Solving Hitori

Applying Answer Set Programming to Hitori

Sappho de Nooij[1]

Supervisor: Dr. A.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
25th January 2026

Name of the student: Sappho de Nooij
Final project course: CSE3000 Research Project
Thesis committee: Dr. A.L.D. Latour, Dr. T. Coopmans

Abstract

We investigate the performance of modelling and solving paradigms on the NP-complete puzzle Hitori. The choice of paradigm can have a significant impact on performance, but it is not always clear which paradigm is most suitable and why. We develop an ASP model to compare to models made in other paradigms. We investigate the effect of redundant constraints and research the effects of different puzzle properties on the solving time of our ASP model. In our experimental evaluation, we compare the ASP model to the models in other paradigms from parallel studies.

We find that redundant constraints have a negative impact on performance. The solving times of the ASP model had little variance and was not or slightly correlated with various puzzle properties.

Our results show that our approach solves 50-by-50 puzzles 38 times faster at PAR-2 than the second-best model and is the only model that never exceeds the specified timeout, showing the suitability of ASP to Hitori.

1 Introduction

There are many NP-complete problems, like Sudoku [1], for which different solvers, that use different modelling-and-solving paradigms, can be used. These paradigms all have their own strengths and weaknesses. A recent Dagstuhl seminar affirms the need for more interaction between the communities of different paradigms and developing a better understanding between different kinds of problems and how different paradigms perform [2].

The effectiveness of paradigms have been explored several times in the past by applying them to logic puzzles [3–5], but these have not been explored through applying them to the logic puzzle *Hitori*.

Hitori is an NP-complete puzzle [6] where certain numbers have to be blacked out so that no row or column contains duplicate numbers. Additionally, two black tiles may not be adjacent and all white tiles must remain connected.

This thesis is part of a parallel study, where every thesis applies one paradigm to Hitori. The overarching goal is to provide a fair comparison of how well different paradigms perform at Hitori. We have explored Hitori through Answer Set Programming (ASP). This technique is used to find an answer set, a minimal set of atoms that when true make a valid model of a propositional logic program [7].

We developed an ASP-model that can solve Hitori and investigated the impact of different redundant constraints on solving time. We analysed what puzzle properties impacted the solvers performance and compared the ASP model against models made in different paradigms.

We found that most redundant constraints have a negative impact on runtime performance. Most puzzle properties had no impact on performance, but limiting the amount of numbers in a puzzle did slightly increase runtime despite reducing the number of conflicts. Furthermore, we found that ASP scales better than models in other paradigms.

The rest of this paper is organised as follows. We give an overview of related work on Hitori and other puzzles. In section 3, we provide a background overview of Hitori and different paradigms. Then we provide a proof sketch that Hitori puzzles with at most one solution are NP-complete. In section 5 we present our ASP model and the redundant constraints we implemented. In Section 6, we elaborate upon our research questions and explain our experimental setup. Then we show our results in section 7 and discuss them by providing potential reasons for the different results.

2 Related Work

Past studies have applied different paradigms to Hitori. In their bachelor thesis [8], the authors used SAT to solve Hitori. To ensure white tiles are connected, they enforce that the black tiles do not form a cycle. This however, lead to an exponential encoding. They also developed a custom algorithm that uses derivations based on detecting certain patterns, guessing, and backtracking.

Van der Knijff developed an SMT model but did not report runtime performance [9].

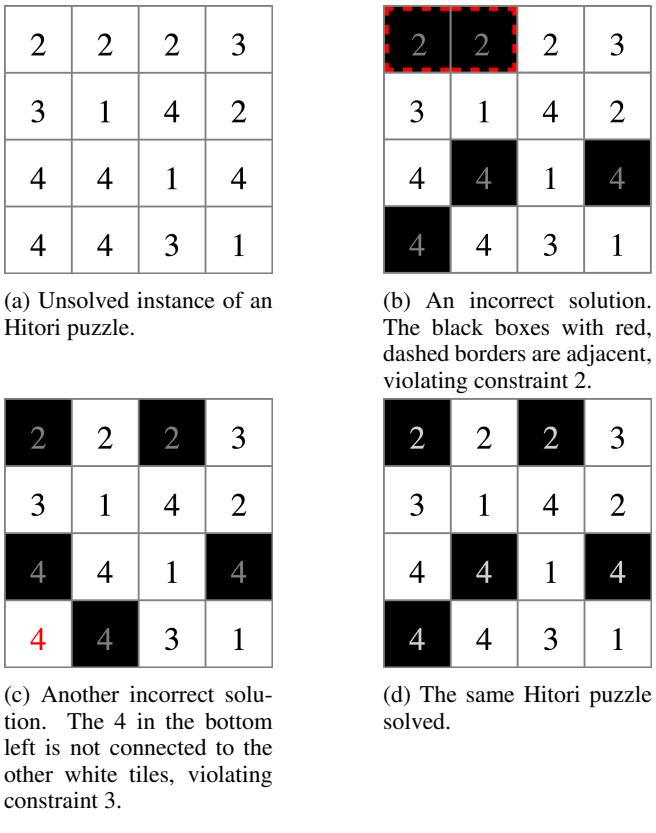
In a recent study on generating Hitori puzzles and classifying their difficulty [10], Wensveen made a rule-based solver in addition to a SAT-based solver. With these she classified different Hitori puzzles based on their difficulty, where more difficult puzzles require more complicated rules to be solved and the most difficult category even *probing*, where a cell is given a color to see if that would lead to a contradiction.

Past studies on Hitori had different ways of gathering puzzles. Some studies used pre-existing puzzles [8, 11]. Van der Knijff tried to generate puzzles by filling in random numbers in a grid and checking whether there is a valid solution, however this did not work because it almost never lead to a good puzzle [9]. In [10], Wensveen took an alternative approach by iterating over all puzzles of a given size, while using *equivalency classes* to skip over symmetrically-equivalent puzzles. The generated puzzles were not guaranteed to be (uniquely) solvable and thus still needed to be filtered. This approach has the advantage that the results were representative of all puzzles, but it limits the size of puzzles that can be generated because of how fast the amount of puzzles grows with the size.

In [12], the authors investigate the runtime performance of ASP and CSP on four grid puzzles. They found that for puzzles with reachability constraints ASP was faster since their CSP solver did not support transitive closures. They do note that solvers with special propagators for reachability might have better runtime performance but left that as future work.

3 Background

In this section, we explain the puzzle Hitori and elaborate on its properties. Additionally, we give an overview of three modelling and solving paradigms, and contrast their strengths and weaknesses.



(a) Unsolved instance of an Hitori puzzle.

(b) An incorrect solution. The black boxes with red, dashed borders are adjacent, violating constraint 2.

(c) Another incorrect solution. The 4 in the bottom left is not connected to the other white tiles, violating constraint 3.

(d) The same Hitori puzzle solved.

Figure 1: A 4-by-4 Hitori puzzle, two incorrect solutions and one correct solution. This Hitori puzzle is uniquely-solvable.

3.1 Hitori

Hitori is a logic puzzle containing an n-by-n grid of numbers where you need to mark tiles black to satisfy the following constraints [13]:

- **Uniqueness**: every number may only appear once in the white tiles of a row and column,
- **Adjacency**: black tiles may not touch each other vertically or horizontally,
- **Connectivity**: all white tiles must be orthogonally connected to all other white tiles.

Figure 1 shows an example of an unsolved instance, two invalid solutions and the correct solution.

Past literature focuses on *uniquely-solvable* Hitori, which is a variant of Hitori such that every puzzle has at most one unique solution [10, 14]. In [14], the authors investigate how many unique numbers a Hitori puzzle can contain. Most websites that publish Hitori puzzles only use Hitori puzzles n unique numbers, where n is the width and height of the puzzle [15, 16]. We focus on uniquely-solvable Hitori puzzles with at most n unique numbers.

In [6], the authors show that non-uniquely-solvable Hitori is NP-complete. In section 4, we provide a proof sketch that shows uniquely-solvable Hitori is NP-complete too. We reduce *Unambiguous SAT* (USAT) to uniquely-solvable Hitori. USAT is a special case of SAT that has at most one solution, which is NP-complete too [17].

3.2 Paradigms

There are several paradigms that can be used to solve logic puzzles. We explored ASP, SMT and CSP. For each we explain what it is, explored its strengths and weaknesses, and investigated examples of it being applied to logic puzzles. Based on these findings, we then decided to use ASP.

3.3 Answer Set Programming (ASP)

One technique is Answer Set Programming (ASP). ASP is used to find an answer set, a minimal set of atoms that when true make a valid model of a propositional logic program [7]. ASP uses a higher-level modeling language, which includes variables, that is turned into propositional logic using a process called *grounding* [18].

Notable is Clasp which improves upon other ASP solvers by incorporating *Conflict-Driven Learning* [7]. Whenever Clasp encounters a *conflict*, an assignment of atoms that cannot occur in a valid model, it analyses the cause of the conflict and remembers the cause so that it can prevent the conflict from occurring again, accelerating performance [7].

```
1 p(1). q(2). r(X) :- p(X), q(X + 1).
```

Figure 2: A simple ASP model.

Figure 2 shows an example of an ASP model. Here $p(1)$ and $p(2)$ are facts, which means that they are always true. $r(X)$:- $p(X), q(X+1)$ is a rule that states that if $p(X)$ and $q(X+1)$ are true, $r(X)$ will be true too.

In [19], M. Gebser et al. describe the language as follows: A model consists of a set of rules. Rules are statements that consist of a head and a body like $p(3)$:- $p(1)$, *not* $p(2)$. The head $p(3)$ is considered true when all the elements of the body are true, i.e. $p(1)$ is true and $p(2)$ is false. Facts are a special type of rule with an empty body such that the head is always true. Furthermore, there are constraints which are rules with an empty head, like :- $p(3), p(1)$. Constraints enforce that the elements of the body are not all true.

As shown above, it is possible to use numbers in the modeling language. However, these numbers cannot have an infinite range. A rule like $p(X-1)$:- $p(X)$ could cause the grounding not to halt [19].

Additionally, ASP supports *aggregates*. An aggregate is a counting operation on multiple elements [19]. The aggregate evaluates to true if the specified lower- and upperbound are fulfilled [19]. This can be used to model a choice, take the following rule:

$$1 \ \#\texttt{sum}\ \{\ p(1),\ p(2),\ a\ \}\ 2$$

This rule evaluates to true when at least 1 of $p(1)$, $p(2)$ and a is true and at most 2 of those, thus disallowing them to all be true. Many solvers have dedicated support for aggregates to avoid the quadratic encoding size caused by converting aggregates to pure propositional logic [20].

Variables must be acyclically supported [20]. Consider the program `a :- a`. The only valid answer set is $\{\}$. $\{a\}$ is a *supported model*, but it is not a valid answer set since it only has a circular derivation. If a partial solution contains a

set of atoms that can not be acyclically supported, the solver adds a *loop nogood*, so that the conflict can be reverted and avoided [20].

In [5], Ç. Merve et al applied ASP to several logic puzzles with some connectivity constraint. They argue that due to aggregates, it is easy to represent them in ASP. However, they also show that a "simple representation" can cause Clasp to generate a lot of loop nogoods of large sizes.

3.4 Satisfiability Modulo Theory (SMT)

Satisfiability Modulo Theory (SMT) solvers check whether a logical formula is satisfiable in some *theory* [21]. This theory may be boolean satisfiability but may also be integer arithmetic or a custom theory defined in the SMT-Lib language [21]. Z3 is an SMT solver which is capable of integer arithmetic, bit-vector, and equality reasoning out-of-the-box [22].

Z3's core consists of two parts, an EUF and a SAT solver [23].

The EUF (equality and uninterpreted function) logic is what allows the SMT solver to reason about the equality of variables and the outputs of functions, even if the exact values of the variables and the functions are unknown [23]. For example, $p = q, f(p) \neq f(q)$ evaluates to false since $f(p)$ always equals $f(q)$ if $p = q$, regardless of what the function f evaluates to. This reasoning can also be performed with nested functions [23].

Secondly, it contains a SAT solver. This SAT solver includes *Conflict Driven Clause Learning* to avoid wasting time on unsatisfiable branches [23], just like Clasp. Whenever a conflict arises, the solver will analyse the causes of the conflict and remember those so it will not reach the same conflict again [24].

In [3], R. Behari applies SMT to a puzzle called Fillomino. They compare the runtime performance of SMT to their own custom solver and show that for small puzzles, their custom solver is faster but SMT's performance becomes similar for larger puzzles and is better at difficult puzzles.

3.5 Constraint Satisfaction Programming (CSP)

In CSP, a model is defined using a set of variables, the values those variables may take and constraints, which limit the valid combinations between two or more variables [25].

One CSP solver is Gecode[26], which can be programmed using the high-level modeling language MiniZinc, which is designed to be able to use multiple solvers with low overhead [27].

Another solver is Pumpkin, which creates a proof while it is solving so that correctness can later be verified [28]. The proof creator of Pumpkin is designed so that it selectively records conflicts in the proof to minimise performance overhead [28]. It not only supports MiniZinc but can also be called directly from Rust.

Models with few variables with large domains tend to perform better than models with many variables with small domains [25]. This means that to use CSP well for Hitori, a different approach should be looked for other than using one binary variable for each cell.

In [29], Crawford et al. apply CSP to Sudoku. They showed that the solver was faster when using *fail-first* heuristics, heuristics that will cause the solver to first explore part of the search space where you are likely to fail quickly. This is, they argue, because unnecessary work in branches that will be pruned is avoided.

3.6 Comparison

All these paradigms could be suitable for Hitori. They have several things in common. They all have support for boolean variables and some form of conflict learning, meaning they *could* have similar performance

One of the important differences is how well they support non-boolean variables. ASP only supports other types by encoding them as booleans. Z3 on the other hand is explicitly designed to be able to combine boolean expression with other theories. Finally, CSP supports both boolean and integers but performs better with fewer variables with larger domains [25]. Therefore, CSP might be slower unless an encoding with no or little booleans can be found.

4 Proof sketch of NP-completeness

In this section, we provide a proof sketch that uniquely-solvable Hitori is NP-complete. We reduce *Unambiguous SAT* (USAT) to uniquely-solvable Hitori. USAT is a special case of SAT that has at most one solution, which is NP-complete too [17].

We construct a *partial Hitori solution* P from a USAT formula ϕ. In this partial solution, every tile is either black, white, or its colour depends on the solution to ϕ. In the third case, the value of a tile is associated with a literal in ϕ and is white iff that literal is true in the solution to ϕ. Later, we construct a uniquely-solvable puzzle such that the full solution to ϕ can be determined from the colours of tiles associated with literals. In our reduction, the puzzle has a solution iff the USAT formula is satisfiable.

For every clause $l^1 \vee l^2 \vee ... \vee l^k$ in ϕ, we add a *clause gadget* to our partial Hitori solution. The clause gadget contains a wall of black squares with k gaps, where the tiles of the gaps are filled with a number associated with one of the clause's literals. We show that a partial solution P can only have a solution iff at least one of the tiles of the literals is white.

We show that we can connect all clause gadgets using *connecting gadgets* such that if in a solution a tile associated with a literal is marked white, all other tiles associated with that literal are white and all tiles associated with the negated literal are black.

Then we show that we can construct a solution to ϕ from a solution to P. We prove that P has at most one solution and that we can construct a uniquely-solvable Hitori puzzle H that has a solution iff the USAT formula ϕ is satisfiable.

We argue that this reduction is computable and takes polynomial time. It is trivial to show that Hitori is in NP. Therefore, Hitori is NP-complete.

In appendix E, we provide a more elaborate proof sketch that uniquely-solvable Hitori is NP-complete.

5 Approach

In this section we describe how we modelled Hitori in ASP. Then we mention which redundant constraints we implemented and how they work.

5.1 Modelling Hitori in ASP

Our model consists of several parts. Every tile in the puzzle is encoded as `cell(X, Y, Z).`, where X and Y represent the column and row index, and Z represents the number in that tile. We define the following helper variables, where `col(1..n)` is shorthand notation for defining `col` for every value from 1 to n.

```
1 col(1..n). row(1..n). number(1..n).
```

The uniqueness constraint is modelled using the following constraints.

```
2 :- is_white(X, Y), cell(X, Y, Z), is_white(X
      , Y2), cell(X , Y2, Z), Y != Y2.
3 :- is_white(X, Y), cell(X, Y, Z), is_white(X2
      , Y ), cell(X2, Y , Z), X != X2.
```

Alternatively, we could have modelled the uniqueness constraint using an aggregate. Line 2, 3 and 4 could be replaced by the rule `0 { is_white(X, Y) : cell(X, Y, Z) } 1 :- col(X), number(Z).` and a similar one for the rows. However, we found that that model, despite the general guideline to use aggregates (see Section 8.4 of [20]), was slower at both grounding and solving itself. This might be because in practice, the amount of atoms in the aggregate is limited and thus the benefits of aggregates are not noticeable.

On line 4 and 5, we ensure that every tile is either marked black or white. Furthermore, we explicitly enforce that a tile cannot be both black and white on line 6.

```
4 is_white(X, Y) :- not is_black(X, Y), col(X),
      row(Y).
5 is_black(X, Y) :- not is_white(X, Y), col(X),
      row(Y).
6 :- is_black(X, Y), is_white(X, Y), col(X),
     row(Y).
```

To enforce the adjacency constraint, we add constraints that disallow two adjacent tiles to be black.

```
7 :- is_black(X, Y), is_black(X, Y + 1).
8 :- is_black(X, Y), is_black(X + 1, Y).
```

Lastly, we model connectivity by using the transitive property of connectedness. We specify that the two tiles in the top left corner are connected if they are white. Since they are adjacent, they may not both be black thus at least one will be marked as connected. Every other tile is considered connected if it is next to a connected white tile. On line 17 we enforce that a white tile may not be disconnected.

```
9  connected(1, 1) :- is_white(1, 1).
10 connected(1, 2) :- is_white(1, 2).
11
12 connected(X, Y) :- connected(X - 1, Y),
       is_white(X - 1, Y).
13 connected(X, Y) :- connected(X + 1, Y),
       is_white(X + 1, Y).
14 connected(X, Y) :- connected(X, Y + 1),
       is_white(X, Y + 1).
15 connected(X, Y) :- connected(X, Y - 1),
       is_white(X, Y - 1).
16
17 :- not connected(X, Y), is_white(X, Y).
```

5.2 Redundant constraints

We compared our base model (as described above), against our base model with several redundant constraints added.

We used the following two constraints based on the fact that the puzzle must be uniquely solvable:

Unique Cell (UC) [10]: If a tile is unique in its row and column, mark it to as white.

Unique Cell Dynamic (UCD) [10]: Idem as UC, but now looking only at tiles that could be white: if all other tiles with the same number in the row/column are already marked black, we force it to be white.

The correctness of UC(D) can be shown using a proof-by-contradiction. Assume that it is possible that the tile is black. This means that there is a solution S such that this tile is black. The adjacent tiles will be white and connected as per the rules of Hitori. We can construct a solution S' which is identical except that this tile is white since the number of the cell does not appear in the row or column. Since the tile is adjacent to connected tile, this tile is also connected. Thus S' is a valid solution too but this contradicts the assumption that Hitori is unique-solvable. Therefore, a tile that can be white must be white.

To check connectivity, we implemented the following redundant constraints:

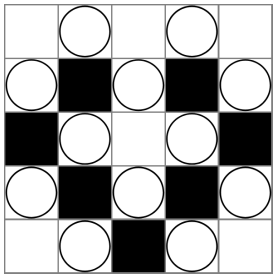
(a) A partial Hitori solution before QM is applied.

(b) A partial Hitori solution after QM is applied.

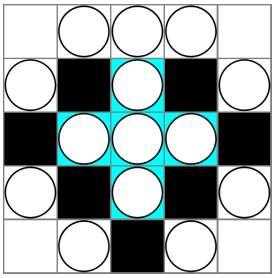
Figure 3: An example of the Quad Middle technique.

Quad Middle (QM)[30]: Since the white tiles must be connected, every white tile has at least one adjacent white tile. We add a constraint *s.t.* no white tile is surrounded by four black squares. See fig. 3.

Isolated Area (IA): IA detects when an area is isolated and makes sure that the only entrance is marked white. In fig. 4a, the middle square could be determined to be white using QM, however the square above must also be white. IA can determine this by marking every square that has 3 adjacent black squares as isolated. Additionally, a white square surrounded by a combination of 3 black or isolated squares is marked isolated. If this is the case, the square that is not black or isolated is marked white since this is the only remain-



(a) A partial Hitori solution before IA is applied.



(b) A partial Hitori solution after IA is applied. The cyan area is marked isolated.

Figure 4: An example of the isolated area technique.

ing square that can guarantee connectivity. Figure 4b shows which squares would be marked white.

Additionally, we implemented some techniques humans use to solve Hitori as described by Hanssen [30]. For each technique listed, we checked whether it was not something our model already does and whether it was concise enough to implementable. Some human techniques were already covered by our base model. For the sake of brevity, we will omit descriptions of the constraints. The human techniques can be grouped in three ways: some techniques work specifically on the corner, some specifically at the edge and others can work everywhere in the grid.

6 Experimental Setup

First, we discuss for each research question how we will answer them. We explain how we generated our puzzles and we mention which software and hardware we used and how we performed the statistical analyses.

6.1 Research Questions

RQ1: *What is the impact of different redundant constraints on solving performance?*

We identified and implemented several redundant constraints. Then we generated 1000 puzzle instances where $n = 50$. We ran every puzzle instance with the base model and the base model plus one redundant constraint. We recorded the CPU time. For each run, we measured the *grounding time* and *solving time* separately. The sum of these values we call the *total solving time*. Additionally, we used the statistics output of Clasp to record the number of *choices* Clasp makes (*i.e.* the number of times Clasp guesses the truth value of an atom because it cannot continue otherwise), the number of conflicts analysed, and the average length of the recorded conflicts.

We report the values averaged over the puzzle instances.

RQ2: *What puzzle characteristics influence the performance of the ASP base model?*

We identified different properties that could potentially influence the solving time and measure the effect on the base model. We looked at the impact of the puzzle size. Secondly, we looked at the number of *adjacent-duplicate tiles*, tiles that have the same value as an adjacent tile. These tiles could make it easier to solve since if two adjacent tiles have the same number, any other tile with the same number in that row or column must be black. Thirdly, we looked at the number of *row-column duplicate tiles*, tiles whose value appears elsewhere in both their row and their column. More of these tiles might make solving faster since the tile can be marked black if the solver marks the other tile in its row with the same value white *and* if the tile with the same number in its column is marked white. Additionally, we looked at the number of *unique tiles*, tiles whose value does not appear in their row or column. Since these are guaranteed to be white, they theoretically reduce the search space, potentially leading to better performance. However, since we do not explicitly model this, it might not affect the base model.

By default, the generator does not generate a very diverse set of puzzles. Therefore, we modified the generator to add bias to the generation. When choosing the number for a black tile, instead of picking a random number there is a chance it will instead pick a value of its neighbour or a number that appears in both its row and column. We made it possible to specify the probability it does this and generated puzzles with different probabilities to get a more diverse set of puzzles for analysis.

We also investigated the range of numbers that appear in a puzzle. Normally, we looked at puzzles with n unique numbers. We investigated how performance differs when less than n unique numbers may be used and when more than n may be used. The solver could be faster for puzzles with less unique numbers since it means more duplicate numbers appear in a row or column, allowing the solver to mark more values as black if a tile becomes white. On the other hand, it may be slower since it could have more conflicts. For this experiment, we modified the generator to allow a smaller or larger range of unique numbers to be used in a puzzle.

RQ3: *How does the runtime performance of different paradigms vary for Hitori puzzles?*

We ran the models in other paradigms from the parallel studies on Hitori puzzles of size $n = 5, 10, 15, ..., 45, 50$. We measured performance using *penalised average runtime 2* (PAR-2), where the runtime is averaged. If a solver takes longer than the timeout, this is penalised by recording the time taken as twice the timeout. We used a timeout of 10 seconds.

We tested five models each in a different paradigm: we tested our base model with the Clasp solver, an SMT model written for the Z3 solver, a CSP model for Pumpkin, an *Integer Linear Programming* (ILP) model written for the solver Gurobi, and a logic program model written in the programming language *Prolog*.

We measured the runtime inside the program of each model, so that the reported time excludes the overhead of starting an interpreter and initialising libraries.

6.2 Generator

We made our own puzzle generator that is capable of generating every uniquely-solvable Hitori puzzle (for proof sketch see appendix D). The generator starts with a solution, fills the white tiles with numbers, then fills in the black tiles and then checks whether the puzzle is uniquely-solvable.

The solution is made by iterating over all tiles in a random order and marking each tile black if this would not violate any constraint. It checks if none of the adjacent tiles are black and whether marking the tile black does not split the white tiles into two disconnected regions. It tries this for all tiles since if a tile that may be black is not black, there are at least two solutions: one with the tile marked black, one without.

When a solution has been found, it fills in the numbers in the white tiles by choosing a random number for every white tile that does not already appear in that row or column. If no such number exists, it backtracks and tries again. Afterwards, it fills in the black tiles. Every black tile is filled with a number that also appears in the tile's row or column. Finally, the uniqueness is verified by running our ASP model and asking for two solutions. If only one is given, the puzzle is uniquely-solvable.

6.3 Software and Hardware

All experiments were run by calling Clingo, an integrated system for the grounder Gringo and the ASP solver Clasp, from Python. We used Clingo version 5.8.0 and Python version 3.14.0. We generated the graphs with Matplotlib and performed our statistical analyses with SciPy.

The experiments for RQ 1 and RQ 2 were run on an Apple M2 laptop on MacOS, while plugged-in. To minimize the effects of thermal throttling, we alternated running the puzzle between different models to spread any performance impact of thermal throttling evenly over all models.

The experiment for RQ 3 was run on a desktop PC with an AMD Ryzen 7 5800X. That experiment was run inside a docker image.

6.4 Statistical Analyses

To compare the performance difference of the different constraints, we calculated significance using a permutation test, since it does not assume the underlying distribution is normal unlike other tests [31].

To measure the impact of the puzzle properties with performance, we used linear regression. The p-value was calculated using the Wald test with t-distribution (SciPy's default). We considered $p < 0.01$ to be significant.

7 Results and Discussion

RQ1: *What is the impact of different redundant constraints on solving performance?*

In table 1, we can see the effect of the redundant constraint on performance. The Base model + Unique Cell is the fastest model, both in grounding time and solving time ($p < 10^{-4}$ for both). Surprisingly, the second-best model is the base model. Although Sandwich Triple has a higher solving time than the base model ($p < 10^{-4}$), it notably performs identically with respect to number of choices, conflicts and mean conflict size. This result could be explained by ASPs support of preprocessing [32]. We tested this hypothesis by disabling equivalence-based preprocessing and found that the base model had bigger and more conflicts than ST without preprocessing (see appendix C for a full table).

A possible explanation for the fact that most redundant constraints perform worse than the base model might be that the solver could be slower the more constraints it has, and that the extra cost of learning constraints is smaller than the overhead by additional grounded constraints, some of which might not end up being used by the solver. This also explains why Unique Cell improves performance, with Unique Cell the grounder generates less variables and constraints since it is already known during grounding that a tile is white.

Notably, M-Pair, Sandwich Pair, Impossible Pair and Paired Isolation all report a mean conflict size near 1, meaning most conflicts are about a single tile. They also have the fewest conflicts. Since they were promising in this regard, we reran them with Unique Cell. This reduced the number of choices and conflicts and solving time, but the solving time was still higher than that of Unique Cell.

On the other end of the spectrum, Isolated Area stands out by having more than 10 times the solving time and number of conflicts than the base model. The poor performance of Isolated Area likely is due the model generating on average 5.4 loop nogoods of mean size 6.0, while none of the other models generate loop nogoods. Additionally, because it introduces an extra variable per tile, the solver might be wasting time on proving whether a tile is isolated instead of whether it is white or black.

In table 2, we can see the effect of redundant constraints for the corners and edges of a puzzle. Despite being run on relatively small, 10-by-10 puzzles, the performance impact is small. Like with the other constraints, the base model is fastest. Double corner reduces the number of conflicts the most, despite only being applicable in the corner. Surprisingly, Close Edge has more conflicts and its average conflict size is more than 8 times as large.

An interesting result in table 1 and 2 is that every constraint related to connectivity (Quad Middle, Isolated Area and Close Edge) heavily increases the number of conflicts and the average conflict size. These results may be due to these constraints interfering with ASPs preprocessing, since Quad Middle and Close Edge do improve upon the base model in these metrics when preprocessing is disabled.

RQ2: *What puzzle characteristics influence the performance of the ASP model?*

In fig. 5, the average solving with respect to puzzle size is shown in a log-log plot. Surprisingly, most time is spent on grounding. The solving time increases more steeply than the grounding time but never exceeds it for the sizes we tested. The model scales well with n, taking less than 300 ms to solve a 100-by-100 puzzle. Figure 6 shows the distribution of solving and grounding time. There is very little variation in solving time and grounding time.

In fig. 7, the correlation between solving time and number of adjacent-duplicate tiles and between the number of row-column duplicate tiles is shown. There were very little statistically significant correlations. The number of adjacent-duplicate tiles, the number of row-column duplicate tiles and the number of unique tiles all have no correlation with the grounding or solving time for the base model. The number of row-column duplicate tiles is negatively correlated with the number of conflicts ($R = -0.064$, $p = 0.003$). The only significant result we found for the number of unique tiles is the number of choices the solver makes, which it slightly

	grounding time (ms)	solving time (ms)	#choices	#conflicts	mean conflict size
Base model	54	6.3	20.0	2.8	2.5
Unique Cell	54	5.5	11.7	2.4	8.4
Unique Cell Dynamic	61	8.6	5.4	1.9	37.0
M-pair	59	8.1	18.8	1.8	1.0
Sandwich Pair	56	8.1	19.0	1.9	1.1
Sandwich Triple	56	6.4	20.0	2.8	2.5
Paired Isolation	56	8.0	18.9	1.9	1.1
Impossible Pair	61	7.9	19.0	1.9	1.0
Quad Middle	57	7.1	21.6	15.8	41.5
Isolated Area	72	103.0	136.4	31.2	29.1
Unique Cell + Sandwich Pair	57	6.6	10.4	1.7	1.1
Unique Cell + M Pair	59	6.6	10.5	1.7	1.1
Unique cell + PI	56	6.6	10.3	1.7	1.1
Unique Cell + IP	62	6.5	10.2	1.8	1.1

Table 1: Performance of the base model and the base model plus one or two of the redundant constraints that are applicable everywhere. The runtime is split into grounding time and solving time. Values represent the mean of 1000 50-by-50 puzzles.

	grounding time (ms)	solving time (ms)	#choices	#conflicts	avg conflict size
Base model	2.7	0.27	3.7	1.4	1.3
Quad Corner	2.8	0.27	3.7	1.4	1.3
Triple Corner	2.8	0.28	3.7	1.4	1.3
Double Corner	2.9	0.30	3.3	1.2	1.2
Double Edge Pair	2.9	0.28	3.7	1.5	1.3
N-Edge Pair	2.9	0.28	3.7	1.4	1.3
Close Edge	2.9	0.29	3.9	2.3	11.3

Table 2: Performance of the base model and the base model plus one of the redundant constraints of the corners or edges. The runtime is split into grounding time and solving time. Values represent the mean of 1000 10-by-10 puzzles.

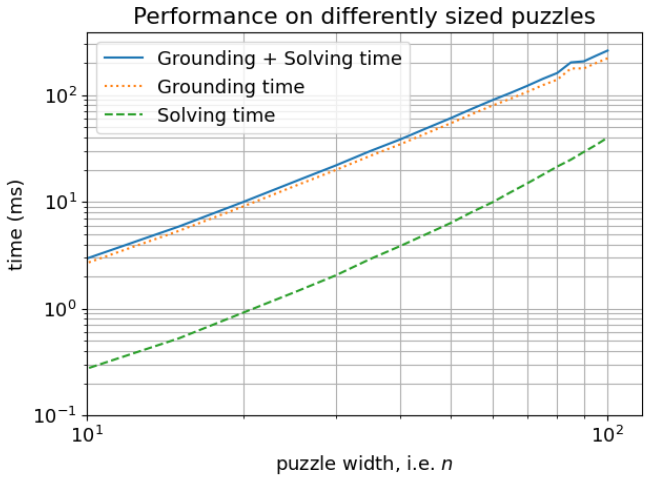


Figure 5: The average grounding and solving time with respect to puzzle size measured in the number of tiles. The figure has logarithmic scaling.

]

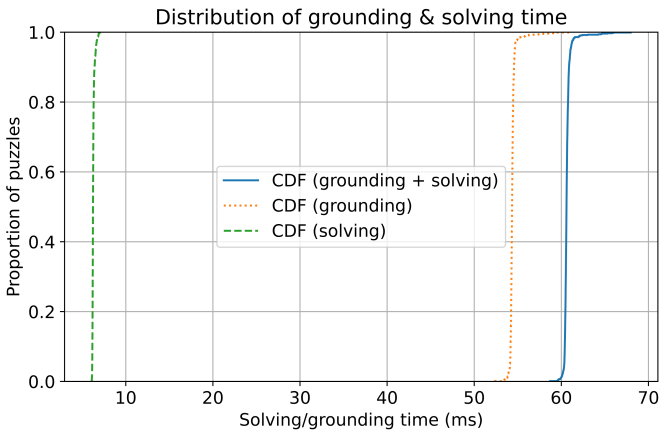


Figure 6: Proportion of 50-by-50 puzzles that are solved within a certain time frame.

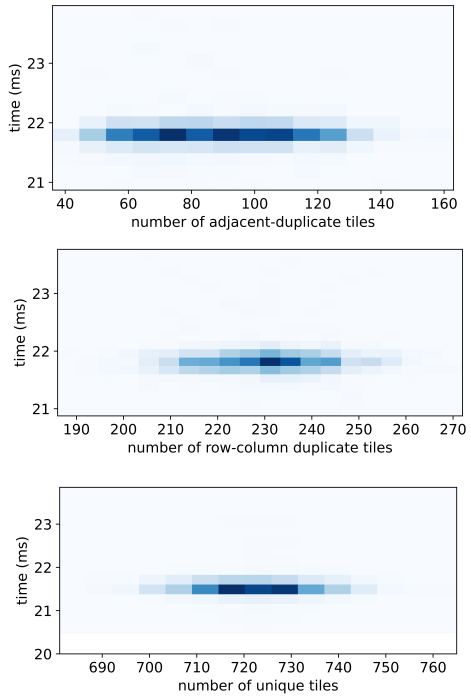


Figure 7: Correlation between solving time and puzzle properties. The puzzles have size $n = 30$.

increases ($R = 0.027$, $p = 0.008$). When using the base model with the Unique Cell constraint, the number of unique tiles is correlated negatively with solving time ($R = -0.064$, $p < 10^{-10}$).

The higher the number of unique numbers that may appear in a puzzle, the lower the grounding and solving time ($p < 10^{-20}$ for both statistics). However, the more numbers that may appear in a puzzle, the higher the amount of conflicts. For a regular 50-by-50 puzzle, the average amount of conflicts is 2.8, whereas this is only 2.1 when a puzzle has at most 40 unique values. On the other hand, if we loosen the rule that the number of a cell is limited by n and allow up to 100 numbers in a 50-by-50 puzzle, there are on average 4.1 conflicts. That the total solving speed decreases while the number of conflicts increases is a surprising result, but could be explained by the overhead extra constraints add since more tiles having duplicate values means more constraints get added during grounding.

RQ3: *How does the runtime performance of different paradigms vary for Hitori puzzles?*

Figure 8 shows the PAR-2 performance of the models made in different paradigms for different puzzle size. ILP and Prolog both scale poorly and always time out for $n = 25$. The SMT, CSP and ASP models are all able to solve puzzles of size $n = 50$, but the ASP model is the only model that never times out. For $n = 50$, the PAR-2 score of the CSP model is 38 times higher than ASP's score. This is in line with the results of [12].

A limitation of the collected data is that the models measured time in slightly different ways, notably the CSP model

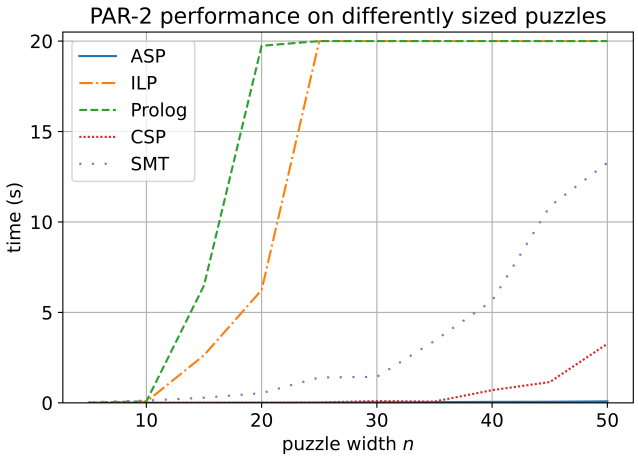


Figure 8: Runtime performance of the models in the different paradigms with respect to puzzle width n. The base model was used for ASP.

measured wall time instead of CPU time. Additionally, the performance is influenced by how well a solver is optimised and thus does not fully reflect the performance potential of other paradigms.

8 Conclusion and Future Work

We explored the suitablity of different modelling and solving paradigms to the NP-complete puzzle Hitori. We made an ASP model and developed a generator that can generate every uniquely-solvable Hitori puzzle.

We investigated the impact of redundant constraints. We showed that they tend to have a negative impact on performance, despite often reducing the amount of guesses and conflicts. However, we found that redundant constraints related to connectivity heavily increase the size of conflicts.

We investigated the relation between puzzle properties and Hitori. We found that the model scales relatively well with the puzzle's size, but other properties had little to no correlation. Limiting the amount of unique numbers used in a puzzle slightly decreased the number of conflicts while increasing performance impact.

Lastly, we showed that ASP performs up to 38 faster in our experiments than models in other paradigms, confirming past research that ASP is well-suited to problems with connectivity constraints.

Additional research is needed to understand why the redundant constraints related to connectivity increase the size of conflicts heavily.

Further studies on Hitori could explore better or alternative ways to generate puzzles. An unbiased generator could be developed, so research results are more generalisable to all Hitori puzzles. This could be realised by using an algorithm for sampling uniformly from CSP solutions, like described by [33].

Responsible Research

We designed our experiments to be reproducible and extendible. The code used for RQ 1 and 2 can be found at https://github.com/sappho3/Thesis-Hitori-ASP. All graphs, tables and statistical analyses can be reproduced by running the python scripts in the benchmarks folder. The used python dependencies and their versions can be found in the `requirements.txt` file. The code for benchmarking the models can be found at https://github.com/sappho3/Thesis-Hitori-shared. The code is licensed under the MIT license so it can be extended by others in the future.

No LLMs were used for this thesis due to the risk of factual errors and due to the high environmental costs of running those models.

Acknowledgements

Thanks to Lesley Smits, Robin Rietdijk, Sophieke van Luenen and Tom Friederich for the feedback, ideas and code they shared. Special thanks my supervisor Anna Latour for supervising the project, giving guidance and the immense amount of feedback on my writing. Thanks to Sarah van de Noort, Jana Dönszelmann and Charlie Ciaś for giving feedback on my writing.

References

[1] T. Yato and T. Seta, "Complexity and completeness of finding another solution and its application to puzzles," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science*, vol. 86-A, no. 5, pp. 1052–1060, 2003.

[2] K. Fazekas, M. Järvisalo, N. Narodytska and P. J. Stuckey, *Interactions in Constraint Optimization*, Schloss Dagstuhl, 2025. [Online]. Available: https://www.dagstuhl.de/25371 (visited on 06/01/2026).

[3] R. Behari, "Solving Fillomino: An Algorithmic and SMT-Based Approach," University Leiden, 31st Jul. 2025.

[4] Rico te Wechel, ""Bridges" as an SMT problem: Solving and generating puzzles using different boolean encodings," 22nd Aug. 2023.

[5] M. Çaylı, A. G. Karatop, A. E. Kavlak, H. Kaynar, F. Türe and E. Erdem, *Solving challenging grid puzzles with answer set programming*, S. Costantini and R. Watson, Eds., Porto: Universidade do Porto, Faculdade de Ciencias, Sep. 2007. [Online]. Available: https://research.sabanciuniv.edu/id/eprint/5086/.

[6] R. A. Hearn, *Games, Puzzles, and Computation*, 1st ed. Florida: CRC Press LLC, 2009, 1 p., ISBN: 978-1-56881-322-6 978-1-4398-6505-7.

[7] M. Gebser, B. Kaufmann and T. Schaub, "Conflict-driven answer set solving: From theory to practice," *Artificial Intelligence*, vol. 187–188, pp. 52–89, Aug. 2012, ISSN: 00043702. DOI: 10.1016/j.artint.2012.04.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0004370212000409 (visited on 10/11/2025).

[8] M. Gander and C. Hofer, "Hitori Solver," BA thesis, Universität Innsbruck, 8th Apr. 2006.

[9] G. van der Knijff, "Solving and generating puzzles with a connectivity constraint," BA thesis, Radboud University, 2021. [Online]. Available: https://www.cs.ru.nl/bachelors-theses/2021/Gerhard_van_der_Knijff___1006946___Solving_and_generating_puzzles_with_a_connectivity_constraint.pdf.

[10] R. Wensveen, "Solving, Generating and Classifying Hitori," Leiden University, 2024. [Online]. Available: https://theses.liacs.nl/3122.

[11] A. Tran, "Enhancing GNNs: An Exploration of Iterative Solving and Augmentation Techniques," BA thesis, ETH Zürich, 24th Aug. 2023.

[12] M. Çelik, H. Erdogan, F. Tahaoglu, T. Uras and E. Erdem, "Comparing ASP and CP on four grid puzzles," in *Proceedings of the 16th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion, RCRA@AI*IA 2009, Reggio Emilia, Italy, December 11-12, 2009*, M. Gavanelli and T. Mancini, Eds., ser. CEUR Workshop Proceedings, vol. 589, CEUR-WS.org, 2009. [Online]. Available: https://ceur-ws.org/Vol-589/paper01.pdf.

[13] "Rules of Hitori puzzle." (2006), [Online]. Available: https://web.archive.org/web/20170812215618/http://www.nikoli.com/en/puzzles/hitori/rule.html (visited on 12/11/2025).

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

[15] V. Hanssen. "Hitori Puzzles." (2006).

[16] S. Tatham. "Singles, from Simon Tatham's Portable Puzzle Collection." (2025), [Online]. Available: https://www.chiark.greenend.org.uk/~sgtatham/puzzles/js/singles.html.

[17] L. G. Valiant and V. V. Vazirani, "NP is as easy as detecting unique solutions," *Theoretical Computer Science*, vol. 47, no. 3, pp. 85–93, 1986. DOI: 10.1016/0304-3975(86)90135-0. [Online]. Available: https://doi.org/10.1016/0304-3975(86)90135-0.

[18] M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub and S. Thiele, "On the Input Language of ASP Grounder Gringo," in *Logic Programming and Nonmonotonic Reasoning*, E. Erdem, F. Lin and T. Schaub, Eds., Berlin, Heidelberg: Springer, 2009, pp. 502–508, ISBN: 978-3-642-04238-6. DOI: 10.1007/978-3-642-04238-6_49.

[19] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and S. Thiele, "A User's Guide to gringo, clasp, clingo, and iclingo," 4th Oct. 2010.

[20] M. Gebser, R. Kaminski, B. Kaufmann and T. Schaub, *Answer Set Solving in Practice* (Synthesis Lectures on Artificial Intelligence and Machine Learning). Cham: Springer International Publishing, 2013, ISBN: 978-3-031-00433-9 978-3-031-01561-8. DOI: 10.1007/978-

3 - 031 - 01561 - 8. [Online]. Available: https : / / link . springer.com/10.1007/978-3-031-01561-8 (visited on 04/01/2026).

[21] C. Barrett, A. Stump and C. Tinelli, *The SMT-LIB Standard*, 21st Dec. 2010.

[22] L. De Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., red. by D. Hutchison *et al.*, vol. 4963, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78799-0 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: http://link.springer.com/10.1007/978-3-540-78800-3_24 (visited on 10/11/2025).

[23] N. Bjørner, L. De Moura, L. Nachmanson and C. M. Wintersteiger, "Programming Z3," in *Engineering Trustworthy Software Systems*, J. P. Bowen, Z. Liu and Z. Zhang, Eds., vol. 11430, Cham: Springer International Publishing, 2019, pp. 148–201, ISBN: 978-3-030-17600-6 978-3-030-17601-3. DOI: 10.1007/978-3-030-17601-3_4. [Online]. Available: https://link.springer.com/10.1007/978-3-030-17601-3_4 (visited on 10/11/2025).

[24] J. P. M. Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, pp. 506–521, 1999. DOI: 10.1109/12.769433. [Online]. Available: https://doi.org/10.1109/12.769433.

[25] S. C. Brailsford, C. N. Potts and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research*, vol. 119, no. 3, pp. 557–581, Dec. 1999, ISSN: 03772217. DOI: 10.1016/S0377-2217(98)00364-6. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0377221798003646 (visited on 10/11/2025).

[26] *Gecode*. [Online]. Available: https://www.gecode.dev/ (visited on 14/12/2025).

[27] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 529–543, ISBN: 978-3-540-74970-7.

[28] M. Flippo, K. Sidorov, I. Marijnissen, J. Smits and E. Demirović, "A multi-stage proof logging framework to certify the correctness of CP solvers," in *30th International Conference on Principles and Practice of Constraint Programming (CP 2024)*, P. Shaw, Ed., ser. Leibniz International Proceedings in Informatics (Lipics), vol. 307, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 11:1–11:20, ISBN: 978-3-95977-336-2. DOI: 10.4230/LIPIcs.CP.2024.11. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2024.11.

[29] B. Crawford, C. Castro and E. Monfroy, "Solving Sudoku with Constraint Programming," in *Cutting-Edge Research Topics on Multiple Criteria Decision Making*, vol. 35, Chengdu, People's Republic of China: Springer, 2009, pp. 345–348, ISBN: 1865-0929. DOI: 10.1007/978-3-642-02298-2_52. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-02298-2_52.

[30] V. Hanssen. "Hitori solving methods," Hitori solving Methods. (2006), [Online]. Available: https://menneske.no/hitori/methods/eng/ (visited on 03/12/2025).

[31] M. D. Ernst, "Permutation methods: A basis for exact inference," *Statistical Science*, vol. 19, no. 4, pp. 676–685, 2004, ISSN: 08834237. JSTOR: 4144438. [Online]. Available: http://www.jstor.org/stable/4144438 (visited on 14/01/2026).

[32] M. Gebser, B. Kaufmann, A. Neumann and T. Schaub, "Advanced preprocessing for answer set solving," in *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, M. Ghallab, C. D. Spyropoulos, N. Fakotakis and N. M. Avouris, Eds., ser. Frontiers in Artificial Intelligence and Applications, vol. 178, IOS Press, 2008, pp. 15–19. DOI: 10.3233/978-1-58603-891-5-15. [Online]. Available: https://doi.org/10.3233/978-1-58603-891-5-15.

[33] V. Gogate and R. Dechter, "A new algorithm for sampling CSP solutions uniformly at random," in *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, F. Benhamou, Ed., ser. Lecture Notes in Computer Science, vol. 4204, Springer, 2006, pp. 711–715. DOI: 10.1007/11889205_56. [Online]. Available: https://doi.org/10.1007/11889205_56.

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

B Redundant constraints

Here we list our ASP implementations of the redundant constraints. The constraints are grouped by whether they apply every in the puzzle or only at the edge or corner.

B.1 Middle Constraints

Unique Cell:

```
is_white(X, Y) :- cell(X, Y, Z), { cell(X, Y2, Z) : Y != Y2; cell(X2, Y, Z) : X != X2} 0.
```

Unique Cell Dynamic:

```
is_white(X, Y) :- cell(X, Y, Z), { is_white(X, Y2) : cell(X, Y2, Z), Y != Y2; is_white(X2, Y) : cell(X2, Y, Z), X != X2} 0.
```

Quad Middle:

```
:- is_white(X, Y), is_black(X - 1, Y), is_black(X, Y - 1), is_black(X + 1, Y), is_black(X, Y + 1), row(Y-1), row(Y+1), col(X-1), col(X+1).
```

Isolated Area:

```
isolated(X, Y) :- is_black(X, Y).
isolated(X, Y), is_white(X, Y + 1) :- is_white(X, Y), isolated(X - 1, Y), isolated(X + 1, Y), isolated(X, Y - 1), col(X), row(Y).
isolated(X, Y), is_white(X, Y - 1) :- is_white(X, Y), isolated(X - 1, Y), isolated(X + 1, Y), isolated(X, Y + 1), col(X), row(Y).
isolated(X, Y), is_white(X - 1, Y) :- is_white(X, Y), isolated(X + 1, Y), isolated(X, Y + 1), isolated(X, Y - 1), col(X), row(Y).
isolated(X, Y), is_white(X + 1, Y) :- is_white(X, Y), isolated(X - 1, Y), isolated(X, Y + 1), isolated(X, Y - 1), col(X), row(Y).
```

Paired Isolation:

```
is_black(X2, Y) :- cell(X, Y, Z), cell(X + 1, Y, Z), X2 != X, X2 != X + 1, row(X), row(X2), col(Y), cell(X2, Y, Z).
is_black(X, Y2) :- cell(X, Y, Z), cell(X, Y + 1, Z), Y2 != Y, Y2 != Y + 1, row(X), row(Y2), col(Y), cell(X, Y2, Z).
```

Flanked Isolation:

```
is_black(X2, Y) : cell(X2, Y, Z1;;X2, Y, Z2) :- cell(X, Y, Z1), cell(X + 1, Y, Z2), cell(X + 2, Y, Z2), cell(X + 3, Y, Z1), 1 <= X2 <= n, X2 < X ; X2 > X + 3.
is_black(X, Y2) : cell(X, Y2, Z1;;X, Y2, Z2) :- cell(X, Y, Z1), cell(X, Y + 1, Z2), cell(X, Y + 2, Z2), cell(X, Y + 3, Z1), 1 <= Y2 <= n, Y2 < Y ; Y2 > Y + 3.
```

Impossible Pair:

```
is_white(X + 1, Y2 ;; X, Y2 + 1) :- cell(X, Y, Z1), cell(X + 1, Y, Z2), cell(X, Y2, Z1), cell( X + 1, Y2 + 1, Z2), Y2 != Y, Y2 + 1 != Y, col(Y2), row(X), X + 1 <= n, col(Y).
is_white(X + 1, Y2 ;; X, Y2 - 1) :- cell(X, Y, Z1), cell(X + 1, Y, Z2), cell(X, Y2, Z1), cell( X + 1, Y2 - 1, Z2), Y2 != Y, Y2 - 1 != Y, col(Y2), row(X), X + 1 <= n, col(Y).
is_white(X2, Y + 1 ;; X2 + 1, Y) :- cell(X, Y, Z1), cell(X, Y + 1, Z2), cell(X2, Y, Z1), cell( X2+ 1, Y + 1, Z2), X2 != X, X2 + 1 != X, col(X2), row(X), X + 1 <= n, col(Y).
is_white(X2, Y + 1 ;; X2 - 1, Y) :- cell(X, Y, Z1), cell(X, Y + 1, Z2), cell(X2, Y, Z1), cell( X2- 1, Y + 1, Z2), X2 != X, X2 - 1 != X, col(X2), row(X), X + 1 <= n, col(Y).
```

Sandwich Pair:

```
is_white(X, Y) :- cell(X-1, Y  , Z), cell(X, Y, _), cell(X+1, Y  , Z), col(X), row(Y).
is_white(X, Y) :- cell(X  , Y-1, Z), cell(X, Y, _), cell(X  , Y+1, Z), col(X), row(Y).
```

Sandwich Triple:

```
is_black(X-1,Y  ), is_white(X, Y  ), is_black(X+1, Y) :- cell(X-1, Y  , Z), cell(X, Y, Z), cell(X+1, Y  , Z), col(X), row(Y).
is_black(X  ,Y-1), is_black(X, Y+1), is_white(X  , Y) :- cell(X  , Y-1, Z), cell(X, Y, Z), cell(X  , Y+1, Z), col(X), row(Y).
```

M-Pair:

```
is_black(X  , Ym) :- cell(X, Y1, Z1), cell(X+1, Y1, Z2), cell(X, Y2, Z1), cell(X+1, Y2, Z2), col(X), col(X+1), row(Y1), row(Y2), cell(X, Ym, Z1), Ym != Y1, Ym != Y2, row(Ym), Y1 != Y2.
is_black(X+1, Ym) :- cell(X, Y1, Z1), cell(X+1, Y1, Z2), cell(X, Y2, Z1), cell(X+1, Y2, Z2), col(X), col(X+1), row(Y1), row(Y2), cell(X+1, Ym, Z2), Ym != Y1, Ym != Y2, row(Ym), Y1 != Y2.

is_black(Xm, Y ) :- cell(X1, Y, Z1), cell(X1, Y+1, Z2), cell(X2, Y, Z1), cell(X2, Y+1, Z2), col(X1), col(X2), row(Y), row(X+1), cell(Xm, Y  , Z1), Xm != X1, Xm != X2, col(Xm), X1 != X2.
is_black(Xm, Y+1) :- cell(X1, Y, Z1), cell(X1, Y+1, Z2), cell(X2, Y, Z1), cell(X2, Y+1, Z2), col(X1), col(X2), row(Y), cell(Xm, Y+1, Z2), Xm != X1, Xm != X2, col(Xm), X1 != X2.
```

B.2 Edge Constraints

Double Edge Pair:

```
edge(1..n, 1, 0, 1).
edge(1, 1..n, 1, 0).
edge(1..n, n, 0, -1).
```

```
edge(n, 1..n, -1, 0).

is_white(X - 2 * DY, Y - 2 * DX), is_white(X + 2 * DY, Y + 2 * DX) :- edge(X, Y, DX, DY), cell(X, Y, Z), cell(X + DY, Y + DX, Z), cell(X + DX, Y + DY, Z2), cell(X + DX + DY, Y + DX + DY, Z2), col(X - 2 * DY), col(X + 2 * DY), row(Y - 2 * DX), row(Y + 2 * DX).
```

N-Edge Pair:

```
edge(1..n, 1, 0, 1).
edge(1, 1..n, 1, 0).
edge(1..n, n, 0, -1).
edge(n, 1..n, -1, 0).

# we have a variable is_edge_pair which detects pairs at the edge
is_edge_pair(X, Y, DX, DY) :- edge(X, Y, DX, DY), cell(X, Y, Z), cell(X + DY, Y + DX, Z), col(X - 2 * DY), col(X + 2 * DY), row(Y - 2 * DX), row(Y + 2 * DX).
# if there is a pair next to an edge pair, we mark that as an "edge pair" and mark tiles white
is_edge_pair(X + DX, Y + DY, DX, DY), is_white(X - 2 * DY, Y - 2 * DX), is_white(X + 2 * DY, Y + 2 * DX):- is_edge_pair(X, Y, DX, DY), cell(X + DX, Y + DY, Z2), cell(X + DX + DY, Y + DX + DY, Z2).
```

Close Edge:

```
edge(1..n, 1, 0, 1).
edge(1, 1..n, 1, 0).
edge(1..n, n, 0, -1).
edge(n, 1..n, -1, 0).
:- is_black(X, Y), is_black(X + 2 * DY, Y + 2 * DX), is_black(X + DX + DY, Y + DX + DY), edge(X, Y, DX, DY),1 <= X + 2 * DY <= n,1 <= Y + 2 * DX <= n,1 <= X + DX + DY <= n,1 <= Y + DX + DY <= n.
```

B.3 Corner Constraints

Quad Corner:

```
is_black(1, 1), is_black(2  ,   2) :- cell(1, 1, Z), cell(1, 2, Z), cell(2, 1, Z), cell(2, 2, Z).
is_black(n, 1), is_black(n-1,   2) :- cell(n, 1, Z), cell(n - 1, 1, Z), cell(n, 2, Z), cell(n-1, 2, Z).
is_black(1, n), is_black(2  , n-1) :- cell(1, n, Z), cell(2, n, Z), cell(1, n-1, Z), cell(2, n-1, Z).
is_black(n, n), is_black(n-1, n-1) :- cell(n, n, Z), cell(n-1, n, Z), cell(n, n-1, Z).
```

Triple Corner:

```
is_black(1, 1) :- cell(1, 1, Z), cell(1, 2, Z), cell(2, 1, Z).
is_black(n, 1) :- cell(n, 1, Z), cell(n - 1, 1, Z), cell(n, 2, Z).
is_black(1, n) :- cell(1, n, Z), cell(2, n, Z), cell(1, n-1, Z).
is_black(n, n) :- cell(n, n, Z), cell(n-1, n, Z), cell(n, n-1, Z).
```

Double Corner:

```
is_white(1, 2) :- cell(1, 1, Z), cell(2, 1, Z).
is_white(1, 2) :- cell(2, 1, Z), cell(2, 2, Z).
is_white(2, 1) :- cell(1, 1, Z), cell(1, 2, Z).
is_white(2, 1) :- cell(1, 2, Z), cell(2, 2, Z).
is_white(n, 2) :- cell(n, 1, Z), cell(n-1, 1, Z).
is_white(n, 2) :- cell(n-1, 1, Z), cell(n-1, 2, Z).
is_white(n-1, 1) :- cell(n, 1, Z), cell(n, 2, Z).
is_white(n-1, 1) :- cell(n, 2, Z), cell(n-1, 2, Z).
is_white(1, n-1) :- cell(1, n, Z), cell(2, n, Z).
is_white(1, n-1) :- cell(2, n, Z), cell(2, n-1, Z).
is_white(2, n) :- cell(1, n, Z), cell(1, n-1, Z).
is_white(2, n) :- cell(1, n-1, Z), cell(2, n-1, Z).
is_white(n, n-1) :- cell(n, n, Z), cell(n-1, n, Z).
is_white(n, n-1) :- cell(n-1, n, Z), cell(n-1, n-1, Z).
is_white(n-1, n) :- cell(n, n, Z), cell(n, n-1, Z).
is_white(n-1, n) :- cell(n, n-1, Z), cell(n-1, n-1, Z).
```

C Performance with equivalence-based preprocessing disabled

Here we list the performance of our different models when equivalence-based preprocessing is disabled. In appendix C the performance is listed for constraints that work everywhere. Appendix C lists the performance for constraints that work in the edge or corner.

	grounding time (ms)	solving time (ms)	#choices	#conflicts	mean conflict size
Base model	57	6.9	21.3	7.6	44.8
Unique Cell	62	7.4	20.3	7.3	43.1
Unique Cell Dynamic	67	9.5	33.7	15.7	5.5
M-pair	62	7.3	21.1	7.6	3.4
Sandwich Pair	58	7.3	27.3	7.8	1.0
Sandwich Triple	59	7.0	20.6	7.4	32.3
Paired Isolation	58	7.2	22.0	6.8	1.0
Impossible Pair	60	6.8	35.0	7.8	1.0
Quad Middle	57	7.1	20.8	7.5	41.8
Isolated Area	76	107.9	230.8	24.4	25.5

Table 3: The performance of the base model and the base model plus one of the redundant constraints that are applicable everywhere. The runtime is split into grounding time and solving time. Values represent the mean of 1000 50-by-50 puzzles. Equivalence-based preprocessing was disabled.

	grounding time (ms)	solving time (ms)	#choices	#conflicts	mean conflict size
Base model	2.7	0.28	3.9	1.1	11.3
Quad Corner	2.8	0.28	3.9	1.1	11.3
Triple Corner	2.8	0.28	3.9	1.1	11.2
Double Corner	2.9	0.29	3.6	1.1	8.2
Double Edge Pair	2.9	0.28	3.9	1.1	11.3
N-Edge Pair	2.9	0.28	3.9	1.1	10.8
Close Edge	2.9	0.29	3.8	1.0	10.4
all	3.7	0.30	3.4	1.0	7.7

Table 4: Performance of the base model and the base model plus one of the redundant constraints of the corners or edges. The runtime is split into grounding time and solving time. Values represent the mean of 1000 10-by-10 puzzles. Equivalence-based preprocessing was disabled.

D 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]
GenerateHitoriInstance Let S = GenerateSolutionTopology Let H = GenerateHitoriInstanceS H is not uniquely solvable
H = GenerateHitoriInstanceS
H

D.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]
GenerateSolutionTopology Let $S[1, \ldots, n][1, \ldots, n]$ be the two-dimensional array of tiles, all unmarked Let C be the collection of all coordinates in S $((1,1),(1,2),\ldots(1,n),(2,1),\ldots,(n,n))$ in random order
$i = C[1]$ to $C[n^2]$ no orthogonally adjacent tile is marked $S[i] = \text{marked}$ the unmarked tiles of S are disconnected $S[i] = \text{unmarked}$
S

Lemma D.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 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. □

D.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.

[1]
GenerateHitoriInstanceFromSS Let $H[1, \ldots, n][1, \ldots, n]$ be a grid of 0s FillUnmarkedTilesH, S, n, 1 FillMarkedTilesH, S, n, 1 H

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

[1] FillUnmarkedTilesH, S, n, k Let $i = \lceil \frac{k}{n} \rceil$ Let $j = ((k-1) \mod n) + 1$
$k > n^2$ true
$S[i][j] == \text{marked}$ FillUnmarkedTilesH, S, n, k + 1
Let row be the numbers used in the row of $H[i][j]$ Let col be the numbers used in the column of $H[i][j]$ $C = \{1, \ldots, n\} \setminus row \setminus col$ $C = \emptyset$ We check if a conflict occurred false this is optimized by analyzing the conflict and returning to the conflict's cause shuffle C $H[i][j] = C[1]$ Assign $H[i][j]$ the first element in C
FillUnmarkedTilesH, S, n, k + 1

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 $1, 2, \ldots, n$ 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 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]
FillMarkedTilesH, S, n, k Let $i = \lceil \frac{k}{n} \rceil$ Let $j = ((k-1) \bmod n) + 1$
$k > n^2$ true
$S[i][j] == \text{unmarked}$ FillMarkedTilesH, S, n, k + 1
Let row be the numbers used in the unmarked tiles of the row of $H[i][j]$ Let col be the numbers used in the unmarked tiles of the column of $H[i][j]$
$C = row \cup col$ shuffle C $H[i][j] = c[1]$ Assign $H[i][j]$ the first element in C
FillMarkedTilesH, S, n, k + 1

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. □

D.3 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 $\square$

E Proof sketch of NP-completeness uniquely-solvable Hitori

We reduce USAT to uniquely-solvable Hitori. We first construct a *partial Hitori solution* P from a USAT formula ϕ. In this partial solution, every tile is either black, white, or contains a number. In the third case, the tile is given a number and labelled with a literal in ϕ. The tile is white iff the literal it is labelled with is true in the solution to ϕ. We note this as a_i, where the tile contains the number i in the (partial) Hitori solution and is white iff a is true in ϕ. From P, we construct a puzzle H *s.t.* the solution to ϕ can be determined from the solution of H by reading off the colours of the tiles labelled with a literal. In our reduction, the puzzle has a solution iff the USAT formula is satisfiable.

Definition E.1 (USAT formula)**.** *Let ϕ be a USAT formula in conjunctive normal form. Every clause c in ϕ consists of one or more* literals*, which is an atom (e.g. a) or a negated atom (e.g. $\overline{a}$). A clause is true if at least one of its literal is true. ϕ is satisfiable if there is an assignment of atoms such that every clause is true. There is at most one assignment of atoms for which every clause is true.*

Definition E.2 (Solution)**.** *Let $S = \langle n, M\rangle$ be a Hitori solution where M is an n-by-n binary matrix such that tile is marked black or marked white. No two adjacent tiles in the matrix may be black (the adjacency constraint). Additionally, all white tiles must be orthogonally connected.*

Definition E.3 (partial Hitori Solution)**.** *Let $P = \langle n, M\rangle$ denote a partial Hitori solution. M is an n-by-n matrix where every tile is marked black, marked white or has a number.*

E.1 Partial Hitori Solution

$S = \langle n, M'\rangle$ is a valid solution to P iff for every tile in M that has a color, M' has the same colour, and for every tile in M that has a number, M' is white or black such that for every pair of tiles in M that have the same number and that appear in the same row, at most one is white in S.

We construct the partial solution P from ϕ as follows. For every clause $l^1 \vee l^2 \vee ... \vee l^k$ in ϕ, we add a *clause gadget* to our partial Hitori solution. The clause gadget contains a wall of black tiles with k gaps, where the tiles of the gaps are filled with a number and labelled one of the clause's literals. We later show that a partial solution P can only have a solution iff at least one of the tiles of labeled with a literal is white.

For every atom x, we add $k-2$ connecting gadgets where k is the amount of clauses atom x is used in. These connecting gadgets ensure that if in solution a tile labeled with a literal is white, all other tiles labelled with that literal are white and all tiles labelled with the negated literal are black. A connecting gadget can connect to at most 3 other gadgets. In between every gadget, before the first gadget and after the last gadget, we add a filler pattern (see fig. 11) to guarantee we can ensure that every tile not labelled with a literal must be black or white. We can ensure P is square by adding filler patterns to the right as needed, and extending all patterns and gadgets downwards according to the filler pattern.

Lemma E.1. *P only has a solution if for all clause gadgets, one of the clause's literals is true.*

	A	B	C	D	E	F	G
1			■				■
2		■			■		
3			a_2	$\overline{a_1}$			■
4		■			■		
5			■			■	
6		■			■		
7			■			■	
8		■			■		
9			■			■	
10		■			■		
11			$\overline{b_1}$	b_2			■
12		■			■		
13			■			■	
14		■			■		
15			■			■	
16		■			■		
17			■			■	
18		■			■		

Figure 9: A clause gadget for the clause $a \vee \overline{b}$.

Figure 9 shows an example of a clause gadget. The black tiles in column B and C of the clause gadget extend from the top of the puzzle to the bottom of the puzzle. For every literal in a clause, a black tile in column C is labelled that literal. To the right of that tile, the tile is labelled with the negated literal.

By the connectivity constraint, a solution is only valid if the white tiles on the left of the gadget are connected to the tiles on the right of the gadget. If for a clause gadget, all tiles labelled with a literal in column C are black, the white tiles on the left of the gadget are disconnected from the white tiles on the right of the gadget. Therefore, P only has a solution if at least one of the tiles labelled with the literals of the clause is white.

E.2 Consistency

Definition E.4 (Consistency)**.** *An atom is* consistent*, iff in a solution to P, all tiles labelled with that atom have the same colour and all tiles labelled with the negated atom have the opposite colour.*

Lemma E.2. *All atoms within a connecting gadget are consistent.*

Proof. Figure 10 shows a connecting gadget. In fig. 10, if both a_1 and $\overline{a_2}$ are black, tile 2B is not connected to the other white tiles violating the connectivity constraint. Therefore, either the tile labelled a_1 or the tile labelled$\overline{a_2}$ must be white. Similarly, either $\overline{a_1}$ or $\overline{a_3}$ must be white, idem for the other literals. Since the tile labelled a_1 has the same number as that of $\overline{a_1}$, if the tile a_1 is white (*i.e.* true), the tile $\overline{a_1}$ is black (*i.e.* false). If $\overline{a_1}$ is black, a_3 in tile 4C is white, thus $\overline{a_3}$ in tile 8C is black. This in turn means a_3 in tile 8A is white, so $\overline{a_3}$ in tile 6A is black and a_2 is white and $\overline{a_2}$ is black. In other words, if the tile labelled a_1 is white, all tiles labelled with a positive

	A	B	C
1		■	
2	a_1		$\overline{a_2}$
3		■	
4	$\overline{a_1}$		a_3
5		■	
6	$\overline{a_3}$		a_2
7		■	
8	a_3		$\overline{a_3}$
9		■	

Figure 10: A connecting gadget. Every connecting gadget can be connected to three other gadgets. The literals on line 2, 4 and 6 can be used for connecting.

	A	B	C	D
1				■
2		■		
3			■	
4		■		
5			■	
6		■		
7			■	
8		■		
9			■	

Figure 11: The filler pattern. This pattern can be extended downwards.

	A	B	C	D
1				■
2		■		
3			■	
4		■		
5			■	
6		■		
7			■	
8		■		
9			■	

Figure 12: A clause gadget for the clause $a \vee \overline{b}$. The clause gadget is connected to two connecting gadgets.

atom are white and all tiles labelled with a negative atom are black. Similarly, it can be shown that if $\overline{a_2}$ is white, all tiles labelled with a negative atom are white and all tiles labelled with a positive atom are white. Thus all literals within the connecting gadget are consistent.

□

Lemma E.3. *All literals are consistent between a connecting gadget and a clause gadget.*

Proof. We connect a connecting gadget to a clause gadget as shown in fig. 12 *s.t.* the literals in the clause gadget are the negated form of the literals that appear in that row in the connected gadget. As shown above, either a_1 or $\overline{a_2}$ must be white. If a_1 is white, $\overline{a_1}$ is black, and by the adjacency constraint a_2 is white. If a_1 is black, $\overline{a_2}$ is white and thus a_2 is black and $\overline{a_1}$ is white.

Thus, the literals in a clause gadget are consistent with those of the connected gadget.

□

Lemma E.4. *All literals are consistent between a connecting gadget and another connecting gadget.*

Proof. Figure 13 shows two connected connecting gadgets. If tile 6A labelled $\overline{a_3}$ is white, a_3 is black and $\overline{a_2}$ is white. Similarly, if tile 6C labelled a_2 is white, $\overline{a_2}$ is black and a_3 is white. Thus the literals in these tiles are consistent among each other. By transitivity, all literals in the first connecting gadget are consistent with all literals in the second connecting gadget. □

Lemma E.5. *All atoms in our partial solution P are* consistent.

	A	B	C	D	E	F	G	H	I	J
1		■			■				■	
2	a_1		$\overline{a_2}$			■				■
3		■			■				■	
4	$\overline{a_1}$		a_3			■				■
5		■			■				■	
6	$\overline{a_3}$		a_2			■		a_3		$\overline{a_2}$
7		■			■				■	
8	a_3		$\overline{a_3}$			■				■
9		■			■				■	
10			■			■		$\overline{a_3}$		a_4
11		■			■				■	
12			■			■		$\overline{a_4}$		a_2
13		■			■				■	
14			■			■		a_4		$\overline{a_4}$
15		■			■				■	

Figure 13: Two connecting gadgets that are connected to each other.

Proof. In P, every tile labelled with an atom is *connected* to every other tile labelled with that atom. If an atom is used in at most three literals, we connect them all to the same connecting gadget. If an atom is used in more than three gadgets, they cannot be connected to the same connecting gadget. In this case, we chain the connecting gadgets without loss of consistency, as shown in fig. 13.

Since literals are consistent within a connecting gadget (theorem E.2, between a connecting gadget and a clause gadget (theorem E.3), literals between a connecting gadget and another connecting gadget are consistent (theorem E.4), and all gadgets are connected in P, all literals are consistent. □

E.3 Constructing the full Hitori puzzle

We now construct the full Hitori puzzle H from our partial solution P. We do this such that P and H will have the same solution. Then we prove that the solution to H can be used to derive the solution to ϕ and vice versa.

Lemma E.6. *We can construct a puzzle H from our partial solution P such that H and P share the same solutions.*

Proof. We construct a puzzle H from our partial solution P. We fill in the numbers of all tiles. For every tile labelled with a literal a_i or $\overline{a_i}$, we put the number i in the tile. In every white tile in H, we put a unique number.

We fill the black tiles in H using the *Sandwich Triple* pattern[30]: If the same number appears three times next to each other, like 1 1 1, the middle number must be white and the outer two must be black. We can apply Sandwich Triple in the filler pattern, the clause gadget and the connecting gadget. Any tile next to these black tiles must be white. Additionally, it can be shown that every other white tile must be white by the connectivity constraint.

The filler pattern and the clause gadget have one black tile in the top row that is not fillable by Sandwich Triple. These tiles can be made black by setting the number of the tile to that of a tile adjacent to a tile forced black by the sandwich triple. The same holds for the rightmost tiles in the clause gadget (see column G of fig. 9).

□

Lemma E.7. *A solution to ϕ can be derived from a solution to H.*

Proof. By lemma E.5, we can construct an assignment of atoms A such that an atom is true in A if the tiles labeled with its positive literals are white, and false in A, if the tiles labeled with its negative literal are white. By theorem E.1, every clause gadget has at least one tile labeled with a literal that is white. Every clause gadget maps one-to-one to a clause in ϕ. Therefore, for every clause in ϕ, there is an atom that is true in A.

□

Lemma E.8. *A solution S of H can be derived from a solution to ϕ.*

Proof. By construction of H, all tiles that have a colour in P, must have the same colour in the S. Every other tile in P is labeled with a literal. Let A be the solution to ϕ. For every tile that is labeled with a literal, we mark it white in S iff the literal is true in A, otherwise we mark the tile black in S. By construction of the clause gadgets, for every clause gadget at least one gap in the wall of black tiles is white thus S is a valid solution.

□

Lemma E.9. *H is* uniquely-solvable*, i.e., it has at most one solution.*

Proof. Assume H has multiple solutions.

The solutions to H can only differ for tiles related to a literal, since we constructed H such that all tiles that have a specific colour in P must be that colour. By theorem E.5, if a tile labelled with a literal changes colour, every tile labelled with that literal or the negated literal must change colour, so we have a different set of literals that is true.

By theorem E.7, we can construct a different solution to ϕ. This violates our premise that ϕ has at most one solution, thus H has at most one solution. □

E.4 Completion of Reduction

Proof. We have reduced USAT formula ϕ to a uniquely-solvable Hitori puzzle H, such that every solution to H maps to a solution to ϕ and vice versa. This reduction can be performed in polynomial time. It is trivial to show that Hitori is in NP. Thus, Hitori is NP-complete. □