Comparative Evaluation of Constraint Programming and Deductive Search for Solving Slitherlink Puzzles with Difficulty Assessment

Bachelor's Thesis

Omar Alkhatib
464143

16.06.2026

Supervisor: Prof. Dr. Benjamin Blankertz
Dr.-Ing. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Kurzfassung

Slitherlink ist ein Logikrätsel, bei dem unter lokalen numerischen Einschränkungen eine einzige geschlossene Schleife konstruiert werden muss. Diese Arbeit untersucht verschiedene Ansätze zum Lösen von Slitherlink-Rätseln. Das Problem wird als Constraint Satisfaction Problem (CSP) formalisiert und in CP-SAT (OR-Tools) modelliert. Drei Methoden zur Durchsetzung der Single-Loop-Bedingung werden vorgestellt: ein Solution-Verification-Ansatz, der die Single-Loop-Bedingung nach dem Lösen lokaler Einschränkungen überprüft, ein Lazy-Constraint-Ansatz, der iterativ ungültige Schleifenstrukturen ausschließt, und eine direkte Kodierung der Single-Loop-Bedingung. Zusätzlich zu diesen rechnerischen Ansätzen wird Deductive Search (DS) als eine vom menschlichen Lösen inspirierte Methode implementiert. Dazu wird eine Menge logischer Deduktionsregeln definiert, und hypothetisches Schließen wird auf begrenzten Einbettungstiefen verwendet, um menschliche kognitive Grenzen zu berücksichtigen. Dadurch kann der Solver den Lösungsprozess nachvollziehen und die Schwierigkeit des Lösens von Rätseln einschätzen. Die Solver werden anhand von Benchmark-Instanzen aus mehreren Datensätzen evaluiert und systematisch verglichen. Dabei wird auch untersucht, wie die von DS berechneten Schwierigkeiten mit den offiziellen Schwierigkeitsgraden der Instanzen zusammenhängen. Die Ergebnisse zeigen, dass der Lazy-Constraint-Ansatz der schnellste, zuverlässigste und konsistenteste Ansatz ist. Der Solution-Verification-Ansatz ist einfach, skaliert jedoch schlecht, während die direkte Kodierung der Single-Loop-Bedingung robust ist, aber aufgrund des größeren Modells langsamer ist. Deductive Search erweist sich als effiziente Methode mit niedrigen Laufzeiten und bietet gleichzeitig ein hohes Maß an Interpretierbarkeit. Die von DS berechneten Schwierigkeiten zeigen eine positive Korrelation mit einigen offiziellen Schwierigkeitsgraden, während der Zusammenhang bei anderen schwächer ist.

Abstract

Slitherlink is a logic puzzle in which a single closed loop has to be constructed under local numerical constraints. This thesis investigates different approaches for solving Slitherlink puzzles. The problem is formalized as a constraint satisfaction problem (CSP) and modeled in CP-SAT (OR-Tools). Three methods for enforcing the single-loop constraint are presented: a solution-verification approach that checks the single-loop condition after solving local constraints, a lazy-constraint approach that iteratively excludes invalid loop structures, and a direct encoding of the single-loop condition. In addition to these computational approaches, Deductive Search (DS) is implemented as a human-inspired solving method, by defining a set of logical deduction rules and using hypothetical reasoning at bounded levels of embedding, reflecting the human cognitive limits. This allows the solver to trace the solving process and assess the difficulty of solving puzzles. The solvers are evaluated using benchmark instances of multiple datasets and compared systematically, where DS can assess the difficulties of the instances compared with their labelled difficulties. The results show that the lazy-constraint approach is the fastest, most reliable and consistent approach. The solution-verifier approach is simple, but does not scale well, while the approach of directly encoding the single-loop constraint is robust but slower due to its larger model. Deductive Search proves to be an efficient method with low runtimes while providing a high level of interpretability. Computed difficulties by DS show a positive correlation with some official difficulty labels, but the relationship is weaker with others.

Contents

1 Introduction **6**
1.1 Puzzle Description and Rules . . . 6
1.2 Motivation and Relevance . . . 7
1.3 Literature Overview . . . 7
1.4 Goal of the Thesis . . . 8
1.5 Structure of the Thesis . . . 9

2 Methodology **10**
2.1 Constraint Programming . . . 10
2.1.1 Clue Constraints . . . 10
2.1.2 Vertex Constraints . . . 11
2.1.3 Single Loop Constraint . . . 11
2.2 Deductive Search . . . 16
2.2.1 Strategies and Patterns . . . 16
2.2.2 Core Operations . . . 22
2.2.3 Algorithmic Procedure . . . 26
2.2.4 Difficulty Assessment . . . 28

3 Results **30**
3.1 Experimental Setup . . . 30
3.2 Benchmark Instances . . . 30
3.3 Experiments . . . 30

4 Discussion **37**

5 Conclusion **40**

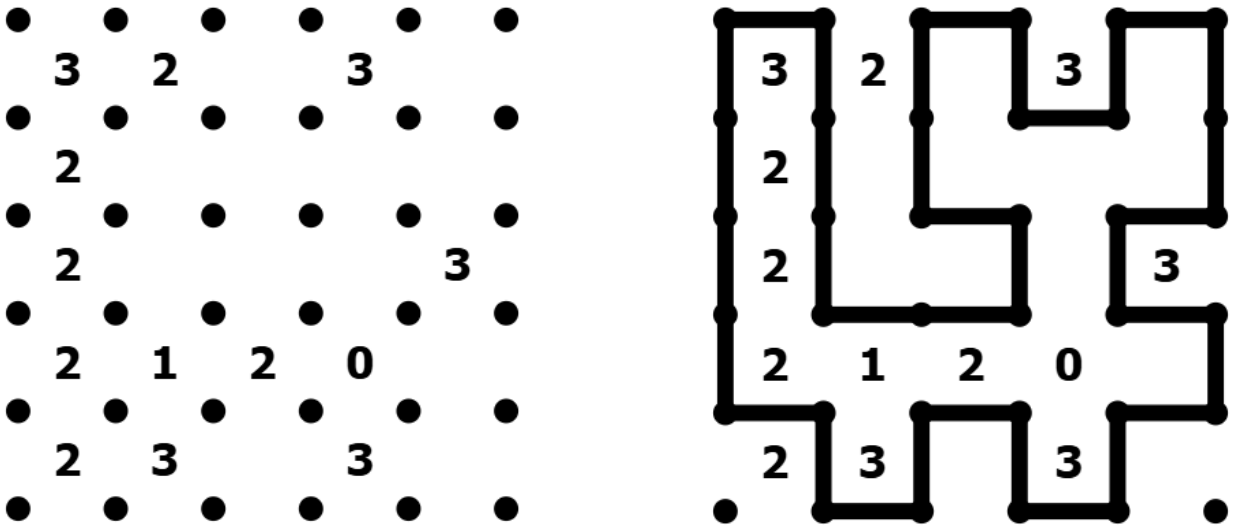
1 Introduction

1.1 Puzzle Description and Rules

Slitherlink, also known as Fences, Loop the Loop, Loopy, Sli-Lin, Rundweg, or Suriza, is one of the most popular and interesting logic puzzles developed by Nikoli[1] in 1989. It belongs to the family of loop puzzles, where the goal is to draw connected edges on a rectangular lattice of vertices to form a single loop while adhering to certain rules. The squares formed by the vertices, also called clues or cells, can contain a whole number from 0 to 3. The rules of the game are simple and can be summarized in the following points:

- The vertices can be connected using horizontal or vertical edges to form only a single loop.
- The loop cannot branch off or intersect itself.
- The numbers in cells indicate how many of its four surrounding edges are part of the loop. Empty cells can use any of its surrounding edges.

Instances of Slitherlink are designed by experts in a way that ensures they only have one valid solution. Figure 1.1 shows an example of a Slitherlink instance and its solution.



(a) Instance (b) Solution

Figure 1.1: Example of a (a) Slitherlink puzzle instance and (b) its corresponding solution. The numbers inside the cells specify how many surrounding edges must be part of the loop.

[1] https://www.nikoli.co.jp/en/puzzles/slitherlink/

1.2 Motivation and Relevance

The simple nature of pencil-and-paper puzzles that transcend the language barrier has not only gained them popularity worldwide as recreational, brain-teasing challenges, but also made them ideal as research subjects in of computer science. Due to their easy rules, clear objective, and high combinatorial complexity, they serve as controlled experimental environments for developing and comparing various methods and algorithms under reproducible conditions, contributing to the broader study of constraint modeling and combinatorial optimization.

Furthermore, the importance of solving puzzles grows as they contribute to artificial intelligence research [3]. The techniques used in these puzzles can often be transferred to real-world problem-solving tasks, as they share similar characteristics. For instance, the global single-loop constraint of Slitherlink resembles connectivity requirements encountered in routing and network design problems.

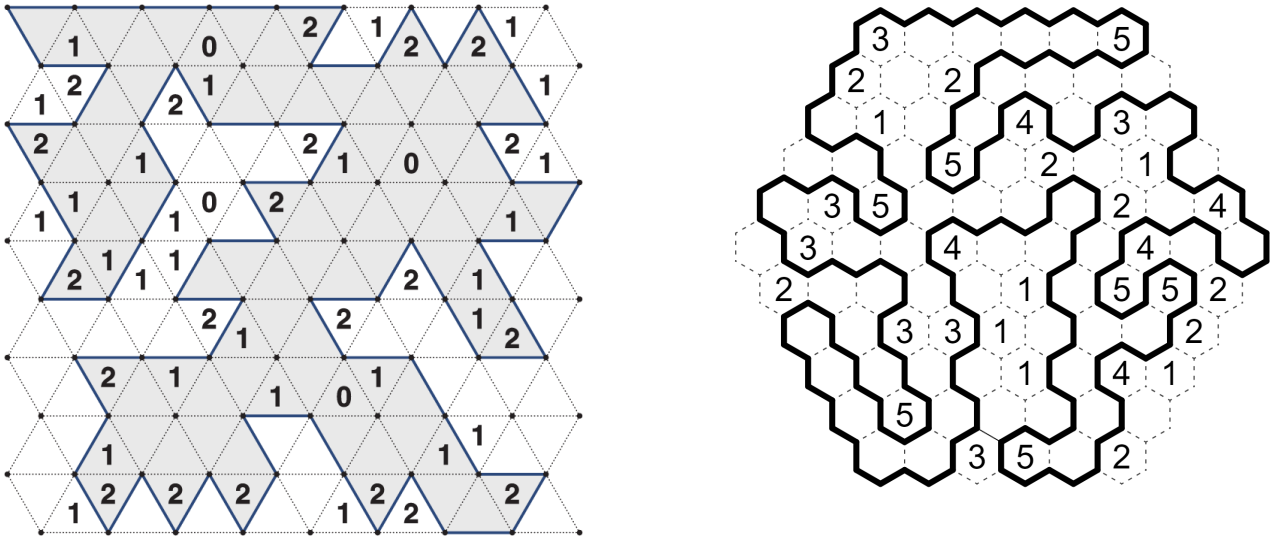
In addition to computational approaches, puzzle solving also offers opportunities to study human-like thinking processes. Methods such as Deductive Search, which was developed by Cameron Browne [1] and is the focus of this thesis, attempt to emulate the reasoning of human solvers, which is valuable for providing more interpretable and explainable solution processes than purely search-based methods.

1.3 Literature Overview

The problem of whether a given Slitherlink instance admits a solution was proven by Yato Takayuki to be NP-Complete [9] through a polynomial time reduction of the Hamiltonian Path Problem. This result indicates that no polynomial-time algorithm is known for solving arbitrary instances unless P = NP. Yato further proved that the Another Solution Problem (ASP) of Slitherlink, which asks whether a second distinct solution exists for an instance whose solution is already known, is also NP-complete [10]. Additionally, Jonas [4] has shown that selected variants of Slitherlink, such as the triangular or hexagonal variants shown in Figure 1.2, are NP-Complete as well.

The brute-force approach to solving Slitherlink is by exploring all possible paths and all combinations of assignments to variables. However, this method is inefficient, since the search space and time explode exponentially as the size of the puzzle increases. For an $m \times n$ board, the number of edge variables is 2mn + m + n. Since each edge can be either on or off, the resulting search space contains $2^{2mn+m+n}$ possible assignments [5]. Therefore, more sophisticated methods have been developed to solve Slitherlink using intelligent techniques, like heuristics or constraint propagation.

David Westreicher demonstrated in his Bachelor's Thesis [8] that Slitherlink can be solved using pure SAT logic by encoding the variables and rules as a boolean formula, converting it to a CNF, and solving it using a SAT solver. In [6], the puzzle was also solved using a *Satisfiability Modulo Theories (SMT)* solver, which is a higher-level extension of SAT that utilizes integers, arrays, and other data structures, allowing for modeling more complex mathematical constraints.



(a) Triangular Slitherlink

(b) Hexagonal Slitherlink

Figure 1.2: (a) Triangular Slitherlink with 3 edges surrounding a cell, and (b) Hexagonal Slitherlink with 6 edges surrounding a cell.

In addition to pure logical approaches, other researchers were able to solve Slitherlink using different methods, such as *Monte-Carlo Tree Search (MCTS)* [3], which is a stochastic search method based on random sampling and heuristic-guided tree exploration. Moreover, it was shown how the problem can also be solved using Integer Programming [2] or using Zero-suppressed Decision Diagrams [11].

1.4 Goal of the Thesis

The goal of this thesis is the implementation and systematic comparison of two fundamentally different approaches for solving Slitherlink puzzles: a computational solver based on CP-SAT[2], developed by Google's OR-Tools, and a Deductive Search solver inspired by the work of Browne [1]. The CP-SAT approach models the puzzle as a constraint satisfaction problem and relies on advanced propagation and search techniques to efficiently explore the solution space. Particular attention is given to the modeling of the global single-loop constraint, for which three different formulations are presented and evaluated.

In contrast, the Deductive Search solver follows a human-inspired reasoning process. Solutions are derived through the repeated application of logical deduction rules and depth-limited hypothetical reasoning. Unlike purely computational approaches, each deduction can be explained and traced, making the solving process highly interpretable and closer to the reasoning strategies employed by human puzzle solvers.

Both approaches are evaluated experimentally under identical conditions and compared with respect to correctness, runtime performance, scalability, and interpretability. Furthermore, the Deductive

[2] https://developers.google.com/optimization/cp/cp_solver

Search solver is used to compute difficulty values for a collection of benchmark instances. These values are then compared to the difficulty ratings assigned by their original puzzle sources in order to investigate the relationship between human-perceived and deduction-based difficulty.

1.5 Structure of the Thesis

- **Chapter 1: Introduction** introduces Slitherlink, describes its rules and motivation, reviews relevant literature, and presents the goal of the thesis.
- **Chapter 2: Methodology** formulates Slitherlink as a constraint satisfaction problem, describes the mathematical constraints used in the model, and presents three different approaches for enforcing the single-loop constraint. Furthermore, it introduces Deductive Search as an approximation of human-like solving, lists different strategies and deduction rules, describes its core operations, and explains how the method is used to assess difficulty.
- **Chapter 3: Results** describes the experimental setup and benchmark instances, and presents the results of the comparison between the proposed methods.
- **Chapter 4: Discussion** interprets the results, highlights the strengths and weaknesses of each method, and addresses limitations.
- **Chapter 5: Conclusion** summarizes the main findings and contributions of the thesis and suggests possible directions for future work.

2 Methodology

2.1 Constraint Programming

A *Constraint Satisfaction Problem (CSP)* is defined as a problem represented by a triple $< X, D, C >$, where X is a set of variables that should be assigned a value from its respective domain that is part of the set of domains D, while satisfying a set of constraints C that connect subsets of variables [7]. Under this definition, Slitherlink can be formulated as a CSP as follows:

- **Variables:** Each edge of the puzzle is represented by a boolean decision variable x_e. For an $H \times W$ board, where H and W denote the height and width of board respectively, the total number of edge variables is

$$(H + 1)W + (W + 1)H.$$

- **Domains:** Each variable has domain

$$x_e \in \{0, 1\},$$

  where 1 indicates that the edge should be on and 0 otherwise.

- **Constraints:** The variables are connected through constraints derived from the rules of Slitherlink (see section 1.1), which are described in the following sections.

2.1.1 Clue Constraints

One of the rules of Slitherlink states that the amount of drawn edges surrounding a clue cell should be equal to the value of the clue. Let $E(c)$ denote the set of the four edges surrounding clue c. The rule can be formally expressed as:

$$\sum_{e \in E(c)} x_e = c$$

The constraint is imposed for every clue cell in the puzzle, while empty cells are ignored.

In the CP-SAT model, this relation is implemented using *addEquality* between the clue value and the sum of the surrounding edge variables.

2.1.2 Vertex Constraints

The second rule of Slitherlink requires the solution to form loops without branching, intersections, or open endpoints. This property can be expressed using vertex degrees.

Let I(v) denote the set of edges incident to vertex v. The degree of a vertex is defined as

$$deg(v) = \sum_{e \in I(v)} x_e$$

A degree of 3 or 4 would indicate branching or intersection points, while a degree of 1 would indicate an endpoint of an open path. Consequently, every vertex must satisfy

$$deg(v) \in \{0, 2\}$$

Equivalently,

$$\sum_{e \in I(v)} x_e \in \{0, 2\}$$

Thus, a vertex either does not belong to the loop at all (degree 0) or belongs to exactly one loop (degree 2).

In the CP-SAT model, an auxiliary integer variable is introduced for each vertex to represent its degree. The variable is linked to the sum of its incident edge variables through *addEquality* constraints. Since CP-SAT integer variables are specified by a lower and upper bound, the degree variables are initially created with domain [0,2]. This domain still permits the value 1, which can be excluded by adding inequality constraints using *addDifferent*.

It is worth noting that even with these constraints, the solver could possibly generate an empty solution with all edge variables set to 0 for an instance with only clues of value 0. This case can be excluded by imposing the following constraint:

$$\sum_{e \in E} x_e \geq 1$$

where E denotes the set of all edge variables. This constraint guarantees at least one edge to be on, and since degree 1 is not allowed, the solver is forced to set other edges on to form a loop, avoiding empty solutions.

2.1.3 Single Loop Constraint

With all the previously defined constraints, the solver is now able to generate solutions with one or more loops satisfying the clues. Exactly one of these solutions corresponds to a valid Slitherlink solution containing a single loop. Figure 2.1 shows possible solutions found for the same instance.

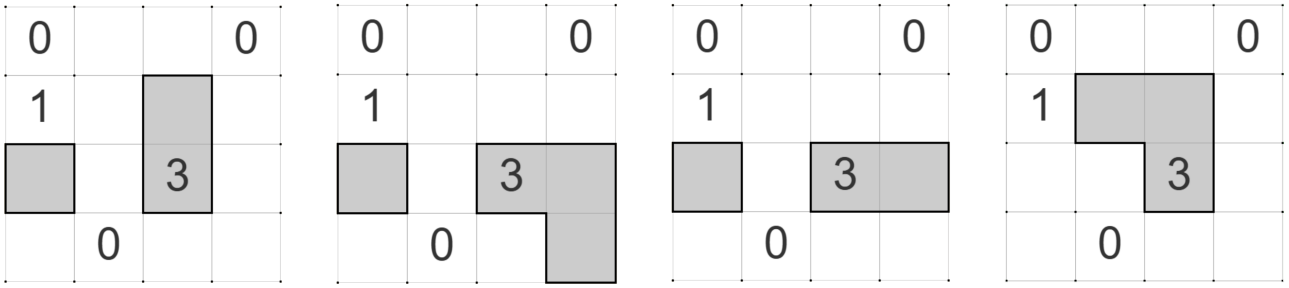


Figure 2.1: Candidate solutions generated after enforcing only the local constraints. Only the solution on the far right forms a single loop and is therefore valid.

All constraints defined so far are local constraints, meaning that they only relate variables within a small neighborhood of the board. In contrast, the single loop constraint is a global constraint, as its satisfaction depends on the structure of the entire graph rather than on local relationships between neighboring variables. Consequently, modeling this constraint within a CP framework is considerably more challenging. This thesis investigates three different approaches for handling the single loop requirement.

Solution Verifier

This approach does not model the single-loop constraint directly. Instead, the CP-SAT solver is instructed to enumerate feasible solutions satisfying all local constraints. Each candidate solution is then verified using a graph traversal procedure. Starting from an arbitrary vertex of degree two, the traversal marks all vertices belonging to the discovered loop. If another unvisited vertex of degree two remains afterwards, multiple loops are present and the solution is rejected. Otherwise, the solution is accepted as valid.

Lazy Constraints

A lazy constraint is a constraint applied dynamically during the solving process. After the solver generates a solution, using the same graph traversal in the previous method and modifying it to store the edges of each loop, we can add the following constraint to the model:

$$\sum_{e \in L} x_e < |L|$$

where L denotes the set of edges belonging to the loop. After running the solver again, it is obliged to set at least one of the loop edges off, which forbids that loop from appearing in future solutions, unlike the solution verifier method that could generate the same loop with a different combination of other loops in future solutions (see Figure 2.1). Figure 2.2 shows the generated solutions by the lazy constraint approach for the same instance.

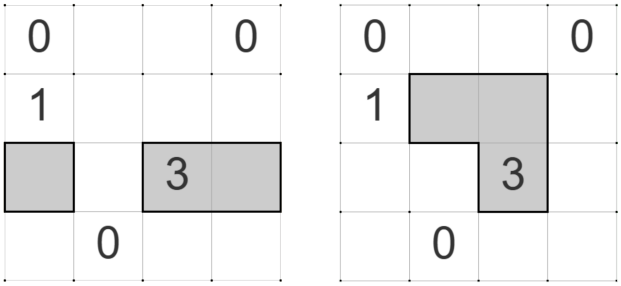


Figure 2.2: Candidate solutions generated using the lazy constraints method. The same loops are forbidden and can never appear in other solutions.

However, this approach suffers from an important edge case, shown in Figure 2.3, where it will fail to find the correct solution. One of the candidate solutions might contain the actual correct loop, while also having a random small loop elsewhere. As a result, the solver will detect more than one loop and apply the lazy constraint on both, which prohibits the correct loop from appearing again.

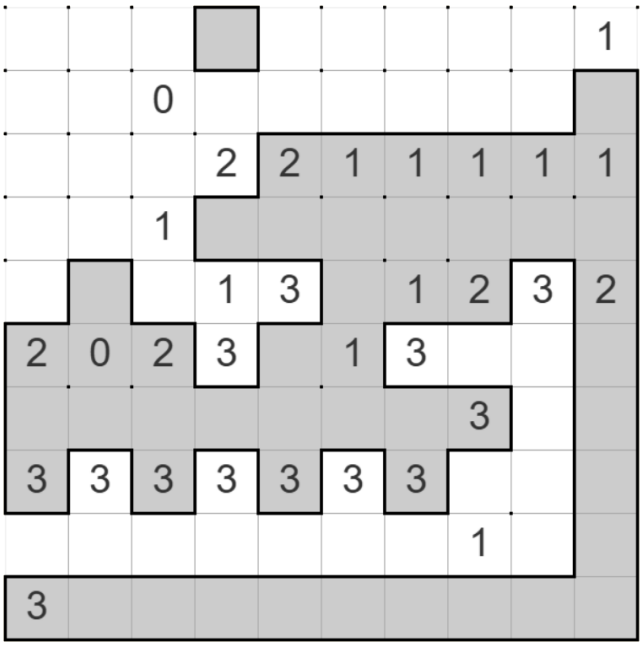


Figure 2.3: Edge case for the lazy constraints method. The big loop satisfies all clues and forms the correct solution, but it is still forbidden due to the other small loop.

To handle this situation, each detected loop is considered individually. The edges of the loop are copied into a separate board representation and verified against the clue constraints. If the loop alone satisfies all clues, it is returned as the correct solution. Otherwise, the corresponding lazy constraint is generated and added to the model.

Single Loop as a Normal Constraint

Unlike the previous two approaches, which verify or eliminate invalid loops after a solution has been generated, this formulation incorporates the single loop requirement directly into the CP-SAT model through additional variables and constraints. The central idea is to transform the undirected loop into a directed graph and enforce connectivity through vertex numbering.

Instead of Boolean variables, each edge is represented by an integer variable

$$x_e \in \{-1, 0, 1\}$$

where:

- $x_e = 1$ means the edge is on and oriented according to a predefined direction.
- $x_e = -1$ means the edge is on in the opposite direction.
- $x_e = 0$ means the edge is off.

The same clue and vertex constraints defined before are applied to the absolute value of the new edge variables.

Let $In(v)$ and $Out(v)$ denote the incoming and outgoing edges of vertex v respectively. The following constraint is added for every vertex:

$$\sum_{e \in In(v)} |x_e| = \sum_{e \in Out(v)} |x_e|$$

This guarantees that every vertex belonging to the loop will have exactly one incoming and one outgoing edge. To ensure that all selected edges belong to a single connected loop, a new integer variable $n(v)$ is assigned to each vertex as a numbering label. By adding the following constraint:

$$n(v) > 0 \iff deg(v) > 0$$

we ensure that vertices belonging to the loop receive positive labels, while vertices that do not belong receive label 0. Furthermore, a Boolean variable $start(v)$ is introduced for each vertex and is true if and only if the vertex has label 1:

$$start(v) = 1 \iff n(v) = 1$$

To restrict the solution to have one loop only, we add the constraint that only one vertex has label 1:

$$\sum_{v} start(v) = 1$$

Finally, for every directed edge from v to u that is set on, vertex u should either have a greater label or label 1:

$$v \to u \implies n(u) > n(v) \quad \vee \quad n(u) = 1$$

This creates a strictly increasing sequence of labels around the loop until it closes at the unique start vertex.

Because only one vertex may receive label 1 as the start vertex and all other labels must increase along directed edges, every selected vertex becomes part of a single numbering sequence. Multiple disjoint loops would require multiple independent starting points and therefore violate the uniqueness of the start vertex. Consequently, the model allows only the valid solution containing a single loop. Figure 2.4 visualizes the idea for better understanding.

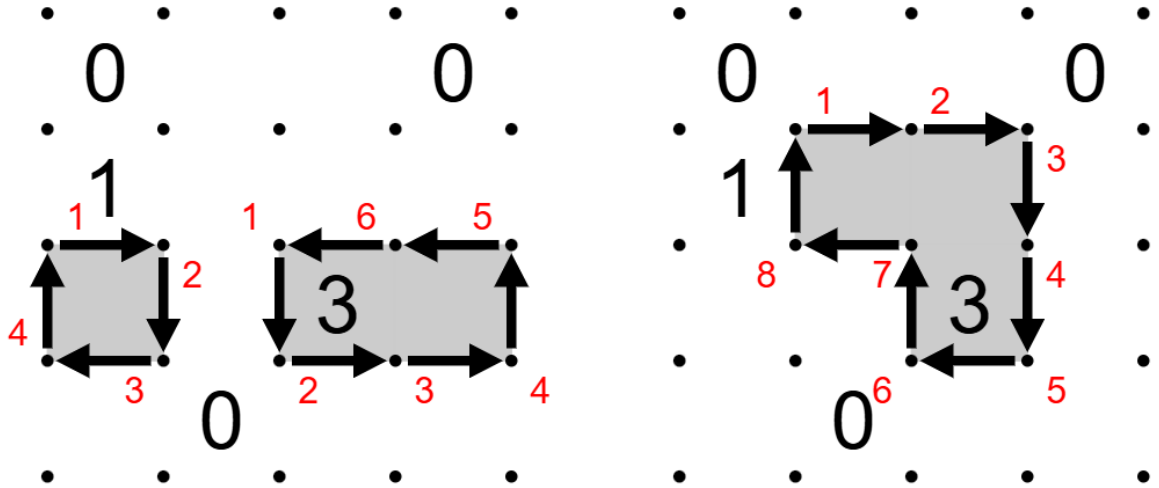


Figure 2.4: The red numbers represent the auxiliary labeling variable for vertices $n(v)$. Each vertex v that belongs to a loop is assigned a positive label. In each loop on the left image, each directed edge leads to a vertex with a higher label until the start vertex $n(v) = 1$ is reached again. By restricting the sum of all $start(v)$ to 1, i.e. allowing only one vertex to have $n(v) = 1$, the solver is forced to generate one loop only, as shown in the right image.

2.2 Deductive Search

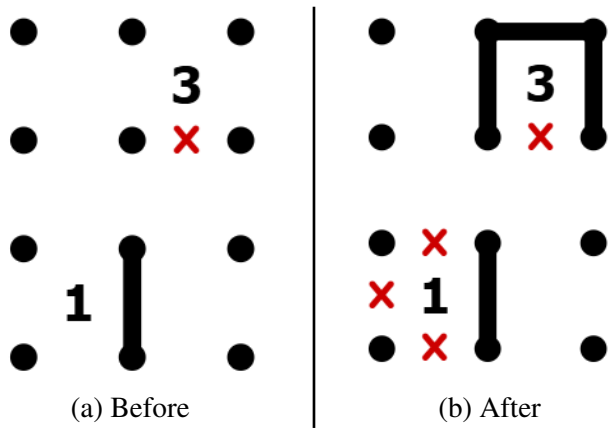
While CP-SAT approaches solve Slitherlink by searching for assignments that satisfy a set of constraints, Deductive Search attempts to emulate the reasoning process of a human player in an iterative breadth-first depth-limited manner. Rather than exploring the search space exhaustively, the method applies local deduction strategies and limited hypothetical reasoning to infer new edge assignments. Introduced by Browne [1], the approach organizes deductions into different Levels of Embedding (LoE), allowing both the solution process and the difficulty of an instance to be quantified. This section presents the deduction strategies, the recursive search procedure, and the difficulty metric used in this thesis.

2.2.1 Strategies and Patterns

Before getting into the algorithmic procedure and implementation details of Deductive Search, it is essential first to illustrate some strategies discovered and employed by human solvers and are fundamental for the simplification part of the algorithm later (see section 2.2.2). These strategies have been adapted from [5] as well as other sources[1].

Clue Rule

Since every clue is surrounded by exactly four edges, knowledge about some of these edges can often determine the state of the remaining ones. This rule is based directly on the clue constraint. If the number of surrounding edges already set on is equal to the value of the clue, all remaining undecided surrounding edges must be set off. Conversely, if the number of surrounding edges already set off is equal to four minus the clue value, all remaining undecided surrounding edges must be set on. Figure 2.5 shows an example. Using this rule as well, all clues of value 0 will immediately have all their surrounding edges set off.



(a) Before (b) After

Figure 2.5: (a) An intermediary state of solving where clue 3 has one edge off and clue 1 has one edge on. (b) After applying the rule, the remaining undecided edges surrounding clue 3 are set on, while the remaining undecided edges surrounding clue 1 are set off.

[1] https://jonathanolson.net/slitherlink/
https://www.conceptispuzzles.com/index.aspx?uri=puzzle/slitherlink/techniques
https://puzzleparasite.blogspot.com/2011/11/slitherlink-pattern-guide_23.html

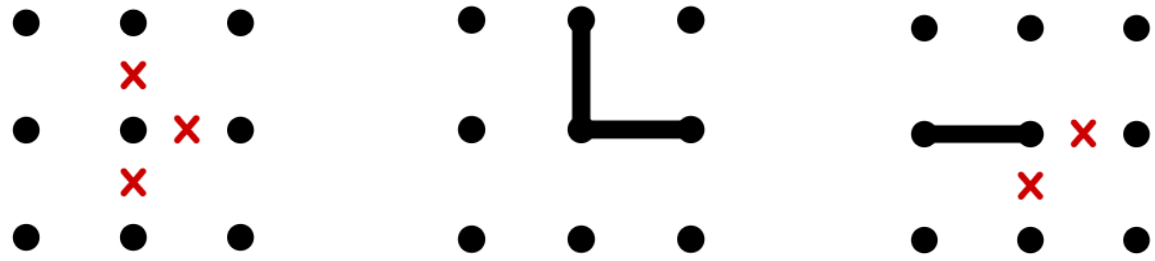
Furthermore, the rule can be used to detect contradictions relating to clues during the solving process. If a clue already has more edges set on than its value permits, or more excluded edges than allowed, the current state cannot lead to a valid solution and is therefore rejected.

Vertex Rule

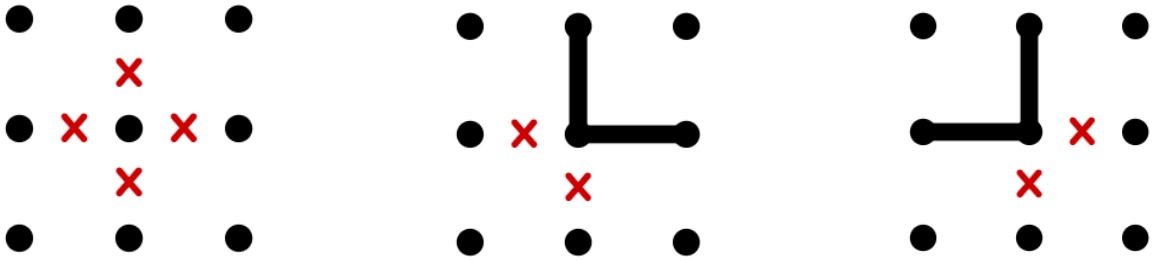
Analogously to the clue rule, this rule is derived directly from the vertex constraints discussed previously. Since every vertex must have degree 0 or 2, knowledge about some incident edges can often determine the state of the remaining ones:

1. If a vertex has three incident edges turned off, the fourth undecided edge must be set off.
2. If a vertex has two incident edges set on, the remaining undecided edges are set off.
3. If a vertex has two incident edges off and one edge on, the fourth undecided edge must be set on.

Figure 2.6 visualizes the three cases.



(a) Before



(b) After

Figure 2.6: Application of the rule on the three cases.

This rule can also be used to detect contradictions in vertices during the solving process. For example, a vertex with more than two incident edges on would create a branch or an intersection. Likewise, if three incident edges are off while the fourth is on, the vertex becomes a dead end that cannot be extended further.

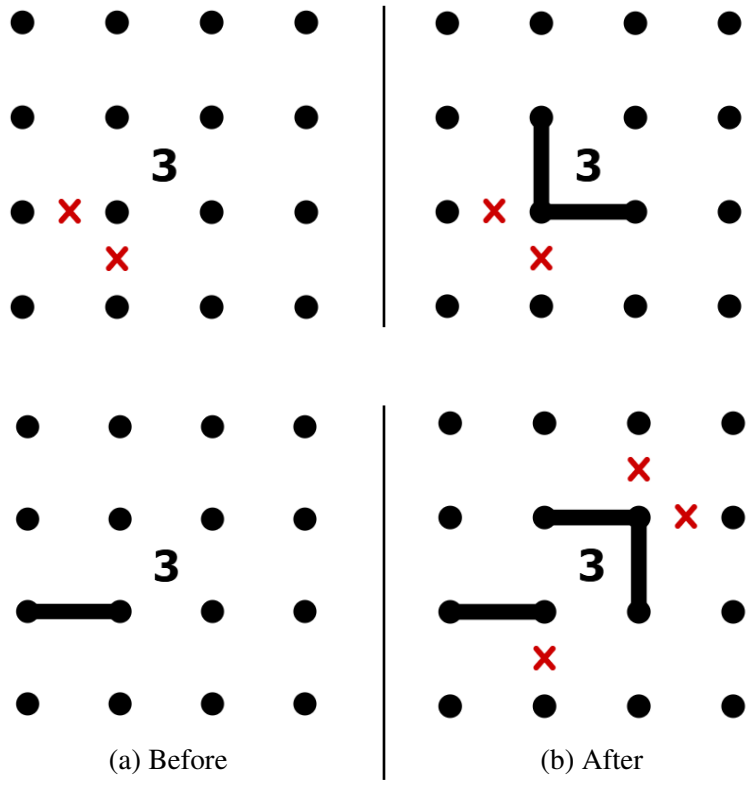
Corners of Three Rule

This rule handles two patterns that occur at one of the corner vertices of a clue with value 3.

The first pattern occurs when the two outer edges of the corner are off. In this case, the two inner edges must be on. Since a clue 3 requires exactly three surrounding edges to be selected, at least one of the two inner edges must be on. However, selecting only one of them would leave the corner vertex with degree 1, violating the vertex constraint. Therefore, both inner edges must be selected.

The second pattern occurs when one of the outer edges is on. In this case, the other outer edge cannot be on, otherwise, the two inner edges will be set off according to the vertex rule, which violates the clue constraint. Therefore, the other outer edge is forced to be off, and the line is forced to continue through one of the inner edges. Regardless of which edge is picked, the inner two edges of the opposite corner vertex will be on.

Figure 2.7 shows a visual example of both patterns.



(a) Before (b) After

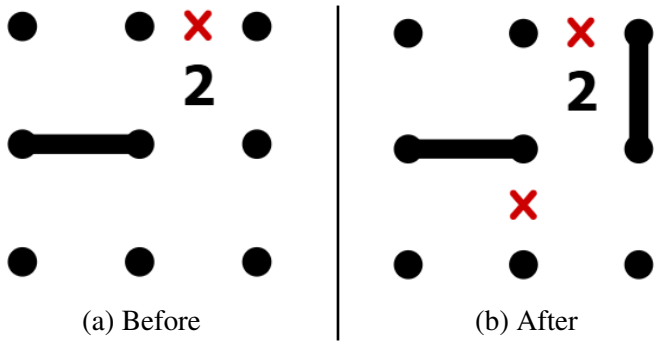
Figure 2.7: Example of the rule on the bottom-left corner for both patterns.

The first pattern also covers clues at the corners of the board. Since the edges outside the board are considered absent and therefore off, the condition of the rule is automatically satisfied. As a consequence, both inner edges incident to the vertex at the corner of the board are on.

Corners of Two Rule

Similarly to the previous rule, this rule handles a pattern that could occur at a corner vertex of a clue with value 2. If one of the outer edges is on, and one of the inner edges of the opposite corner is off, it can be deduced that the second outer edge of the current corner must be off. If the second outer edge were also on, the clue would be violated because three surrounding edges would already be determined incorrectly. Therefore, the second outer edge must be off, which in turn forces the remaining inner edge of the opposite corner to be on.

Figure 2.8 shows a visual example of this pattern.



(a) Before (b) After

Figure 2.8: Application of the Corners of Two rule. One outer edge is on and one opposite inner edge is off, forcing the remaining outer edge off and the remaining opposite inner edge on.

Corners of One Rule

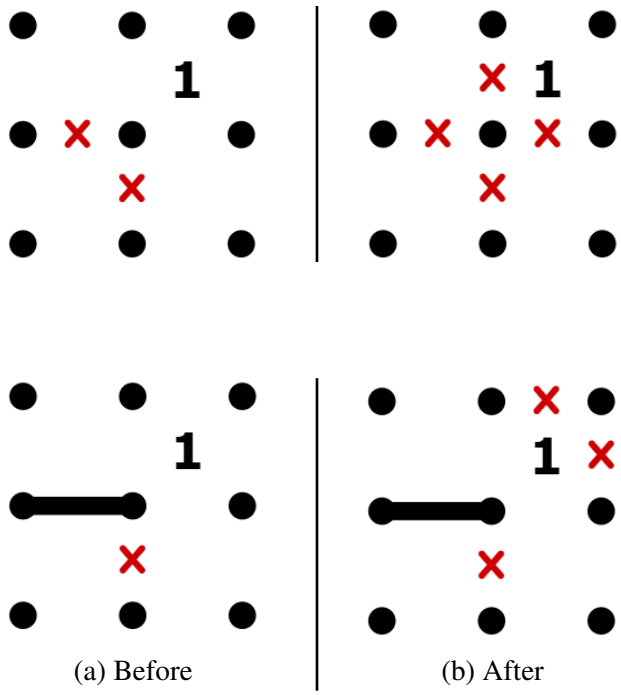
This rule also handles two patterns that could occur at a corner vertex of a clue with value 1.

The first pattern occurs when both outer edges of the corner are off. If one of the inner edges is on, the other is forced to be on as well by the vertex rule, but this violates the clue. Therefore, both inner edges must be off.

The second pattern occurs when one of the outer edges is on and the other is off. As a result, the line is forced to continue its path through one of the inner edges. In either case, the clue will be satisfied. Thus, the inner edges of the opposite corner must be set off.

Figure 2.9 gives a visual example of both patterns.

Similarly to the corners of three rule, this rule also applies to clues of value 1 located at the corners of the board. Since the edges outside the board are considered off, the two inner edges incident to the vertex at the corner of the board are forced to be off.

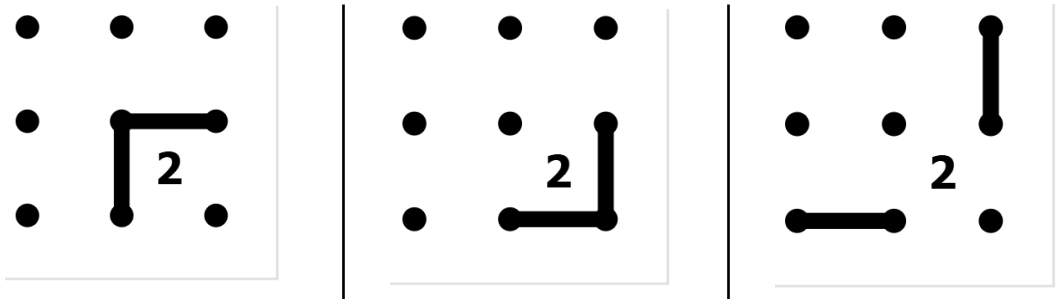


(a) Before (b) After

Figure 2.9: Example of the rule on the bottom-left corner for both patterns.

Two at Corners Rule

Unlike the previous corner rules, this deduction is made by considering all valid local configurations of a clue of value 2 located at a corner of the board. Since the clue requires exactly two surrounding edges to be selected, only two configurations satisfy both the clue and vertex constraints. Although these configurations differ locally, they share a common property: in both cases the line continues along the border of the board. Consequently, the two border-adjacent edges are forced on and can be assigned immediately, as shown in Figure 2.10.

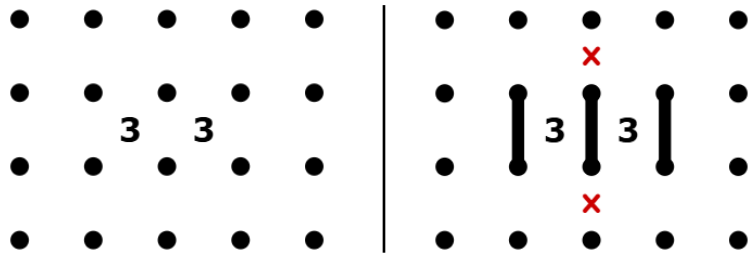


(a) Configuration A (b) Configuration B (c) Result

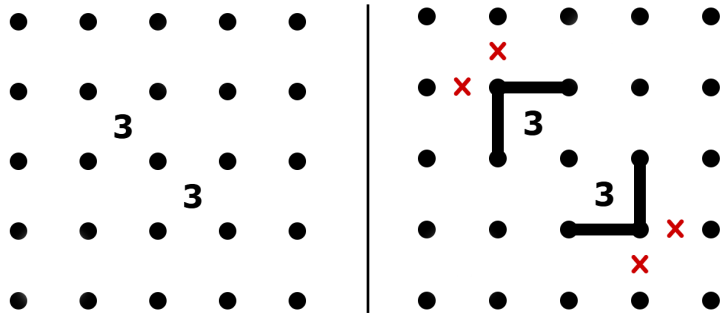
Figure 2.10: A clue of value 2 at a corner of the board has two valid local configurations, shown in (a) and (b). Both configurations force the border-adjacent edges shown in (c).

Neighbors of Three Rule

This rule captures deduction patterns that arise from pairs of neighboring clues with value 3, either orthogonally or diagonally. These patterns are obtained by repeatedly applying the clue and vertex rules until no further deductions are possible. The resulting forced edge assignments are shown in Figure 2.11.



(a) Orthogonal

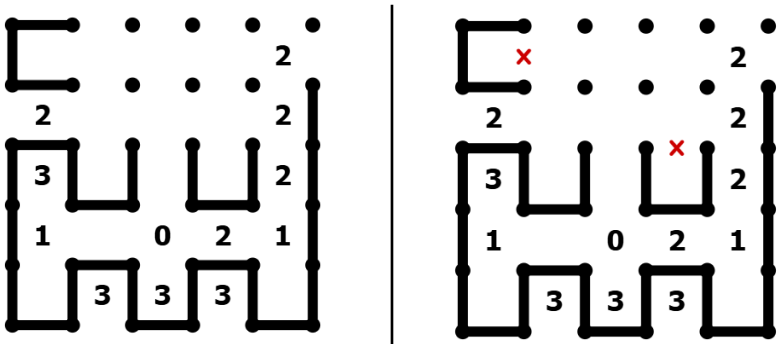


(b) Diagonal

Figure 2.11: Application of the Neighbors of Three rule.

1x1 Rule

This rule simply avoids closing small 1x1 loops around any cell and forming multiple disjoint loops. If any cell has three of its surrounding edges on, the fourth will be forced off, as demonstrated in Figure 2.12.

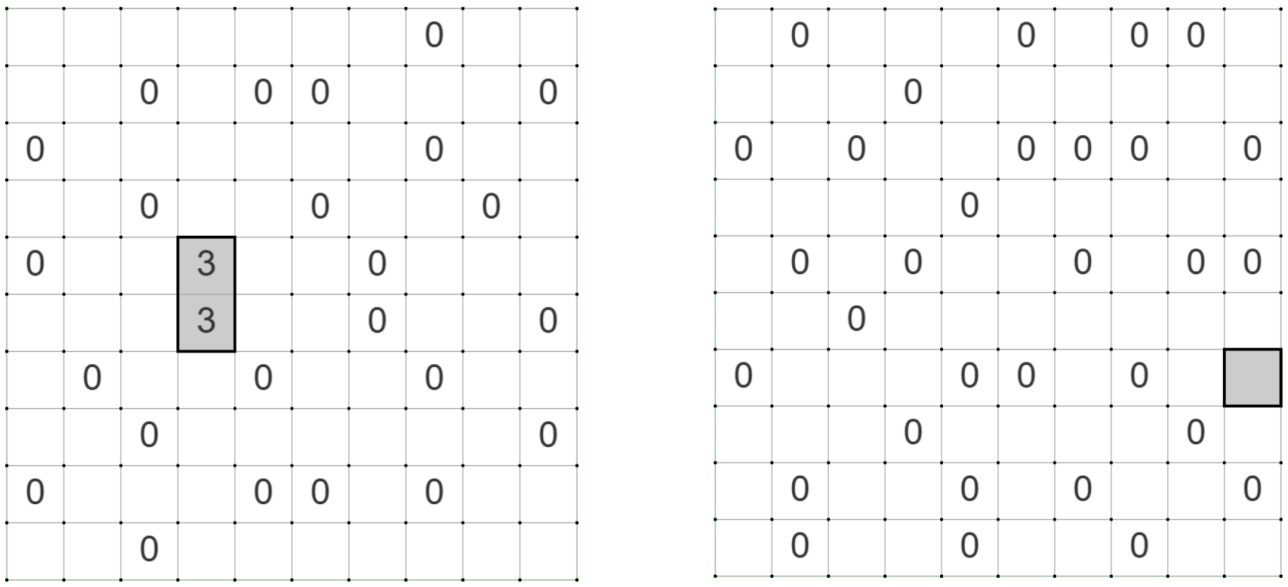


(a) Before (b) After

Figure 2.12: Application of 1x1 rule.

Edge Cases

While both Neighbors of Three Rule and 1x1 Rule are considered useful for deducing values for some edges and helping reach the solution, especially in large instances, it is important to note that these rules are not always valid in every Slitherlink instance, as there exist edge cases that should be accounted for, such as the ones in Figure 2.13, where the application of the rules would lead to a contradictory state.



(a) Neighbors of Three Rule Edge Case

(b) 1x1 Rule Edge Case

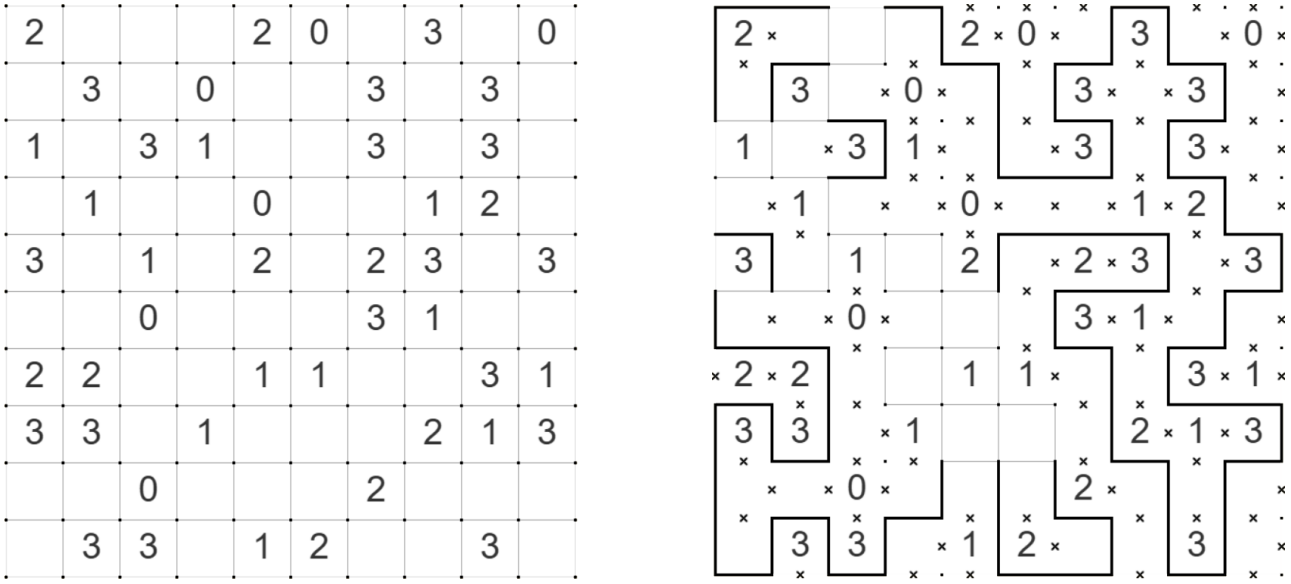
Figure 2.13: (a) Applying the Neighbors of Three Rule would set the shared edge on, which is not part of the solution. (b) All other edges are off, and the only loop that can be formed is 1x1.

2.2.2 Core Operations

While the deterministic deduction rules presented in the previous section among other strategies discovered by players build the foundation for solving Slitherlink, oftentimes they are insufficient and many instances reach a state where no more moves or deductions can be made, and hypothetical assignments are needed to progress further and derive additional information. This section presents the three main operations proposed by Browne [1]: *Simplification, Shaving,* and *Agreement*.

Simplification

Simplification is the process of applying the logical deduction rules previously defined and propagating constraints to prune domains and assign forced values to variables without any hypotheses until a fixed point is reached, where the puzzle is either solved, a contradiction arises, or no more updates can be made and hypothetical assignments are needed. Figure 2.14 illustrates an example of applying the simplification process.



(a) Before (b) After

Figure 2.14: Applying Simplification on a Slitherlink instance.

Simplification is not only performed at the start of the solving process, it is also performed after the application of *Shaving* or *Agreement* to propagate their consequences. It repeatedly applies the deduction rules until none of them can update any more variables, as shown in Algorithm 1.

Algorithm 1: Simplification

```
1  Function Simplify(Board S):
2      totalUpdates ← 0
3      repeat
4          updates ← 0
5          updates ← updates + VertexRule(S)
6          updates ← updates + ClueRule(S)
7          updates ← updates + CornersOfThreeRule(S)
8          updates ← updates + CornersOfTwoRule(S)
9          updates ← updates + CornersOfOneRule(S)
10         updates ← updates + 1x1Rule(S)
11         totalUpdates ← totalUpdates + updates
12     until updates = 0
13     return totalUpdates
```

Note that *Neighbors of Three Rule* and *Two at Corners Rule* are not included. These rules depend exclusively on the relative positions of clue values and not on intermediate edge assignments. Their deductions can be computed once before the search process begins. Reapplying them during solving does not produce additional deductions and only increases computational overhead. Therefore, they have their own function *preprocessing*, shown in Algorithm 2, which is executed once at the start.

Algorithm 2: Preprocessing

```
1 Function Preprocess(Board S)
2     return NeighborsOfThreeRule(S) + TwoAtCornersRule(S)
```

Shaving

Shaving is the breadth-first depth-limited process of testing a hypothesis by assigning a value to a variable from its domain and simplifying. The operation is formally expressed [1] as follows:

$$(\exists v \in X_i^* : X_i \cdot v \Rightarrow \bot) \Rightarrow X_i \leftarrow X_i \backslash v$$

where X_i denotes a variable, and X_i^* denotes the subset of the remaining values of its domain. $X_i \cdot v$ denotes the instantiation or assignment of value v to the variable, while $X_i \backslash v$ denotes the elimination of value v from its domain. $\bot$ means a contradiction. Thus, the expression states that if assigning a value v from the domain of X_i leads to a contradiction, then v can be eliminated from the domain of X_i.

In Slitherlink, edge variables have binary domains {0,1}. So if a hypothetical assignment of a value to an edge and simplifying after leads to a contradiction, the other value can be directly assigned as the correct value. Figure 2.15 shows an example of Shaving in Slitherlink.

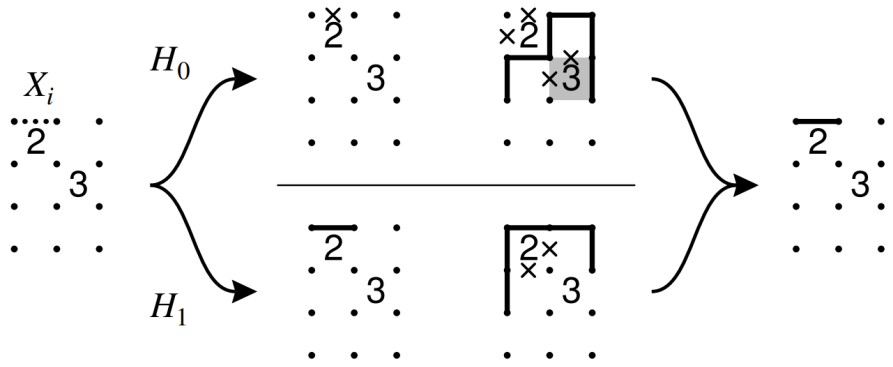


Figure 2.15: Example of Shaving. H_0 and H_1 represent hypotheses for testing edge X_i off (0) or on (1) and simplifying. H_0 leads to a contradiction, which forces X_i to be on (1) in the original board. Source: [1]

.

After each assignment of a value and simplifying in Slitherlink, there are three outcomes possible:

- It leads to an inconsistent or contradictory state, which results in forcing the other value, as stated before.
- It leads to the solution of the puzzle.
- It does not cause any inconsistency. In that case, *Agreement* is applied.

Agreement

If all hypothetical branches in Shaving are consistent, any common assignments across all hypotheses are directly applied to the original board. This can be formally expressed [1] as follows:

$$(\forall v \in X_i^* : X_i \cdot v \Rightarrow X_j \cdot w) \Rightarrow X_j \leftarrow X_j \cdot w$$

where if the assignment of every value from X_i^* to X_i leads to the assignment of value w to X_j in hypothetical branches, then w is directly assigned to X_j in the original board as the correct value, even if no conclusion can be drawn about X_i.

In Slitherlink, agreement compares the results of two consistent hypothetical branches. Any deduction that appears in both branches must also hold in the original board and can therefore be applied directly. An example is demonstrated in Figure 2.16.

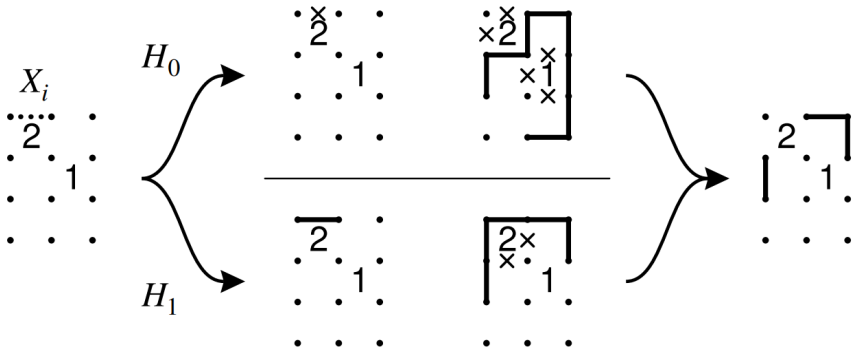


Figure 2.16: Example of Agreement. In both hypotheses, the same three edges are forced to be on (1) after testing X_i and simplifying, so these edges can be safely assigned to this value in the original board. Source: [1]

.

Algorithm 3 illustrates how the agreement function works in our implementation, where S_1 and S_2 represent copies of the original board S for testing hypotheses, and $S(e)$ denotes the value assigned to edge e in S. In this work, only two hypotheses are generated for each edge, corresponding to the values 0 and 1. Agreement therefore reduces to comparing the resulting states after simplifying both hypotheses.

Algorithm 3: Agreement

```
1 Function Agreement(S, S1, S2)
2     foreach edge e do
3         if S1(e) = S2(e) then
4             S(e) ← S1(e)
5     Simplify(S)
```

2.2.3 Algorithmic Procedure

Algorithm 4: Deduce

```
1  Function Deduce(Board S, depth)
2      foreach unknown edge e do
3          S_1 ← copy(S)
4          S_2 ← copy(S)                                  // copy boards for hypotheses
5          S_1(e) ← 1
6          Simplify(S_1)                                  // first hypothesis
7          if depth > 1 then
8              contra_1 ← Deduce(S_1, depth − 1)          // 2-LoE
9          if contra_1 ∨ Contradiction(S_1) then
10             S(e) ← 0
11             Simplify(S)
12             if Solved(S) then
13                 return false
14         if Solved(S_1) then
15             S ← S_1
16             return false
17         S_2(e) ← 0
18         Simplify(S_2)                                  // second hypothesis
19         if depth > 1 then
20             contra_2 ← Deduce(S_2, depth − 1)          // 2-LoE
21         if contra_2 ∨ Contradiction(S_2) then
22             S(e) ← 1
23             Simplify(S)
24             if Solved(S) then
25                 return false
26         if Solved(S_2) then
27             S ← S_2
28             return false
29         if ¬contra_1 ∧ ¬contra_2 then
30             Agreement(S, S_1, S_2)                     // if both tests consistent, apply agreement
31         if contra_1 ∧ contra_2 then
32             return true                                // if both tests contradictory, invalid board
33         if depth > 1 ∧ Δ(S) ≠ ∅ then
34             return false                               // early exit for 2-LoE
35     return false
```

After defining the main operations in Deductive Search, they are all combined in a function *Deduce*, shown in Algorithm 4, where they operate together to produce new deductions. For each undecided edge, two copy boards S_1 and S_2 of the original are created for testing both hypotheses. For each hypothesis, the board is checked whether it is contradictory or solved. If no value causes a contradiction and no solution is found, Agreement is applied. If both hypotheses lead to contradictions, $Deduce$ returns true, indicating that the current board state is inconsistent. Otherwise, it returns false. If no more deductions can be found at 1-LoE, $Deduce$ is recursively called at 2-LoE, where $contra_1$ and $contra_2$ carry the result of the recursive call. $\Delta(S)$ denotes the set of changes made in S.

Contradiction and Solution Checks

$Contradiction(S)$ checks for any contradiction in S through the contradiction flags set by *Clue Rule* and *Vertex Rule* during any simplification process, as well as a graph traversal similar to the one used in Constraint Programming that checks how many loops are formed. Unlike the CP formulations, intermediary states in Deductive Search are typically incomplete and therefore do not necessarily contain any loop. Consequently, the graph traversal distinguishes between three cases: no loop, exactly one loop, and multiple loops. If any premature loop is formed, $Contradiction(S)$ returns true.

The same graph traversal is also used in $Solved(S)$ to check if a single loop is formed. If exactly one loop is detected, $Solved(S)$ additionally verifies that all clue constraints are satisfied, and returns true if that was the case.

Levels of Embedding (LoE)

A *Level of Embedding* represents the depth of hypothetical reasoning that requires mental storage for future backtracking. Browne [1] argues that humans are effectively limited to approximately two levels of embedding, since deeper hypothetical reasoning rapidly exhausts short-term memory. which is why the algorithm is executed at a maximum of 2-LoE. The $Solve$ function, shown in Algorithm 5, attempts to solve the puzzle at different LoEs:

- **0-LoE:** simple preprocessing and simplification till a fixed point. Some puzzles are solved at this level already.
- **1-LoE:** calls $Deduce$ in several passes for testing hypotheses until the puzzle is solved or no more changes are made. Any deductions made by shaving or agreement at this level are applied directly to the original board. If $Deduce$ returns a contradiction, the puzzle cannot be solved.
- **2-LoE:** If the puzzle is not solved and no more changes occur in 1-LoE, 2-LoE calls $Deduce$ recursively to test hypotheses inside hypotheses. Whenever a deduction is propagated back to the original board, the recursive search is terminated early and control returns to 1-LoE. This reduces the computational cost associated with deeper hypothetical reasoning.

Algorithm 5: Solve

```
1  Function Solve(S)
2      Preprocessing(S)
3      Simplify(S)
4                                                          // 0-LoE
5      if Contradiction(S) then
6          return CONTRADICTORY
7      if Solved(S) then
8          return SOLVED
9      while ¬Solved(S) and Δ(S) ≠ ∅ do
10         while ¬Solved(S) and Δ(S) ≠ ∅ do
11             Deduce(S, 1)
12                                                         // 1-LoE
13             if Contradiction(S) then
14                 return CONTRADICTORY
15         if ¬Solved(S) then
16             Deduce(S, 2)
17                                                         // 2-LoE
18             if Contradiction(S) then
19                 return CONTRADICTORY
20     if Solved(S) then
21         return SOLVED
22     return NON-DEDUCIBLE
```

Algorithms 4 and 5 together constitute the complete deductive solver. The solver first exhausts all deductions obtainable through direct simplification, then progressively applies hypothetical reasoning at increasing Levels of Embedding. If a solution cannot be reached after exhausting deductions at 2-LoE, the puzzle is classified as non-deducible within the limits of the implemented reasoning model.

2.2.4 Difficulty Assessment

The purpose of the deductive solver is not only to solve Slitherlink instances but also to estimate their difficulty from a human-solving perspective. Since the solver operates through a collection of deduction rules and hypothetical reasoning at different levels of embedding, the amount and complexity of reasoning required can be used as a measure of puzzle difficulty. Browne [1] proposes the following formula for measuring difficulty:

$$diff(S) = \frac{S_0 + 4(S_1 + V_1 + A_1) + 9(S_2 + V_2 + A_2)}{\sum_{i=1}^{n} D_i}$$

where S_n, V_n, and A_n denote the number of deductions made by simplification, shaving, and agreement respectively at n-LoE on the original board. The cognitive cost of reasoning increases with the level of embedding. Thus, deductions performed at n-LoE are penalized with a weight factor of

$(n + 1)^2$. The denominator represents the sum of all domains and serves as a normalization factor according to the puzzle size.

A difficulty value close to zero indicates that the puzzle can be solved almost entirely through direct deductions, whereas higher values indicate increasing reliance on hypothetical reasoning and deeper levels of embedding. Puzzles that cannot be solved by the deductive solver within the imposed limit of 2-LoE are classified as non-deducible and are not assigned a finite difficulty score.

To calculate the difficulty score, counters are passed as parameters to the *Deduce* function and incremented whenever a deduction is produced by simplification, shaving, or agreement. If a hypothetical branch leads to a contradictory state, its counters are discarded. Consequently, only deductions that ultimately contribute to progress on the original board are included in the final difficulty score.

3 Results

3.1 Experimental Setup

All experiments were conducted on a machine equipped with an Intel Core i5-12500H processor, 16 GB RAM, and Windows 11. The implementation[1] was developed in Java SE 17 and Apache Maven. Constraint programming experiments were performed using Google OR-Tools CP-SAT[2] version 9.14.6206. Java was selected due to the availability of an official Google OR-Tools API, platform independence through the JVM, and its object-oriented design, which facilitated the representation of puzzle states and solving procedures.

3.2 Benchmark Instances

Three collections of benchmark instances are used in experiments with a total of 2626 instances varying in size and difficulty:

- **Janko**[3]: 1230 instances, with difficulty labels from 0 (*leicht*) to 9 (*schwer*). 10 instances were not ranked in the difficulty order.
- **Nikoli**[4] **Fresh**: 1000 instances, with difficulty labels from 1 to 10. There are some instances with a missing difficulty label that are ranked under difficulty 1, as well as among the most difficult ones, so they were assigned a difficulty of 0 and 10 respectively for consistency with other instances.
- **Nikoli Tokoton**: 396, with difficulty labels from 1 to 32.

3.3 Experiments

The four solvers: Constraint Programming with Solution Verifier (CP-SV), Lazy Constraints (CP-Lazy), Single Loop as a Normal Constraint (CP-Normal), as well as the Deductive-Search solver (DS) were evaluated in different experiments under the same conditions and compared in terms of correctness, performance, and scalability. The DS solver was also used to measure the difficulty of the instances for comparisons with the difficulty labels given to the instances by their designers.

[1] https://github.com/Omarr-Alkhatib/Slitherlink-Solver
[2] https://developers.google.com/optimization/install/java
[3] https://www.janko.at/Raetsel/Slitherlink/index-2.htm
[4] https://www.nikoli.co.jp/en/puzzles/slitherlink/

Correctness

Table 3.1 summarizes the number of instances solved by each solver within a time limit of 60 seconds. Across all completed runs, no incorrect solutions were observed.

Solver	Solved (%)	Correct	Incorrect	Unsolved	Timeout (60s)
CP-SV	66.98	1759	0	0	867
CP-Lazy	100.00	2626	0	0	0
CP-Normal	100.00	2626	0	0	0
DS	99.89	2623	0	1	2

Table 3.1: Correctness results over all benchmark instances.

CP-Lazy and CP-Normal solved all benchmark instances correctly within the imposed time limit. DS solved almost all instances, but exceeded the time limit on two instances and could not fully deduce one instance within the implemented 2-LoE search. CP-SV performed the worst, exceeding the time limit for many instances.

Three instances from the Tokoton collection were found to have multiple valid solutions. For these instances, different solvers produced different solutions. The alternative solutions produced within the time limit were manually checked and counted as correct.

Performance

Table 3.2 summarizes the runtime performance of the solvers over all correctly solved non-timeout instances. Runtime values are reported in seconds. The table includes the mean, median, 90th percentile, 99th percentile, and maximum observed runtime.

Solver	Solved	Mean	Median	P_{90}	P_{99}	Max
CP-SV	1759	2.379	0.019	3.985	47.643	59.167
CP-Lazy	2626	0.241	0.122	0.556	1.831	4.208
CP-Normal	2626	3.230	1.393	9.330	22.087	49.283
DS	2623	0.194	0.007	0.125	2.280	51.239

Table 3.2: Performance results over solved instances.

DS and CP-Lazy achieved the best overall runtime results. DS had the lowest median runtime, solving a large portion of the benchmark set within only a few milliseconds. CP-Lazy had a higher median runtime, but showed more stable behavior on difficult instances, as indicated by its lower maximum runtime. In contrast, CP-Normal solved all instances but required considerably more time on average. CP-SV solved fewer instances within the time limit, so its runtime statistics only describe the subset of instances it completed successfully.

Figure 3.1 provides a runtime profile of the solvers. The x-axis shows the runtime in seconds on a logarithmic scale, while the y-axis shows the cumulative number of solved instances.

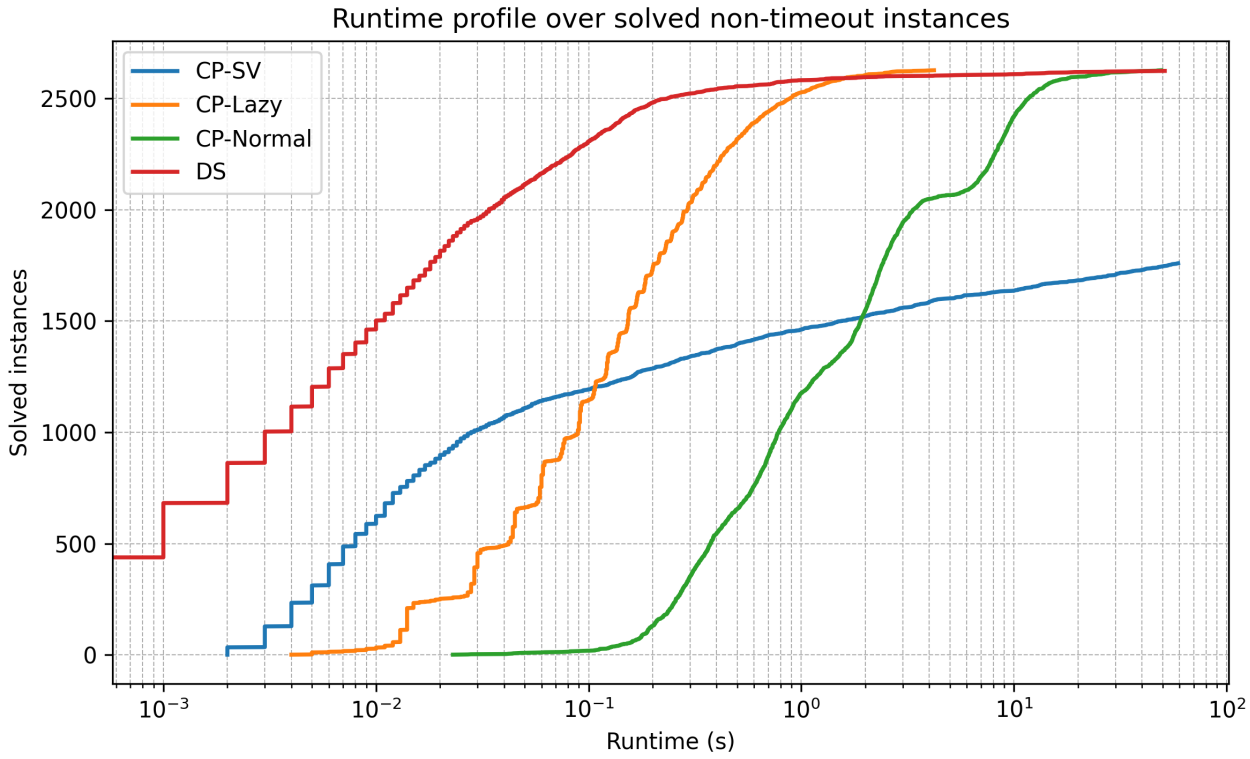


Figure 3.1: Cumulative number of solved instances vs the runtime performance using a logarithmic scale

The runtime profile confirms the behavior shown in Table 3.2. DS rises very steeply at the beginning, which shows that it solves many instances faster than the other approaches. However, its curve stretches far to the right near the end, indicating a small number of expensive outlier instances. CP-Lazy starts slower than DS, but its curve remains more compact and reaches all benchmark instances without large runtime spikes. CP-Normal also solves all instances, but its curve is shifted further to the right, showing consistently higher runtimes. CP-SV solves many easy instances quickly, but its curve stops earlier because many instances exceed the 60-second time limit.

Scalability

To evaluate scalability, the benchmark instances were grouped by puzzle size, according to the number of cells. Table 3.3 reports the median runtime for each solver and size group. Only correctly solved non-timeout instances are included.

Solver	≤ 100	101–400	401–900	> 900
CP-SV	0.006	0.029	0.336	–
CP-Lazy	0.038	0.137	0.547	2.018
CP-Normal	0.290	1.526	9.000	27.750
DS	< 0.001	0.007	0.095	0.759

Table 3.3: Median runtime (s) grouped by puzzle size.

The median runtimes show that DS has the lowest typical runtime in every size group. CP-Lazy also scales well, with a moderate increase in runtime as puzzle size grows. CP-Normal shows a stronger increase, especially for instances with more than 400 cells. CP-SV has no value for the largest size group because it did not solve any instance in this group within the time limit. Figure 3.2 shows the individual runtimes and median trends for each solver separately.

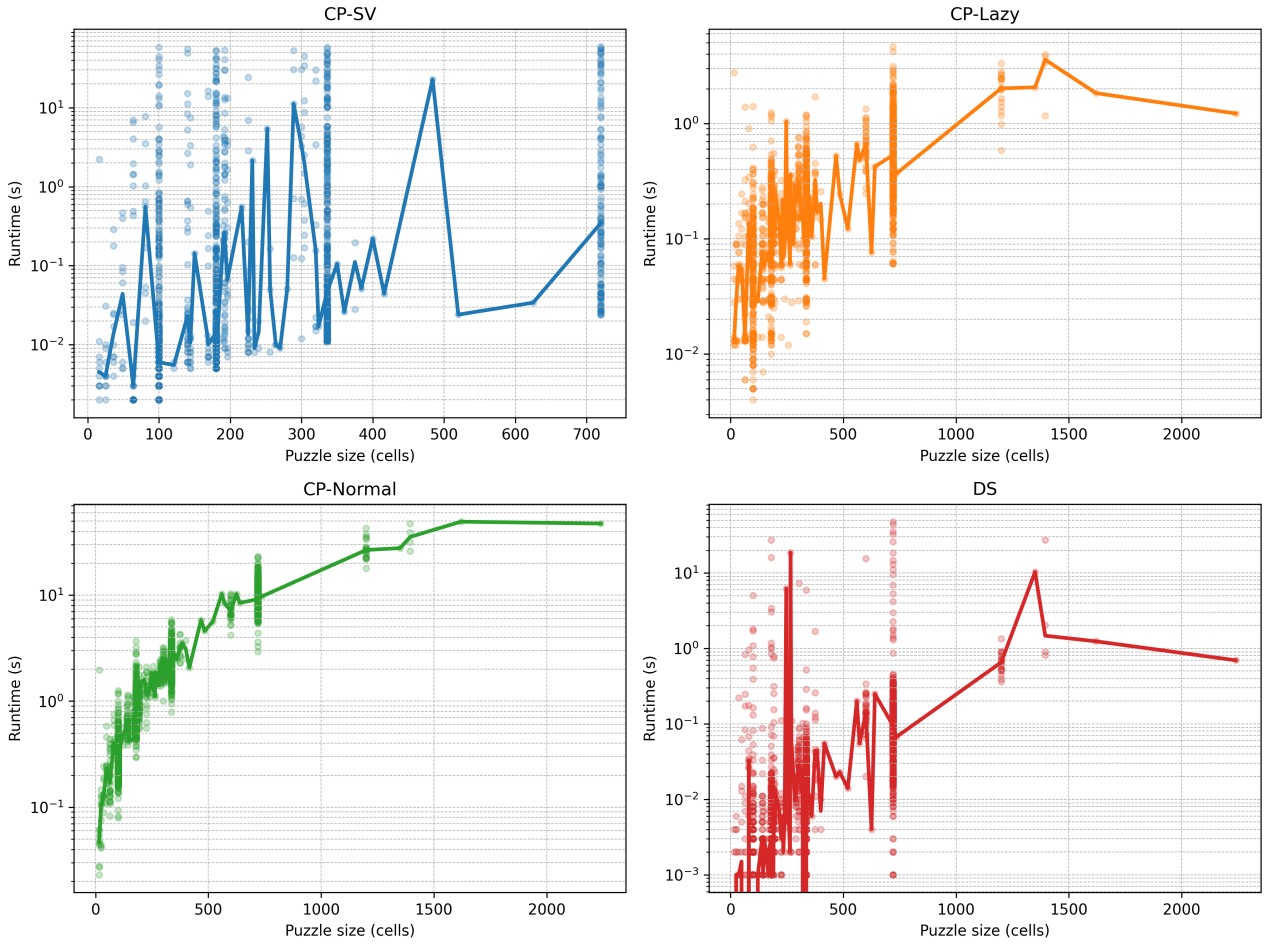


Figure 3.2: Runtime (logarithmic) scalability by puzzle size for each solver. Points represent individual solved non-timeout instances, while the lines show the median runtime trend.

The plots show the same overall trend as the table, but they also reveal the variance within each size group. In particular, DS and CP-SV show several outliers, indicating that puzzles with the same number of cells can still have very different runtimes. CP-Lazy and CP-Normal show more regular behavior within each size group, although CP-Normal remains consistently slower than CP-Lazy.

Deductive Search and Difficulty

Figure 3.3 shows the maximum depth of reasoning required for solving different instances from each dataset. Many instances required 1-LoE reasoning, where the solver temporarily assigns edges in order to obtain new deductions. A smaller number of instances could already be solved at 0-LoE using only direct deduction rules, while some instances required deeper reasoning at 2-LoE.

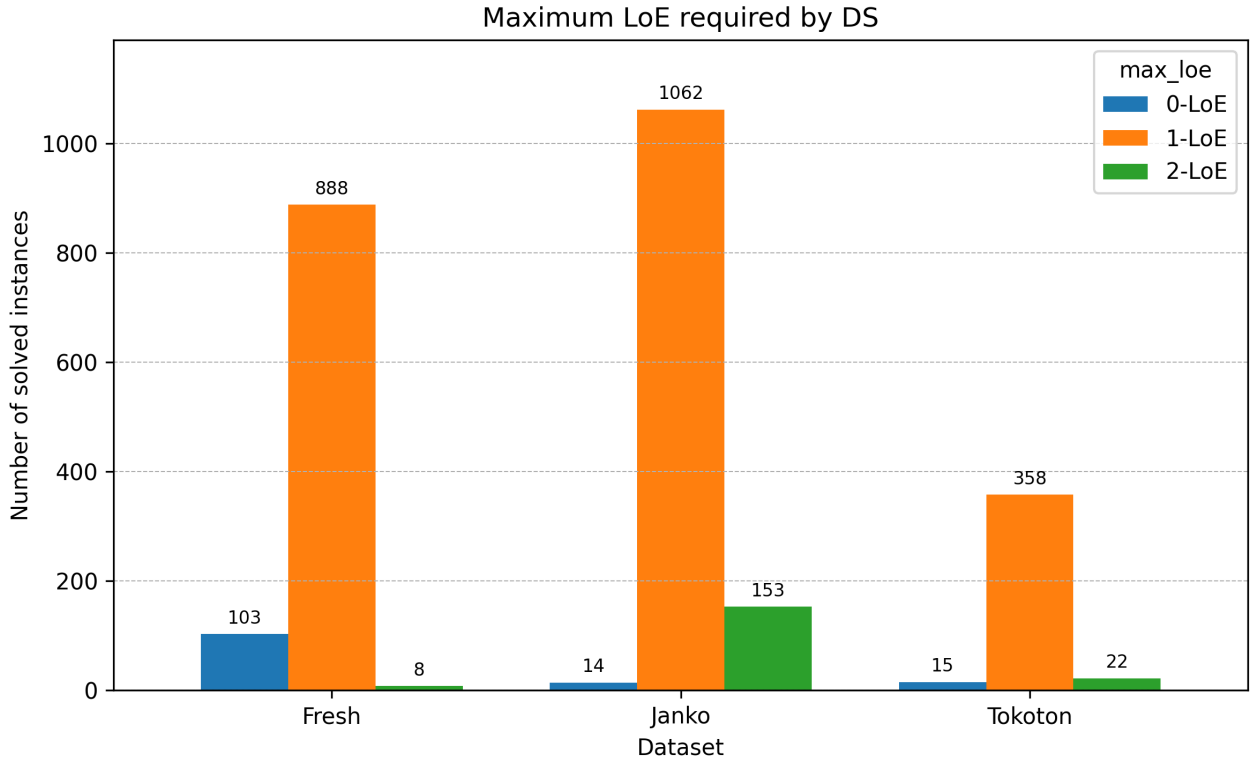


Figure 3.3: Number of solved instances per LoE for each dataset

In addition to the maximum LoE, the internal behavior of DS can be examined through the number of successful shaving and agreement operations. Simplification forms the basis of the solver and is responsible for many direct deductions. However, shaving and agreement are especially relevant for measuring hypothetical reasoning, since both rely on testing assumptions to derive new edge assignments. Figure 3.4 compares the number of successful shaving and agreement operations over all solved instances.

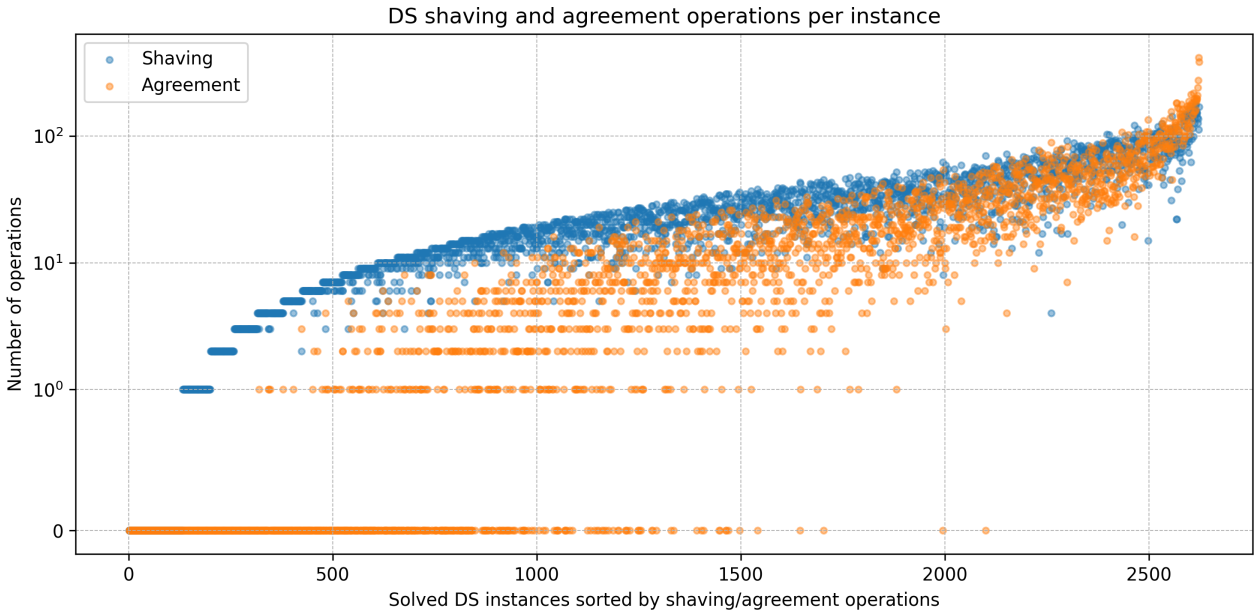


Figure 3.4: Number of shaving and agreement operations (logarithmic) for all instances. Instances are sorted by the combined amount of deductions of both operations.

The plot shows that shaving is used more consistently than agreement. The number of shaving operations increases relatively smoothly, while many instances require no agreement operations at all. The long tail on the right side of the plot indicates that a smaller number of instances require substantially more hypothetical reasoning. Overall, this supports the observation that DS is fast for many instances, but that some instances form bottlenecks because they require a large number of shaving and agreement operations.

As described in the methodology, DS can be used to estimate the difficulty of solved instances based on the amount and depth of reasoning required by the solver. However, the computed difficulty values are not directly comparable to the official difficulty labels, since the datasets use different scales. Therefore, the evaluation focuses on whether higher official difficulty labels tend to correspond to higher computed DS difficulty values. Table 3.4 shows the Spearman rank correlation between the official difficulty and the computed DS difficulty for each dataset, which measures whether both variables follow a similar ordering.

Dataset	Instances	Official range	DS range	Spearman ρ
Janko	1219	0–9	0.500–4.161	0.323
Fresh	999	0–10	0.500–1.678	0.715
Tokoton	395	1–32	0.500–1.689	0.473

Table 3.4: Spearman correlation between official difficulty and computed DS difficulty.

Figure 3.5 visualizes the relationship using boxplots. For Tokoton, instances are grouped into intervals of three due to the wider scale.

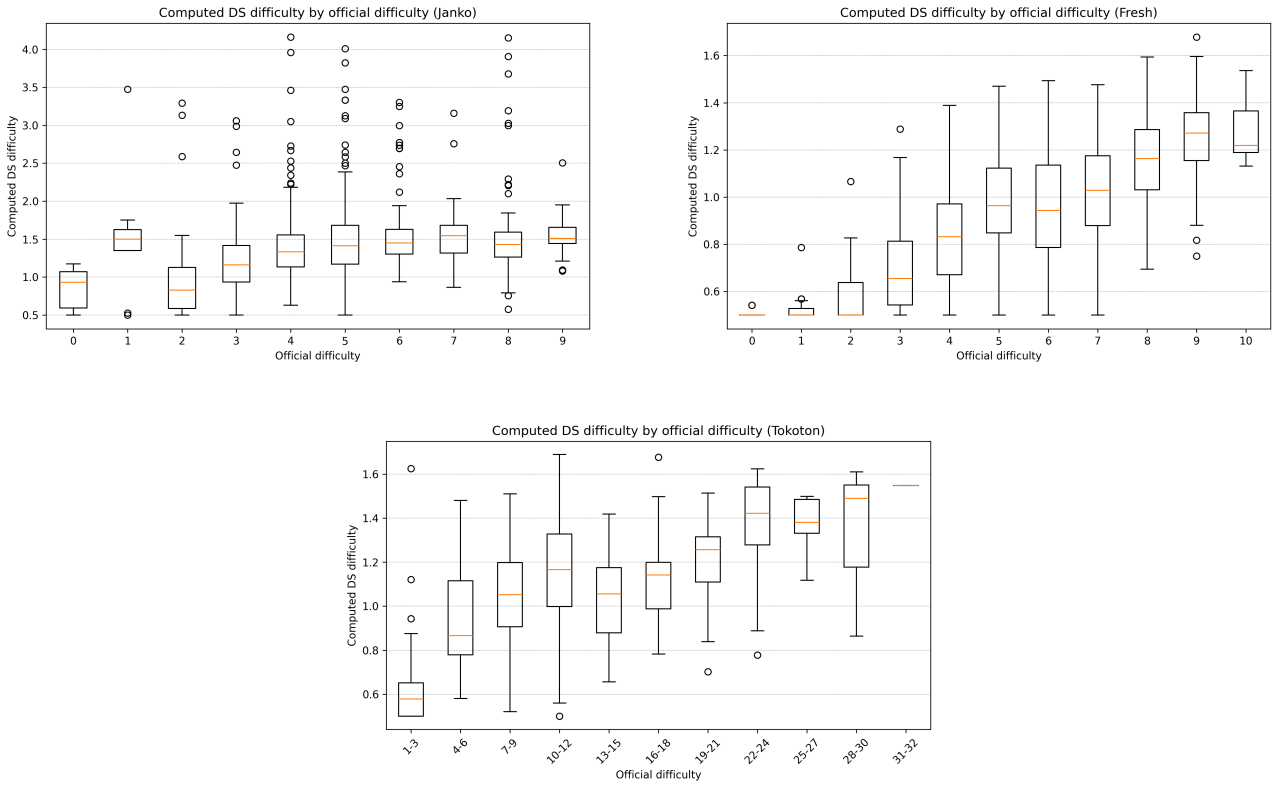


Figure 3.5: Computed DS difficulty grouped by official difficulty. Boxes represent 25-75% of the Interquartile range (IQR). Orange lines represent the median. Whiskers represent the largest to smallest values within 1.5 × IQR. Dots represent outliers.

Fresh has the clearest relationship, as the median computed DS difficulty increases with the official difficulty for most labels, which is consistent with the Spearman correlation of $\rho = 0.715$. Tokoton shows a moderate positive relationship with $\rho = 0.473$, and the boxplots also show an increasing trend across the grouped difficulty intervals, although with some variation in its groups. Janko has the weakest relationship with $\rho = 0.323$. Its boxplots are noisier and contain several outliers, indicating that puzzles with the same official difficulty can require very different amounts of DS reasoning effort.

4 Discussion

This chapter serves the purpose of interpreting results and drawing meaningful conclusions from them, highlighting important differences and addressing important limitations.

Correctness

The correctness results (Table 3.1) show that all implemented solvers are capable of producing valid Slitherlink solutions. No incorrect solutions were discovered when a solver returned a solution within the time limit. However, the solvers differ strongly in their ability to find a solution within the imposed limit. CP-Lazy and CP-Normal solved all benchmark instances, while CP-SV produced many timeouts. DS solved almost all instances, with two timeouts and one instance that could not be deduced using the implemented 2-LoE search.

The results show that the implementation of an efficient global single-loop condition is essential for a complete solver. CP-SV enforces the local clue and vertex constraints during the CP-SAT search, but checks the single-loop condition only afterwards. This allows many locally valid but globally invalid candidates with many different combinations of loops to be generated. In contrast, CP-Lazy and CP-Normal both incorporate the single-loop requirement into the solving process, either incrementally through lazy cuts or directly through the normal single-loop encoding. This explains why both approaches are much more reliable on the benchmark set.

For DS, the unsolved instance highlights a limitation of the implemented reasoning depth. The solver is restricted to 2-LoE in order to align with the mental limitation of human reasoning. Extending the solver to 3-LoE or beyond could allow it to solve additional instances, but this would also increase the search effort and move the solver further away from the intended cognitive restriction.

In instances with multiple valid solutions, every solver is capable of returning one of the valid solutions, and can be extended to find the other solutions by forbidding the previous ones.

Runtime and Performance

The runtime results (Table 3.2) show that DS competes strongly with the CP-based solvers on many instances. It has very low mean, median, and P_{90} runtimes and solves many puzzles almost immediately. This is expected for instances where the implemented deduction rules are sufficient or where only limited hypothetical reasoning is needed. Unlike the CP-SAT models, DS does not need to construct a large constraint model before solving. It can immediately start applying deduction rules and assigning forced edge values. This can also be obeserved in Figure 3.1, where DS can solve many instances in the time required for CP solvers to build the model.

However, DS is less robust than CP-Lazy. The runtime profile and the P_{99} and maximum runtime values show that DS has expensive outliers. These outliers occur when the implemented deduction rules are not sufficient to make fast progress and the solver has to perform many shaving and agreement operations. This also explains why puzzles of similar size can have very different runtimes for DS. The runtime is not determined only by the number of cells, but also by the clue structure and by the kind of reasoning required.

CP-Lazy provides the best overall balance between completeness, runtime, and scalability. It solves all instances and remains relatively stable even on larger ones. The lazy cut approach is effective because it starts with a compact model and only adds additional constraints when invalid loop structures are found. This avoids the generation of numerous candidate solutions like CP-SV, while also keeping the model shorter and simpler than CP-Normal.

CP-SV performs well on some easier instances because its model is simple and contains fewer variables and constraints. However, this advantage disappears when many locally valid candidates exist. Since the single-loop condition is only checked after candidate generation, CP-SV may enumerate a large number of invalid solutions before finding a valid one. This explains the high number of timeouts and shows that separating local feasibility from global loop verification is not scalable.

CP-Normal is the most direct complete CP-SAT model, since it encodes the single-loop condition inside the model itself. This makes it reliable, as shown by its ability to solve all instances. At the same time, the model contains many additional variables and constraints, which leads to higher runtime. Therefore, CP-Normal is robust but less efficient than CP-Lazy in this evaluation.

Scalability

The scalability results (Table 3.3, Figure 3.2) show that puzzle size affects runtime, but it is not the only relevant factor. Larger puzzles generally require more time, especially for CP-Normal and CP-Lazy. However, the scatter plots also show high variance among instances with the same number of cells. This indicates that the distribution and sparsity of clues along with the number of possible loop structures and the required reasoning depth also play an important role.

This is especially visible for DS and CP-SV. For DS, some large puzzles are solved quickly if the deduction rules apply effectively, while some smaller or medium-sized puzzles become difficult if they require many hypothetical operations. For CP-SV, the number of locally valid candidates can vary strongly even for puzzles of similar size, leading to large differences in runtime. CP-Lazy shows the most stable scaling behavior among the CP-SAT approaches, while CP-Normal scales less favorably because of its larger single-loop encoding.

Deductive Search and Explainability

A major advantage of DS is its interpretability. While CP-SAT solvers provide solver logs and parameters, they do not naturally explain the solution as a sequence of logical steps. DS, on the other hand, can trace every simplification step, hypothetical assignment, contradiction, and agreement. This

makes it useful not only as a solver, but also as a tool for analyzing how an instance can be solved by a sequence of logical deductions.

The DS results (Figure 3.4) show that shaving is used more consistently than agreement, while agreement is applied more selectively. The long tail in the number of shaving and agreement operations indicates that a small number of instances require substantially more hypothetical reasoning. This also supports the observation made before, where DS has very low runtime on many instances but still produces outliers on others.

One limitation of the implemented DS strategy is that it uses an exhaustive ordered search over undecided edges when applying hypothetical reasoning in deduce passes. Human solvers usually do not test edges uniformly. Instead, they tend to choose regions where a hypothesis is likely to produce useful consequences, for example around constrained clues or partially determined loop segments. The current strategy may therefore spend time testing assumptions that do not lead to useful deductions. A possible improvement would be to use heuristics for selecting promising edges before applying shaving or agreement.

Difficulty Assessment

The computed DS difficulty should be interpreted carefully. It is useful as a solver-specific measure of deductive effort, but it should not be understood as an objective measure of human puzzle difficulty. It only measures how difficult an instance is for the particular method and set of rules defined in this work, based on the number of deductions and the maximum LoE required.

The correlation results support this interpretation. The computed difficulty correlates strongly with the Fresh difficulty labels, moderately with Tokoton, and only weakly with Janko. This suggests that the metric captures some aspects of puzzle difficulty, but not all. In particular, the Janko labels are reported to be based on solving time. Solving time depends on many factors that are not fully captured by the DS metric, such as visual search, pattern recognition and strategies, and expertise of the human solvers.

Another limitation is that the metric treats all simplification deductions similarly. In practice, different simplification rules may require different levels of cognitive effort. A simple clue rule may be much easier for a human solver to recognize than a more complex structural pattern, even if both are counted similarly by the metric. The implemented rule set also contains only a subset of possible Slitherlink deduction patterns. More advanced patterns used by experienced human players are not represented, and adding them could change the computed difficulty values. Furthermore, the metric counts successful deductions, but it does not fully capture unsuccessful hypotheses that may be tested before a useful deduction is found. Thus, two instances may receive similar computed difficulty values even if one required substantially more internal search.

5 Conclusion

Contributions: This thesis investigated different approaches for solving Slitherlink puzzles. Slitherlink was formalized as a constraint satisfaction problem and modelled using CP-SAT. Three CP-SAT approaches were implemented: CP-SV verifies the single-loop condition after generating locally feasible candidates, CP-Lazy adds cuts to exclude invalid loop structures, and CP-Normal encodes the single-loop condition directly in the model. In addition, Browne's Deductive Search framework [1] was adapted to Slitherlink using human-inspired deduction rules and hypothetical reasoning at different LoEs.

Main Findings and Implications: The experiments showed that CP-Lazy was the most effective complete solver in this evaluation. It solved all benchmark instances, achieved low runtimes, and scaled well across different puzzle sizes. CP-Normal also solved all instances, but its larger model led to higher runtimes. CP-SV was the simplest approach and worked well on some easier instances, but it did not scale reliably because many locally valid candidates had to be checked before finding a valid single loop.
DS solved almost all instances and achieved very low runtimes on many puzzles. Its main advantage is explainability, since it can trace the solving process through explicit deduction rules, hypothetical assignments, contradictions, and agreements. However, DS is limited by its rule set and bounded reasoning depth. The computed difficulty metric should therefore be interpreted as a solver-specific measure of deductive effort, not as an objective measure of human puzzle difficulty.

Future Work: Future work could extend the DS solver with additional deduction rules and better heuristics for selecting promising edges during hypothetical reasoning. The difficulty metric could also be improved by comparing it with empirical human solving data and by assigning different cognitive weights to different deduction rules.

Main Takeaway: Overall, CP-Lazy provides the best balance between completeness, runtime, and scalability among the implemented solvers. DS offers a complementary human-like and interpretable approach, while CP-SV and CP-Normal illustrate the trade-off between model simplicity and complete global constraint encoding.

List of Figures

1.1 Slitherlink instance and solution 6
1.2 Triangular and hexagonal variants of Slitherlink 8

2.1 Candidate solutions of an instance using local constraints only 12
2.2 Candidate solutions using the lazy constraints method 13
2.3 Edge case for the lazy constraints method 13
2.4 Single loop constraint 15
2.5 Example of Clue Rule 16
2.6 Example of Vertex Rule 17
2.7 Example of Corners of Three Rule 18
2.8 Example of Corners of Two Rule 19
2.9 Example of Corners of One Rule 20
2.10 Example of Two at Corners Rule 20
2.11 Example of Neighbors of Three Rule 21
2.12 Example of 1x1 Rule 21
2.13 Edge cases for Neighbors of Three and 1x1 rules 22
2.14 Simplification example 23
2.15 Shaving example 24
2.16 Agreement example 25

3.1 Comparison of performance 32
3.2 Comparison of scalability 33
3.3 Solved instances per LoE 34
3.4 Number of deductions from shaving and agreement 35
3.5 Computed difficulty by official difficulty 36

List of Tables

3.1 Correctness results over all benchmark instances. 31
3.2 Performance results over solved instances. 31
3.3 Median runtime (s) grouped by puzzle size. 33
3.4 Spearman correlation between computed and official difficulty 35

Statement on the Use of Generative AI Tools

ChatGPT (GPT-5.5 Thinking by OpenAI) was used in the scope of this work during the development process for generating a few initial code snippets, debugging, syntax support, and generating Python scripts for running and analyzing experiments.

Additionally, it was used during documentation and writing of the thesis for improving the style and professionalism of the text, finding linguistic mistakes, LaTeX syntax support, and an overall final review.

Bibliography

[1] Cameron Browne. Deductive search for logic puzzles. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[2] T Ishihama and T Kuno. A faster solution to the slitherlink puzzle using integer programming. *IPSJ Journal*, 54(8):2103–2108, 2013.

[3] Mohammad Sina Kiarostami, Mohammadreza Daneshvaramoli, Saleh Khalaj Monfared, Aku Visuri, Helia Karisani, Simo Hosio, Hamed Khashehchi, Ehsan Futuhi, Dara Rahmati, and Saeid Gorgin. On using monte-carlo tree search to solve puzzles. In *Proceedings of the 2021 7th International Conference on Computer Technology Applications*, pages 18–26, 2021.

[4] Jonas Kölker. Selected slither link variants are np-complete. *Journal of information processing*, 20(3):709–712, 2012.

[5] Tung-Ying Liu, I-Chen Wu, and Der-Johng Sun. Solving the slitherlink problem. In *2012 Conference on Technologies and Applications of Artificial Intelligence*, pages 284–289. IEEE, 2012.

[6] Tetsuo Miyauchi and Kiyofumi Tanaka. Solving slitherlink with fpga and smt solver. *Journal of Information Processing*, 28:959–969, 2020.

[7] Nagesha Shivappa. A survey on classic examples of constraint satisfaction problems with solutions and applications. In *2023 3rd International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, pages 1–7. IEEE, 2023.

[8] David Westreicher. Slitherlink reloaded. *Slitherlink Reloaded*, 16, 2011.

[9] Takayuki Yato. On the np-completeness of the slither link puzzle. *IPSJ SIGNotes ALgorithms*, 74:25–32, 2000.

[10] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.

[11] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5(20):176–213, 2012.