. Further information regarding individual contributions can be found in the Shared Project Contributions (Appendix A).

References

[1] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, December 1999.

[2] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *CoRR*, cs.MS/0102001, 2001.

[3] Blowey Dumas et al. Reasoning about connectivity constraints. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 2015.

[4] Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 352–366, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] Maarten Flippo, Konstantin Sidorov, Imko Marijnissen, Jeff Smits, and Emir Demirović. A multi-stage proof logging framework to certify the correctness of CP solvers. In Paul Shaw, editor, *30th international conference on principles and practice of constraint programming (CP 2024)*, volume 307 of *Leibniz international proceedings in informatics (lipics)*, pages 11:1–11:20, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969 tex.urn: urn:nbn:de:0030-drops-206969.

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

[7] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, August 2012.

[8] Vegard Hanssen. Menneske.no.

[9] Robert A. Hearn. *Games, Puzzles, and Computation*. CRC Press LLC, Florida, 1st ed edition, 2009.

[10] John N Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

[11] Helmut Simonis. Sudoku as a constraint problem. 2005.

[12] Roos Wensveen. Solving, generating and classifying hitori. Master's thesis, Leiden Institute of Advanced Computer Science (LIACS), Leiden University, August 2024.

A Shared Project Contributions

This research was developed as part of a collaborative project. While this manuscript is the work of the author, the puzzle instance generation, benchmarking software and theoretical proofs were developed collectively. The distribution of contributions of this collective development are:

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 Generator Proof

Below, we prove that our generator can generate only any single-solution Hitori puzzle. We do so in three 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. In the last step we put everything together to prove the following theorem:

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

Algorithm 1 Algorithm that exactly any valid Hitori instance H

```
1: function GENERATEHITORIINSTANCE
2:     Let S = GENERATESOLUTIONTOPOLOGY
3:     Let H = GENERATEHITORIINSTANCE(S)
4:     while H is not uniquely solvable do
5:         H = GENERATEHITORIINSTANCE(S)
6:     end while
7:     return H
8: end function
```

B.1 Generating Solution Topologies

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 topology, 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 adjacency and uniqueness constraints defined by the Hitori rules.

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.

```
1: function GENERATESOLUTIONTOPOLOGY
2:     Let S[1,...,n][1,...,n] be the two-dimensional array of tiles, all unmarked
3:     Let C be the collection of all coordinates in S ((1,1),(1,2),...(1,n),(2,1),...,(n,n)) in random order
4:     for i = C[1] to C[n^2] do
5:         if no orthogonally adjacent tile is marked then
6:             S[i] = marked
7:             if the unmarked tiles of S are disconnected then
8:                 S[i] = unmarked
9:             end if
10:        end if
11:    end for
12:    return S
13: end function
```

Lemma B.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 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 B.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. Marking any of the tiles in M does not violate the adjacency or connectivity constraints, and as such 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 after already marking the tiles in M, 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 B.2 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. □

B.2 Generating Hitori instances

A puzzle instance of Hitori H is an $n \times n$ grid of numbers where each element $H_{i,j} \in [1..n]$ with $i, j \in [1..n]$. 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
    FillUnmarkedTiles(H, S, n, 1)
    FillMarkedTiles(H, S, n, 1)
    return H
end function
```

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

```
function FillUnmarkedTiles(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 FillUnmarkedTiles(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 FillUnmarkedTiles(H, S, n, k + 1)
end function
```

Lemma B.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 create 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 B.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. □

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

```
1:  function FillMarkedTiles(H, S, n, k)
2:      Let i = ⌈k/n⌉
3:      Let j = ((k − 1) mod n) + 1
4:      if k > n^2 then
5:          return true
6:      else if S[i][j] == unmarked then return FillMarkedTiles(H, S, n, k + 1)
7:      else
8:          Let row be the numbers used in the unmarked tiles of the row of H[i][j]
9:          Let col be the numbers used in the unmarked tiles of the column of H[i][j]
10:         C = row ∪ col
11:         shuffle C
12:         H[i][j] = c[1]
13:                                          ▷ Assign H[i][j] the first element in C
14:     end if
          return FillMarkedTiles(H, S, n, k + 1)
15: end function
```

Lemma B.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 of unmarked tiles 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 B.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 B.8. *Given a valid solution topology S, Algorithm 3 can generate any valid puzzle instance H.*

Proof. Lemma B.4 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 B.5 proves that we can generate nothing but valid combinations of numbers.

Lemma B.6 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 B.7 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. □

B.3 Proving Theorem B.1

Proof. Lemma B.3 has proven that our algorithm can generate exactly any valid solution topology S, and Lemma B.8 has proven that, given any valid solution topology S we can generate exactly only any valid puzzle instances H. In the last part of Algorithm 1 we keep generating new instances H from S until we have found one that is uniquely solvable. Once we have found such an H, we return it.

Given this, we know that the generator can only return uniquely-solvable valid instances H, and as such we have proven Theorem B.1 □

C Redundant Constraints

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).

White Neighbours (WN)

For every puzzle with $n > 1$, every white cell has **at least** one white neighbour.

Corner Close (CC) [12]

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

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

(Top-right corner)

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

(Bottom-left corner)

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

(Bottom-right corner)

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

Corner Check (CCH) [12; 6]

Generalisation of Triple Corner and Quad Corner. Is visualised in Figure 8. Only applicable in corners of puzzles of size $n > 2$.

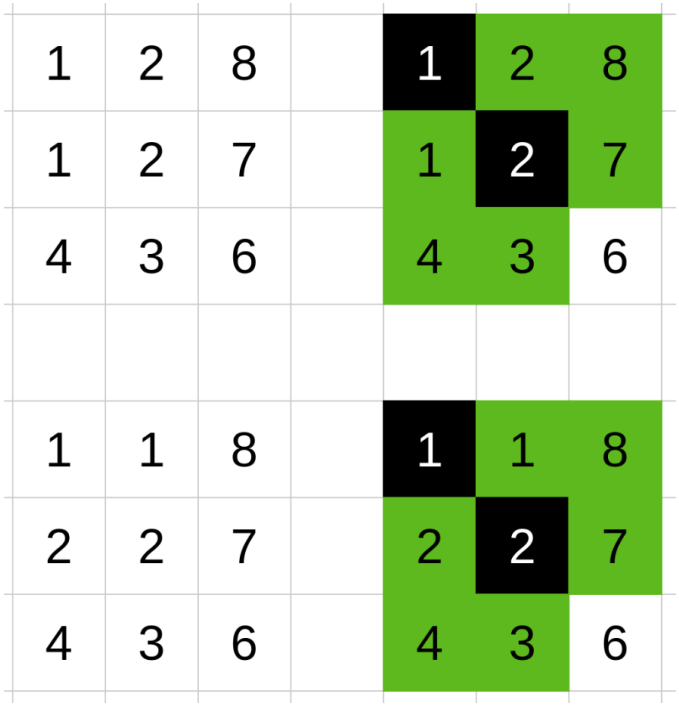


Figure 8: Pattern 9 and implication

Sandwich Pair (SP) [6; 12]

Similar to the Sandwich Triple, but now the centre tile has a different symbol than the sandwiching tiles. The sandwiched tile is always white.

Edge Pair (EP)

This constraint is a generalisation of the fourth and fifth pattern described by Gander and Hofer [6], it can be seen in 9.

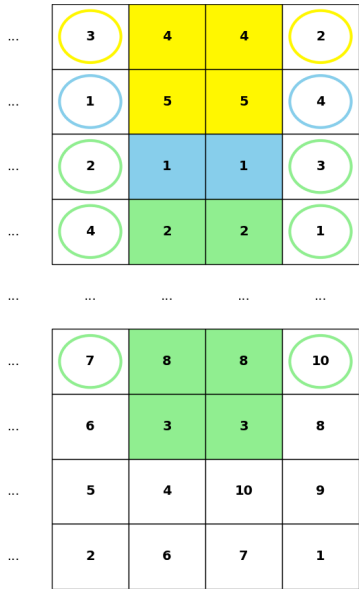


Figure 9: Edge pair constraint, yellow shows DEP, blue shows TEP and green shows how this works for longer “edge pair blocks”.

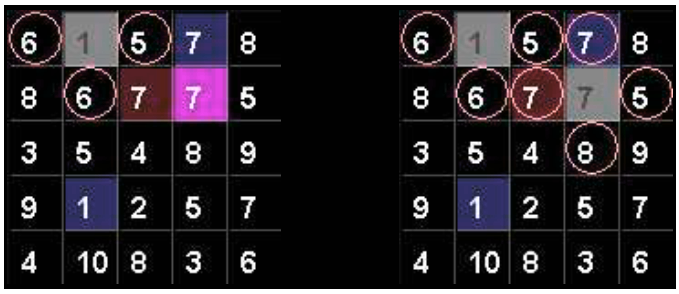


Figure 10: Pattern 8 and implication

Flanked Isolation (FI)
If there are two adjacent pairs, any other value must be black [8].

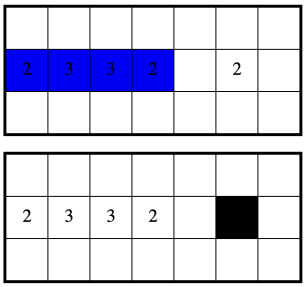


Figure 11: Flanked Isolation

Least Whites (LW)
for every row and column, there are **at least** $\lfloor \frac{n}{2} \rfloor$ white cells. The logic used to implement this also implies *Black Count (BC)*: For every row and column, there are **at most** $\lceil \frac{n}{2} \rceil$ black cells.

Unique Cell [12]
If a tile's number is unique in its row and column, set it to white. This can be done dynamically, in which case cells marked black are ignored when checking uniqueness.

Pair Isolation (PI) [6; 12]
The sixth pattern as described by Gander and Hofer [6] and Wensveen [12], visualized in Figure 12. This pattern is applicable anywhere in a puzzle of size $n > 3$ for both rows and columns.

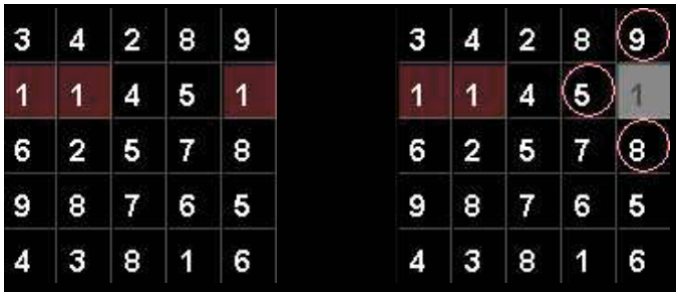


Figure 12: Pattern 6 and implication

Close Isolation (CI)
The seventh pattern as described by Gander and Hofer [6], visualized in Figure 13. This pattern is applicable anywhere in a puzzle of size $n > 4$ for both rows and columns.

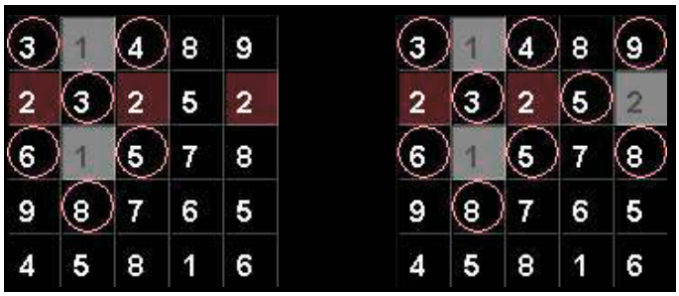


Figure 13: Pattern 7 and implication

White Bridges (WB)
For every pair of adjacent rows, there exists at least one pair of orthogonally adjacent white cells.

Diagonal Wall (DW)
For any full diagonal D, meaning a diagonal that touches both sides it spans, spanning the board with size $n > 1$, the following constraint holds:

$$\exists (i, j) \in D \text{ such that } (i, j) \text{ is white.} \tag{1}$$