Eating Soup with a Fork: Solving Hitori with Integer Linear Programming

An Investigation into the Suitability of Integer Linear Programming for Solving Single-Solution Hitori Instances

Sophieke van Luenen[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: Sophieke van Luenen
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L. D. Latour, Dr. Tim Coopmans

Abstract

We study the advantages and disadvantages of using Integer Linear Programming for solving instances of the NP-complete puzzle Hitori. We do so as part of a larger comparison between different modelling-and-solving (M&S) paradigms that aims to find what solving techniques do (not) work for different paradigms so as to guide effective modelling. To this end we create three models: a *path* model that is quartic with regards to instance size, a quadratic and probabilistic *optimised naive* model, and an exponential *duplicates* model which uses proven properties of the game. We measure their runtime performance with and without different optimisations on 1000 10×10 Hitori instances, compare it to the runtime performance of models in four other M&S paradigms, and correlate properties of puzzle instances to runtime. We find that there is no performance difference between our optimised naive and duplicates models though our path model is significantly slower. Adding redundant constraints tends to slow models down. Our optimised naive model performs better than a Logic Programming model, but worse than a Satisfiability Modulo Theories, a Constraint Satisfaction Problem, and an Answer Set Programming model. Runtime correlates strongly with the number of non-unique tiles in the problem instance, though the correlation's direction differs per model. We conclude that ILP is an unintuitive and slow tool for solving Hitori instances, especially when compared to other M&S paradigms.

1 Introduction

Modelling-and-solving (M&S) paradigms are powerful tools in which users model a problem as a set of variables and rules, and a solver then tries to find an assignment of those variables which satisfies those rules. In M&S, the way one models a problem influences how well one can solve it [1], and that is where the motivation for our study lies: we investigate what solving techniques (do not) work well when applying Integer Linear Programming (ILP) to Hitori. We do so as part of a larger study in which five colleagues model Hitori in five M&S paradigms in order to compare solving strategies between different paradigms. This is in response to a recent Dagstuhl seminar [2] which reaffirmed the need for different M&S communities to create more cross-paradigm interactions in order to keep up rapid progress on M&S research. It's also in response to a call for more comparisons between M&S paradigms by Çoban et al. [3].

This study investigates ILP (though other paradigms were considered, for more information see appendix A) and uses the *Gurobi* solver to do so. We chose Gurobi because it is very popular[1] and as such we are interested in its strengths and weaknesses. We define two research questions:

[1]They claim 80% market share, see [4]

Research question 1: How fast can we solve Hitori puzzles using the Integer Linear Programming solver Gurobi?

Research question 2: What properties of Hitori instances influence the speed at which the Gurobi models can solve the instance?

The rest of this work is organised as follows: we first examine relevant prior work on this topic. Then in the background section we detail ILP, the rules of Hitori, we provide a formal definition of Hitori, and prove three properties of the game. We define our three models and seven redundant constraints in our approach section and elaborate on our research questions and experiment parameters in the experimental setup. We show what modelling-decisions and instance properties affected performance in our results which we reason about in our discussion. Finally, we conclude that ILP is comparatively slow in solving Hitori, though the properties that slow our models down depend on the encoding we use, and provide ideas for further research in our conclusion.

2 Prior Work

Wensveen [5] has looked at Hitori before, identifying solving techniques and using them to create a difficulty classification for instances. They also created a 2-SAT solver, and a rules-based solver that rely solely on human-solving strategies such as those described in Section 4.4. Our work will build upon the solving rules defined in their work, and their method of encoding the connectivity constraint. Gander and Hofer [6] also created two Hitori solvers: a SAT-based one similar to Wensveen, and an imperative one which relies on other solving patterns. They use a different method to model the connectivity constraint which inspires one of our models.

Beyond Hitori there is a large number of studies that consider solving other logic problems with M&S paradigms. For instance, Crawford et al. [7] and Bartlett et al. [8] both solve Sudoku using *Constraint Satisfaction Problems* (CSP) and ILP respectively, though no performance was reported on the latter, so no comparison can be made. Smith et al. [9] solve the Progressive Party Problem using both ILP and CSP and compares both paradigms. They find CSP more intuitive to model the problem in, and the resulting model to be faster.

Çoban et al. [3] compare ASP with the ILP/Constraint Programming solver ILOG OPL. On their specific problem set they find ASP consistently faster, though both solvers have their merits in their language's expressiveness. For future research they recommend extending comparisons to different problems.

3 Background

In this section we first describe ILP. We then move on to describing the rules of Hitori after which we give a formal description of Hitori. Last, we prove three properties of the game that we use in one of our models.

3.1 Integer Linear Programming

ILP is an M&S paradigm where users model their problems through integer variables such as $x_1 \in [1..10]$ and

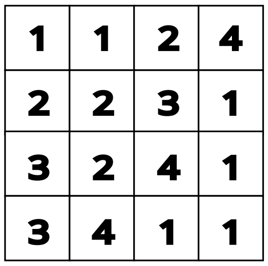
$x_2 \in [-4..2]$ and through rules as linear (in)equalities such as $x_1 \leq x_2$ and $x_2 \geq -3$. Optionally, they may add an objective function such as $\text{maximise}(x_1 - x_2)$ or $\text{minimise}(x_1 + x_2)$. The Gurobi solver then uses a *simplex algorithm* (for more information see Dantzig [10]) to find an assignment of values to the defined variables that optimises the objective function. If no objective function is given, the model returns when it has found any assignment that satisfies all rules.

3.2 The Rules of Hitori

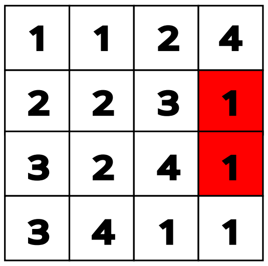
Hitori is an NP-complete puzzle [11] where the player is given an $n \times n$ grid with each tile containing a number between 1 and n inclusive. The player is tasked with marking tiles such that three constraints hold:

- Uniqueness constraint: no unmarked number appears twice in its row and column,
- Adjacency constraint: no marked tiles are orthogonally adjacent,
- Connectivity constraint: all unmarked tiles are connected through an orthogonal path.

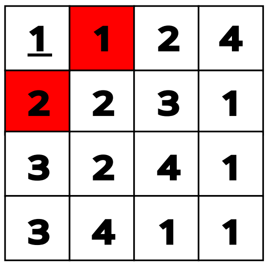
We visualise these rules in Figure 1. On top of these rules, we only consider Hitori instances with exactly one solution. This is the most common version of the puzzle [5, 12] and it also removes trivial instances from our problem-space.



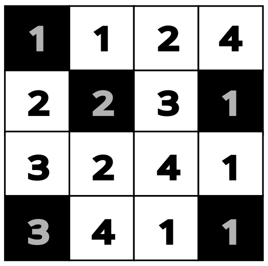
(a) An unsolved Hitori board.



(b) The adjacency constraint forbids two adjacent marked tiles.



(c) The connectivity constraint requires all unmarked tiles to be orthogonally connected.



(d) A solved Hitori board.

Figure 1: A visual explanation of the rules of Hitori.

3.3 Formal Description of Hitori

An n-Hitori instance can be described as an $n \times n$ matrix $\mathbf{H}$ where variable $\mathbf{H}_{i,j}$ corresponds to the tile at the i^{th} row and j^{th} column of the instance and shares its value. A solution $\mathbf{S}$ is then another $n \times n$ matrix consisting of 1s and 0s where $\mathbf{S}_{i,j} = 0$ means the tile on the i^{th} row and j^{th} column is marked, and a $\mathbf{S}_{i,j} = 1$ means it is not.

The adjacency constraint is described as

$$\forall_{i\in[1..n]j\in[1..n-1]}\mathbf{S}_{i,j} + \mathbf{S}_{i,j+1} \geq 1$$
$$\forall_{i\in[1..n-1]j\in[1..n]}\mathbf{S}_{i,j} + \mathbf{S}_{i+1,j} \geq 1$$

For the uniqueness constraint we define n^3 auxiliary variables $t_{i,j,k} \in [0,1]$ where $i, j, k \in [1..n]$. $t_{i,j,k} = 1$ iff $\mathbf{H}_{i,j} = k$ and 0 otherwise. Using these variables we describe the constraint as

$$\forall_{i,k} \sum_{j=1}^{n} t_{i,j,k} \cdot \mathbf{S}_{i,j} = 1$$
$$\forall_{j,k} \sum_{i=1}^{n} t_{i,j,k} \cdot \mathbf{S}_{i,j} = 1$$

The connectivity constraint requires a different approach where we encode the solution $\mathbf{S}$ as a graph $\mathcal{C}$ with a set of vertices V and a set of edges E. To create V we add a vertex v for each field in $\mathbf{S}$ where $\mathbf{S}_{i,j} = 1$. $v_{i,j}$ is then a vertex created from $\mathbf{S}_{i,j}$. We then create E by adding an edge between all vertices created from orthogonally adjacent fields in $\mathbf{S}$:

$$\forall_{i\in[1..n]j\in[1..n-1]}\mathbf{S}_{i,j} + \mathbf{S}_{i,j+1} = 2 \iff \{v_{i,j}, v_{i,j+1}\} \in E$$

$$\forall_{i\in[1..n-1]j\in[1..n]}\mathbf{S}_{i,j} + \mathbf{S}_{i+1,j} = 2 \iff \{v_{i,j}, v_{i+1,j}\} \in E$$

The connectivity constraint is then satisfied if there is a path from every node n to every other node n' in $\mathcal{C}$, or in other words, if there is only one connected component in $\mathcal{C}$. Last, since we only consider uniquely-solvable instances of Hitori, if multiple different instances of $\mathbf{S}$ satisfy the above three constraints for $\mathbf{H}$, $\mathbf{H}$ itself was not a valid instance.

3.4 Properties of Hitori

Below we provide proofs for three properties: First, a *unique tile* (one with a number that is unique in its row and column) will never be marked. Second, if we forbid the marking of all tiles in a cycle in the instance graph, the connectivity constraint will be satisfied. Third, as a corollary, we prove that instance graphs do not need to contain unique tiles.

Theorem 3.1. *Given a Hitori instance* $\mathbf{H}$*, a tile* $\mathbf{H}_{i,j}$ *that is unique in its row* i *and column* j *will never be marked.*

Proof. We will use a proof by contradiction. Let $\mathbf{S}$ be the solution of $\mathbf{H}$, and assume that $\mathbf{H}$ contains a tile $\mathbf{H}_{i,j}$ that is unique in its row and column, i.e. $\mathbf{H}_{i,j} \neq \mathbf{H}_{k,j}$ for $k \in [1..n]$, $k \neq i$, and $\mathbf{H}_{i,j} \neq \mathbf{H}_{i,k}$ for any $k \in [1..n]$, $k \neq j$, and that tile is marked, i.e. $\mathbf{S}_{i,j} = 0$.

We will now use this assumption to create a contradiction by showing that we can create a second valid solution for this instance $\mathbf{H}$, which is impossible for any of the Hitori instances we consider. We do so by setting $\mathbf{S}_{i,j}$ to 1.

Changing $\mathbf{S}_{i,j}$ to 1 cannot break the uniqueness constraint as $\mathbf{H}_{i,j}$ is unique in its row and column by its definition. It also does not break the adjacency constraint: since

$\mathbf{S}$ is a valid solution, removing a marked tile can never create a solution where two marked tiles are adjacent. The connectivity constraint is also preserved. As $\mathbf{S}_{i,j}$ is marked as by our assumption, we know that all tiles orthogonally adjacent are not. Since we know of all of those tiles that they are part of an orthogonal path connecting all unmarked tiles (as is required by the existence of a valid solution $\mathbf{S}$), when $\mathbf{S}_{i,j}$ is unmarked we can add it to the path simply through its neighbours.

We have now constructed a second valid solution to $\mathbf{H}$ which we know is a contradiction. As such, we can conclude that unique tiles in any given instance $\mathbf{H}$ will never be marked in the corresponding solution $\mathbf{S}$. □

Next we introduce instance graphs. It should be noted that these are distinctly different from the graph $\mathcal{C}$ from our formal description. For each Hitori instance $\mathbf{H}$ there exists an instance graph $\mathcal{G}$ with a set of vertices V and a set of edges E. To create V we add a vertex v for each field in $\mathbf{S}$ where $\mathbf{S}_{i,j} = 0$. $v_{i,j}$ is then a vertex created from $\mathbf{S}_{i,j}$. We add one more vertex to V: b which represents the edge of the board. We then create E by adding an edge between all vertices originating from diagonally adjacent fields in $\mathbf{S}$, that is:

$$\forall_{i\in[1..n-1]j\in[1..n-1]}\mathbf{S}_{i,j} + \mathbf{S}_{i+1,j+1} = 0 \iff \{v_{i,j}, v_{i+1,j+1}\} \in E$$

$$\forall_{i,\in[1..n-1]j\in[2..n]}\mathbf{S}_{i,j} + \mathbf{S}_{i+1,j-1} = 2 \iff \{v_{i,j}, v_{i+1,j-1}\} \in E$$

Last, we add an edge between each marked tile on the edge of the board and b, that is:

$$\forall_{i\in[1..n],j\in[1,n]}\mathbf{S}_{i,j} = 0 \iff \{v_{i,j}, b\} \in E$$

$$\forall_{i\in[1,n],j\in[1..n]}\mathbf{S}_{i,j} = 0 \iff \{v_{i,j}, b\} \in E$$

Theorem 3.2. *Given a Hitori instance* $\mathbf{H}$ *and its graph representation* $\mathcal{G}$*, a solution that forbids all tiles in a given simple cycle in graph* $\mathcal{G}$ *to be marked will guarantee the* connectivity *constraint [6].*

Proof. We will use a proof by contradiction. Take an arbitrary Hitori instance $\mathbf{H}$, its graph $\mathcal{G}$, and its solution with respect to only the first two constraints $\mathbf{S}'$. Fix $\mathbf{S}'$ so that it does not satisfy the connectivity constraint, that is, there is a set of tiles L in $\mathbf{S}'$ which are all unmarked, and orthogonally disjoint from the set K which contains every other unmarked tile in $\mathbf{S}'$. Now assume that for each cycle in $\mathcal{G}$, at least one of the tiles present in that cycle is unmarked, i.e. $\mathbf{S}'_{\text{tile}} = 1$.

There are two features that can make L disjoint from K: marked tiles and the edge of the board. Therefore, for L to be orthogonally disjoint from K it must be orthogonally adjacent to only marked tiles or the edge of the board. Define Q to be the set of features orthogonally surrounding L. Since Q consists solely out of marked tiles and possibly the edge of the board, $\mathcal{G}$ must contain nodes representing all elements within Q. We now define $\mathcal{G}'$ to be the subgraph of $\mathcal{G}$ which consists only of the elements in Q and the edges between them.

We will now create a contradiction by showing that, in order for L to exist, $\mathcal{G}'$ and thus $\mathcal{G}$ must contain a cycle of marked tiles and possibly the board edge.

If there is no cycle of marked tiles and possibly the board edge in $\mathcal{G}'$, then Q does not form a cycle. By its definition, Q is the set of features surrounding L. If Q does not form a cycle in $\mathcal{G}'$, L must be orthogonally adjacent to something that is not a marked tile, or the edge of the board. This must then be an unmarked tile. But this is not possible, because L is disjoint from all other unmarked tiles. As such Q must form a cycle in $\mathcal{G}'$, and thus in $\mathcal{G}$.

We have now shown that $\mathcal{G}$ contains a cycle of nodes that represent marked tiles and potentially the edge of the board, and as such a contradiction has formed. Thus we can now conclude that if $\mathcal{G}$ contains no cycles, the connectivity constraint is satisfied. □

Corollary 3.2.1. *Given a Hitori instance* $\mathbf{H}$ *and its graph representation* $\mathcal{G}$*, the variables in* $\mathbf{H}$ *that are unique in their row and column do not need to be added to* $\mathcal{G}$*.*

Proof. Theorem 3.2 states that only cycles of marked tiles and potentially the edge of the board can break the connectivity constraint. Theorem 3.1 states that unique tiles will never be marked. As such, unique tiles will never be part of a cycle that breaks the connectivity constraint and thus do not need to be mapped in $\mathcal{G}$. □

4 Approach

We are interested in how well we can solve Hitori with Gurobi, and what properties of Hitori influence the solving speed of our models. To this end we define three models and a number of redundant constraints we will use in our experiments.

4.1 Translation to the Path Model

We define an $n \times n$ grid G of binary variables $x_{i,j}$ (with $i, j \in [1..n]$) with domain $[0, 1]$, which represent whether a tile is marked ($x_{i,j} = 1$) or not ($x_{i,j} = 0$). This is swapped from the formal definition, because this allows for more efficient code through the representation of unique tiles in the duplicates model as an integer 0.

We model the adjacency constraint by enforcing

$$\forall_{i\in[1..n],j\in[1..n-1]}x_{i,j} + x_{i,j+1} \leq 1$$
$$\forall_{i\in[1..n-1],j\in[1..n]}x_{i,j} + x_{i+1,j} \leq 1$$

For the uniqueness constraint we create k expressions e_k for each row and column. If a tile in the row/column has $\mathbf{H}_{\text{tile}} = k$, e_k sums x_{tile}. If an expression e_k contains $o > 1$ variables we enforce $o - e \leq 1$.

To model the connectivity constraint we define a new grid P of dimensions $n \times n \times n^2$. If a tile is marked, it will not be on the path, so we define

$$\forall_{i,j\in[1..n]z\in[1..n^2]}P_{i,j,z} \leq 1 - G_{i,j}$$

We define that there may be only one source by forcing $\sum_{i=1}^{n}\sum_{j=1}^{n} P_{i,j,0} \leq 1$ and then build the path away from the source, increasing z by 1 for each tile we traverse. We do so by requiring:

$$\forall_{i\in[2..n-1] z\in[2..n^2]} \quad P_{i,1,z} \leq P_{i-1,1,z-1} + P_{i,2,z-1} + P_{i+1,1,z-1}$$

$$\forall_{i\in[2..n-1] z\in[2..n^2]} \quad P_{i,n,z} \leq P_{i-1,n,z-1} + P_{i,n-1,z-1} + P_{i+1,n,z-1}$$

$$\forall_{j\in[2..n-1] z\in[2..n^2]} \quad P_{1,j,z} \leq P_{1,j-1,z-1} + P_{2,j,z-1} + P_{1,j+1,z-1}$$

$$\forall_{j\in[2..n-1] z\in[2..n^2]} \quad P_{n,j,z} \leq P_{n,j-1,z-1} + P_{n-1,j,z-1} + P_{n,j+1,z-1}$$

$$\forall_{i,j\in[2..n-1] z\in[2..n^2]} \quad P_{i,j,z} \leq P_{i+1,j,z-1} + P_{i-1,j,z-1} + P_{i,j+1,z-1} + P_{i,j-1,z-1}$$

$$\forall_{z\in[2..n^2]} P_{1,1,z} \leq P_{1,2,z-1} + P_{2,1,z-1}$$

$$\forall_{z\in[2..n^2]} P_{n,1,z} \leq P_{n,2,z-1} + P_{n-1,1,z-1}$$

$$\forall_{z\in[2..n^2]} P_{1,n,z} \leq P_{1,n-1,z-1} + P_{2,n,z-1}$$

$$\forall_{z\in[2..n^2]} P_{n,n,z} \leq P_{n,n-1,z-1} + P_{n-1,n,z-1}$$

To count the number of variables on the path we iterate over all tiles in the board and create a new binary variable $s_{i,j}$ and expression $e_{i,j} = \sum_{z=1}^{n^2} P_{i,j,z}$. We enforce $s_{i,j} \leq e_{i,j}$. The number of tiles on the path is now $l_1 = \sum_{i=1}^{n} \sum_{j=1}^{n} s_{i,j}$. We also create an expression counting the number of marked tiles $l_2 = \sum_{i=1}^{n} \sum_{j=1}^{n} G_{i,j}$. In order to ensure all unmarked tiles are on the path we require $l_1 + l_2 = n^2$.

4.2 Translation to the Naive Model

The naive model enforces the adjacency and uniqueness constraints in the same way as the path model. It does not have an encoding for the connectivity constraint. Instead it returns proposed solutions which are checked by a separate breadth-first search algorithm. If a proposed solution does not satisfy the connectivity constraint we try again after forbidding the proposed solution as follows: Define new decision variable v, two expressions $e_{\text{marked\_tiles}}$ and $e_{\text{unmarked\_tiles}}$ which sum the decision variables of all marked and unmarked tiles respectively, value l which is the number of marked tiles in the proposed solution, and $m \gg n$. Enforce $e_{\text{unmarked\_tiles}} \geq 1 - (v \cdot m)$ and $e_{\text{marked\_tiles}} \leq l - 1 + ((1 - v) \cdot m)$

We include the naive model in our experiment because we are curious how a smaller, probabilistic model compares with our two more involved models. We expect that this model will perform comparatively fast on smaller problem instances.

4.3 Translation to the Duplicates Model

The duplicates model is named so because its constraints only consider non-unique (duplicate) tiles. Similar to the path translation we define an $n \times n$ grid, though now only with binary variables $x_{i,j}$ on those locations where $\mathbf{H}_{i,j}$ is a non-unique value (given again $i, j \in [0..n-1]$). We model the uniqueness and adjacency constraints the same as in the path model, though using only the variables we have defined. Unique tiles are thus not considered in these constraints.

We then use Theorem 3.2 to model the connectivity constraint. For each cycle in $\mathcal{G}$ we create an expression e summing its constituent variables. We define value l as the number of variables in e, and enforce $e \leq l - 1$.

Table 1 shows an overview of our three models and their main properties.

Model	Has connectivity constraint	Uses Hitori properties
Path	Yes	No
Naive	No	No
Duplicates	Yes	Yes

Table 1: Our three models and their properties.

4.4 Redundant Constraints

This section defines seven redundant constraints that provide hints to the solver. Since we cannot change the solver itself, we are limited to constraints that rely only on the initial board. Appendix B provides a visualisation of these constraints.

Corner Close [5]: The connectivity constraint dictates $x_{0,1} = 1 \implies x_{1,0} = 0$ and $x_{1,0} = 1 \implies x_{0,1} = 0$. The same holds for $x_{n-2,0}$ and $x_{n-1,1}$, $x_{0,n-2}$ and $x_{1,n-1}$, as well as $x_{n-2,n-1}$ and $x_{n-1,n-2}$.

Corner Check [5,6]: Given $\mathbf{H}_{0,0} = \mathbf{H}_{0,1} \wedge \mathbf{H}_{1,0} = \mathbf{H}_{1,1}$, or $\mathbf{H}_{0,0} = \mathbf{H}_{1,0} \wedge \mathbf{H}_{0,1} = \mathbf{H}_{1,1}$, all three Hitori constraints together dictate $x_{0,0} = x_{1,1} = 1 \wedge x_{0,1} = x_{0,2} = x_{1,2} = x_{1,0} = x_{2,0} = x_{2,1} = 0$.

Most Blacks: Given the adjacency constraint we know that at most $\lceil \frac{n}{2} \rceil$ tiles can be marked in any row or column.

Least Whites: The inverse of most blacks, determines that at least $\lfloor \frac{n}{2} \rfloor$ tiles must be unmarked in any row or column.

Sandwiches [5,6]: Covers both Sandwich Pair and its more specific case Sandwich Triple. Given $\mathbf{H}_{i,j} = \mathbf{H}_{i+2,j}$ then $x_{i+1,j} = 0$. If $\mathbf{H}_{i,j} = \mathbf{H}_{i+1,j}$ as well, then we also know that $x_{i,j} = x_{i+2,j} = 1$. The same goes columnwise.

Pair Isolation [5,6]: Given $\mathbf{H}_{i,j} = \mathbf{H}_{i+1,j}$ and $\mathbf{H}_{k,j} = \mathbf{H}_{i,j}$ with $k \in [1..i-2] \cup [i+3..n]$, then the uniqueness and adjacency constraints require $x_{k,j} = 1$. The same goes columnwise.

Edge Pairs [5,6]: Given $\mathbf{H}_{i,1} \neq \mathbf{H}_{i+1,1} = \mathbf{H}_{i+2,1} \neq \mathbf{H}_{i+3,1} \wedge \mathbf{H}_{i+1,2} = \mathbf{H}_{i+2,2}$, then the three Hitori constraints require $x_{i,1} = x_{i+3,1} = 1$. For each consecutive k where $\mathbf{H}_{i,k} \neq \mathbf{H}_{i+1,k} = \mathbf{H}_{i+2,k} \neq \mathbf{H}_{i+3,k}$ we know that $x_{i,k-1}$ and $x_{i+3,k-1}$ are also 1. The above description only describes scenarios on the top edge of the board, but this constraint holds for each edge of the board.

5 Experimental Setup

In this section we elaborate on our research questions and explain what experiments we run. We also provide more details on how we conduct our experiments including what parameters we used.

5.1 Detailing our Research Questions

Using the models and properties above we can now more precisely define our research questions:

Research question 1: How fast can we solve Hitori puzzles using the Integer Linear Programming solver Gurobi?

1. How does solving speed depend on the encoding of Hitori?
2. How does the ILP model compare to an *Answer Set Programming* (ASP), CSP, SMT, and *Logic Programming* (LP) model in terms of solving speed?

We answer these research questions through 2 experiments. In our first experiment we test our base models with and without each of the redundant constraints defined in Section 4.4 on 1000 $n = 10$ instances. In our second experiment we run our naive model with an ASP (in Clingo), CSP (in Pumpkin), SMT (in Z3), and LP (in Prolog) model and compare their solving speed over 50 instances of size 5,10, 15,…,45,50. The experiment was run by a colleague and thus uses different hard- and software than described in Section 5.2. As such, the results of the naive model in this experiment should not be compared with our other experiments.

Research question 2: What properties of Hitori instances influence the speed at which the Gurobi models can solve the instance?

1. How does the average solving speed of our path, naive, and duplicates models scale with instance size?
2. How does solving speed depend on the number of non-unique tiles in the instance for the path, naive, and duplicates model?
3. How does solving speed depend on the number of marked tiles in the solution for the path, naive, and duplicates model?
4. How does solving speed depend on the number of cycles in the instance graph for the duplicates model?

We conduct two experiments to collect the data needed to answer these questions. First, we run our base models on instances of size 5 to 10 inclusive. Second, we collect data on the number of non-unique tiles in instances, the number of marked tiles in solutions, and the number of cycles. We combine this data with other experiments that measured runtime to run correlation tests.

5.2 Experiment Parameters

Our experiments were run on an *AMD Ryzen 5 7640U* processor connected to the grid and running in performance mode hosting a Linux environment. To avoid thermal throttling affecting our results we interleaved our models when running experiments. Modelling was done with GurobiPy 12.0.3, the Python API of Gurobi. The code that we used, including all the instances that we ran our models against, are available on GitHub[2]. The Hitori instances used were created using a generator that is available here[3]. We generated our instances from commit *a2eb900*[4].

Experiments run on this hardware used instances of sizes up to $n = 10$. This size was chosen to ensure that no model would run out of memory, and our three models would be able to solve most instances within 10 seconds, our defined timeout limit. We used penalised aggregate runtime for models that did exceed 10 seconds, meaning that we cut off their process at 10 seconds and instead recorded 20 seconds as their result. To closer emulate common benchmarking setups in competitions, we imposed an 8 GB memory limit on the model and forced it to run on a single thread.

When comparing models we look at the mean time it took to solve an instance. This represents poor performing instances and worst-case scenarios better than other statistics such as *most instances evaluated fastest*, especially when models perform well on the vast majority of instances. In Appendix C we detail why we use *permutation tests* and *Spearman's correlation* tests to verify our results.

6 Results

In this section we first run two preliminary tests to optimise our naive model. After that we run experiments to collect data necessary to answer our research question. These experiments are organised per research question.

6.1 Preliminary Experiments

We ran two preliminary tests to optimise our naive model. First we compared the runtime of the naive model with the same model if we added a heuristic to prioritise proposed solutions with more or fewer marked tiles. A permutation test showed that our model with no heuristic was 8.397 seconds faster than our model that prioritised solutions with more marked tiles ($p < 0.001$). Prioritising proposed solutions with fewer marked tiles was the fastest, gaining 9.350 seconds on the base model ($p < 0.001$).

We also tried three different algorithms for checking the connectivity constraint after the naive model had returned a potential solution. These algorithms were breadth-first search (BFS), counting connected components (#CC) in the graph of unmarked tiles in the solution, and counting cycles (cycles) in the graph of marked tiles of the solution. A permutation test found BFS was 0.001 seconds faster than cycles ($p < 0.001$), and < 0.001 seconds faster than #CC ($p < 0.001$). Given these results, all follow-up experiments with the naive model used the minimisation heuristic and BFS. This variant of the naive model will be referred to as the *optimised naive model*.

RQ1.1: *How does solving speed depend on the encoding of Hitori?*

[2]https://github.com/Sophieke32/Gurobi_Hitori

[3]https://github.com/sappho3/Thesis-Hitori-shared

[4]https://github.com/sappho3/Thesis-Hitori-shared/commit/a2eb9006ef758783665ef2287a0dbf1f0d555641

Constraint	**Occurrences**	**Path**		**Optimised Naive**		**Duplicates**	
		$\Delta\mu$	p	$\Delta\mu$	p	$\Delta\mu$	p
Corner Close	4000[5]	0.049	< 0.001	0.040	0.104	-	-
Corner Check	2	-0.003	< 0.001	< 0.001	0.419	< 0.001	0.215
Sandwich Pair	2648	0.017	0.059	-0.025	0.214	0.021	0.031
Sandwich Triple	85						
Edge pairs (any k)	0	-0.003	< 0.001	< 0.001	0.410	< -0.001	0.433
Most blacks	20000[5]	-0.008	0.227	-0.017	0.289	< 0.001	0.348
Least whites	20000[5]	-0.009	0.192	-0.018	0.272	< -0.001	0.062
Pair isolation	1748	0.027	< 0.001	-0.004	0.489	< -0.001	0.403

Table 2: Number of occurrences of each constraint in 1000 $n = 10$ instances, and their effect on solvers. $\Delta\mu$ is the difference in mean compared to the base case. p is the significance level.

When comparing the optimised naive model ($\mu = 0.140$, $\sigma^2 = 1.831$), with the duplicates model ($\mu = 0.117$, $\sigma^2 = 0.889$) we find no statistically significant difference in solving speed ($p = 0.331$). Both the optimised naive model (-0.308, $p < 0.001$) and the duplicates model (-0.331, $p < 0.001$) are significantly faster than the path model ($\mu = 0.448$, $\sigma^2 = 0.248$).

Figure 2 shows how long it took each model to solve instances if all instances were sorted based on how fast the model could solve them. It shows that the optimised naive model is initially faster, but it spends over 87% of its total runtime on just 10 instances, allowing the duplicates model to close the gap.

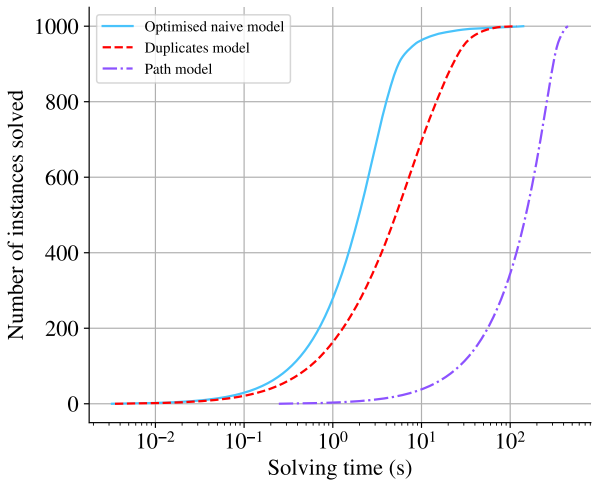


Figure 2: Runtime on $n = 10$ instances of the path, optimised naive, and duplicates model running on Gurobi. Note that the horizontal axis uses logarithmic scaling.

Table 2 shows the effects of the implemented redundant constraints. The path model had four significant results, the optimised naive model none, and the duplicates model one. The only constraints with meaningful effect size were sandwiches on the duplicates model and corner close and pair isolation on the path model. None of these sped up the respective model.

RQ1.2: *How does the ILP model compare to an ASP, CSP, SMT, or LP model in terms of solving speed?*

Figure 3 shows the mean solving time of the five tested models. Permutation testing shows that the ILP model is 1.739 seconds faster than the LP model ($p < 0.001$), but slower than the SMT model ($\Delta\mu = -9.191$, $p < 0.001$), the CSP model ($\Delta\mu = -12.357$, $p < 0.001$), and the ASP model ($\Delta\mu = -12.862$, $p < 0.001$).

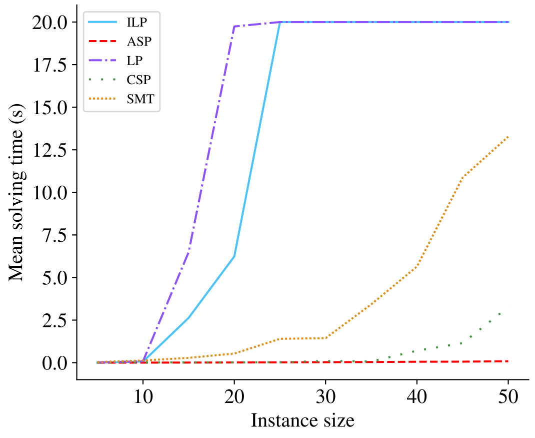


Figure 3: Mean solving time over 1000 instances of different sizes of the ILP, ASP, LP, CSP, and SMT models.

RQ2.1: *How does the average solving speed of our path, naive, and duplicates models scale with instance size?*

Runtime grows exponentially as instance size increases as shown in Figure 4, which makes sense as Hitori is an NP-Complete problem. The optimised naive model seems to perform worse on instances with odd sizes as displayed by the spikes in runtime for instance size 7 and 9.

RQ2.2: *How does solving speed depend on the number of non-unique tiles in the instance for the path, naive, and duplicates model?*

Testing for correlation between the number of non-unique tiles in the instance and the runtime of the three models with

[5]This constraint occurs a set number of times on every board regardless of the contents of the instance.

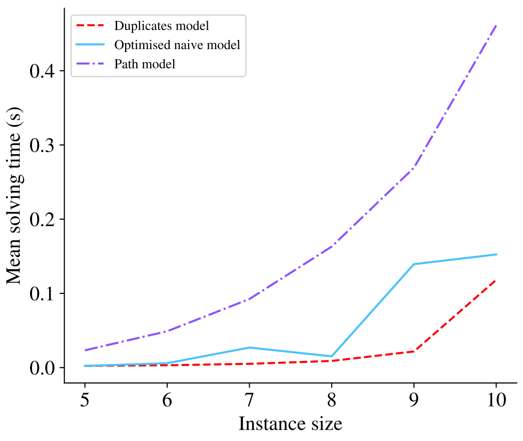


Figure 4: Mean solving time over 1000 instances of different sizes of the path, optimised naive, and duplicates model running on Gurobi.

a Spearman's correlation test yields $\rho = -0.121$ ($p < 0.001$) for the path model, $\rho = -0.110$ ($p < 0.001$) for the optimised naive model, and $\rho = 0.532$ ($p < 0.001$) for the duplicates model. Figure 5 shows this correlation for the duplicates

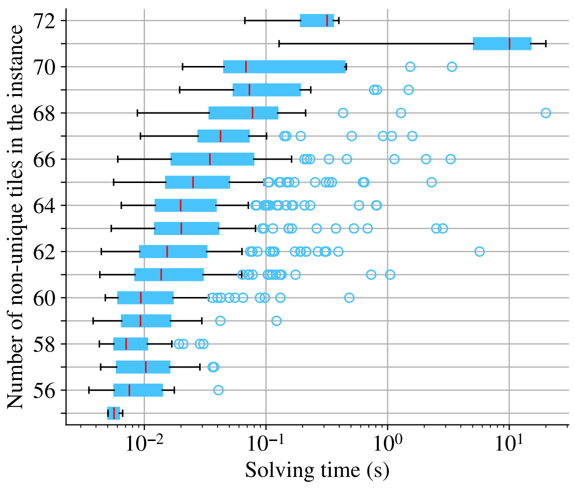


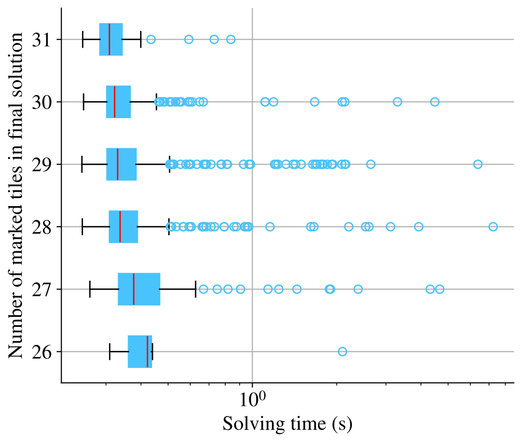
Figure 5: The effect of the number of non-unique tiles in a problem instance on the duplicates model's runtime running in Gurobi. Note the logarithmic scale on the horizontal axis.

RQ2.3: *How does solving speed depend on the number of marked tiles in the solution for the path, naive, and duplicates model?*

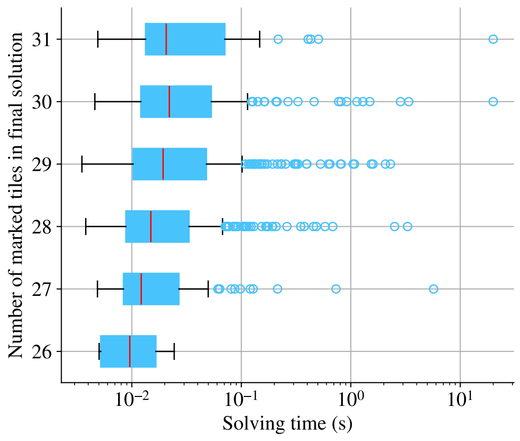
Figure 6a shows that runtime of the path model decreases with more marked tiles in the solution ($\rho = -0.173$ $p < 0.001$), whereas the duplicates model has the opposite relation ($\rho = 0.163$ $p < 0.001$) as visible in Figure 6b. The optimised naive model is not affected ($\rho = -0.038$ $p = 0.224$).

RQ2.4: *How does solving speed depend on the number of cycles in the instance graph for the duplicates model?*

A spearman's correlation test into the effect of the number of cycles in the instance graph on the runtime of the duplicates model shows that they are positively correlated $\rho = 0.978$ ($p < 0.001$).



(a) The path model



(b) The duplicates model

Figure 6: The effect of the number of marked tiles in the final solution on model runtime running in Gurobi. Note the logarithmic scale on the horizontal axes.

7 Discussion

Heuristics for the naive model: The fact that prioritising solutions with fewer marked tiles speeds up the solver hints at a property of the instances we generated: boards tend to have fewer, rather than more, marked tiles in their solution. This makes sense in light of the connectivity constraint, which favours boards with fewer marked tiles.

Checking the connectivity constraint: It is not surprising that our BFS algorithm was the fastest. cycles and #CC both required the creation of a graph before using a breadth-first search algorithm. Therefore it makes sense that plain BFS was the fastest.

Comparing the path, optimised naive, and duplicates model The path model performed significantly worse than the other two models. We hypothesise that this is because the model is quartic compared to input size. On large instance sizes this might be an advantage as other encodings scale worse, but for the smaller instances from our experiment this may prove a disadvantage.

An interesting property of the optimised naive model is that its runtime seems to spike on instances with an odd size. We recreated the optimised naive model in a different library named *ScipOpt* (version 6.0.0) and ran it on the same instances. Doing so did not reveal the same pattern. This insinuates that the behaviour originates from the Gurobi solver.

Adding redundant constraints Almost none of the redundant constraints led to a significant, let alone substantial difference on the runtime. This implies that the time gained through the constraints about evens out with the overhead created by them. Since none of the substantial differences were a decrease in runtime, redundant constraints do not seem a successful strategy for increasing solver speed.

Comparing our ILP model to an ASP, LP, CSP, and SMT model ILP fares better than LP while SMT, CSP, and ASP far outperform both with regards to solving speed. One explanation for this is that our ILP model closely resembled SAT models, using only boolean variables and involving small linear equations. As such, paradigms more optimised for SAT solving will perform better.

An interesting observation also arises when comparing the ILP and ASP models. The ASP model used a similar method of modelling the connectivity constraint as our path model, but in a much more lightweight manner.

The ASP model defined a grid $P_{i,j}$ for $i, j \in [1..n]$ and force $t_i \leq t_j \land t_j \leq t_j$ on all neighbouring tiles. Gurobi reduces $t_i \leq t_j \land t_j \leq t_i$ to $t_i = t_j$, and as such any two adjacent tiles not on the path can put each other on the path. Clasp prevents such circular arguments and can thus model the connectivity constraint much more efficiently.

Effect of number of non-unique tiles in instance The fact that the number of non-unique tiles in an instance correlates with the solving time of the duplicates model makes sense. More non-unique tiles means a larger graph, likely with more cycles. We hypothesise that for the path model it is merely a proxy for the positive effect of extra marked tiles in the solution on runtime, given that the number of non-unique tiles in an instance is positively correlated with the number of marked tiles in its solution ($\rho = 0.417, p \leq 0.001$).We are not sure why more non-unique tiles correlates with faster solving times for the optimised naive model.

Effect of number of marked tiles in solution The number of marked tiles in the final solution correlates inversely with solving speed of the path model. We suspect that this is because a marked tile in the path model instantly assigns a value to n^2 variables, since none of them will be on the path.

We suspected that an increase in the number of marked tiles would slow down the optimised naive model, since it prioritises solutions with fewer tiles. This did not turn out to be the case. We hypothesise that a given instance has a relatively small range of proposed solutions with a different number of marked tiles. As such, instances where the solution had many marked tiles did not have proposed solutions with very few. As a result of this, the number of marked tiles in the solution does not correlate with the runtime performance of the optimised naive model.

The duplicates model's solution time does correlate with the number of marked tiles in the solution, but we hypothesise that this is a proxy of the number of non-unique tiles in the instance.

Number of cycles The result that the duplicates model's runtime correlates with the number of cycles is also intuitive. For every extra cycle there are more constraints to make and keep track off.

Using ILP for Hitori Hitori is, at its core, a boolean combinatorial problem: mark a tile or not. ILP is made for optimising linear equations. As a result, encodings are unintuitive and many of the strengths of ILP are underused: all variables were booleans and the path and duplicates model did not optimise any equation. As such, though it is possible to solve Hitori with ILP, it is not necessarily intuitive or fast.

8 Conclusion

In this study we created and tested three ILP models on 1000 $n = 10$ instances of the Hitori puzzle. We did so to test how well ILP can solve Hitori puzzles. We found that ILP is not a good option for this problem: modelling Hitori in ILP is unintuitive and impractical, and the resulting models do not scale well with instance-size. We used three properties of Hitori to create the duplicates model which performed equally fast as its naive alternative though with higher memory usage. Our path model was quartic with regards to instance size and did not rely on probabilistic methods, yet it was much slower than the other two models. Adding redundant constraints to our model yielded no significant, substantial, positive results. Our optimised naive model tested faster than an LP model, but slower than an SMT, CSP, and an ASP model. The conclusion is straightforward: it is hard to solve Hitori with ILP, especially compared to other M&S paradigms. This is because Hitori is a binary combinatorial problem, and ILP is made for optimising linear equations.

This conclusion makes room for further research. We have compared five paradigms, but there are more out there, which might each bring new insights when applied on this puzzle. Furthermore, Hitori is a very specific problem which favours some paradigms over others. But it is merely one problem, and there are many more out there to be solved. What conclusions hold for Hitori might not for Sudoku, the Progressive Party Problem, or yet other problems.

Third, we do not claim our models to be optimal. New and different insights from M&S communities might lead to new optimisations, also in the ILP model. These too, might lead to new conclusions.

9 Responsible Research

During this study we found three ethical considerations that touched upon our work: reproducibility, extensibility, and the use of Large Language Models (LLMs).

Throughout this study, we have tried to document our steps as extensively as necessary for any future researchers to be able to reproduce our work. This includes descriptions of our models and experiments as well as links to the GitHub repository where we ran our experiments. This repository also includes a list of resources that may aid in the understanding and reproduction of our work. These aids consist of a README which specifies how to run the experiments, the code to run the experiments, the full set of problem instances we used for our experiments, the data files we collected as a result of our experiments, and the code we used to generate our visualisations.

We also linked the code we used to generate our Hitori instances, and provided a prove sketch in appendix D that this method can yield any valid $n \times n$ instance, and no invalid instance. In doing so we hope that others can extend our research to include more instances.

In our project we did not make any use of LLMs. This was a conscious decision informed by the ethical questions raised by LLMs regarding for example copyright infringements [13] and impact on our climate [14].

Acknowledgements

I'd like to thank my committee for their time in reviewing my paper. Special thanks go to Dr. Anna L. D. Latour who provided guidance and generous feedback on my work throughout its creation.

I am also grateful to my colleagues for their critical thoughts and good ideas. A special mention goes to Sappho who was a great sparring partner throughout the project.

Last, I want to thank all the people who supported me personally, whether though moral support or by reviewing my paper. They kept my spirits high and my work sharp.

References

[1] T. Mancini, D. Micaletto, F. Patrizi, and M. Cadoli, "Evaluating ASP and commercial solvers on the csplib," *Constraints An Int. J.*, vol. 13, no. 4, pp. 407–436, 2008. [Online]. Available: https://doi.org/10.1007/s10601-007-9028-6

[2] N. N. K. Fazekas, M. Järvisalo and P. J. Stuckey, "Interactions in Constraint Optimization," Schloss Dagstuhl. [Online]. Available: https://www.dagstuhl.de/25371

[3] E. Coban, E. Erdem, and F. Ture, "Comparing asp, cp, ilp on two challenging applications: Wire routing and haplotype inference," 01 2008.

[4] G. O. LLC, "Gurobi Optimization," Gurobi Optimization LLC. [Online]. Available: https://www.gurobi.com/

[5] R. Wensveen, "Solving, Generating and Classifying Hitori." [Online]. Available: https://theses.liacs.nl/3122

[6] M. Gander and C. Hofer, "Hitori Solver."

[7] B. Crawford, C. Castro, and E. Monfroy, "Solving Sudoku with Constraint Programming," in *Cutting-edge research topics on multiple criteria decision making proceedings*, vol. 35. Springer, pp. 345–348. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-02298-2_52

[8] A. C. Bartlett, T. P. Chartier, A. N. Langville, and T. D. Rankin, "An Integer Programming Model for the Sudoku Problem." [Online]. Available: https://people.sc.fsu.edu/~jburkardt/latex_src/polyominoes_2018_upg/sudoku5.pdf

[9] B. M. Smith, S. C. Brailsford, P. M. Hubbard, and H. P. Williams, "The progressive party problem: Integer linear programming and constraint programming compared," *Constraints An Int. J.*, vol. 1, no. 1/2, pp. 119–138, 1996. [Online]. Available: https://doi.org/10.1007/BF00143880

[10] G. Dantzig, "Origins of the simplex method," in *A History of Scientific Computing*, S. G. Nash, Ed. Association for Computing Machinery, pp. 141–151. [Online]. Available: http://dl.acm.org/doi/10.1145/87252.88081

[11] R. A. Hearn, *Games, Puzzles, and Computation*, 1st ed. Florida: CRC Press LLC, 2009.

[12] A. Suzuki, M. Kiyomi, Y. Otachi, K. Uchizawa, and T. Uno, "Hitori numbers," *J. Inf. Process.*, vol. 25, pp. 695–707, 2017. [Online]. Available: https://doi.org/10.2197/ipsjjip.25.695

[13] R. Reed. Chatnyt: Harvard law expert in technology and the law says the new york times lawsuit against chatgpt parent openai is the first big test for ai in the copyright space. [Online]. Available: https://hls.harvard.edu/today/does-chatgpt-violate-new-york-times-copyrights/

[14] J. Morrison, C. Na, J. Fernandez, T. Dettmers, E. Strubell, and J. Dodge, "Holistically evaluating the environmental impact of creating language models," in *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. [Online]. Available: https://openreview.net/forum?id=04qx93Viwj

[15] G. B. Dantzig, "Reminiscences about the origins of linear programming," in *Mathematical Programming The State of the Art, XIth International Symposium on Mathematical Programming, Bonn, Germany, August 23-27, 1982*, A. Bachem, B. Korte, and M. Grötschel, Eds. Springer, 1982, pp. 78–86. [Online]. Available: https://doi.org/10.1007/978-3-642-68874-4_4

[16] R. Dogaru, *Systematic Design for Emergence in Cellular Nonlinear Networks: With Applications in Natural Computing and Signal Processing*, ser. Studies in Computational Intelligence. Springer, 2008, vol. 95. [Online]. Available: https://doi.org/10.1007/978-3-540-76801-2

[17] G. B. Dantzig, "Linear Programming and Extensions," p. 611. [Online]. Available: https://www.rand.org/content/dam/rand/pubs/reports/2007/R366part1.pdf

[18] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *Eur. J. Oper. Res.*, vol. 119, no. 3, pp. 557–581, 1999. [Online]. Available: https://doi.org/10.1016/S0377-2217(98)00364-6

[19] M. G. Buscemi and U. Montanari, "A survey of constraint-based programming paradigms," *Comput. Sci. Rev.*, vol. 2, no. 3, pp. 137–141, 2008. [Online]. Available: https://doi.org/10.1016/j.cosrev.2008.10.001

[20] P. Meseguer, "Constraint satisfaction problems: An overview," *AI Commun.*, vol. 2, no. 1, pp. 3–17, 1989. [Online]. Available: https://doi.org/10.3233/AIC-1989-2101

[21] L. M. de Moura and N. S. Bjørner, "Satisfiability modulo theories: introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, 2011. [Online]. Available: https://doi.org/10.1145/1995376.1995394

[22] L. M. de Moura and N. S. Bjorner, "Satisfiability modulo theories: An appetizer," in *Formal Methods: Foundations and Applications, 12th Brazilian Symposium on Formal Methods, SBMF 2009, Gramado, Brazil, August 19-21, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. V. M. Oliveira and J. Woodcock, Eds., vol. 5902. Springer, 2009, pp. 23–36. [Online]. Available: https://doi.org/10.1007/978-3-642-10452-7_3

[23] M. Davis, G. Logemann, and D. W. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, 1962. [Online]. Available: https://doi.org/10.1145/368273.368557

[24] D. Macherki, T. M. L. Diallo, J.-Y. Choley, M. Barkallah, and M. Haddar, "Self-reconfiguration of manufacturing systems using Satisfiability Modulo Theory," pp. 1–26. [Online]. Available: https://www.tandfonline.com/doi/full/10.1080/00207543.2025.2561190

[25] E. J. G. Pitman, "Significance tests which may be applied to samples from any populations," *Supplement to the Journal of the Royal Statistical Society*, vol. 4, no. 1, pp. 119–130, 1937. [Online]. Available: http://www.jstor.org/stable/2984124

[26] ——, "Significance tests which may be applied to samples from any populations. ii. the correlation coefficient test," *Supplement to the Journal of the Royal Statistical Society*, vol. 4, no. 2, pp. 225–232, 1937. [Online]. Available: http://www.jstor.org/stable/2983647

[27] ——, "Significance tests which may be applied to samples from any populations: Iii. the analysis of variance test," *Biometrika*, vol. 29, no. 3/4, pp. 322–335, 1938. [Online]. Available: http://www.jstor.org/stable/2332008

[28] Scipy.stats permutation test. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.permutation_test.html

[29] Weak or strong? how to interpret a spearman or kendall correlation. [Online]. Available: https://blogs.sas.com/content/iml/2023/04/05/interpret-spearman-kendall-corr.html

Appendix

A Literature Review

At the start of the project we reviewed literature to better understand different M&S paradigms. This helped us make an informed decision when selecting the paradigm we would use. As such, this literature review investigates three dominant constraint modelling-and-solving (M&S) paradigms. We briefly discuss their origins and inner workings, compare them with one another, and investigate their application to Hitori. Last, we look into earlier studies where M&S paradigms are used to solve logic problems.

A.1 Linear Programming

Linear programming (LP) is at its heart a paradigm that tackles problems by solving linear (in)equalities [15]. The user defines a list of linear constraints in the form of

$$2x_1 + 3x_2 \leq 9$$
$$x_1 \leq 3$$

and the programme then optimises the variables for a minimal or maximal objective function[15, 16], such as:

$$\text{maximise}(x_1 + 2x_2)$$

The paradigm originally used the *Simplex algorithm* to attain this, though alternatives exist as LP has been an active field of research since the late 40's [17, 18] (for more information on the Simplex algorithm see Dantzig [10]). The approach was popularised by Dantzig around the 1950's [15] and variants were made, including *integer linear programming*, where variables are restricted to integer values, *mixed integer linear programming*, where some variables are restricted to integers and others are not, and *quadratic programming*, which allows for quadratic (in)equalities.

For Hitori, however, LP is not the most intuitive choice. The paradigm grew out of resource allocation problems and as such quickly evolved to be built around objective functions [15]. Hitori, however, is not about allocating resources. Answers in Hitori are correct or incorrect, there is no objective to minimise or maximise. Even though solving Hitori in LP is possible, the paradigm is not made for solving such types of problems, and solutions will likely involve unusual workarounds.

A.2 Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSPs) put variables at the centre. Problem instances of CSPs have a finite set of variables, each with a finite domain of possible values [19, 20]. Only on top of that are constraints defined [19, 20]. A crucial concept in solving CSPs is *arc consistency*, which is when two variables x_i and x_j share a constraint C_{ij}, and for each element in the domain of x_i, there exists an element in the domain of x_j such that C_{ij} is satisfied [18].

CSP solvers in brief consist of two parts. The first is a *backwards-facing algorithm* such as backtracking, where, when a contradiction has been created through the elimination of elements within variable domains, the program undoes decisions made before until the contradiction is resolved [18].

The second is a *forward-facing algorithm* such as forward-checking, which, when assigning a variable, ensures that arc consistency is maintained by removing elements from variables' domains [18, 20].

Unlike with LP, the type of problem that Hitori represents is closely aligned with what CSP is made to solve. When solving Hitori, one looks merely for satisfiability. A single answer that satisfies the given constraints is enough, nothing needs to be optimised. As such CSP provides an interesting option as a paradigm for solving Hitori.

A.3 Satisfiability Modulo Theories Solvers

Satisfiability Modulo Theories (SMT) are a generalisation of SAT which also include solving capacity for, for example, strings, integers, and pattern matching through the implementation of theories [21]. To achieve this, an SMT solver transforms its input by building a propositional skeleton in conjunctive normal form, and then iteratively uses SAT solving and theory solving to come to a conclusion on whether the given input is satisfiable or not [21, 22]. How exactly the SAT and theory solving are done depends heavily on the solver itself, though most modern models do SAT solving with the *Davis–Putnam–Logemann–Loveland* (or DPLL) algorithm [21] (for more information on DPLL refer to [23])

Compared to CSP, SMT is more generalised. Different solvers which implement different theories support different features, but all of them support theories, and thus variable-types, that CSP cannot (easily) model. But that does not necessarily make it more suitable for solving Hitori. A lot of the functionality that comes with state of the art SMT solvers are simply redundant for solving Hitori, and is thus not of interest when solving it.

A.4 Prior Work on Solving Problems Using M&S Paradigms

This section discusses four papers that applied M&S paradigms. For each, notable findings that relate to solving Hitori are marked.

Sudoku: Crawford et al. [7] solves sudoku using CSP. In their paper they investigate using different variable and value selection heuristics when making guesses. They find that the *fail-first* heuristic, where variables with a smaller domain are prioritised for selection, perform better due to their ability to prune the search-space quicker.

The Progressive Party Problem: Smith et al. [9] also find that fail-first is a good heuristic to use. In their paper they solve the Progressive Party Problem (for more information, refer to their paper) using both ILP and CSP. In doing so they find that CSP provides a better framework for solving the problem. CSP is faster, and, contrary to the ILP model, able to find a solution for an instance with 13 host boats. This is in large part contributed to the fact that CSP allows for a more compact model, and the fact that its constraint propagation is quick to prune large parts of the search-space. On top of this, they also note that CSP also has a more intuitive model, whereas mapping from the ILP model to the problem is more difficult.

An important part for both of the two models is the removal of symmetries in the search-space, for instance due to two host boats with the same spare capacity, or two guest boats with the same crew size. This will not be relevant for solving Hitori, however, as there is only one possible solution for each problem instance.

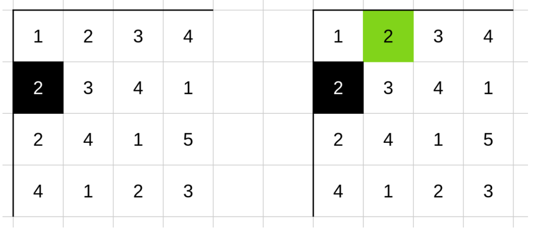
Self-reconfiguring Manufacturing Systems Using SMT: In their paper, Macherki et al. [24] propose an SMT model which is able to reconfigure a manufacturing plant when given internal or external changes to the needs and abilities of that plant (such as urgent orders or machine failures). Notably this system is incredibly complex and as such the model uses many of the features that state of the art SMT solvers offer. This is a stark difference to Hitori, which in itself is a very simple combinatorial problem.

Hitori Last in this section, Wensveen studied the generation, solving, and classification of Hitori [5]. In her work she made a thorough analysis of a more generalised version of the game which allows for $n \times m$ boards. She discusses a variety of solving rules and builds a 2-SAT and a rules-based model to solve the puzzles. Her work, however, is more focussed on classifying the difficulty level of an existing dataset of Hitori puzzles, whereas this paper is more interested in modelling intuitiveness and speed. Nevertheless, this work will build upon the solving rules defined in her work.

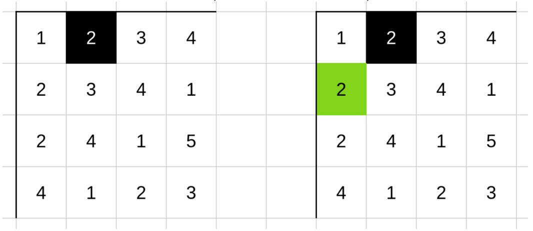
B Visualisation of Redundant Constraints

This appendix visualises the redundant constraints we have implemented. Black tiles are marked, green tiles are definitively unmarked, and white tiles are unknown. The grid on the left defines a board state, and the grid on the right defines a conclusion we can draw from that board state.

Corner Close [5]: The connectivity constraint dictates $x_{0,1} = 1 \implies x_{1,0} = 0$ and $x_{1,0} = 1 \implies x_{0,1} = 0$. The same holds for $x_{n-2,0}$ and $x_{n-1,1}$, $x_{0,n-2}$ and $x_{1,n-1}$, as well as $x_{n-2,n-1}$ and $x_{n-1,n-2}$. See also Figure 7.



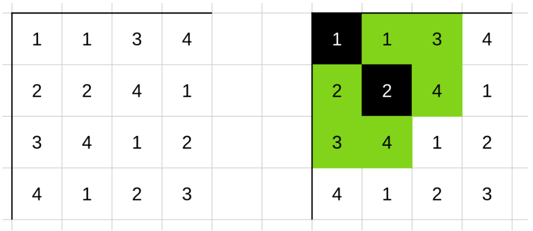
(a) $x_{1,0} = 1 \implies x_{0,1} = 0$.



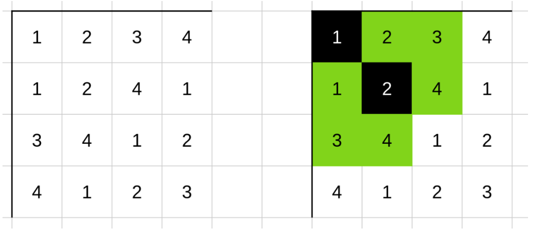
(b) $x_{0,1} = 1 \implies x_{1,0} = 0$.

Figure 7: Corner Close is enforced by the connectivity constraint, and holds in all corners.

Corner Check [5,6]: Given $\mathbf{H}_{0,0} = \mathbf{H}_{0,1} \wedge \mathbf{H}_{1,0} = \mathbf{H}_{1,1}$, or $\mathbf{H}_{0,0} = \mathbf{H}_{1,0} \wedge \mathbf{H}_{0,1} = \mathbf{H}_{1,1}$, all three Hitori constraints together dictate $x_{0,0} = x_{1,1} = 1 \wedge x_{0,1} = x_{0,2} = x_{1,2} = x_{1,0} = x_{2,0} = x_{2,1} = 0$. See also Figure 8.



(a) Horizontal corner check.



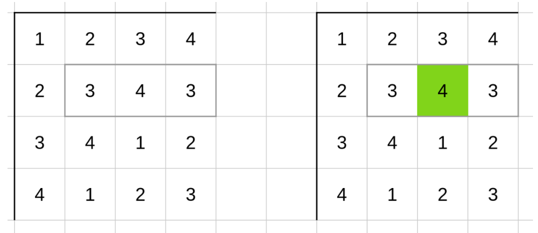
(b) Vertical corner check.

Figure 8: Corner Check is enforced by the connectivity constraint, and holds in all corners.

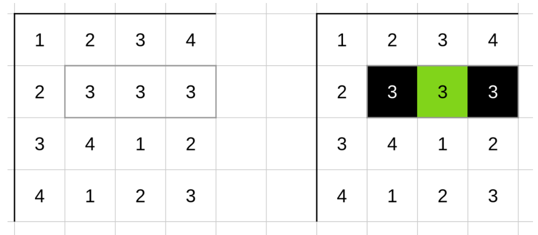
Most Blacks: Given the adjacency constraint we know that at most $\lceil \frac{n}{2} \rceil$ tiles can be marked in any row or column.

Least Whites: The inverse of most blacks, determines that at least $\lfloor \frac{n}{2} \rfloor$ tiles must be unmarked in any row or column.

Sandwiches [5,6]: Covers both Sandwich Pair and its more specific case Sandwich Triple. Given $\mathbf{H}_{i,j} = \mathbf{H}_{i+2,j}$ then $x_{i+1,j} = 0$. If $\mathbf{H}_{i,j} = \mathbf{H}_{i+1,j}$ as well, then we also know that $x_{i,j} = x_{i+2,j} = 1$. The same goes columnwise. See also Figure 9.



(a) A sandwich pair. The adjacency constraint requires the middle tile to be left unmarked.



(b) A sandwich triple. The adjacency constraint requires the middle tile to be marked, and the outer two to be unmarked.

Figure 9: Variants of the sandwich pair and sandwich triple. Both hold in columns as well.

Pair Isolation [5,6]: Given $\mathbf{H}_{i,j} = \mathbf{H}_{i+1,j}$ and $\mathbf{H}_{k,j} = \mathbf{H}_{i,j}$ with $k \in [1..i-2] \cup [i+3..n]$, then the uniqueness and adjacency constraints require $x_{k,j} = 1$. The same goes columnwise. See also Figure 10.

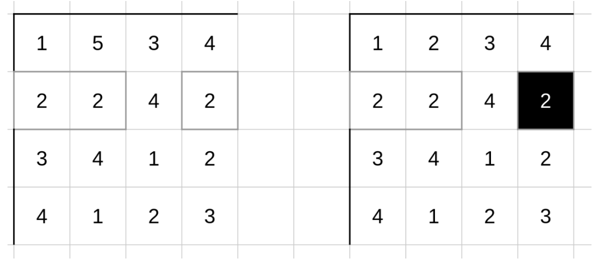


Figure 10: The uniqueness constraint requires that the single 2 on the right must be marked, for at least one of the 2s on the left must be unmarked.

Edge Pairs [5,6]: Given $\mathbf{H}_{i,1} \neq \mathbf{H}_{i+1,1} = \mathbf{H}_{i+2,1} \neq \mathbf{H}_{i+3,1} \wedge \mathbf{H}_{i+1,2} = \mathbf{H}_{i+2,2}$, then the three Hitori constraints require $x_{i,1} = x_{i+3,1} = 1$. For each consecutive k where $\mathbf{H}_{i,k} \neq \mathbf{H}_{i+1,k} = \mathbf{H}_{i+2,k} \neq \mathbf{H}_{i+3,k}$ we know that $x_{i,k-1}$ and $x_{i+3,k-1}$ are also 1. The above description only describes scenarios on the top edge of the board, but this constraint holds for each edge of the board. See also Figure 11.

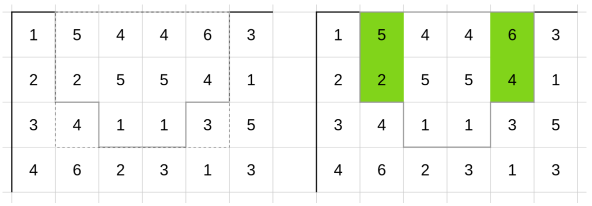


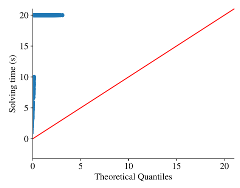
Figure 11: The adjacency and uniqueness constraints require the middle two 4s, 5s, and 1s to have at least one marked, and one unmarked tile. The green-marked cells must be unmarked as a result.

C Statistics

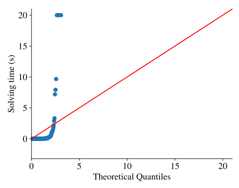
This appendix details the process of selecting statistical tests and interpreting their results. The first observation made is that we have two types of tests: Tests in which we compare the means of datasets (when comparing the runtime performance of two models), and testing for correlation between two variables (for instance the effect of the number of cycles in an instance graph on the runtime of the duplicates model). As such, this appendix will be split in two, dealing with each of these two types of tests. All statistical tests were run in Scipy version 1.16.3.

C.1 Comparing means

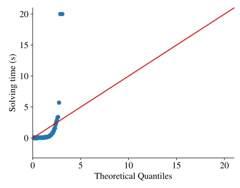
When choosing a test to compare means it is crucial to determine whether the data is parametric. To this end we ran a *Shapiro-Wilk* test on each of our data-sets. The highest value we got was $W = 0.69$, which is well-below the threshold. We visually confirmed these findings using *Quantile Quantile plots*. As shown in Figure 12 our data strayed far from the 45°line, and hence was not normally distributed.



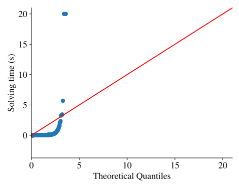
(a) Naive no heuristic.



(b) Naive, minimisation heuristic.



(c) Duplicates model.

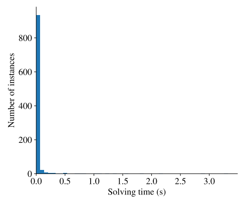


(d) Duplicates, on $n = 5, 6, 7, 8, 9, 10$.

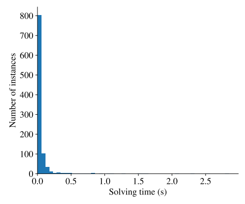
Figure 12: A selection of quantile quantile plots.

Since our data was non-parametric, we next checked to see whether it followed a symmetrical distribution. To this end we plotted our data in histograms, a few of which are shown in Figure 13. They clearly do not follow a symmetrical distribution. The next decision-point is whether our data is dependent or independent. As we test all our models on the same set of 1000 instances, they are paired (and thus dependent).

Given non-parametric, non-symmetric data, the best statistical tests is a permutation test [25–27]. Permutation tests make no assumptions about data, but do allow the comparison of means. We performed our permutation tests using SciPy's permutation_test method [28]. We used 99999 resamples (which, together with the original sample make 100000 samples) and the samples permutation type to reflect the dependence of our data. The method returns a p-value which we combine with the difference in mean ($\Delta\mu$) to present the significance and magnitude of our results.



(a) Optimised naive model, does not include 7 outliers.



(b) Duplicates model, does not include 5 outliers.

Figure 13: Histogram of runtime performance of two models.

C.2 Testing for correlations

When testing for correlations the distribution of our data again matters. Here too we have non-parametric data (we use the same runtime data as before). The type of variable also matters. Since we have an *interval* variable (number of marked tiles or number of cycles) and a *ratio* variable (runtime), we can use the *Spearman's correlation test*. The Spearman's ρ's interpretation is detailed in Table 3.

ρ	**Effect size**
$\lvert\rho\rvert = 0.00$	no effect / very small effect
$\lvert\rho\rvert = 0.10$	small effect
$\lvert\rho\rvert = 0.38$	medium effect
$\lvert\rho\rvert = 0.68$	large effect
$\lvert\rho\rvert = 0.89$	very large effect

Table 3: Interpretation of ρ by [29].

ρ may also be negative, which indicates a negative correlation of the same strength.

D Cooperative work

As mentioned prior, this thesis is part of a larger project to compare the performance of different M&S paradigms on Hitori. Five of us contributed directly to this project, each writing our own paper and working on a shared code base. This code base included three programmes that aided our research. First was an instance generator which could generate single-solution Hitori puzzles of arbitrary size. Second was a rudimentary solver in MiniZinc. This was meant as a proof of concept, and was used by the solver to check that a generated puzzle had exactly one solution. Last was a solution-checker, which was used to confirm that the MiniZinc solver created correct solutions to the generated puzzles.

Below we present a proof sketch for the correctness of our generator. It was pivotal to our research that our generator could generate every possible single-solution Hitori puzzle and no invalid puzzle, and so we proved this property. Last, we present author's contributions.

D.1 Generator Proof Sketch

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 D.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
```

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.

Lemma D.2. *Algorithm 2 only generates valid solution topologies.*

Algorithm 2 Algorithm that generates a solution topology S.

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

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 D.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 D.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. □

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 $H[1,\dots,n][1,\dots,n]$ 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 $i = \lceil \frac{k}{n} \rceil$
    Let $j = ((k-1) \mod n) + 1$
    if $k > n^2$ then
        return true
    else if $S[i][j] == \text{marked}$ 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 $H[i][j]$
        $C = \{1,\dots,n\} \setminus row \setminus col$
        if $C = \emptyset$ 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 D.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 $1, 2, \dots, n$ excluding any number that is already present in the row or column on an unmarked tile.

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 D.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 $i = \lceil \frac{k}{n} \rceil$
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:         $C = row \cup col$
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 D.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 D.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 D.8. *Given a valid solution topology S, Algorithm 3 can generate any valid puzzle instance H.*

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

Lemma D.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 D.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. □

Proving Theorem D.1

Proof. Lemma D.3 has proven that our algorithm can generate exactly any valid solution topology S, and Lemma D.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 D.1 □

D.2 Author Contributions

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; visualisation. **Tom Friederich:** software (equal); formal analysis (equal).