



Evaluating Logic Programming in PROLOG for solving Hitori puzzles
A comparison of competing modeling-and-solving paradigms

Tom D. Friederich

Supervisor: Dr. A.L.D. Latour

¹EEMCS, 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
June 12, 2026

Name of the student: Tom D. Friederich
Final project course: CSE3000 Research Project
Thesis committee: Dr. A.L.D. Latour, Examiner: Dr. Tim Coopmans

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Abstract

We evaluate the performance of Logic Programming in PROLOG for solving Hitori puzzles. We analyze the runtime of our model as puzzle size increases, and compare performance to models from competing modeling-and-solving paradigms. We discuss our model approach, and show that incorporating redundant constraints into our model can reduce average runtime. We discuss PROLOG’s backtracking depth first search and compare it with techniques used by SAT solvers and CLASP, and briefly discuss options to incorporate some of these search techniques in PROLOG to potentially improve its performance in problems with large search spaces. In the comparison with other paradigms, Answer Set Programming and Constraint Programming greatly outperformed PROLOG.

1 Introduction

In the academic tradition of using puzzles as test problems for evaluating algorithms, we use Hitori puzzles to evaluate modeling and solving paradigms for solving NP-hard problems. The different kinds of constraints on a Hitori solution are an interesting challenge for modeling and solving paradigms, especially the connectivity constraint that requires the existence of a path between unmarked tiles. As discussed in a recent Dagstuhl seminar, communities for different modeling and solving paradigms rarely interact [11]. To encourage some cross-pollination between these paradigms, a comparison evaluating solving techniques and discussing possible reasons for the differences in performance can be beneficial to the field as a whole. PROLOG is one of the earliest modeling and solving paradigms, and has been a major inspiration to later paradigms such as ASP. Because of that, PROLOG is an interesting data point in this comparison.

1.1 Research questions and goals

The goal of this thesis is to evaluate the usage of LP in PROLOG for solving Hitori puzzles, and to compare its performance to newer modeling and solving paradigms. This brings us to following main research question:

How does a Logic Programming PROLOG model perform for solving Hitori puzzles, and how does it compare to models using competing paradigms?

Apart from runtime when compared to other paradigms, we are also interested in runtime scaling as puzzle size increases, and whether applying redundant constraints could potentially improve performance. This brings us to the following sub questions:

- For each redundant constraint tested: how does applying it influence penalized average runtime (PAR-2)?
- How does penalized average runtime of the PROLOG model scale with puzzle size?
- How does the performance of the PROLOG model compare to other models from competing paradigms?

1.2 The Hitori Puzzle

Hitori is a NP-complete logic puzzle [9] invented by Nikoli Co. Ltd, first published in 1990. Nikoli Co. Ltd is a Japanese company famous for developing logic puzzles and for naming and popularizing Sudoku. It is featured in many newspapers, puzzle books, and puzzle websites.

The Rules of Hitori

A Hitori puzzle is a $N \times N$ grid of numbers, where some rows and columns contain duplicate numbers. The goal of the puzzle is to mark tiles as black until the puzzle satisfies all of the following constraints:

- Uniqueness: the numbers on unmarked tiles must be unique in their row and column.
- Non-adjacency: there may never be two marked tiles in orthogonally adjacent places on the grid.
- Connectivity: the set of unmarked tiles must remain orthogonally connected.

An example Hitori puzzle and its corresponding solution are shown in Figure 1.

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

2	■	4	3
1	4	■	2
■	2	1	4
4	3	2	■

Figure 1: A Hitori puzzle (left), and its solution (right). In the solution, each number is unique its is row and column, there are no adjacent black squares, and the unmarked tiles remain orthogonally connected to each other.

Solution uniqueness

The uniqueness of solutions is often unmentioned for Hitori, but we only consider puzzles with unique solutions. Puzzles by Nikoli Co. Ltd have unique solutions too.

2 Background

We discuss PROLOG, its backtracking search algorithm, and compare it to methods employed by other, often newer paradigms and languages. For a more detailed explanation of logic programming in PROLOG, we refer the reader to the SWI-PROLOG website.¹

2.1 PROLOG

The first version of PROLOG was released in 1972. It is a programming language based on predicate logic, where the user defines logical relations. After defining these relations, the user can query PROLOG for possible solutions in the same language. PROLOG can then find one or all existing solutions that can be derived from the model. To find these solutions, PROLOG uses a depth first backtracking algorithm. PROLOG and LP has been a clear inspiration for other later paradigms, such as Constraint Programming (CP) [1] and Answer Set Programming (ASP) [13].

¹<https://www.swi-prolog.org/>

PROLOG: rules, facts, querying

PROLOG models contain rules to define logical relations. Rules can be read from left to right, where “:-” can be read as “IF”. The “;” operator can be read as “AND”, separating a number of goals. The “,” operator can be read as “OR”. A fact is equivalent to a rule with only the goal ‘true’. We include an example rule from our Hitori model that models two diagonally adjacent positions:

```
diagonallyAdjacent((X1,Y1),(X2,Y2)) :-  
    (X2 is X1 + 1, Y2 is Y1 + 1);  
    (X2 is X1 + 1, Y2 is Y1 - 1);  
    (X2 is X1 - 1, Y2 is Y1 + 1);  
    (X2 is X1 - 1, Y2 is Y1 - 1).
```

After defining this rule, the user can query for adjacent positions, asking PROLOG for all possible assignments of the variable “Adjacent”:

```
?- diagonallyAdjacent((1,2), Adjacent).  
    Adjacent = (2, 3) ;  
    Adjacent = (2, 1) ;  
    Adjacent = (0, 3) ;  
    Adjacent = (0, 1).
```

PROLOG: predicates and more complex relations

In the example above, we used the “;” operator to model the four diagonally adjacent positions, but we could have also used a combination rules and facts with the same name. This set of rules/facts (clauses) is called a predicate. Facts are especially useful for defining base cases for recursive relations. An example is the recursive factorial predicate below:

```
factorial(0,1). %base case: 0! = 1.  
  
factorial(X, Result) :-  
    Prev is X - 1,  
    Prev >= 0,  
    factorial(Prev, PrevFactorial),  
    Result is X * PrevFactorial.
```

PROLOG also has higher order predicates, to apply operations such as map, filter, or to find all solutions to a query.

PROLOG: evaluation

PROLOG queries consist of one or more goals, which are evaluated using a combination of backtracking and the unification algorithm. Unification attempts to recursively unify subterms until it proves that two terms can be unified. This process can fail, for example $c(1,2,3)$ and $c(X,Y)$ cannot be unified, because the arity does not match. $t(1,2,p(\text{"PRO"}, \text{"LOG"}))$ and $t(X,Y,Z)$ can be unified, and PROLOG will report the variable bindings $X = 1$, $Y = 2$ and $Z = p(\text{"PRO"}, \text{"LOG"})$. If a goal fails, PROLOG will backtrack to a previous choice point. If all choice points are exhausted PROLOG will print `false`, meaning there are no more possible solutions. There is a distinction between instantiated and uninstantiated variables: when the unification process unifies a value with a variable, the variable becomes instantiated. When two variables are already fully instantiated, they can only unify when the values are equal. When two variables are uninstantiated, PROLOG will give them a shared alias.

To ease the evaluation process, PROLOG implementations commonly compile PROLOG to some simpler intermediary language. This is often the Warren Abstract Machine (WAM) language, but the SWI-PROLOG implementation used by this project targets another intermediary language [12].

The order in which goals are defined within a rule influences the shape of the search tree. The order in which the clauses that make up a predicate are defined determines the search tree traversal order: the first clause is traversed first.

2.2 Search in PROLOG versus other modeling-and-solving paradigms

PROLOG’s search strategy is depth first: the clause that is defined first in the model is explored first. This causes problems when the search tree for the first clause is large: PROLOG will first exhaust that tree completely before considering a second clause. When this search tree is infinite, PROLOG will not find a solution, even if a direct fact satisfying the query is defined immediately below it. Although PROLOG is considered a declarative language, it has some imperative elements, and it is possible to write code in a more imperative style. ASP and CP are more purely declarative. Because of that, they can move most of the consideration of the search process from the model programmer to the developers of the solver. These solvers can be highly optimized, with considerable resources poured into improving their performance. Many CP and ASP languages use SAT (Boolean satisfiability problem) solvers to find solutions, so these solvers can be used by multiple modeling languages. Solvers developed by large corporations such as Google and Microsoft compete against academic and open source solvers in competitions such as the yearly MiniZinc challenge² and the International SAT Competition³.

2.3 SAT Solver search strategies

Most SAT solvers use the DPLL (Davis Putnam Logemann Loveland) algorithm [2], which uses a backtracking search method just like PROLOG. Modern SAT solvers implement various improvements on basic DPLL that generally allow for a much smarter and a more even exploration of the options in the search tree (in terms of runtime/effort). Simply trying the first clause defined in the model is out of the picture: modern SAT solvers apply various heuristics to decide on which subtree to explore first [10]. We discuss some enhancements made to DPLL below.

Backjumping

A common enhancement of DPLL is Backjumping or non chronological backtracking. Instead of going back one up one level when backtracking, the search process can immediately jump back multiple levels, to explore other subtrees of the search tree. The search process can jump back to the origin of the backjump if no solution is found after the jump. Backjumping may allow the Hitori puzzle search tree to be explored more evenly: It can avoid the solver spending all of its time investigating an incorrect first decision.

²<https://www.minizinc.org/challenge/>

³<https://satcompetition.github.io/>

Conflict driven clause learning (CDCL)

CDCL is an enhancement to regular backjumping. Instead of jumping back to an arbitrary number of decisions, a CDCL solver can jump back to the source of the conflict, and then learn this incompatibility for the future. CDCL seems to be even more interesting for solving Hitori puzzles, as the set of possible solutions for each individual row/column is generally in conflict with those of other rows and columns. CDCL is employed by multiple SAT solvers, including Z3. Clasp, the solver for ASP, does not rely on a SAT solver, but implements its own conflict driven solver instead [15].

Randomization

Multiple randomization techniques exist for DPLL. Gomes et al. [6] describe how they randomly pick from the possible options in the search tree, if heuristics have no clear preference for one over the other. They also randomly restart the search process after a certain amount of backtracks occur: this can indicate that the search might be spending too much time there. Some bookkeeping can be applied to avoid the search process exploring the same areas again. In their tests, they demonstrate that these randomization techniques can provide speedups of several orders of magnitude on real world problems. Random restarts can be especially powerful when combined with CDCL, as clauses learned in previous runs can be applied to later runs. Modern SAT solvers restart very often because of this [7].

3 Prior work

We discuss prior research in Hitori puzzles, and also at enhancing PROLOG’s search algorithm.

3.1 Hitori

Wensveen [16] discusses Hitori puzzles, analyzes puzzle solving and puzzle generation techniques, attempts to classify Hitori puzzle difficulty, and presents techniques to encode the Hitori puzzle so that it can be solved by a SAT solver. The techniques discussed require spotting various properties or patterns of puzzles to deduce whether or not a tile must be marked. These rules follow from the 3 constraints for a Hitori solution, so we call these ‘redundant constraints’. We discuss the redundant constraints in our model in subsection 4.3.

In their Bachelors thesis, [5] implement two Hitori solvers, an imperative one and one that uses an interesting technique to encode the difficult connectivity constraint that is difficult for a SAT solver. Their method generates all possible cycles of duplicates that break connectivity. They mention that the file containing these cycles can reach gigabytes in size for larger puzzles. We use the same cycles idea to detect when connectivity has been broken in our PROLOG model in an iterative way. We describe how it works in subsection 4.2.

3.2 PROLOG search enhancements

Nakamura [14] implements a version of PROLOG that incorporates heuristic search. The model developer can enable this search for a predicate manually using a special predicate. This shows that it is possible to create a version of PROLOG that does not execute in depth first order.

Foong [4] also implements a version of heuristic search for PROLOG, but with the aim of improving parallelism. The focus lays on attempting to estimate which sub trees are the most work to compute, so that those can be computed on other processors.

Drabent [3] describes using exceptions to implement manual backjumps (by the model developer) in some restricted cases, and shows that it is impossible in the current version of PROLOG in general. This does not mean that it is impossible for the PROLOG solver to do backjumping, only for the model developer. We did not find any papers describing implementations of any randomization techniques, random restarts or CDCL like techniques in PROLOG.

4 Approach

We describe the puzzle solving approach used by our PROLOG model, and explain the redundant constraints that we selected to try to improve performance. The model uses SWI-PROLOG, which was chosen because it is a non-proprietary and mature PROLOG implementation.

Different versions of our PROLOG model use different approaches, but we focus on describing the techniques used by the latest version of the model. For an overview of all versions, their puzzle solving approach, major changes and a performance comparison, see Appendix B and Appendix C.

4.1 Iterative marking

The latest version of the model uses an iterative marking strategy: puzzles are solved by marking tiles in one row/column after the other. As proven in Appendix E, all non duplicate numbers can be safely assumed to be white, so they are completely ignored.

The model pre-computes a dictionary for each row and column that contains the amount of times each number occurs in each row/column. From that, it creates a set of duplicate tiles, a subset of which needs to be marked to satisfy the uniqueness constraint. Since PROLOG uses linked lists that have expensive random lookups, the process is designed in such a way that the model never needs to compute or look up if a value is still a duplicate, all the required information is kept in the dictionaries and the sets of duplicates that have fast lookups.

After pre-computing the dictionary and the set, the iterative marking process starts: a solution for each rows/columns is found one by one, moving from top to bottom, and from left to right. A solution for a row/column is found by repeatedly attempting to mark a duplicate tile. When a tile is marked, the number of times the number occurs in this row/column is updated in the dictionary, and the number is removed from the set of duplicates when it is no longer a duplicate. When the whole set of duplicates is empty, the row/column satisfies the uniqueness constraint. Before a tile is marked, it is ensured to not be adjacent to other tiles marked in the current row/column and in other rows/columns. PROLOG will backtrack when this non-adjacency constraint is not met. It will backtrack to previous rows/columns when it finds that no possible solutions exist for the current row/column given the choices made before.

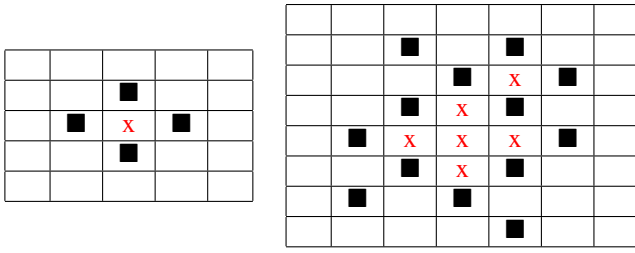


Figure 2: Minimal example (left) and a larger example (right) of how a cycle can 'cut off' some unmarked tiles from the rest, violating the connectivity constraint.

Algorithm 1 Iterative cutoff detection. This function uses CutoffSearch on all newly marked start positions to check if a path or cycle can be found exploring from any of them.

```

function ISSTILLCONNECTED(newlyMarkedPositions,
AllMarkedPositions, N)
  for all startPosition in newlyMarkedPositions do
    queue  $\leftarrow$  new FIFO queue
    enqueue (startPosition, NULL) into queue
    if CUTOFFSEARCH(queue,
AllMarkedPositions,  $\emptyset$ ,  $\emptyset$ , N, false) then
      return false
  return true

```

```

function CUTOFFSEARCH(queue, markedPositions,
usedEdges, visited, N, foundSide)
  (current, from)  $\leftarrow$  pop from queue
  if (current, from)  $\in$  usedEdges then
    return CUTOFFSEARCH(queue,
markedPositions, usedEdges, visited, N,
foundSide)
  if queue is empty then
    return false  $\triangleright$  No cutoff found
  cIsOnSide  $\leftarrow$  current.x == 0 OR current.x == (N-1) OR current.y == 0 OR current.y == (N-1)
  if cIsOnSide AND foundSide then
    return true  $\triangleright$  Cutoff path found! (found a side of the board 2 times)
  if current  $\in$  visited then
    return true  $\triangleright$  Cutoff cycle found! (already visited this node without reusing edges)
  neighbours  $\leftarrow$  { e  $\in$  markedPositions | e is diagonally adjacent to current }
  for all neighbour in neighbours do
    enqueue (neighbour, current) into queue
  add current to visited
  add (current, from) to usedEdges
  newFoundSide  $\leftarrow$  foundSide OR cIsOnSide
  return CUTOFFSEARCH(queue, markedPositions,
usedEdges, visited, N, newFoundSide)

```

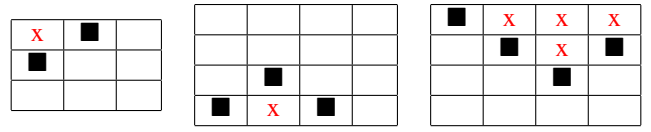


Figure 3: Examples of paths from and to a side of the board that 'cut off' some unmarked tiles from the rest of the unmarked tiles, violating the connectivity constraint. These paths must have at least length 2, a single tile in the corner does not violate connectivity.

4.2 Cutoff detection for the connectivity constraint

Starting from V3.0 of the PROLOG model, we use a method to enforce the connectivity constraint that we call "iterative cutoff detection". After satisfying all other constraints for a row/column, it checks if its solution including the newly marked tiles still satisfies the connectivity constraint. It does so in a way that is similar to [5], but integrated in the search process.

Instead of checking if the unmarked tiles remain connected (like the connectivity constraint dictates), it attempts to find ways that marked tiles 'cut off' some set of the unmarked tiles from the rest. Using the fact that marked tiles can never be orthogonally adjacent because of the adjacency constraint, unmarked tiles can only be 'cut off' in two ways:

1. A cycle that encloses some amount of tiles, shown in Figure 2.
2. A path of marked tiles from a side of the board to a side of the board, shown in Figure 3. This path 'cuts off' at least one unmarked tile from the rest of the unmarked tiles. The path can reach two different sides or the same side of the board twice, but it needs to have at least length 2: a single marked tile in the corner reaches two sides of the board but does not cut off any tiles.

Any other path of marked tiles does not 'cut off' tiles from the rest, so if none of these paths exist, a potential solution satisfies the connectivity constraint.

The iterative search process only continues to the next row/column when it satisfies all constraints. Because of that, the cutoff detection process only needs to search for cycles or paths to the side of the board starting from newly marked tiles in the newest row/column. We include imperative pseudocode for iterative cutoff detection algorithm here, to illustrate the idea. Note that the PROLOG code is significantly different.

4.3 Redundant constraints

The latest version of our PROLOG model can apply up to 3 redundant constraints. The redundant constraints that we selected can all occur many times in one puzzle and do not depend on partial solutions. More redundant constraints can be derived from human Hitori solving techniques, (see [8]) but many require partial solutions, more complex situations that are more difficult to detect, or only occur a limited amount of times in the puzzle (for example ones involving the corners). We describe each selected redundant constraint in detail in the following sections. All 3 redundant constraints can be applied both vertically and horizontally.

Sandwich pair

The sandwich pair redundant constraints allows us to determine a tile that must always remain unmarked when it occurs. An example is shown in Figure 4. In the situation on the left, there are two 1's. At least one of them needs to be marked to satisfy the uniqueness constraint. This makes it impossible to mark the middle tile, because that would violate the adjacency constraint, requiring the middle 2 to remain unmarked.

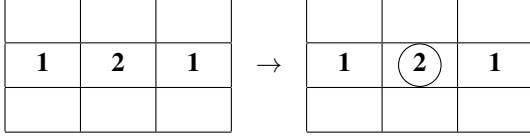


Figure 4: Example application of the sandwich pair redundant constraint. A circle denotes a tile that is known to always remain unmarked.

Sandwich triple

The sandwich triple redundant constraint occurs less often than the sandwich pair, but it provides us with more information when it does. When three adjacent equal tiles occur, it lets us determine that two tiles must remain marked, and one unmarked. When combined with the adjacency constraint, we can definitively unmark the neighbors of the marked tiles too. An example of the sandwich triple redundant constraint is shown in Figure 5. In the situation on the top, the middle tile can never be marked, as doing so still leaves two 1's in one row that cannot be marked because of the adjacency constraint. So, the two outer tiles need to be marked, and the middle one needs to remain unmarked.

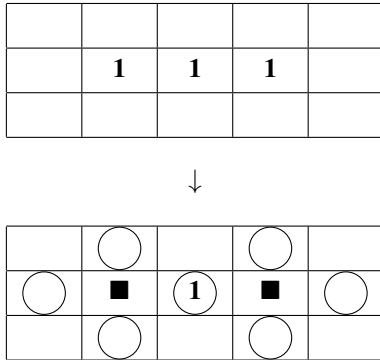


Figure 5: Example application of the sandwich triple redundant constraint. A circle denotes a tile that is known to always remain unmarked.

This check can be partially combined with the sandwich pair check: a sandwich triple also always contains a sandwich pair. The difference is the tile in between the pair. The PROLOG model takes advantage of this dependency: it only checks for occurrences of a sandwich triple when it finds a sandwich pair.

Pair isolation

An example of the pair isolation redundant constraint is shown in Figure 6. When two adjacent tiles have the same value, it is impossible to mark both, as that would violate the adjacency constraint. Since all but one tile of the same value in that row/column needs to be marked to satisfy the uniqueness constraint, we know that all tiles that are not the two adjacent tiles need to be marked. Just like with a sandwich triple, we can also definitively unmark the neighbors of the marked tiles.

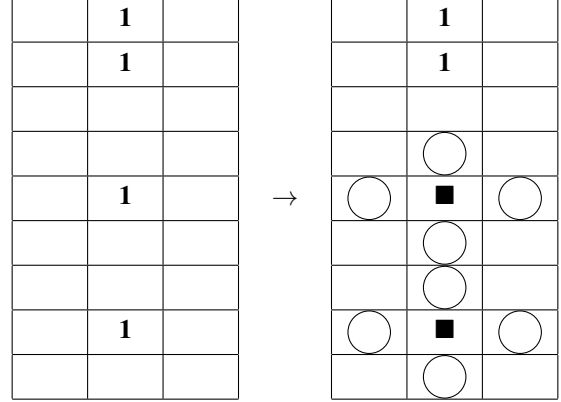


Figure 6: Example application of the pair isolation redundant constraint. A circle denotes a tile that is known to always remain unmarked.

5 Experimental setup

We describe the experiments we conducted to answer our research questions, and discuss time measurement and our benchmark environment.

5.1 Influence of redundant constraints

To determine the influence on performance of the redundant constraints described in subsection 4.3, we created 8 variants of the latest version of our PROLOG model, V4.0, to evaluate the performance impact of applying every individual constraint and every possible combination of redundant constraints. We tasked each variant with solving the same 1000 randomly generated puzzles created by our puzzle generator. For these 1000 puzzles we chose $N=15$, because we found that to be a puzzle size where our model can solve most but not all puzzles within the timeout. We report on PAR-2, the amount of puzzles solved, and include a cactus plot to show the distribution of solving times measured.

5.2 PAR-2 scaling with puzzle size

To determine how penalized average runtime scales with puzzle size, we used our generator to create a puzzle set of puzzles ranging from $n=5$ to $n=25$. The set contains 250 puzzles of each size, totaling 5250 puzzles. To confirm that the variant with redundant constraints outperforms the variant without redundant constraints at all puzzle sizes tested, we tasked both models with solving all puzzles. We used a 10 second timeout, and we report on PAR-2 and the amount of puzzles solved for each puzzle size.

5.3 Comparison with other paradigms

To determine how the performance of the PROLOG model compares to other models using competing paradigms, we evaluate how reliably puzzles of increasing size can be solved by each model within a timeout. We randomly generated 25 puzzles of each size in increments of 5, totaling 500 puzzles. We then tasked each model with solving all of these puzzles with a timeout of 10 seconds. We report on the percentage of puzzles solved instead of PAR-2, because PAR-2 becomes meaningless when (almost) no puzzles of size N can be solved. Unfortunately, we could not use the best performing PROLOG model because of practical limitations: the PROLOG model used was V3.0.1, which performs slightly worse than V3.0.2 (the best version without redundant constraints).

5.4 Benchmark environment

Each model was limited to a single CPU core, and 8GB of memory. All PROLOG-only benchmarks were ran natively on an M2 Macbook Pro on macOS Sequoia. The comparison benchmark from subsection 6.3 was ran in a docker container on a desktop PC equipped with an AMD Ryzen 7 5800x processor. The SWIPL version used was 9.2.9.

5.5 Time measurement

Each model is responsible for their own time measurement. Time measurement includes only the puzzle solving process itself: it excludes reading and parsing of the puzzle file and writing solution files to disk. It also excludes any startup time of interpreters (if applicable). In tests where time was limited, each model received the allotted time (10 seconds for all experiments) plus one extra second per puzzle to account for startup and reading/writing the files. If a puzzle was not solved, their process group was killed. If the measured puzzle solve time exceeded the allotted time, it was counted as a failure.

6 Results

We discuss the results for each research question separately. The results for an additional experiment measuring the performance of various versions of the PROLOG model can be found in Appendix B.

6.1 For each redundant constraint tested: how does applying it influence penalized average runtime (PAR-2)?

Figure 7 shows the results for the experiment measuring the influence of redundant constraints on PAR-2 at $N=15$, as described in subsection 5.1. The table shows that all 3 redundant constraints tested decrease PAR-2 and increase the amount of puzzles that could be solved. The cactus plot confirms this visually: the variant without redundant constraints corresponds to the gray line on the left. This line is consistently above the variants with redundant constraints, meaning it takes more time to solve puzzles for this variant, and it can solve fewer puzzles within the 10 second timeout. The cactus plot also shows that the shapes of the graphs are similar: this means the distribution of runtime remains is largely unchanged when applying redundant constraints.

Both the table and the cactus plot show that the influence of Pair Isolation is much larger than the influence of Sandwich pair and Sandwich Triple. The influences also seem independent: every combination of redundant constraints is better than the variant with only its constituent parts. The variant with all redundant constraints enabled performs the best.

6.2 How does penalized average runtime of the PROLOG model scale with puzzle size?

Figure 9 shows the results of the experiment measuring PAR-2 for various puzzle sizes for both a variant with and without redundant constraints, as described in subsection 5.2. The top graph shows what percentage of puzzles could be solved for each N . The experiment shows that most puzzles up to about $N=15$ can be solved within 10 seconds, and that the model with redundant constraints performs better or equal to the model without redundant constraints at all puzzle sizes tested. The second graph shows PAR-2 scaling as N becomes larger. We will focus on the better model with redundant constraints: it solved more puzzles and its PAR-2 is consistently lower. We can discern 3 phases as N grows in this graph: a 'trivial' phase, a 'growth' phase and a 'timeouts' phase. In the first phase, small puzzles up to about $N=10$ are solved almost instantly: all puzzles of this size are solved, and the average runtime is below 0.02 seconds. In the last phase, most of the PAR-2 growth can be explained by an increasing portion of puzzles resulting in timeouts, until PAR-2 approaches 2 times the timeout. This begins around $N=15-16$.

Empirical analysis of PAR-2 scaling in the growth phase

The most interesting phase to analyze is the growth phase, where puzzles are nontrivial but timeouts are rare: around $N=10$ to $N=15$. In this region, the fourth column of the table shows that the growth factor of PAR-2 increases as n increases. This indicates PAR-2 scales faster than exponential with N . However, the amount of tiles in a puzzle scales with N^2 , and the amount of ways to mark a puzzle is 2^{n^2} , so it would not be unreasonable to expect PAR-2 to scale exponentially with n^2 . Our measurements seem to support that hypothesis: adding a tile in the growth phase roughly corresponds to an increase in PAR-2 of around 4-5%. For example, when comparing $N=11$ to $N=12$, 23 tiles are added ($12^2 - 11^2 = 23$), and runtime scales by a factor of 1.0425^{23} which corresponds to an increase of 160%. See the last column in the table. This indicates that PAR-2 in this region may scale exponentially with the amount of tiles in the puzzle.

6.3 How does the performance of the PROLOG model compare to other models from competing paradigms?

Figure 8 shows the results for the experiment comparing the performance of the PROLOG model against Hitori models created using competing paradigms, as described in subsection 5.3. The PROLOG model performs the worst of all, just behind Gurobi. The PROLOG model solved all 10×10 puzzles, around 80% of the 15×15 puzzles and rarely solved a 20×20 puzzle. The CLASP model performs the best, solving all puzzles tested, even up to the largest 50×50 size. PUMPKIN finishes second, and Z3 third.

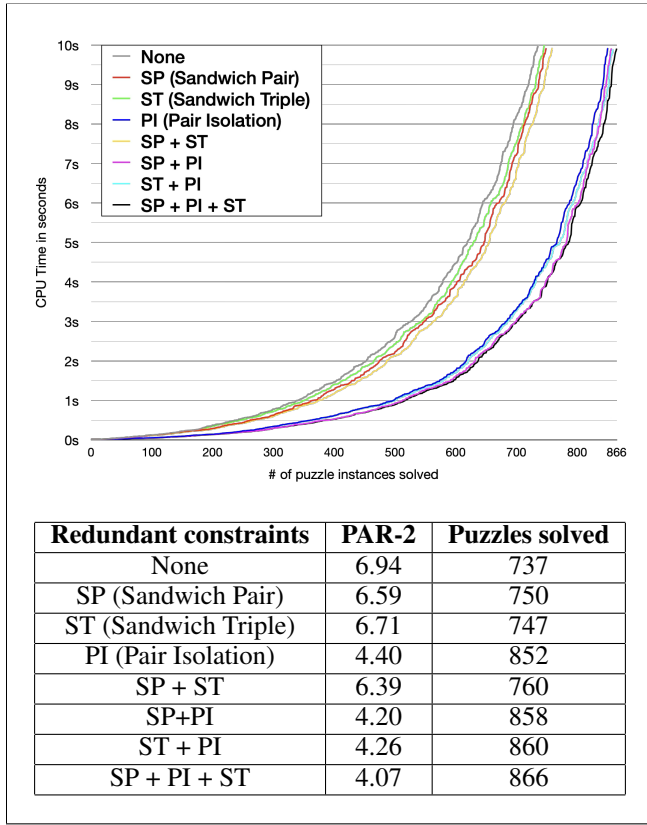


Figure 7: Results for the experiment described in subsection 5.1: a cactus plot and a table with the PAR-2 measurements for all variants of V.4.0 of the PROLOG model. The cactus plot shows how many puzzles could be solved within the amount of time on the Y axis. The timeout for this experiment was 10 seconds, and the puzzle set consisted of 1000 puzzles.

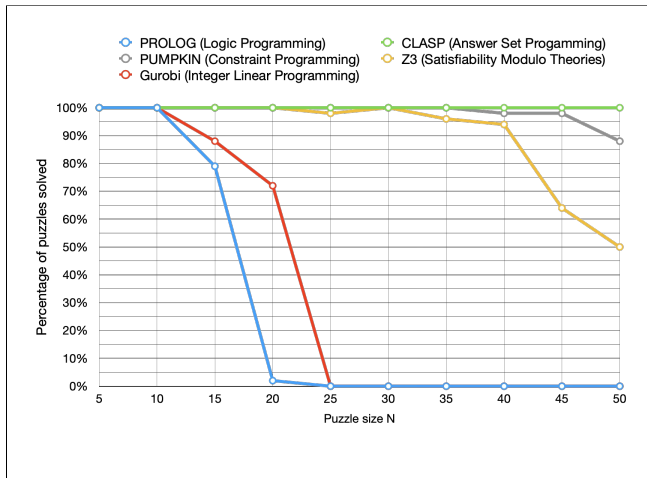


Figure 8: Results for the experiment described in subsection 5.3, showing how reliably puzzles of various sizes can be solved by the competing paradigms within 10 seconds. The PROLOG model used was V3.0.1. This experiment was ran in a docker container on a desktop PC equipped with an AMD Ryzen 7 5800x processor

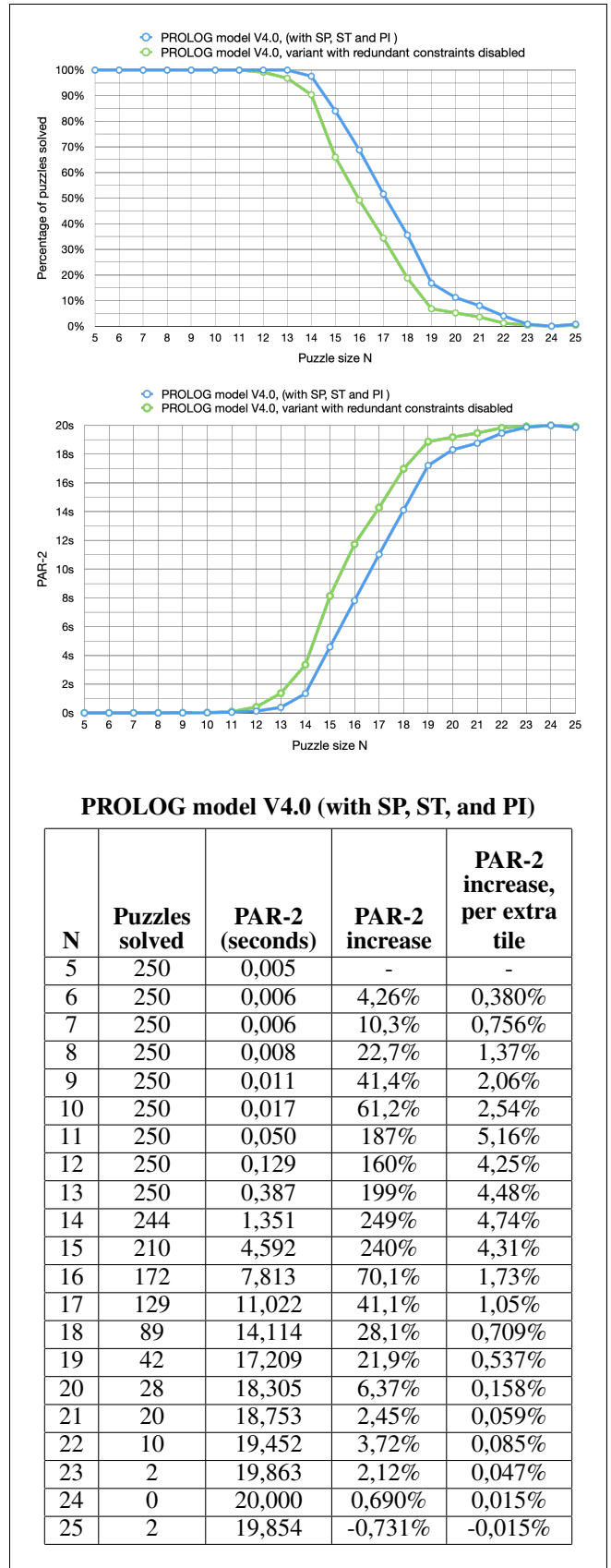


Figure 9: Results for the experiment described in subsection 5.2: a plot showing the percentage of puzzles solved as N grows, a plot of PAR-2 as N grows, and a table with exact measurements for the model with redundant constraints, along with increases in PAR-2 over the previous data point. Puzzle set of 250 puzzles for each N, timeout of 10 seconds.

7 Conclusion

Our experiments show that the performance of our PROLOG model can be improved using redundant constraints, and that our model performed best at all relevant puzzle sizes when all redundant constraints were enabled. Depending on the timeout used, our model can fairly reliably solve puzzles up to $N=15$. Our empirical PAR-2 analysis shows that PAR-2 scales in phases: a phase where puzzles are trivial, a phase where timeouts are rare/ non-existent, and a phase where PAR-2 is dominated by timeouts. In the middle phase, runtime seemed to increase by about 4-5% for each tile added to the puzzle. Finally, our comparison experiment shows models from other modeling and solving paradigms performed much better than our PROLOG model: The CLASP model could solve 100% of largest puzzles tested at $N=50$. Because of that, we can conclude that the performance of our Logic Programming PROLOG model does not perform very well at solving Hitori Puzzles, at least when compared to other paradigms. It seems to be limited by its basic search method when compared to CLASP, Z3, and PUMPKIN.

7.1 Limitations and further research

We are especially interested in the possibility of applying various search optimizations from other paradigms to PROLOG. We discuss this further in section 8.

Another interesting area of research that we did not explore is Constraint Logic Programming (CLP). This paper focuses on pure Logic Programming, but there exist libraries that extend PROLOG with some concepts from the Constraint Programming paradigm: CLPFD (Constraint Logic Programming over Finite Domains). The performance of these libraries compared to pure Logic Programming and other paradigms for solving Hitori puzzles remains unexplored.

A limitation of our research is that we tested 3 redundant constraints, but more exist for Hitori puzzles. Our experiments show that the performance of the PROLOG model can be improved using the redundant constraints we investigated. How much performance is still left on the table by applying more redundant constraints is unclear.

8 Discussion

In the spirit of the recent Dagstuhl seminar, we wonder if it may be possible to apply some search optimizations from more recent paradigms to PROLOG. We imagine giving the PROLOG programmer the possibility to mark individual predicates as 'safe to traverse in any order'. An advanced search algorithm could then freely explore the search tree for these predicates in an efficient manner, using techniques inspired by DPLL enhancements such as backupjumping and randomization. By requiring this mark, it would not break compatibility with existing PROLOG code, and the programmer would still have access to the imperative elements of PROLOG with some direct control over search if desired. We envision that this 'marking' process works similarly to the 'table' predicate in SWI-PROLOG that prepares a specified predicate for tabled execution.

8.1 Beyond performance: other criteria

The main goal of this research project is to compare PROLOG to other paradigms. We focus on performance as the main criterium. Performance is easy to measure, but other factors less so. We include some observations and opinions on PROLOG when compared with the other paradigms.

Paradigm Problem match

In a vacuum, expressing the Hitori problem using logical rules is relatively natural. The initial version of the model is fairly simple and declarative, and models the rules of Hitori closely. This decreased somewhat with newer faster versions, adopting a more imperative style that focusses on efficiently passing the relevant information to various predicates in the pursuit of more performance.

It is our opinion that our model more closely matches the logic of Hitori than a hypothetical model in an imperative language, because the search part of the hypothetical imperative model is abstracted away by PROLOG. The redundant constraints were also intuitive to implement in a declarative style. The code seems fairly intuitive for those familiar with PROLOG, except for the manipulation and preparation of all the required datastructures to get the correct data at the correct place. When comparing the PROLOG model to CLASP, the CLASP model is much more high level and easy to understand than the PROLOG model. It is also much shorter. Another comparison can be made with the naive MiniZinc model that we developed for usage in the puzzle generator. We found our MiniZinc model to be much easier to write and much more concise (25 times less lines of code!). Both much other models perform much better than the PROLOG model too.

Resource, documentation, and Ecosystem quality

SWIPL features a useful trace tool for debugging, and the interactive nature of PROLOG queries is especially useful during the development process. Since PROLOG is well established and it has been used for many years, there are numerous wonderful resources available online. The exact documentation depends on the PROLOG distribution used. SWIPL documentation for the language and its libraries is easily searchable, clear and centralized.

8.2 Is PROLOG still relevant?

It is my opinion that learning PROLOG is still valuable, even if its performance seems to be outclassed by other paradigms for this NP-hard problem. I find its unique programming style to be extremely interesting, and it can definitely still be a good choice for some (probably not NP-hard) use cases. The way queries work is perfect for gathering insights on some kind of set of business rules. Identifying those use cases benefits from experience with PROLOG. Apart from that, it is my opinion that learning PROLOG and developing this puzzle solving model has also improved my (imperative and functional) programming skills. And it was fun too!

Responsible Research

We found no serious ethical issues regarding this research project. We discuss reproducibility, describe all LLM usage and include an personal reflection on that LLM usage.

Reproducibility

The source code for the PROLOG model can be found on GitHub⁴ and in Appendix G. Software versions, the exact benchmarking method and hardware specs used to reproduce these findings can be found in section 5.

Description of LLM usage

LLMs can be a valuable tool, but output generated by LLMs may be inaccurate, and we consider undisclosed LLM usage unethical. In accordance with the official LLM policy of the CSE3000 course, all LLM prompts were recorded and included in Appendix F. They were categorized into either 'code (older versions only)', 'research' or 'formatting and LaTeX'. Some old versions of the PROLOG model contained up to 3 lines of code in the puzzle solving logic, plus a predicate for writing a puzzle solution to a file that were inspired by an LLM. All these lines were removed or replaced in or before v3.0.2. Similarly, part of a python script for parsing a puzzle file was written by an LLM, but this was also replaced in V3.0.2. The latest version of the project and tools do not contain any code that written by or directly inspired by an LLM. LLMs were never considered as a factual source. The research category consists of potential ideas on paradigm evaluation criteria, summaries of fields on day one of the project, and an impression of CLPFD. All words are purely my own, all writing was done without LLM assistance.

Personal reflection on LLM usage

I think my usage of LLMs could have been more thoughtful, especially considering code. Some usages like formatting puzzles and translating pseudocode to LaTeX I think were valuable and I do not see any issues with those. I never asked a LLM for any help with the writing of this paper, because I think writing an academic document is one of the learning goals for doing a thesis like this, and I preferred every word of it to be my own. I am also glad I recorded every single use of LLMs well, documenting all of my prompts in an appendix, in compliance with the rules of the course. I do regret using an LLM for code for this project.

Apart from my actual usage of LLMs, I think the communication in previous versions of this paper about my LLM usage was subpar. Previously, it seemed like I let an LLM generate more code than I actually did, because outside of the appendix, I only specified that I generated 'a few code snippets'. In actuality, this only consisted of a single line of code in the (then latest) model logic that was influenced by an LLM, and two trivial file format input/output parsing routines. This has impacted the optics of my project, which I think was nowhere near worth the minuscule amount of time I saved by using an LLM for these purposes. There is no way to undo that, but this version of my paper has much improved

documentation, and to remove all doubt I have replaced everything that was influenced by an LLM, which was possible due to the records I kept. I have also improved the appendix, visually distinguishing LLM output and prompts, and adding more output. For the future, I think a better way to use LLMs for code in an educational project like this would be to use it in a style similar to pull request reviews, since that makes it clear to any supervisors that the project was developed fully by the student, and not by an LLM.

Acknowledgements

I would like to thank all the research group members for our great collaboration on the shared work, shared benchmarking, and all the wonderful discussions we had. I would also like to thank the committee for taking the time to review my paper. A special thanks goes to Dr. A.L.D. Latour for her supervision, feedback and guidance during this project.

References

- [1] Roman Barták. *History of Constraint Programming*. 01 2011.
- [2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.
- [3] Włodzimierz Drabent. Implementing backjumping by means of exception handling. *CoRR*, abs/2305.16137, 2023.
- [4] Wai-Keong Foong. Or-parallel prolog with heuristic task distribution. In *RCLP*, volume 592 of *Lecture Notes in Computer Science*, pages 193–200. Springer, 1991.
- [5] M. Gander and C. Hofer. Hitori solver. Bachelor's thesis, Universität Innsbruck, 2006.
- [6] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437. AAAI Press / The MIT Press, 1998.
- [7] Shai Haim and Marijn Heule. Towards ultra rapid restarts. 02 2014.
- [8] Vegard Hanssen. "hitori solving methods". <https://menneske.no/hitori/methods/eng/index.html>.
- [9] Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation*. A K Peters, 2009.
- [10] MJH Heule and H van Maaren. *Look-Ahead Based SAT Solvers*, pages 155–184. IOS Press, Netherlands, 2009.
- [11] Nina Narodytska Peter J. Stuckey Katalin Fazekas, Matti Järvisalo. Interactions in constraint optimization, 2025.
- [12] Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, José F. Morales, Jan Wielemaker, Daniel Diaz, and Salvador Abreu. Fifty years of prolog and beyond. *Theory Pract. Log. Program.*, 22(6):776–858, 2022.

⁴<https://github.com/tom0334/PrologHitori>

- [13] Victor W. Marek, Ilkka Niemelä, and Miroslaw Truszczynski. Origins of answer-set programming - some background and two personal accounts. *CoRR*, abs/1108.3281, 2011.
- [14] K. Nakamura. Heuristic prolog: Logic program execution by heuristic search. In *LP*, volume 221 of *Lecture Notes in Computer Science*, pages 148–155. Springer, 1985.
- [15] Potassco technologies. Potasco website. <https://potassco.org/clasp/>.
- [16] Roos Wensveen. Solving, generating and classifying history. Master’s thesis, Leiden University, 2024.

A Author Contribution

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 Comparison of runtime between PROLOG model versions

Apart from the experiments we conducted to answer the research questions, we also compared the runtime of each model version to verify if subsequent versions of the model are faster than previous versions. We share some comparison results in this appendix.

B.1 Experimental setup

To compare runtime of multiple PROLOG model versions, we used two benchmark sets of 100 puzzles generated by our puzzle generator. One set of $N=5$, and one of $N=13$. These values were chosen, because $N=5$ and $N=13$ seemed to be the largest puzzles that could be solved in a reasonable time-frame by V1.0 and V2.0 respectively. We ran the benchmark set 3 times for each model version, and took the median average solving time. Models that could not solve each puzzle within 10 minutes were excluded from the results. We validated the results using the inference count, which is consistent from run to run. We did not observe any significant run to run variance in the average solving times.

B.2 Results

The performance difference between newer and older versions of the PROLOG model versions spans multiple orders of magnitude. We illustrate that using Figure 10 and Figure 11, which show the average puzzle solving times for different model versions using a logarithmic scale. As seen in Figure 10, puzzles of $N=5$ take over 100 seconds on average for V1.0, but are trivially solved within around 5 milliseconds starting at V2.0. V4.0 seems to have a very slight regression at this small puzzle size, potentially because of the overhead from redundant constraints. Figure 11 shows a similar graph for puzzles of size $n=10$. The continuous improvements for every version after v2.0 become visible at this larger puzzle size, but even this puzzle size becomes fairly trivial for

later model versions. The largest performance improvement is seen between V2.3 and v3.0, when iterative cutoff detection was introduced.

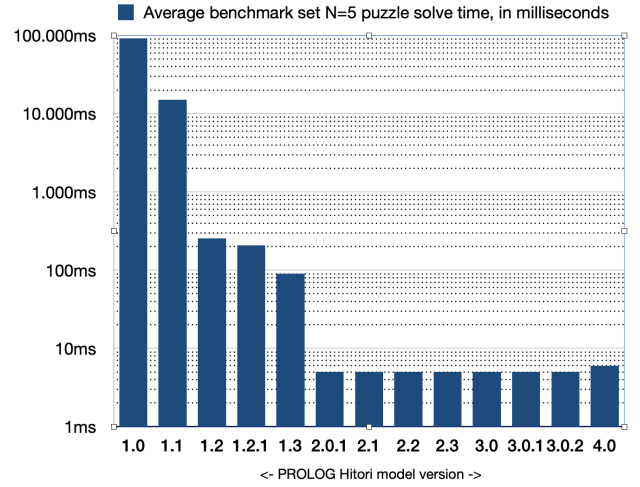


Figure 10: Average PROLOG model puzzle solving time for different model versions, on a benchmark set of 100 small $N=5$ puzzles. Scale is logarithmic. Average time was measured on an Apple M2 PRO Macbook Pro, median of 3 runs.

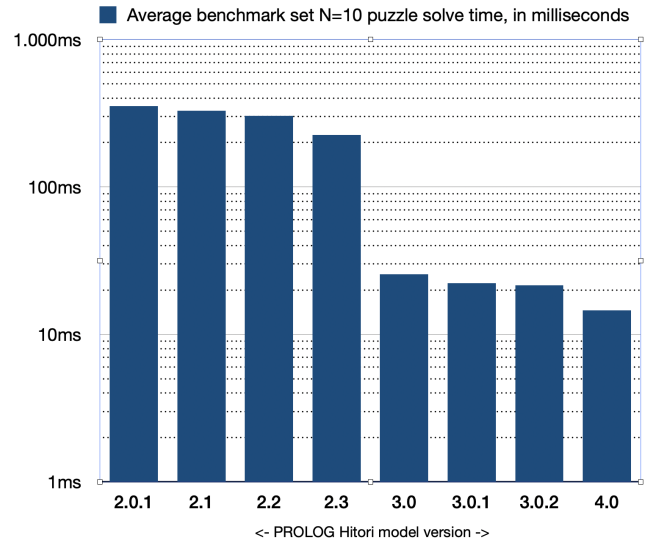


Figure 11: Average PROLOG model puzzle solving time for different model versions, on a benchmark set of 100 $n=10$ puzzles. Scale is logarithmic. Average time was measured on an Apple M2 PRO Macbook Pro, median of 3 runs.

C PROLOG model version overview

Version	Approach	Major changes
1.0	Generate and check (matrices), DFS matrix connectivity check	-
1.1	Generate and check (matrices), DFS matrix connectivity check	Combined check of adjacency and uniqueness constraints
1.2	Generate and check (matrices), DFS matrix connectivity check	Only consider numbers that are duplicate in their row/column
1.3	Generate and check (sets), DFS matrix connectivity check	Precompute duplicates, and consider numbers that are duplicate in both their row and column first
2.0	Iterative marking, DFS matrix connectivity check	Iterative marking using sets of duplicates, but still check connectivity using DFS on a Matrix
2.1	Iterative marking, DFS matrix connectivity check	Data reuse and more efficient checks
2.2	Iterative marking, DFS matrix connectivity check	Speedup DFS matrix connectivity check
2.3	Iterative marking, DFS set connectivity check	Rewrote DFS matrix connectivity check to run on a set of unmarked tiles instead of a full board matrix
3.0	Iterative marking and connectivity	Iterative cutoff detection for connectivity
4.0	Iterative marking and connectivity with RCs	Sandwich pair, pair isolation, and sandwich triple redundant constraints

Figure 12: Overview of major model versions and their approach, along with major changes made in each release. All versions can be found on <https://github.com/tom0334/PrologHitori>.

D Overview of collaboratively developed tools

To facilitate a comparison between the paradigms, the research group worked together to create the following:

1. Two shared file formats, one for Hitori puzzles and one for Hitori solutions. These shared formats makes it easy to compare the various models. The answer file format allows for extra metadata to be added below the solution, such as which model solved it, a timestamp, and the measured solving time.
2. Checker: A Hitori solution checker that reads a solution

from a file, and determines if a solution is valid. Used by the generator and testing script.

3. Puzzle solver: A basic MiniZinc based model to act as a comparison point to validate our puzzle generator.
4. Generator: A Hitori puzzle generator that creates random puzzles and writes them to a file in the shared Hitori file format. Our puzzle generator creates puzzles by picking a random set of marked tiles that satisfies all Hitori constraints. It then randomly fills in the numbers on the marked and unmarked tiles, in such a way that the unmarked tiles remain unique and the marked tiles contain non unique numbers in their row and column. Finally, it checks if the puzzle is uniquely solvable using our MiniZinc model. For a proof that our generator can generate all puzzles, see Appendix E. The proof was also created collaboratively.
5. Testing script: A Python script that evaluates the performance of a model on a set of puzzle files, and uses the checker to verify the reported solution. Solving time is measured by the models themselves, to avoid Python interpreter startup time and time for reading/writing the solution to a file.

See Appendix A for the Author Contribution.

The code for all these elements can be found on the shared GitHub page⁵

E 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 E.1. *Our generator is complete. That is, Algorithm 2 can generate exactly only every uniquely-solvable puzzle H .*

Algorithm 2 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:   return  $H$ 
```

E.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 $H_{i,j}$

⁵<https://github.com/sappho3/Thesis-Hitori-shared>

unmarked. A solution topology S is valid if it adheres to the adjacency and uniqueness constraints defined by the Hitori rules.

Algorithm 3 is a pseudo-code representation of the algorithm with which we generate our solution topologies.

Algorithm 3 Algorithm that generates a solution topology S .

```

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

```

Lemma E.2. *Algorithm 3 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. $\square$

Lemma E.3. *Algorithm 3 can generate exactly only any valid solution topology.*

Proof. Algorithm 3 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 3 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 E.2 proves that Algorithm 3 can only generate valid solution topologies, we have now proven that the algorithm can generate exactly only any valid solution topology. $\square$

E.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 4 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 5 which generates numbers for the tiles in H which correspond to unmarked tiles in S , and Algorithm 6 which generates numbers for the tiles in H which correspond to marked tiles in S .

Algorithm 4 Algorithm that generates a Hitori instance H from a solution topology S .

```

1: function GENERATEHITORIINSTANCEFROMS(S)
2:   Let  $H[1, \dots, n][1, \dots, n]$  be a grid of 0s
3:   FILLUNMARKEDTILES( $H, S, n, 1$ )
4:   FILLMARKEDTILES( $H, S, n, 1$ )
5:   return  $H$ 

```

Algorithm 5 Algorithm that fills in the unmarked tiles given a partial Hitori instance H and a solution topology S .

```

1: function FILLUNMARKEDTILES( $H, S, n, k$ )
2:   Let  $i = \lceil \frac{k}{n} \rceil$ 
3:   Let  $j = ((k - 1) \bmod n) + 1$ 
4:   if  $k > n^2$  then
5:     return true
6:   else if  $S[i][j] == \text{marked}$  then return FILLUNMARKEDTILES( $H, S, n, k + 1$ )
7:   else
8:     Let  $row$  be the numbers used in the row of  $H[i][j]$ 
9:     Let  $col$  be the numbers used in the column of  $H[i][j]$ 
10:     $C = \{1, \dots, n\} \setminus row \setminus col$ 
11:    if  $C = \emptyset$  then
12:       $\triangleright$  We check if a conflict occurred
13:      return false
14:       $\triangleright$  this is optimized by analyzing the conflict and returning to the conflict's cause
15:    else
16:      shuffle  $C$ 
17:       $H[i][j] = C[1]$ 
18:       $\triangleright$  Assign  $H[i][j]$  the first element in  $C$ 
return FILLUNMARKEDTILES( $H, S, n, k + 1$ )

```

Lemma E.4. *Given a valid solution topology S , Algorithm 5 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.

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 5 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 5 can generate all valid combinations of numbers in unmarked tiles. $\square$

Lemma E.5. *Given a valid solution topology S , Algorithm 5 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 5 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. $\square$

Algorithm 6 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  $j = ((k - 1) \bmod n) + 1$ 
4:   if  $k > n^2$  then
5:     | return true
6:   else if  $S[i][j] == \text{unmarked}$  then return FILL-
   MARKEDTILES( $H, S, n, k + 1$ )
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:     $\triangleright$  Assign  $H[i][j]$  the first element in  $C$ 
   return FILLMARKEDTILES( $H, S, n, k + 1$ )

```

Lemma E.6. *Given a valid solution topology S , and a valid partial Hitori instance H with numbers assigned to each unmarked tile, Algorithm 6 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 6 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 6 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 6 can generate any valid combination l of numbers for in the marked tiles. $\square$

Lemma E.7. *Given a valid solution topology S , and a valid partial Hitori instance H with numbers assigned to each unmarked tile, Algorithm 6 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 6 cannot generate an invalid combination of numbers for in the marked tiles. $\square$

Lemma E.8. *Given a valid solution topology S , Algorithm 4 can generate any valid puzzle instance H .*

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

Lemma E.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 E.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 . $\square$

E.3 Proving Theorem E.1

Proof. Lemma E.3 has proven that our algorithm can generate exactly any valid solution topology S , and Lemma E.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 2 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 E.1 $\square$

F LLM Usage

A complete and categorized overview of all prompts is included below. We also include responses for some prompts where we consider that useful, but we exclude some for brevity. The categories are:

1. Code for older versions of the model (5 prompts)
2. Research (3 prompts)
3. Formatting and LaTeX (4 prompts)

F.1 LLMS: Code

As of v3.0.2, all code that was directly influenced by LLM output was removed. We still provide an overview of all prompts related to code for older versions of both the PROLOG model and testing scripts. The specific contributions to older versions were:

- A suggestion to use functional notation for dictionary access ("Dictionary.get")
- A suggestion to swap the order of 2 goals for better performance.
- A predicate to write a matrix to a file.
- Some Python code that parses a Hitori file.

8-12-2025: Dicts in SWI PROLOG

ChatGPT introduced me to a useful function to get from a dict with a default value that is only available when using functional notation. This was removed in 3.0.2, since the default value was no longer needed.

```
in prolog, i am "looping" (with recursion)
over all elements in a list. I want to
put the value that is currently in a dict
+1. So i want to to something like
get_dict(dict,Value), put_dict(dict,
value +1). But if it's not in the dict
yet we run into issues. How can i do a
get with default value 0?
```

A response using a subquery was generated, which was undesired. Followup prompt:

```
another way without using the () subqueries?
```

ChatGPT generated a response using `get_dict/4`, which does not exist. Followup prompt:

```
thats bs that doesn't even exist
```

Useful response:

```
In SWI-Prolog you can get a value with a
default, but not via get_dict/4.
Use the functional notation instead:

increment_key(Key, Dict0, Dict) :-
    V0 = Dict0.get(Key, 0), % returns 0 if Key
    is missing
    V is V0 + 1,
    put_dict(Key, Dict0, V, Dict).
```

Example:

```
?- increment_key(a, _{}, D1),
    increment_key(a, D1, D2).
```

Gives:

```
D1 = _{a:1},
D2 = _{a:2}.
```

This is the clean built-in way to do a 'get with default' for dicts.

8-12-2025: Help debugging

Asked ChatGPT for help debugging. This predicate was removed in v3.0.1, because it was no longer needed after the changes made in v3.0. It did not find the bug: I later found it to be a typo in how I called the `duplicatesInAll` predicate. However, it did point out a small improvement in my code: swapping the order of two goals slightly improved performance.

```
I am making a solver for hitori puzzles in
PROLOG. I have some predicates that
create ord_sets. Those run instantly. I
have two sets, that i compute. I am
interested in creating 3 sets based on
that, one contains the intersection of
the two original sets and the other two
are the original ones except what's in
the intersection. Somehow it seems to
loop infinitely. Do you understand why?
duplicatesInAll(RowsOrColumns,
    AllDuplicatesSet) :- maplist(
    findDuplicatePositions, RowsOrColumns,
    DuplicatesPerSubList), append(
    DuplicatesPerSubList, AllDuplicatesList),
    list_to_ord_set(AllDuplicatesList,
    AllDuplicatesSet). duplicatesInAllRows(
    AllPositions, DupsInRows):- allRows(
    AllPositions, AllRows), duplicatesInAll(
    AllRows,DupsInRows).
duplicatesInAllColumns(AllPositions,
    DupsInColumns):- allColumns(AllPositions,
    AllColumns), duplicatesInAll(AllColumns,
    DupsInColumns). allDuplicateSets(
    AllPositions, RowOnly, ColOnly, InBoth)
:- duplicatesInAllRows(AllPositions,
    DupsInRows), duplicatesInAllColumns(
    AllPositions, DupsInColumns),
    ord_subtract(DupsInColumns, DupsInRows,
    ColOnly), ord_subtract(DupsInRows,
    DupsInColumns, RowOnly).
```

ChatGPT generated a long response suggested things to fix, that were incorrect. It did point out a small improvement in my code: order of operations. Followup prompt:

```
Okay, you are correct that this way of
ordering the operations is better.
However, that doesn't solve the issue. I
will give you some more info: DupsInRows
```

```
and DupsInColumns are definitely ord_sets
, and AllPositions is known
```

The output following that was not useful, and I fixed the issue myself afterwards.

15-12-2025: Syntax error

```
What is wrong with the syntax here? I don't
see it. ERROR: /Users/tfriederich/
Documents/GitHub/PrologHitori/solver.pl
:146:65: Syntax error: Illegal start of
term
```

```
%For translating a set of chosen duplicates
to a board (list of rows). %%%%%%%%%%%
translateToBoard(Board, Zerod, Solution)
:- sameShape(Board, Solution),
allPositionsWithValue(Board, Positions),
maplist(translatePositionToBoardValue(
Solution, Zerod), Positions).
translatePositionToBoardValue(Solution,
Zerod, (X,Y,V)):- elementAt(Solution, X, Y
, SolutionValue), boardValueOrZerod((X,Y,
V),Zerod, SolutionValue).
boardValueOrZerod((X,Y,V), Zerod, 0) :-
ord_memberchk(X,Y), Zerod, !.
boardValueOrZerod( (_,_,V), _, V).
sameShape([], []). sameShape([A|As], [B
Bs]) :- same_length(A, B), sameShape(As,
Bs).
```

ChatGPT responded correctly that I had mismatched parenthesis.

16-12-2025 Python parsing helper script

Used CHATGPT to generate some trivial Python code that parses a matrix stored in a file. I adapted the output to use in a the script that calls the model for testing purposes. I rewrote this myself in V3.0.2.

```
write a python script that will read a file
like this when you run ./read PATH It
must have a array of arrays for the
matrix.
```

```
5 3 3 1 4
2 4 2 2 3
5 2 1 4 2
3 3 2 5 4
4 4 2 2 2
```

```
@125654977787056548792306960039095462909
```

16-12-2025 Help with predicate that writes to file

Used ChatGPT to modify my PROLOG that writes a matrix to a file in a certain format. I adapted the output to fit my use case. It was replaced in v3.0.2.

```
I am writing a matrix to a file in prolog.
I each element spaced by a space,
with a newline between the rows. It is
almost perfect, but after each row
there is now an extra space. Do you
know of a convenient way to fix this?
loop_through_list(_File, []) :- !.
```

```
loop_through_list(File, [Head|Tail]) :-
maplist(writeCell(File), Head),
nl(File),
loop_through_list(File, Tail).
```

```
writeCell(File, Value):-
write(File, Value),
write(File, " ").
```

```
write_list_to_file(Filename,List) :-
open(Filename, write, File),
loop_through_list(File, List),
close(File).
```

F.2 LLMs: Research

10-11-2025: First day information

Asked ChatGPT to summarize all the paradigms in a paragraph on the first day of the project. This was useful to get an initial feel for them, as some do not have Wikipedia pages, and especially no easily understandable ones.

I regret doing this prompt, because I copy pasted this list of paradigms from an introduction to the subject written by my supervisor. On further reflection, I should not have done that without permission.

```
"I am researching solving a series of puzzles
, and I want to compare different
paradigms. Can you explain the listed
paradigms in one paragraph each?
```

```
Modelling and solving Each student selects a
modelling-and-solving paradigm from this
list: Conjunctive Normal Form (CNF) [2],
solved with kissat
Constraint Satisfaction Problem (CSP),
modelled and solved with Oscar [3]
Constraint Satisfaction Problem (CSP),
modelled with MiniZinc and solved with
one or more of MiniZinc's supported
solvers
[4] Integer (Linear) Program (ILP), modelled
and solved with gurobipy or PySCIPOpt
Satisfiability Modulo Theories (SMT),
modelled and solved with Z3 [5]
Answer Set Program (ASP), modelled and solved
with clasp [6]
Logic Program (LP), modelled and solved with
Prolog [7]
```

14-11-2025: Paradigm evaluation criteria

Asked ChatGPT to come up with criteria for assessing paradigms, to see if it could come up with additional criteria. The suggestions did not contain anything not already covered. To be specific:

- "Expressiveness for constraints", "Declarative power", "Algorithmic flexibility", "Implementation complexity", and "State management clarity" was all already covered by "How natural is it to express a Hitori puzzle in this paradigm?"
- "Parallelization support" and "Performance potential" were covered by "solver performance".

- "Debuggability and transparency" was covered by "Test and debug ability".
- "Tooling maturity" was covered by resource and documentation quality and maintenance level.

I am researching multiple competing programming paradigms in the context of solving a Hitori puzzle. I would like to compare the different programming paradigms. I am interested in how well suited they are for creating a solver for this puzzle, and also how difficult it would be to create such a solver. For this, I would like to create a ranking system. Can you suggest some criteria for this ranking system?}

Useful Response:

Suggested criteria:

- Expressiveness for constraints
 - Ability to encode Hitori rules (uniqueness, adjacency, connectivity) directly and clearly.
- Search-space control
 - How well the paradigm supports pruning, backtracking, heuristics, and incremental solving.
- Declarative power
 - How close the code can stay to the problem specification without low-level plumbing.
- Algorithmic flexibility
 - Ease of implementing multiple strategies (constraint propagation, SAT/ILP encoding, heuristic search, brute force).
- State management clarity
 - How cleanly the paradigm handles evolving board states, trial assignments, and reversions.
- Performance potential
 - Practical ability to implement fast solvers without excessive effort.
- Tooling maturity
 - Availability of libraries for constraint solving, search, or symbolic reasoning.
- Implementation complexity
 - How difficult it is to write and maintain a solver in that paradigm.

Debuggability and transparency
Ease of inspecting intermediate states and understanding solver decisions.

Parallelization support
How naturally the paradigm can distribute or parallelize the search.

Use these to score paradigms such as
imperative, functional, logic/constraint, object-oriented, dataflow, and SAT/LP-based approaches.

11-12-2025: CLPFD Hitori model

Asked ChatGPT to try to create a CLPFD Hitori model, as an indication of what would a CLPFD model would look like compared to the traditional PROLOG model.

```
create a hitori puzzle solver using prologs
constraint programming library. You can
use this sudoku solver as an example.
puzzles must be square but can have any
size. The input is a list of lists of
numbers. I want to be able to run puzzle(
Board), isSolution(Board, Solution). and
have it print the correct solution. the
puzzle fact is defined like puzzle
([ [1,2,3], [...]. Here's the sudoku one
for inspiration: sudoku(Rows) :- length(
Rows, 9), maplist(same_length(Rows), Rows
), append(Rows, Vs), Vs ins 1..9, maplist
(all_distinct, Rows), transpose(Rows,
Columns), maplist(all_distinct, Columns),
Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
blocks(As, Bs, Cs), blocks(Ds, Es, Fs),
blocks(Gs, Hs, Is). blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [
N7,N8,N9|Ns3]) :- all_distinct([N1,N2,N3,
N4,N5,N6,N7,N8,N9]), blocks(Ns1, Ns2, Ns3
). problem(1, [[_,_,_,_,_,_,_,_,_], [_,_,
_,_,3,_,8,5], [_,_,1,_,2,_,_,_], [_,_,
_,5,_,7,_,_,_], [_,_,4,_,_,_,1,_,_], [_,
9,_,_,_,_,_,_], [5,_,_,_,_,_,7,3], [
_,2,_,1,_,_,_,_], [_,_,_,4,_,_,9]])
. Sample query: ?- problem(1, Rows),
sudoku(Rows), maplist(portray_clause,
Rows). [9, 8, 7, 6, 5, 4, 3, 2, 1]. [2,
4, 6, 1, 7, 3, 9, 8, 5]. [3, 5, 1, 9, 2,
8, 7, 4, 6]. [1, 2, 8, 5, 3, 7, 6, 9, 4].
[6, 3, 4, 8, 9, 2, 1, 5, 7]. [7, 9, 5,
4, 6, 1, 8, 3, 2]. [5, 1, 9, 2, 8, 6, 4,
7, 3]. [4, 7, 2, 3, 1, 9, 5, 6, 8]. [8,
6, 3, 7, 4, 5, 2, 1, 9]. Rows = [[9, 8,
7, 6, 5, 4, 3, 2|...], ... , [...|...]].
```

The output contained multiple syntax errors and singleton variables, but it did give me a better idea of what CLPFD models look like.

F3 LLMs: Formatting and LaTeX

(MULTIPLE DAYS): Converting puzzle to text

Variations of the following prompt were used multiple times to convert Hitori puzzles found online to a format that can be understood by the model. This was used to validate that the model works as expected for a variety of puzzles, in addition to puzzles we generated ourselves.

```
Please convert this puzzle to a double array
format:
--Image of Hitori puzzle--
```

ChatGPT always correctly responded with the puzzle in the image in the desired double array format.

12-12-2025: LaTeX formatting help

```
For my thesis, I want to include my LLM usage
in a appendix. I am writing it in
overleaf. My copy pasted LLM output is
all mushed together on one line though,
so now it is unreadable. Using verbatim
also doesn't work, because some lines are
too long to fit. What would be the best
way to have copy pasted output in LATEX
without having to reformat everything
manually?
```

ChatGPT responded that the 'lstlisting' package intended for code could be used for that, which was useful.

12-12-2025: LaTeX table help

This was used for showing the Hitori Puzzle and its solution next to each other.

```
How can i use this to show two tables next to
each other?
-- LATEX subfigure example --
```

ChatGPT responded with a useful example of subfigures with tables with a caption.

06-02-2025: LaTeX pseudocode

Used CHATGPT to help with formatting some pseudocode that I wrote in plain text to LaTeX. The result was a great starting point, but the output was heavily modified.

```
Please convert my pseudocode for an algorithm
into latex. I want to use the
algpseudocode package. You can leave in
placeholders or comments if things are
unclear, I will revise them. Do not
change the meaning of the code. Here is
the pseudocode:
//NewlyMarkedPositions is a list of cell
positions to run DFS from
//AllMarkedPositions are ALL marked cell
positions from previous iterations.
isStillConnected(newlyMarkedPositions,
AllMarkedPositions):
    for all startsPositions:
        queue = new queue with
            startPosition in it
    //
```

```
        if edgeBasedDfsCutSearch(queue,
            visitedEdges = empty set,
            markedPositions =
                AllMarkedPositions)
            return false
        return true

//Queue is a FIFO queue of edges(
    ToCellPosition,FromCellPosition) to be
    explored
// MarkdPositions is a set of positions that
    are currently marked so far
// UsedEdges: a set of edges that was already
    used (to avoid exploring the same edge
    twice)
// VisitedCells a set of cells that are
    visited. If we find the same one twice,
    we found a cycle!
edgeBasedDfsCutSearch(queue,markedPositions,
    usedEdges, VisitedCells) :
    (currentCel, currentFrom) = pop from
        queue
    if usedEdges contains (currentcell,
        currentFrom):
        return edgeBasedDfsCutSearch(
            queue,usedEdges,visitedCells)

    if queue is empty:
        return false: //No cutoff found.

    currentCellIsOnEdge = check if current
        cell is on the edge of the board

    if currentCell is on the edge and
        foundSideOfBoardBefore:
        return true // cutoff path found!

    if visitedEdges contains (currentCell,
        currentFrom):
        return true // cutoff cycle found
        !

    foundSideOfBoardBefore =
        foundSideOfBoardBefore OR
        currentCellIsOnEdge
    neighbours = find all diagonally
        adjacent positions to currentNode
        in markedPositions
    add all pairs (neighbour, currentCell)
        to the queue
    add currentCell to visited cells
    add (currentCell currentFrom) to
        usedEdges
    return DfsCutSearch(Queue,
        markedPositions, usedEdges,
        VisitedCells)
```

Useful Response:

```
Heres a straight conversion into LaTeX
using 'algorithm' + 'algpseudocode'.
Ive kept the structure and logic as-is
, even where things are a bit unclear
or inconsistent, and Ive left comments
```

where appropriate so you can tweak later.

```
'''latex
\begin{algorithm}
\caption{Connectivity and Cut Search}
\begin{algorithmic}[1]

\Function{isStillConnected}{
  newlyMarkedPositions, AllMarkedPositions}
  \Comment{newlyMarkedPositions: list of
    start cell positions}
  \Comment{AllMarkedPositions: all marked
    positions from previous iterations}
  \ForAll{startPosition in
    newlyMarkedPositions} \Comment{
    placeholder: ``startsPositions''}
    \State queue $\gets$ new FIFO queue
      containing $startPosition$
    \If{\Call{edgeBasedDfsCutSearch}{queue,
      AllMarkedPositions, $\emptyset$, $\emptyset$}}
      \State \Return false
    \EndIf
  \EndFor
  \State \Return true
\EndFunction

\vspace{1em}

\Function{edgeBasedDfsCutSearch}{queue,
  markedPositions, usedEdges, visitedCells}
  \Comment{queue: FIFO queue of edges (
    ToCellPosition, FromCellPosition)}
  \Comment{markedPositions: set of currently
    marked positions}
  \Comment{usedEdges: set of edges already
    explored}
  \Comment{visitedCells: set of visited
    cells}

  \State $(currentCell, currentFrom) \gets$
    pop from queue

  \If{$(currentCell, currentFrom) \in$
    usedEdges}
    \State \Return \Call{
      edgeBasedDfsCutSearch}{queue,
      markedPositions, usedEdges,
      visitedCells}
  \EndIf

  \If{queue is empty}
    \State \Return false \Comment{No cutoff
      found}
  \EndIf

  \State currentCellIsOnEdge $\gets$ check
    whether currentCell is on the edge of
    the board

  \If{currentCellIsOnEdge \textbf{and}
    foundSideOfBoardBefore}
    \State \Return true \Comment{Cutoff
      path found}
```

```
\EndIf

\If{$(currentCell, currentFrom) \in$
  visitedEdges} \Comment{placeholder:
  visitedEdges vs visitedCells}
  \State \Return true \Comment{Cutoff
    cycle found}
\EndIf

\State foundSideOfBoardBefore $\gets$
  foundSideOfBoardBefore $\lor$
  currentCellIsOnEdge

\State neighbours $\gets$ all diagonally
  adjacent positions to currentCell in
  markedPositions
\ForAll{neighbour in neighbours}
  \State enqueue $(neighbour, currentCell
    )$ into queue
\EndFor

\State add currentCell to visitedCells
\State add $(currentCell, currentFrom)$ to
  usedEdges

\State \Return \Call{edgeBasedDfsCutSearch}
  {queue, markedPositions, usedEdges,
  visitedCells}
\EndFunction

\end{algorithmic}
\end{algorithm}
'''
```

If you want, I can also:

- * Normalize naming ('visitedEdges' vs 'visitedCells', 'Queue' vs 'queue')
- * Turn comments into '\Require' / '\Ensure'
- * Make it iterative instead of recursive while keeping semantics identical

Just say the word.

G PROLOG model source code

The full model, benchmarking setup and generator for puzzles can be found on Github ⁶. We include the code for v.4.0 of the PROLOG model, along with some short descriptions of each file in case the Github repo ever becomes unavailable.

G.1 hitoriProlog.pl

The 'main' file. Contains the mean search algorithm, uses some predicates from other files.

```
% This is the main file for HitoriProlog. It
% can find solutions to a Hitori puzzle.
% Use the isSolution(Board, SolutionBoard)
% predicate to print the solution as a
% board,
% or if you are satisfied with only a list of
% positions to mark, you can use the
% isSolutionZerodPositions predicate.
% If you want to write the solution to a file
% too, see the testingUtils.pl file

% Alternatively, you can use the
% runHitoriPrologOnFile.py script to run
% HitoriProlog on a .singles file.

:- use_module(library(apply)).
:- use_module(library(lists)).
:- include('src/includedPuzzles/
  examplePuzzles.pl').
:- include('src/includedPuzzles/debugPuzzles.
  pl').
:- include('src/utils.pl').
:- include('src/duplicateFinding.pl').
:- include('src/testingUtils.pl').
:- include('src/cutoffConnected.pl').
:- include('src/redundantConstraints/
  sandwichPair.pl').
:- include('src/redundantConstraints/
  sandwichTriple.pl').
:- include('src/redundantConstraints/
  pairIsolation.pl').
:- include('src/redundantConstraints/
  redundantConstraints.pl').
:- include('src/redundantConstraints/
  neighboursUtils.pl').

modelVersion("4.0").

%if you just want to see the solved puzzle
%board in your console, use this.
isSolution(Board, SolutionBoard) :-
  isSolutionZerodPositions(Board,
    ZerodPositions),
  translateToBoard(Board, ZerodPositions,
    SolutionBoard).

%Main predicate to find a solution to a
%hitori puzzle, where the solution is a
%list of positions to mark.
%Board is a list of lists of numbers
%corresponding to the puzzle.
```

```
%positionsToZero is an ord_set of all
%positions to mark as black.
isSolutionZerodPositions(Board,
  PositionsToZero) :-
  allPositionsWithValue(Board, Positions),
  allRows(Positions, RowList),
  allColumns(Positions, ColumnList),
  length(ColumnList, N),
  append(RowList, ColumnList,
    AllRowsAndColumns),
  maplist(countInList, AllRowsAndColumns,
    AllCountMaps),
  maplist(findDuplicatePositions,
    AllRowsAndColumns, AllCountMaps,
    AllDuplicateLists),
  applyRCsToCountMapsAndDupLists(N,
    AllCountMaps, AllDuplicateLists,
    AllCountMapsWithRC,
    AllDuplicateListsWithRC, PreMarked),
  maplist(dupValuesOnly([]),
    AllDuplicateListsWithRC, AllDupNums),
  solveAll(AllDuplicateListsWithRC,
    AllCountMapsWithRC, AllDupNums, N,
    PreMarked, PositionsToZero).

% Solves a hitori puzzle iteratively row/
% column wise using a number of precomputed
% datastructures.
% [DuplicateLists]: ONLY the duplicate
% positions (X,Y,V) for each row and column
% . So the first element is a list of all
% duplicates in the first row.
% [Countmaps]: the countmaps for the amount
% of times each number occurs in a the row
% or column. So the first element is a
% dictionary containing the counts that each
% number occurs in the first row
% [Dupnums] all duplicate numbers only (V) in
% each row. This may contain numbers that
% are already solved, but that is okay.
% Testing found that to be better than
% sorting them out again.
%The first element contains the duplicate
% numbers in the first row.

% N: puzzle size
% ChosenSoFar: a bag for the currently chosen
% positions.
% Result: Will be unified with result when
% done: a list of all positions to make
% zero

%Base case: when the first 3 params are empty
% lists, that means that we solved all
% rows and columns!
solveAll([], [], [], _N, Result, Result).

solveAll([HDupPositions|TDupPositions], [HCm|
  TCm], [HDupNums | TDupNums], N,
  ChosenSoFar, Result):-
  recursivelySolveRowOrColumn(ChosenSoFar,
    Hcm, HDupNums, [], Chosen ,
    HDupPositions),
  ord_union(ChosenSoFar, Chosen, NewChosen),
```

⁶<https://github.com/tom0334/PrologHitori>

```

isStillConnectedFast(Chosen, N, NewChosen)
',
solveAll(TDupPositions, TCm, TDupNums, N,
NewChosen, Result).

% Base case where we there are no more
duplicates left in this row/column. This
row/column is now solved!
recursivelySolveRowOrColumn(
_AlreadyZerodInOther, _CountMap, [], Result,
Result, _DuplicatePositions):-
!.

%Case where we DO (attempt to) mark the
current duplicate.
recursivelySolveRowOrColumn(
AlreadyZerodInOther, CountMap, DupNums,
ChosenZeros, Result, [(X,Y,V)|
TDuplicatePositions]) :-
get_dict(V, CountMap, CountLeft),
CountLeft > 1,
notNextToOther((X,Y), AlreadyZerodInOther)
',
notNextToOther((X,Y), ChosenZeros),

%We can pick this one to mark!
NewCount is CountLeft - 1,
( NewCount < 2
-> ord_del_element(DupNums, V, NewDupNums)
; NewDupNums = DupNums
),

ord_add_element(ChosenZeros, (X,Y),
NewChosenZeros),
put_dict(V, CountMap, NewCount,
NewCountMap),

recursivelySolveRowOrColumn(
AlreadyZerodInOther, NewCountMap,
NewDupNums, NewChosenZeros, Result,
TDuplicatePositions).

%Case where we do NOT mark the current
duplicate.
recursivelySolveRowOrColumn(
AlreadyZerodInOther, CountMap, DupNums,
ChosenZeros, Result, [(_,_,_)|
TDuplicatePositions]) :-
recursivelySolveRowOrColumn(
AlreadyZerodInOther, CountMap, DupNums,
ChosenZeros, Result,
TDuplicatePositions).

%Checks if a position is NOT next to any of
the positions in AllWithoutValue.
%Implements the adjacency constraint.
%IMPORTANT: the list in which you check needs
to be (X,Y) only, NOT (X,Y,V)
notNextToOther((X,Y), AllWithoutValue):-
YPrev is Y-1,
(\+ ord_memberchk((X,YPrev),
AllWithoutValue)),
XPrev is X -1,

```

```

(\+ ord_memberchk((XPrev,Y),
AllWithoutValue)),
XNext is X + 1,
(\+ ord_memberchk((XNext,Y),
AllWithoutValue)),
YNext is Y+1,
(\+ ord_memberchk((X,YNext),
AllWithoutValue)).

```

G.2 duplicateFinding.pl

Contains everything related to finding the duplicate positions in each row and column, and returning that in an efficient way.

```

% This file contains predicates for finding
duplicates in list in an efficient way

% Finds duplicates in a list by creating a
dictionary with the amount of times a
number occurs
% creates a dictionary that has the following
values:
% key: the value of the position
% value: the number of times that number
occurs in the list
% So countInList([1,1,1,4,7], Result) gives
{1:3, 4:1, 7:1}
countInList(PositionList, ResultDict):-
dict_create(EmptyDict, _, []),
countOccurancesIn(PositionList, EmptyDict,
ResultDict).

% Counts the occurances of all numbers in a
list, generates a dictionary with the
counts. A CountMap!
countOccurancesIn([], Result, Result).
countOccurancesIn([(_,_,V)|Tail], CountMap,
Result):-
% 0 is default value, if its not in the
map yet.
OldCount = CountMap.get(V, 0),
NewCount is OldCount + 1,
put_dict(V, CountMap, NewCount, NewDict),
countOccurancesIn(Tail, NewDict, Result).

% Uses a CountMap to find which positions are
duplicates.
findDuplicatePositions(PositionList, Dict,
DuplicatesOnly):-
include(positionIsDuplicateAccordingToDict
(Dict), PositionList, DuplicatesOnly).

positionIsDuplicateAccordingToDict(Dict, (_,_,
V)) :-
get_dict(V, Dict, Count),
Count > 1.

% Finds the duplicate positions (X,Y,V) in a
list, returns only a list of values of
the positions that are duplicate.
% So, [(1,2,3), (1,1,3), (7,8,9)] will give
[3], since there are 2 positions with
value 3 so it is duplicate.

```

```

dupValuesOnly(Result, [], Result).

dupValuesOnly( SoFar, [ (_,_,V) | Tail ],
  Result):-
  ord_add_element(SoFar,V, NewSoFar),
  dupValuesOnly(NewSoFar, Tail,Result ).

% Some helper predicates to find all
% positions on the board,
% and to convert those positions to lists of
% positions corresponding to each row and
% column.
% These are inefficient, but they are only
% used once per puzzle so that is OK.

% Gives you all positions on the board with
% value: [(X,Y,V)]
allPositionsWithValue(Board,
  PositionsWithValue) :-
  findall( (X,Y,V), elementAt(Board,X,Y,V),
    PositionsWithValue).

% Just gives you [0, 1, 2, 3] if your board
% is size 4.
% Assumes the board is square!
allIndices(Positions, Sorted) :-
  findall(X, member((X, _, _), Positions),
    AllIndices),
  sort(AllIndices, Sorted).

allColumns(Positions, ColumnList) :-
  allIndices(Positions, ColumnIndices), %
  get 0,1,2,3...
  maplist(onlyPositionsInColumnX(Positions),
    ColumnIndices, ColumnList).

allRows(Positions, RowList):-
  allIndices(Positions, RowIndices), % get
  0,1,2,3...
  maplist(onlyPositionsInRowY(Positions),
    RowIndices, RowList).

%Some simple helper predicates to find only
% positions in a certain row or column
onlyPositionsInRowY(Positions, Y, Filtered)
:-
  include(positionIsInRowY(Y), Positions,
    Filtered).

onlyPositionsInColumnX(Positions, X, Filtered
) :-
  include(positionIsInColumnX(X), Positions,
    Filtered).

positionIsInRowY(Y, (_, RY, _)) :-
  RY = Y.

positionIsInColumnX(X, (CX, _, _)) :-
  CX = X.

```

G.3 cutofConnected.pl

Contains everything related to inside out cutoff detection described in subsection 4.2.

```

%This file contains predicates for checking
% the connectivity constraint using the
% Iterative cutoff detection method.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Call this to check if the chosenPositions do
% NOT cut off any white squares
%Params:
%[StartPositions]: A list of newly chosen
% positions(X,Y) to make black. These are
% the start positions to do DFS from to
% look for cutting off paths
%N : size of board, used to detect if a
% position is on the edge of the board or
% not.
%[ChosenPositions]: A list of positions (X,Y)
% that were in previous iterations, and
% that we already know do not cut off
% anything
%Run DFS from all startpositions.
isStillConnectedFast([], _N, _ChosenPositions
).

isStillConnectedFast([Start | TStartPositions
], N, ChosenPositions) :-
  dfsCutSearch([(Start,Start)], N,
    ChosenPositions, 0,[],[]),
  isStillConnectedFast(TStartPositions, N,
    ChosenPositions),
  !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%DFS Search. This is an edge based DFS search
% that will explore to find one of two
% things:
% 1. Two sides of the board. May be the same
% side, that does not matter. If two sides
% of the board are found,
% then that path cuts of some amount of white
% squares from the rest.
% 2. A node that it already encountered
% before. That means there is a cycle
% encircling some amount of white squares
% on the board.

%This predicate will FAIL when it finds one
% of these things, and succeed when it
% ensures the black squares chosen are OK.
%Params:
%[Queue] A list of edges to explore. The
% start has itself as FROM, but that does
% not really matter
%N: board size
%[ChosenPositions]: Positions chosen in
% previous iterations to be black. These
% can be explored.
%[SidesOfBoardFoundSoFar] the number of sides
% of the board found so far. When this
% reaches 2 we fail.

```

```

%[UsedEdges]: A set of edges that we explored
    so far, to avoid it going backwards.
%[VisitedNodes]: All nodes that we explored
    so far

%If the current edge is already used, that
    means we already considered this edge, so
    we can do nothing and continue...
dfsCutSearch([(CurrentNode, From) | TQueue],
    N, ChosenPositions,
    SidesOfBoardFoundSoFar, UsedEdges,
    VisitedNodes):-
    member( (From, CurrentNode), UsedEdges), %
        list with member is faster than
        ord_set here since these are often
        small lists. I tested it!
    dfsCutSearch(TQueue, N, ChosenPositions,
        SidesOfBoardFoundSoFar, UsedEdges,
        VisitedNodes),
    !.

%End case 1: when we encounter a position on
    the side of the board, AND we already
    found one,
% There is a cutting of path!
% we know for sure there is a path from start
    to two sides, so there is a cutoff path!
dfsCutSearch( [ (CurrentNode, _From) |
    _TQueue], N, _ChosenPositions, 1 ,
    _UsedEdges, _VisitedNodes):-
    isOnSideOfBoard(CurrentNode,N),
    !,
    fail.

%End case 2: when we encounter a position
    that was already VisitedNodes, that means
    we found a cycle!
% So there is at least white element not
    connected to the rest!
dfsCutSearch([(CurrentNode, _From) | _TQueue
], _N, _ChosenPositions,
    _SidesOfBoardFoundSoFar, _UsedEdges,
    VisitedNodes):-
    member(CurrentNode, VisitedNodes),
    !,
    fail.

% End case 3: when the queue is empty, we can
    say we did NOT find a cutting path. So
    we do succeed!
dfsCutSearch([], _N, _ChosenPositions,
    _SidesOfBoardFoundSoFar, _UsedEdges,
    _VisitedNodes):-
    !.

%Regular case: increment the counter for
    sidesOfBoard found if needed, find all
    neighbouring edges to explore, and call
    yourself recursively.
dfsCutSearch([(CurrentNode, From) | TQueue],
    N, ChosenPositions,
    SidesOfBoardFoundSoFar, UsedEdges,
    VisitedNodes):-

```

```

    calcNewSidesOfBoardFoundSoFar(CurrentNode,
        N, SidesOfBoardFoundSoFar,
        NewSidesOfBoardFoundSoFar),
    findAllPositionsToExplore(ChosenPositions,
        CurrentNode, Neighbors),
    append(TQueue, Neighbors, Queue),
    dfsCutSearch(Queue, N, ChosenPositions,
        NewSidesOfBoardFoundSoFar, [(
        CurrentNode,From)|UsedEdges], [
        CurrentNode|VisitedNodes]),
    !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Smaller helper predicates for the cutoff
    connection detection.

diagonallyAdjacentPos((X1,Y1),(X2,Y2)) :-
    (X2 is X1 + 1, Y2 is Y1 + 1);
    (X2 is X1 + 1, Y2 is Y1 - 1);
    (X2 is X1 - 1, Y2 is Y1 + 1);
    (X2 is X1 - 1, Y2 is Y1 - 1).

isOnSideOfBoard((X,Y),N):-
    (X = 0,!);
    (Y = 0,!);
    (X is N -1,!);
    (Y is N -1,!).

getPositionToExplore(ChosenPositions, From, (
    Adjacent,From)) :-
    diagonallyAdjacentPos(From, Adjacent),
    ord_memberchk(Adjacent, ChosenPositions).

%Finds the neighbour edges to explore from a
    position
findAllPositionsToExplore(ChosenPositions,
    From, AllAdjacentFromInfoPairs) :-
    findall(
        Adjacent,
        getPositionToExplore(ChosenPositions,
            From, Adjacent),
        AllAdjacentFromInfoPairs
    ).

%Adds one to the counter if it is on the edge
    of the board...
calcNewSidesOfBoardFoundSoFar(Pos, N,
    SidesOfBoardFoundSoFar,
    NewSidesOfBoardFoundSoFar):-
    isOnSideOfBoard(Pos,N),
    NewSidesOfBoardFoundSoFar is
        SidesOfBoardFoundSoFar + 1,
    !.

calcNewSidesOfBoardFoundSoFar(_, _,
    EdgesSoFar,EdgesSoFar).

```

G.4 redundantconstraints.pl

Contains everything related to applying redundant constraints to datastructures. The main logic for each redundant constraint are in separate files.

```

% This file contains anything related to
  applying redundant constraints to
  countMaps and to Lists of duplicates (
  duplists).

% Applies all RCs to the countmaps, duplicate
  lists, and also returns which tiles can
  be preMarked (known black (without values
  ))
applyRCsToCountMapsAndDupLists(N,
  AllCountMaps, AllDuplicateLists,
  AllCountMapsWithRC,
  AllDuplicateListsWithRC, PreMarked):-
  redundantConstraints(N,AllCountMaps,
    AllDuplicateLists,[],[],
    KnownWhiteWithoutValues,
    KnownBlackWithValues),
  writeln("Known_white:"),
  writeln(KnownWhiteWithoutValues),
  writeln("Known_black:"),
  writeln(KnownBlackWithValues),

  %A tile may be marked black by multiple
    RCs, filter out those duplicates
  list_to_ord_set(KnownBlackWithValues,
    KnownBlackWithValuesSet),

  %Now we will update the countmaps and
    duplicate lists to account for the
    newly known black tiles.
  %Change the lists of countmaps and the
    lists of duplists to a list of (
    CountList, DupList)
  %So that we can use maplist on the pair
  zipped(AllCountMaps, AllDuplicateLists,
    Pairs),
  maplist(updateCMAndDupList(
    KnownBlackWithValuesSet), Pairs,
    ResultPairs),
  zipped( AllCountMapsWithRC,
    AllUpdatedDuplicateLists, ResultPairs)
    ,

  %Remove the known white tiles from the
    duplists, since we know that we never
    have to consider them to be marked.
  maplist(removeThisFromListNoValues(
    KnownWhiteWithoutValues),
    AllUpdatedDuplicateLists,
    AllDuplicateListsWithRC),

  %Remove the values for black tiles. This
    may now contain duplicates again, so
    we have remove them again.
  maplist(stripValue,KnownBlackWithValuesSet
    , KnownBlackWithoutValues),

  %After removing those duplicates, we know
    the premarked
  list_to_ord_set(KnownBlackWithoutValues,
    PreMarked).

%Base case, unify the bag with the result

```

```

redundantConstraints(_N, [], [],
  KnownWhiteBag, KnownBlackBag,
  KnownWhiteBag, KnownBlackBag).

%Find known values because of redundant
  constraints in every countmap and
  duplicatelist, Store results in two bags.
redundantConstraints(N, [HCountMap|
  TCountMaps] , [HDuplicateList |
  TDuplicateLists], KnownWhiteBag,
  KnownBlackBag, KnownWhiteRes,
  KnownBlackRes):-
  redundantConstraintsForRowOrColumn(N,
    HCountMap, HDuplicateList,
    KnownWhiteBag, KnownBlackBag,
    NewKnownWhiteBag, NewKnownBlackBag),
  redundantConstraints(N, TCountMaps,
    TDuplicateLists, NewKnownWhiteBag,
    NewKnownBlackBag, KnownWhiteRes,
    KnownBlackRes).

%Find all known values because of RCS on one
  row/column.
redundantConstraintsForRowOrColumn(N,
  CountMap, DuplicateList, KnownWhiteSofar,
  KnownBlackSoFar, KnownWhiteRes,
  KnownBlackRes):-
  sandwichPair(N, CountMap, DuplicateList,
    KnownWhiteSPWithValues, KnownWhiteSP),
  sandwichTriple(N, DuplicateList,
    KnownWhiteSPWithValues, KnownBlackST,
    KnownWhiteST),
  pairIsolation(N, CountMap, DuplicateList,
    KnownBlackPI, KnownWhitePI),

  %write("Pair isolation res (black):"),
  %writeln(KnownBlackPI),
  %write("white:"),
  %writeln(KnownWhitePI),
  %write("sandwichPair result (white): "),
  %writeln(KnownWhiteSP),
  %write("sandwichTriple result (black): "),
  %writeln(KnownBlackST),
  %write("sandwichTriple result (white): "),
  %writeln(KnownWhiteST),

  flatten([KnownBlackSoFar, KnownBlackPI,
    KnownBlackST ], KnownBlackRes),
  flatten([KnownWhiteSofar, KnownWhitePI,
    KnownWhiteST, KnownWhiteSP],
    KnownWhiteRes),
  !.

% Updates the pair of a countmap and a
  duplist for a row or column, according to
  now known black values
% Does so by updating the counts in the
  countmap,
  %and then removing the tiles that are now no
  longer duplicates from the duplist.
updateCMAndDupList(KnownBlackWithValuesSet, (
  CountMap,DupList), (ResCountMap,
  ResDupList) ):-
  updateCM(KnownBlackWithValuesSet, DupList,
    CountMap, ResCountMap),

```

```

%remove all known black tiles from the
duplist
subtract(DupList, KnownBlackWithValuesSet,
DupListWithoutMarked),

%Remove all that are now no longer
duplicates from the duplist
findall(Pos, noLongerNeedsToBeconsidered(
ResCountMap, DupList, Pos),
PositionsToBeRemoved),
subtract(DupListWithoutMarked,
PositionsToBeRemoved, ResDupList).

% A tile no longer needs to be considered to
be marked ifs no longer a duplicate in
the row/column
noLongerNeedsToBeconsidered(CountMap, DupList
, (X,Y,V)):-
member((X,Y,V), DupList ),
get_dict(V, CountMap, CountLeft),
CountLeft < 2.

% Base case. This predicate updates a
countMap to account for tiles that we
determined have to be black.
% Its essentially the same as marking a list
of tiles in one go.
updateCM([],_, ResCountMap, ResCountMap).

updateCM([HKnownBlack | TKnownBlack], DupList
, Countmap, ResCountMap):-
member(HKnownBlack, DupList),
decrementValueInCountmap(Countmap,
HKnownBlack, NewCountMap),
updateCM(TKnownBlack, DupList, NewCountMap
, ResCountMap),
!.

updateCM([_HKnownBlack | TKnownBlack],
DupList, CountMap, ResCountMap):-
updateCM(TKnownBlack, DupList, CountMap,
ResCountMap),
!.

decrementValueInCountmap(CountMap, (_X,_Y,V),
NewCountMap):-
get_dict(V, CountMap, CountLeft),
NewCount is CountLeft - 1,
put_dict(V, CountMap, NewCount,
NewCountMap).

% Different order than regular subtract,
useful for when using it with maplist
removeThisFromList(ToRemove, From, Result):-
subtract(From, ToRemove, Result).

removeThisFromListNoValues([], Result ,Result
).

% Removes positions without value from a
lists with positions with value
% Example: remove (1,1) from [(1,1,1),
(5,5,5)] becomes [(5,5,5)]

```

```

removeThisFromListNoValues( [ PosXY | TRemove
], From, Result):-
exclude( equalIgnoringValue(PosXY) , From,
PartialResult),
removeThisFromListNoValues(TRemove,
PartialResult, Result ).

```

```

equalIgnoringValue( (X, Y), (X,Y,_)).

```

G.5 pairIsolation.pl

Contains everything related to the Pair Isolation redundant constraint.

```

%The Pair isolation redundant constraint

```

```

%Finds a position that can be marked black
because there exists a adjacent pair with
the same value in the row/column

```

```

blackPositionsBecauseOfPairIsolationForV(
AllDuplicatePositions, V, (BlackX, BlackY
, BlackV)):-
member((X,Y,V), AllDuplicatePositions),
member((X2,Y2,V), AllDuplicatePositions),
neighbours((X,Y), (X2,Y2)),
%Cut since we found a pair, so others can
be marked black!
%(without the cut we would find both (A,B)
and (B,A))
%This will not find cases where 2 adjacent
pairs exist. That's OK because
puzzles with 2 pairs are unsolvable!
!,
member((BlackX, BlackY, V),
AllDuplicatePositions),
notTheSamePosition((BlackX,BlackY), (X,Y))
,
notTheSamePosition((BlackX,BlackY), (X2,Y2
)),
BlackV = V.

```

```

% Finds ALL positions with value V that can
be marked black, because there exists a
adjacent pair with the same value.

```

```

findAllBlackPositionsForValue(
AllDuplicatePositions, V, AllBlackForPair
):-
findall(Position,
blackPositionsBecauseOfPairIsolationForV(
AllDuplicatePositions,V, Position)
,
AllBlackForPair).

```

```

valueOccursMoreThanTwoTimes(CountMap, V):-
get_dict(V, CountMap, Count),
Count > 2. % we need a pair and at least
one other one

```

```

notTheSamePosition((X,_Y), (X2,_Y2)) :-
\+(X is X2),
!.

```

```

notTheSamePosition((_X,Y), (_X2,Y2)) :-
\+(Y is Y2),
!.

```

```

%Simple XY neighbour checks used only in this
  redundant constraint.
neighbours((X,_Y), (X2, _Y2)):-
  X is (X2 + 1),
  !.
neighbours((_X,Y), (_X2, Y2)):-
  Y is (Y2 + 1),
  !.
neighbours((X,_Y), (X2, _Y2)):-
  X is (X2 - 1),
  !.
neighbours((_X,Y), (_X2, Y2)):-
  Y is (Y2 - 1),
  !.

% Main Pair isolation predicate. Finds all
  KnownBlack positions that must be marked
  in a row/column,
% and also returns all positions that must be
  white because of adjacency to the black
  ones.
pairIsolation(N, CountMap, DuplicatePositions
, KnownBlack, KnownWhite):-
  % For pair isolation to be possible, we
  need at least 3 occurrences. Otherwise
  there is nothing to isolate...
  findall(Value, valueOccursMoreThanTwoTimes
    (CountMap,Value), Values),

  % Now that we know the values that occur
  at least 3 times in this row/column,
  we can check each one for possible
  pair isolation opportunities...
  maplist(findAllBlackPositionsForValue(
    DuplicatePositions), Values,
    BlackPositionLists),

  %The result is a list of lists, where each
  element is a list of positions that
  we can mark black with some Value.
  Flatten it
  flatten(BlackPositionLists, KnownBlack),

  %All adjacent white neighbours. Already
  gives us the positions in XY as
  desired.
  findAllWhiteNeighboursOfPositions(N,
    KnownBlack, KnownWhite).

```

G.6 sandwichPair.pl

Contains everything related to the Sandwich Pair redundant constraint.

```

%Sandwich pair redundant constraint

% Finds a position that must be white because
  it is surrounded by two positions with
  the same value

% HORIZONTAL case
isWhiteBecauseSandwichPair(N,
  AllDuplicatePositions, (X,Y,V)):-

```

```

  member((X,Y,V), AllDuplicatePositions),
  leftNeighbour(N, (X,Y,V), (LNX,LNY,LNV)),
  rightNeighbour(N, (X,Y,V), (RNX,RNY,RNV)),
  member((LNX, LNY, LNV),
    AllDuplicatePositions),
  member((RNX, RNY, RNV),
    AllDuplicatePositions),
  RNV = LNV.

```

```

% VERTICAL case
isWhiteBecauseSandwichPair(N,
  AllDuplicatePositions, (X,Y,V)):-
  member((X,Y,V), AllDuplicatePositions),
  upNeighbour(N, (X,Y,V), (UNX,UNY,UNV)),
  downNeighbour(N, (X,Y,V), (DNX,DNY,DNV)),
  member((UNX, UNY, UNV),
    AllDuplicatePositions),
  member((DNX, DNY, DNv),
    AllDuplicatePositions),
  UNV = DNv.

```

```

% Main sandwichPair predicate. Finds all
  knownWhite positions (XY) in a row or
  column, using a set of duplicate
  positions.
% Gives back both the sandwich pair positions
  with and without value. We need the one
  without values throughout the model,
% but the list with values is useful for the
  sandwich triple RC.
sandwichPair(N, _CountMap, DuplicatePositions
, KnownWhiteWithValue, KnownWhiteXY):-
  findall(X, isWhiteBecauseSandwichPair(N,
    DuplicatePositions, X),
    KnownWhiteWithValue),
  maplist(stripValue, KnownWhiteWithValue,
    KnownWhiteXY).

```

G.7 sandwichTriple.pl

Contains everything related to the Sandwich Pair redundant constraint.

```

%Sandwich triple redundant constraint.
  Depends on the SandwichPair redundant
  constraint!

% Finds (ALL) sandwich tripple black
  positions around a single middle known
  white sandwichpair position.
% There can be up to 4, (both vertical and
  horizontal).

```

```

% VERTICAL CASE
sandwichTripleBlackPositionsAround(N,
  AllDuplicatePositions, (X,Y,V),
  BlackPositions):-
  member((X,Y,V), AllDuplicatePositions),
  upNeighbour(N, (X,Y,V), (UNX,UNY,UNV)),
  downNeighbour(N, (X,Y,V), (DNX,DNY,DNV)),
  member((UNX, UNY, UNV),
    AllDuplicatePositions),
  member((DNX, DNY, DNv),
    AllDuplicatePositions),

```

```

UNV = DNV,
V = UNV,
BlackPositions = [(UNX, UNY, UNV), (DNX, DNY,
DNV)].

% HORIZONTAL CASE
sandwichTripleBlackPositionsAround(N,
    AllDuplicatePositions, (X,Y,V),
    BlackPositions):-
member((X,Y,V), AllDuplicatePositions),
leftNeighbour(N, (X,Y,V), (LNX,LNY,LNV)),
rightNeighbour(N, (X,Y,V), (RNX,RNY,RNV)),
member((LNX, LNY, LNV),
    AllDuplicatePositions),
member((RNX, RNY, RNV),
    AllDuplicatePositions),
RNV = LNV,
V = RNV,
BlackPositions = [(LNX,LNY,LNV), (RNX,RNY,
RNV)].

% Main predicate to get ALL the sandwich
tripples known black positions, around a
set of known sandwich pair white
positions.
% Also finds all adjacent white positions to
the black positions.
sandwichTriple(N, DuplicatePositions,
    SandwichPairWhitePositions,
    SandwichTripleBlackPositions,
    SandwichTripleWhitePositionsXY):-
% This results in a list of lists, where
each element is a list of known black
positions around one white
sandwichpair position
maplist(findBlackPositionsAroundMiddle(N,
    DuplicatePositions,
    SandwichPairWhitePositions, Result),
    %Flatten that list of lists into one list.
    flatten(Result,
        SandwichTripleBlackPositions),
    %Find all neighbours of the black positions
    (these must be white)
    findAllWhiteNeighboursOfPositions(N,
        SandwichTripleBlackPositions,
        SandwichTripleWhitePositionsXY).

% FindAll black positions (up to 4, for both
vertical and horizontal) around ONE
single white sandwichpair middle position
.
findBlackPositionsAroundMiddle(N,
    DuplicatePositions, WhitePos,
    BlackPositions):-
findall(BlackPosForDirection,
    sandwichTripleBlackPositionsAround(N,
    DuplicatePositions, WhitePos,
    BlackPosForDirection), BlackPositions)
.

```

G.8 neighboursUtils.pl

Contains everything related to finding neighbours of tiles, used by redundant constraints.

%This file contains predicates for finding neighbours within the board. Used for some redundant constraints.

```

leftNeighbour(_N, (MeX, MeY, _MeV), (LNX,LNY
, _LNV) ):-
    LNX is MeX -1,
    LNY is MeY,
    LNX >= 0.

rightNeighbour(N, (MeX, MeY, _MeV), (RNX,RNY,
    _RNV) ):-
    RNX is MeX +1,
    RNY is MeY,
    RNX < N.

upNeighbour(N, (MeX, MeY, _MeV), (RNX,RNY,
    _RNV) ):-
    RNY is MeY +1,
    RNX is MeX,
    RNY < N.

downNeighbour(_N, (MeX, MeY, _MeV), (LNX,LNY
, _LNV) ):-
    LNY is MeY -1,
    LNX is MeX,
    LNY >= 0.

neighbour(N, (MeX, MeY), (NX,NY)):-
(
    leftNeighbour(N, (MeX, MeY, _), (NX,NY,
        _));
    rightNeighbour(N, (MeX, MeY, _), (NX,NY
        ,_));
    upNeighbour(N, (MeX, MeY, _), (NX,NY,_
        ));
    downNeighbour(N, (MeX, MeY, _), (NX,NY,
        _))
).

% Finds all neighbours (XY) of a list of known
black positions. These can be marked as
white because of the adjacency constraint
.
% This predicate is used by multiple
redundant constraints that find known
black positions.
% These white neighbours may contain
duplicates, but that is OK, because we
will have to remove the duplicates anyway
% when we apply multiple redundant
constraints.
findAllWhiteNeighboursOfPositions(N,
    BlackPositions, WhiteNeighboursNoValues):-
    maplist(
        findAllWhiteNeighboursOfBlackPosition(N
        ), BlackPositions, ListOfLists),
    flatten(ListOfLists,
        WhiteNeighboursNoValues).

%Finds the known white neighbours (X,Y) of a
single black position (X,Y,V)
findAllWhiteNeighboursOfBlackPosition(N,
    PositionWithValue, NeighboursWithoutValue)

```

```

:-
stripValue(PositionWithValue,
  PositionsWithoutValue),
findall(Neighbor, neighbour(N,
  PositionsWithoutValue, Neighbor),
  NeighborsWithoutValue).

```

G.9 testingUtils.pl

Contains everything related to testing and measuring the model.

```

%Testing and writing to file utilities.

%Finds a solution to a puzzle, and writes it
to a file in the .singlesol format
isSolutionAndWrite(Board,N,Seed, OutputFile,
  Verbose, Solution):-
  (Verbose = 1, writeln("PROLOG:_Solving..."
    ); true),
  call_time(
    isSolutionZerodPositions(Board,Solution
    ),
    TimeDict
  ),
  modelVersion(Version),
  string_concat("Solved_by:_v", Version,
    Comment),
  translateToBoardWithBlackLetter(Board,
    Solution, SolutionInGenericFormat),
  writeSolution(OutputFile, N,Seed, Comment,
    SolutionInGenericFormat, TimeDict),

  %Print some info to the console so you can
see live what is happening:
  (Verbose = 1,
    write("Solved!:_Took_"),
    write(TimeDict.cpu),
    write("_seconds,_"),
    write(TimeDict.inferences),
    writeln("_inferences"),
    maplist(writeln,
      SolutionInGenericFormat),
    writeln(""),
    writeln(Solution);
    true
  ).

```

```

%Writing board with chosen zeros to solution
output file
#####
translateToBoardWithBlackLetter(Board, Zerod,
  SolutionInGenericFormat):-
  sameShape(Board, SolutionInGenericFormat),
  allPositionsWithValue(Board, Positions),
  maplist(translateToBoardValueOrBoardBlack(
    SolutionInGenericFormat, Zerod),
    Positions).

translateToBoardValueOrBoardBlack(Solution,
  Zerod, (X,Y,V)):-
  elementAt(Solution, X, Y, SolutionValue),
  boardValueOrBlackedBoardValue((X,Y,V),
    Zerod, SolutionValue).

```

```

boardValueOrBlackedBoardValue((X,Y,V), Zerod,
  Res) :- ord_memberchk((X,Y), Zerod),
  string_concat(V,"B", Res), !.

boardValueOrBlackedBoardValue((_,_,V), _, V).

```

```

writeSolution(Filename, N, Seed, Comment,
  SolutionInGenericFormat, TimeDict) :-
  open(Filename, write, File),
  writeln(File, N),
  writeln(File, ""),
  writeMatrix(File, SolutionInGenericFormat
    ,
    writeln(File, ""),
    writeln(File, Seed),
    writeln(File, Comment),
    write(File, "#prologTime="),
    writeln(File, TimeDict.cpu),
    write(File, "#prologInferences="),
    writeln(File, TimeDict.inferences),
    close(File).

```

```

writeMatrix(File, Matrix):-
  maplist( writeRow(File), Matrix).

```

```

%Write the elements (1W, 2B, 3W...etc to a
file),
%with a space as seperator between elements
and a \n at the end of the row
writeRow(File, [Head | Tail]):-
  write(File, Head),
  writeSeparatorAndContinue(File, Tail).

```

```

writeSeparatorAndContinue(File, []):-
  writeln(File, ""), !.

```

```

writeSeparatorAndContinue(File, NonEmptyList)
:-
  write(File, "_"),
  writeRow(File, NonEmptyList).

```

G.10 utils.pl

Some small utils predicates used throughout the model.

%General Utils used throughout the model

```

% Converts a X,Y,Z position into a X,Y
position.
stripValue((X,Y,_), (X,Y)).

```

```

% Converts multiple X,Y,Z positions into
UNIQUE x,y positions.
convertToValueOnlyPositions(
  PositionsWithValue, Result) :-
  maplist(stripValue, PositionsWithValue,
    PositionsWithoutValue),
  list_to_ord_set(PositionsWithoutValue,
    Result).

```

% Same as a zip function in any functional programming language.

```

% but enforces the lists to be the same
length!
zipped([], [], []).
zipped([X|Xs], [Y|Ys], [(X,Y)|Zs]) :-
    zipped(Xs, Ys, Zs).

%WARNING: This is slow!
getRow(Board, X, Row):- nth0(X,Board,Row).

% Get from board at X,Y. Top Left is 0,0.
% WARNING: This is slow!
elementAt(Board, X,Y,Element):-
    getRow(Board,Y,Row),
    % row is now the entire row, get the
    element from that:
    nth0(X,Row,Element).

% For translating a set of chosen duplicates
to a board (list of rows).
%%%%%%%%%%%%%
translateToBoard(Board, Zerod, Solution):-
    sameShape(Board, Solution),
    allPositionsWithValue(Board, Positions),
    maplist(translatePositionToBoardValue(
        Solution, Zerod), Positions).

translatePositionToBoardValue(Solution,Zerod
    , (X,Y,V)):-
    elementAt(Solution, X, Y, SolutionValue),
    boardValueOrZerod((X,Y,V),Zerod,
        SolutionValue).

boardValueOrZerod((X,Y,_), Zerod, 0) :-
    ord_memberchk((X,Y), Zerod), !.
boardValueOrZerod( (_,_,V), _, V).

sameShape([], []).
sameShape([A|As], [B|Bs]) :-
    same_length(A, B),
    sameShape(As, Bs).

```