TECHNISCHE UNIVERSITÄT BERLIN

Abstraction of Puzzle Rules for a General Solver

Bachelor's Thesis

Lorenzo von Flocken (472574)

18. November 2025

Slightly revised for publishing, 16. December 2025

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

Supervisor: Noah Schlegel

Technische Universität Berlin
Faculty IV – Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology Group

Eigenständigkeitserklärung

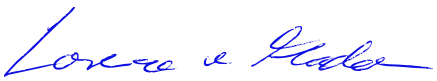
Hiermit versichere ich, dass ich die vorliegende Arbeit eigenständig ohne Hilfe Dritter und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Alle Stellen, die den benutzten Quellen und Hilfsmitteln unverändert oder sinngemäß entnommen sind, habe ich als solche kenntlich gemacht.

Sofern generative KI-Tools verwendet wurden, habe ich Produktnamen, Hersteller, die jeweils verwendete Softwareversion und die jeweiligen Einsatzzwecke (z. B. sprachliche Überprüfung und Verbesserung der Texte, systematische Recherche) benannt. Ich verantworte die Auswahl, die Übernahme und sämtliche Ergebnisse des von mir verwendeten KI-generierten Outputs vollumfänglich selbst.

Die Satzung zur Sicherung guter wissenschaftlicher Praxis an der TU Berlin vom 15. Februar 2023[1] habe ich zur Kenntnis genommen.

Ich erkläre weiterhin, dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Berlin, den 18. November 2025



[1] https://www.static.tu.berlin/fileadmin/www/10002457/K3-AMBl/Amtsblatt_2023/Amtliches_Mitteilungsblatt_Nr._16_vom_30.05.2023.pdf

Kurzfassung

Bisherige Ansätze zum Lösen von Puzzlen müssen für verschiedene Puzzletypen einzeln implementiert werden. In dieser Abschlussarbeit wird ein generalisierter Ansatz vorgestellt, welcher auf dem *Deductive Search*-Algorithmus beruht, und unterschiedliche Puzzles auf Grundlage einer Definition ihrer Spielregeln als Zusammensetzung von Constraints löst. Der generalisierte Ansatz des Algorithmus ermöglicht es, Constraints für verschiedene Puzzletypen wiederzuverwenden, was eine einfachere Implementierung von Lösern erlaubt. Zusätzlich wird eine konkrete Menge an Constraints vorgeschlagen, welche anhand ihrer Eignung, eine sinnvolle Anzahl verschiedener Puzzletypen abzubilden, bewertet werden. Die Gesamtanwendbarkeit des Algorithmus wird anhand seiner Laufzeit für das Lösen ausgewählter repräsentativer Puzzles bewertet.

Abstract

Current approaches to solving Logic Puzzles have to be implemented for each puzzle type separately. A generalized approach based on the *Deductive Search* algorithm is proposed, which can solve puzzles based on a definition of the puzzle's rules as a composition of constraints. The generalized nature of the algorithm allows constraints to be reused for many different puzzle types, resulting in simpler implementation of solvers. A set of constraints is proposed, which are evaluated based on their ability to effectively model a reasonable number of different puzzle types. The overall practicality of the algorithm is evaluated based on performance of solving representative puzzles.

Contents

List of Figures

1 Introduction

Pencil puzzles are culture-agnostic deduction puzzles that are designed to be solved with pencil on paper [1]. An especially interesting subset of pencil puzzles for generalized approaches are those played on rectangular grids. Examples include *Sudoku* and *Slitherlink* (Figure 1), which were popularized by Japanese publisher Nikoli [2], but other much older examples exist, most notably the *Magic Square* [3].

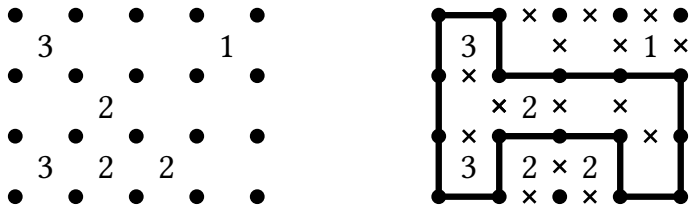


Figure 1. A Slitherlink challenge (left) and its solution (right). Adapted from [2, Fig. 1].
The player connects dots to form a single closed non-intersecting loop, such that the numbers inside the cells indicate the number of lines surrounding the cell [4].

One popular approach to solving pencil puzzles is modeling them as constraint satisfaction problems [5], and solving those using methods like reduction to SAT problems [6], or genetic algorithms [5], [7].

C. Browne [2] presents the *Deductive Search* algorithm to solve, among others, pencil puzzles, along with calculating a heuristic for their difficulty. While other approaches to difficulty metrics use hard-coded strategies [8], Deductive Search derives difficulty scores using only the constraints. C. Browne [2] defines some generalized constraints that are useful for modeling a variety of different puzzles, however their puzzle definitions also include some specialized constraints that are specific to a single puzzle type. In Section 2.1, some additional generalized constraints are proposed, to allow for puzzle definitions consisting entirely of general constraints, making it easier to adapt the solver to additional puzzle types.

1.1 Background

Deductive Search is a breadth-first search approach for solving logic puzzles [2]. The algorithm only makes moves that are proven to be correct and is designed to mimic how a human player would solve a given puzzle.

By limiting the search depth, Deductive Search can be used to analyze the solvability of puzzle instances (some instances might be solvable through brute force, but practically impossible to solve by human players) [2, sec. I A]. The number and kind of operations Deductive Search has to apply to solve an instance can serve as a heuristic for determining the difficulty of a puzzle instance. However, the

quality of such analyses depends largely on the choice of constraints and their usefulness has to be evaluated separately for each puzzle.

Deductive Search models puzzles as constraint satisfaction problems. The game state is represented by integer *variables* X_i, each with a corresponding set D_i of potential values. D_i is referred to as the *domain* of X_i. The puzzle is solved by iteratively removing values from the domains until each domain contains exactly one value. This remaining value v is the value of the corresponding variable X_i. The act of setting D_i to $\{v\}$ is called *instantiating* X_i *to* v. If there are multiple possible solutions, or a puzzle is not solvable by deduction, Deductive Search terminates in a state containing at least one domain with more than one value.

Constraints C_i are defined over a set of variables called C_i's *associated variables*. Every constraint implements a *propagate* method, which is called whenever a variable associated with the constraint is updated, i.e. when a value is removed from that variable's domain. This method eliminates all values v from associated domains D_i that, if X_i was set to v, would violate the constraint. A constraint's implementation ensures that states containing contradictions inevitably lead to empty domains.

To solve a given puzzle instance, the hints are first translated to a set of constraints derived from the puzzle's rules. Translation has to be implemented separately for each puzzle type. Because translation is only necessary once per run, it is allowed to be a relatively expensive operation.

Three methods are used to propagate variable updates:

Simplification For every updated variable X_i, the update is propagated over the constraints. Values in the domains of other variables are removed if, together with X_i, they would violate a constraint. This is achieved by tracking all variables whose domains have changed in a queue S^*, and successively applying the *propagate* method of every constraint associated with the variables in S^*, until S^* is empty.

Shaving Every unresolved variable X_i is successively instantiated to every remaining potential value $v \in D_i$ and the *Simplification* method is applied. If a contradiction arises (i.e. any domain becomes empty after the Simplification), v is eliminated from D_i. Figure 2 shows a Shaving operation on a game of Slitherlink.

Agreement If some value $w \in D_j$ is *always* eliminated from D_j during Simplification after instantiating X_i to *any* potential value in D_i, then X_j can not be equal to w and w is eliminated from D_j.[2] Figure 3 shows an Agreement operation on a game of Slitherlink.

These three operations are combined into a *Deduce* step of depth d that iterates over all unresolved variables. For each variable X_i, a copy S'_v of the state is made for every potential value $v \in D_i$. X_i is instantiated to v in S'_v, and Simplification and Shaving are applied to S'_v. If no contradiction arises, no values are shaved from D_i, and a recursive Deduce step of depth $d - 1$ is applied to S'_v. Finally, an Agreement step is applied to the result of that recursive Deduce operation.

Deductive Search starts with a Deduce step of depth 0: a single Simplification operation. Deduce steps of depth 1 are then applied, until no additional variable updates are produced. If Deduce steps of depth 1 no longer produce any updates and the puzzle is not solved, a single Deduce step of depth 2 is applied, followed by the previous loop of steps of depth 1.

According to C. Browne [2], findings in psycholinguistics suggest 2 *levels of embedding* to be a natural limit for human players. They suggest that limiting the search depth to 2 results in Deductive Search finding solutions to puzzles that are solvable by human players, and finding no solutions to puzzles

[2] This explanation differs from Equation (3) in [2], which appears to be wrong. See Section 4 for further discussion.

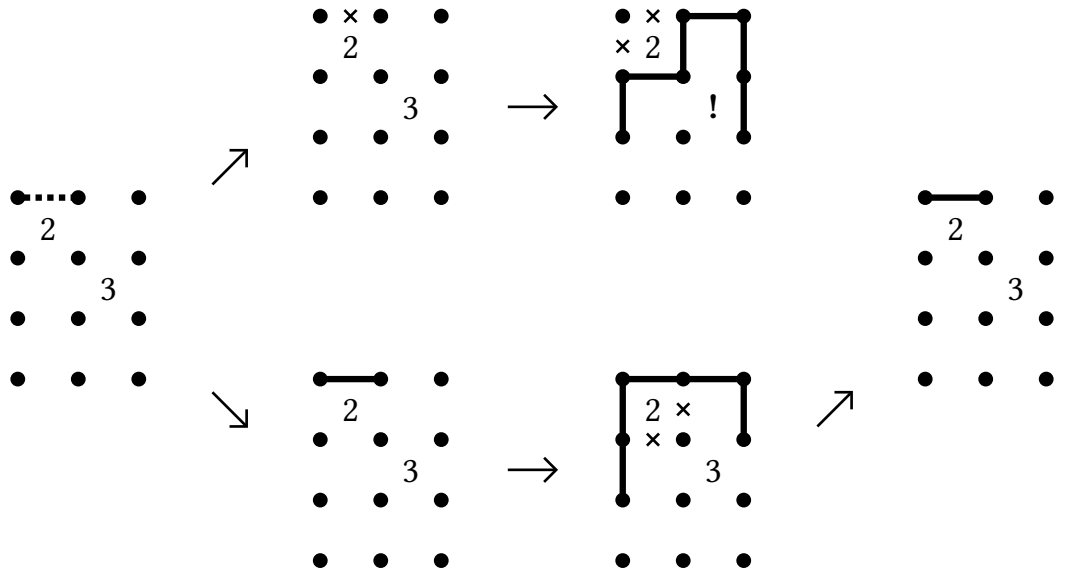


Figure 2. A Shaving operation. Adapted from [2, Fig. 2]

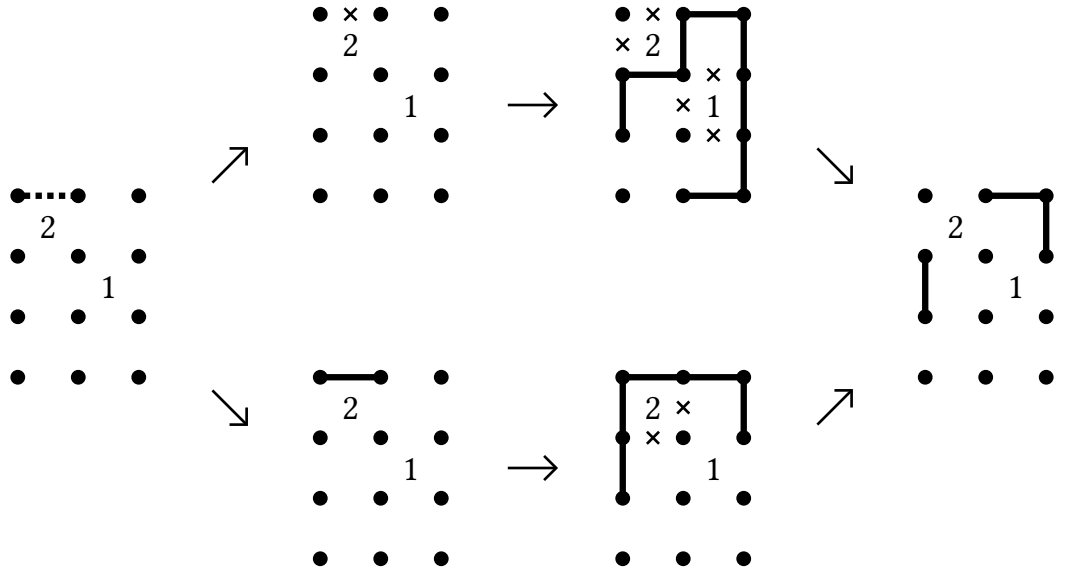


Figure 3. An Agreement operation. Adapted from [2, Fig. 3]

that are too difficult for human players to solve. This property depends on the choice of constraints and their ability to model human reasoning. As discussed in Section 2.1, this is not necessarily the case with some of the constraints explored here, as they need at least one level of embedding to function properly. Naturally, a hard limit of two levels of embedding does not exist for computers, and the algorithm works with arbitrarily large limits, albeit with significantly reduced performance.

2 Methodology

A lot of Pencil Puzzles use only a small set of similar elements: The cells making up the grid may be filled and contain shapes (such as circles, letters, or arrows) and numbers. Figure 4 shows some examples of these elements. The grid cells may be connected with lines. Alternatively, the lines making up the grid may be thickened. A group of adjacent cells surrounded by a single closed loop of thickened lines is referred to as a *region*.

These elements are sometimes drawn differently, depending on puzzle. For example, instances of *Slitherlink* are usually drawn as a grid of dots (see Figure 1), which are then connected horizontally or vertically by the player. For modeling purposes, this is functionally equivalent to the thickened grid lines found, for example, in *Sudoku*. While this means that the visualization has to be implemented separately for each puzzle, a *general solver* can use a generalized data structure to represent both variants.

In order to model puzzles as constraint satisfaction problems, these elements are represented as integer variables. Cell backgrounds and symbols (collectively referred to as *shapes*) are assigned values on a per-puzzle basis. For example, an empty cell can be represented as 0, a filled cell as 1. This *nominal* value is stored in one variable per cell. If cells are allowed to contain numbers, one additional *ordinal* variable is added per cell. A cell containing *no* number is then represented using a special value, such as -1. Letters are modeled as numbers, by mapping each letter to its position in the alphabet.

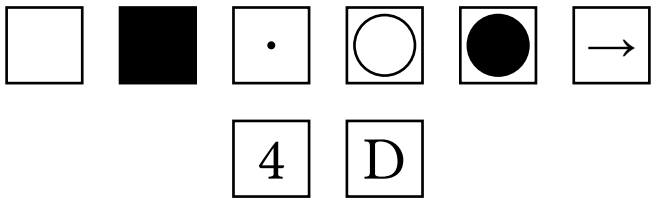


Figure 4. Some elements. The top row consists of *nominal* elements which are arbitrarily mapped to integers. Arithmetic operations can be meaningfully applied to the *ordinal* elements in the bottom row.

Additionally, some elements – namely cell-connecting lines (Figure 5), grid lines (Figure 6), regions (Figure 6), and polyominoes made up of adjacent grid cells with the same color or shape (Figure 7) – can be represented as grid graphs. An $n \times m$ grid graph (i.e. a graph with $n \cdot m$ vertices) can be modeled as $n \cdot (m-1) + m \cdot (n-1)$ variables with domains $\{0, 1\}$.

A puzzle definition has to define what elements the challenge is made up of, what elements are allowed to be used to solve the puzzle, and a function to generate a list of constraints from the challenge.

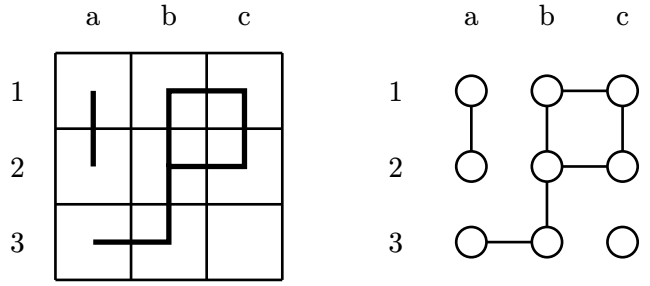


Figure 5. A grid with lines between cell centers (left).
The lines connecting the cells form a grid graph (right).

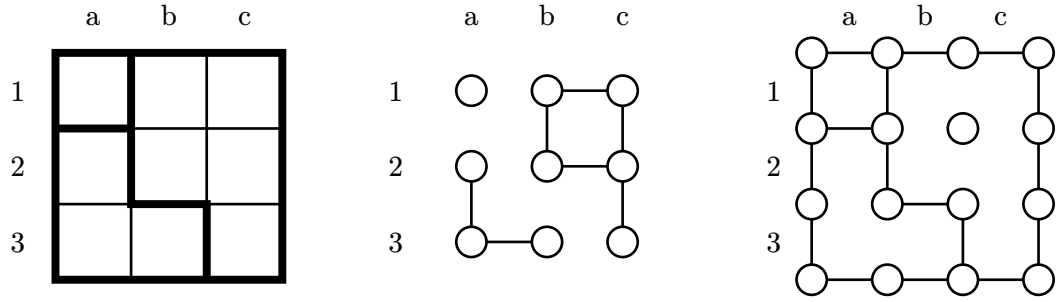


Figure 6. A grid is divided into regions (left). Regions can be represented as a grid graph of connected cells (center), or of grid lines (right).

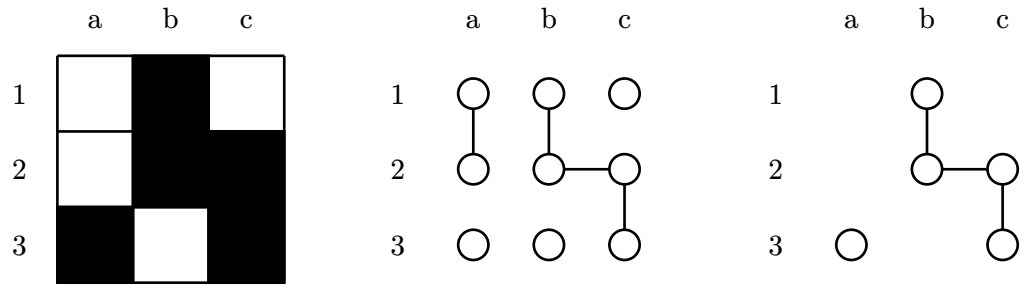


Figure 7. Some cells are filled in, creating polyominoes (left). A corresponding grid graph can be computed (center, right). The graph shown in the middle incorporates both black and white polyominoes, while the graph on the right only considers polyominoes made up of black cells.

Connected Structures

Some puzzles, such as *Nurikabe* (Figure 8), require connected structures similar to the closed loop in Slitherlink, but made up of cells. This can be modeled using the same constraints with a polyomino graph instead of Slitherlink's grid line graph. In the case of Nurikabe, a black polyomino graph is constructed from the current game state. Vertices corresponding to cells that are known to be white (including hint cells) are removed from the graph. Edges are then created between vertices corresponding to (horizontally or vertically) adjacent filled cells (see Figure 9).

The cell marked **?** in Figure 9 is filled because the corresponding vertex has a neighbor.

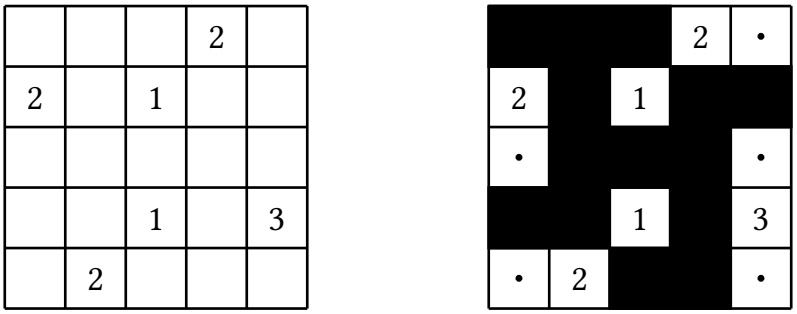


Figure 8. A Nurikabe challenge (left) and its solution (right). The player fills in empty cells to form a single connected wall, such that no black 2×2 squares exist and each white polyomino contains exactly one number corresponding to the number of white cells in that polyomino [9].

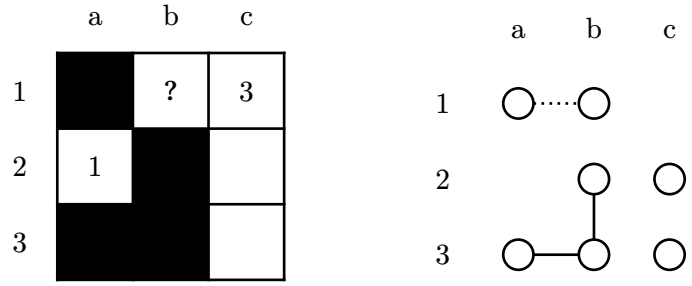


Figure 9. Propagation of the Graph Connected constraint in a game of Nurikabe.
For the graph (shown right) to stay connected, the vertex a1 has to be connected to its only remaining neighbor b1, so the dotted edge is added.

2.1 Constraints

Puzzle rules can generally be classified as either *local* or *global*. Local rules usually apply to each hint element separately (e.g. Slitherlink's *numbers indicate how many lines surround a cell* rule [4]) and are translated into many constraints, each of which only applies to a small subset of variables. Global rules apply to the solution as a whole (e.g. Slitherlink's *lines form a single closed loop* rule), and therefore have to consider all variables, and so are translated into a single constraint.

Global rules are usually not satisfied until the puzzle is solved completely. They are however often required for making local decisions. Global constraints detect moves that would result in a violation of the global rule once all variables are resolved. If such a violation is detected, *any* single domain is set to $\{\}$, resulting in an invalid state and the offending value being shaved from its domain. This mechanism is referred to as *panicking*. One downside of relying on panicking is that global constraints can never make proactive moves, and all their decisions are made through shaving, which requires one level of embedding.

A general solver consists of an implementation of the Deductive Search algorithm, along with a set of generalized constraints that can be used to model different puzzles. The set of constraints $\mathcal{C}$ is considered optimal if it is minimal and the rules of a reasonable number of Nikoli puzzles can be completely implemented using these constraints. $\mathcal{C}$ is *not* considered optimal if any $C \in \mathcal{C}$ is only useful for implementing a single puzzle.

The following section lists all constraints of the set $\mathcal{C}^*$ explored in this thesis. The All Different, Even, Less Than, Sum Equals, and Relation constraints are adapted from [2, sec. II E].

All Different ($\mathcal{X}$) All $X_i \in \mathcal{X}$ have different values.

For each *instantiated* variable $X_i \in \mathcal{X}$, X_i's value is removed from all other domains:

$$D_{j \neq i} \leftarrow D_j \setminus \{X_i\}.$$

Even ($\mathcal{X}$) The sum of all $X_i \in \mathcal{X}$ is even.[3]

The residue class of each domain D_i of $X_i \in \mathcal{X}$ is computed as

$$\operatorname{res}(D_i) := \{n \bmod 2 \mid n \in D_i\},$$

assigning each domain to either $\{0\}$, $\{1\}$, or $\{0, 1\}$.

If there is more than one domain belonging to residue class $\{0, 1\}$, no values can be eliminated at all, because there are multiple possible variable assignments with an even sum, and every value from every domain appears in at least one possible assignment.

Domains D_i with $\operatorname{res}(D_i) = \{0\}$ are disregarded, because its value can not affect the sum's parity, and therefore no decision can be made regarding D_i's value. Likewise, pairs of domains D_1, D_2 are disregarded if $\operatorname{res}(D_1) = \operatorname{res}(D_2) = \{1\}$, because their combined value can not affect the sum's parity. If only one domain D_j belonging to class $\{0, 1\}$ remains after elimination, all odd values are eliminated from D_j, because instantiating X_j to one of them would lead to an odd sum. If, in addition to D_j, a second domain belonging to class $\{1\}$ remains, all *even* values are eliminated from D_j.

Less Than ($\mathcal{X}$, Σ) The sum of all $X_i \in \mathcal{X}$ is less than Σ.

The lower bound for the sum of all associated variables is computed as

$$\Sigma_{\min} := \sum_{X_i \in \mathcal{X}} \min D_i.$$

All values $v \in D_j$ are removed if $|D_j| > 1$ and $\Sigma_{\min} - \min D_j + v$ is larger than Σ.

Sum Equals ($\mathcal{X}$, Σ) The sum of all $X_i \in \mathcal{X}$ is equal to Σ.

A lower and upper bound for the sum of all associated variables are computed as

$$\Sigma_{\min} := \sum_{X_i \in C} \min D_i, \qquad \Sigma_{\max} := \sum_{X_i \in C} \max D_i.$$

All values $v \in D_j$ are removed if $\Sigma_{\min} - \min D_j + v$ is larger or $\Sigma_{\max} - \max D_j + v$ is smaller than Σ.

Relation (X_i, X_j, v, w) $X_i = v \Rightarrow X_j = w$.

If $X_i = v$, then X_j is set to w.

[3] [2, sec. II E] defines the Even constraint as "all $X_i \in C$ have even values", though this is likely a mistake, as the puzzle definitions in [2, sec. III] would not work with this definition.

Connected (G, u, v) Two vertices u and v in grid graph G remain connected (a path between them exists).

A grid graph G^* containing all edges *and all potential edges* in G is constructed. Whenever edges are removed from G^*, a breadth-first search is performed to test if u and v are still (potentially) connected. If no path between u and v is found, the constraint panics.

Disconnected (G, u, v) Two vertices u and v in G are not connected (no path between them exists).

The connected components of G are modeled as an equivalence relation. Whenever an edge is added, the equivalence classes of its two endpoints are joined. If u and v end up in the same equivalence class, the constraint panics.

Graph Connected (G) The graph G remains connected.

A grid graph G^* containing all *potential* edges in G is constructed. The connected components of G^* are modeled as an equivalence relation. If G^* consists of more than one component, the constraint panics.

Tree (G) The graph G contains no loops.

The components of connected edges in G are modeled as an equivalence relation. Whenever an edge is added, the equivalence classes of the two neighboring edges are joined. If an edge is inserted that connects two vertices that already belong to the same equivalence class, that edge is closing a loop, and the constraint panics.

Closed Loop (G) The edges in G form a single, non-intersecting closed loop.

This constraint implies a Less Than (3) and an Even constraint per grid point.

The components of connected edges in G are modeled as an equivalence relation. Whenever an edge is added, the equivalence classes of the two neighboring edges are joined. Once an edge is inserted that connects two vertices that already belong to the same equivalence class, that edge is closing a loop. To ensure that only one loop exists, all unresolved variables in G are set to 0, leading to a contradiction with an Even constraint if there are any edges not belonging to the loop.

Component Size Equals (G, v, Σ) The component of G containing v contains exactly Σ vertices.

A grid graph G^* containing all *potential* edges in G is constructed. The connected components of G and G^* are modeled as equivalence relations. If v's equivalence class in G^* becomes smaller or v's equivalence class in G becomes larger than Σ, the constraint panics. If v's equivalence class in G contains exactly Σ vertices, all potential edges connecting v's component to the rest of the graph are removed (their variables are set to 0).

Conditional (X_i, v, C) Constraint C must be satisfied if $X_i = v$.

If $X_i = v$, the *propagate* method of C is called.

2.2 Implementation

Different aspects of the solver have different priorities: While the puzzle definition and visualization mainly benefit from quick development times, reducing the amount of time needed to implement a solver for a new puzzle, the Deductive Search part of the solver benefits from type safety and execution speed while the development time is less important because the algorithm and constraints do not need to be updated when implementing new puzzles.

A dedicated graphics-focused software like Godot can be used for visualization, making it easier to tweak graphical details for single puzzles. Puzzle definitions (including the translation function) can then be implemented using a higher level scripting language like GDScript. Because visualization is largely independent from the solver, a compiled language was used for the reference implementation. Rust, which can be compiled to a binary file compatible with Godot's GDExtension API using the `godot-rust` library, was chosen for its memory safety and relative ease of development.

This separation allows for higher flexibility and less boilerplate code in puzzle definitions, while keeping the performance and type safety that Rust offers for the solver.

Grid Graph Indices

Graphs are usually implemented by assigning indices to the vertices and storing edges as pairs of vertex indices. Because the structure of a grid graph can always be derived from its width and height, it is sufficient to assign indices to the edges. Figure 10 shows such a numbering scheme on a 3×2 grid.

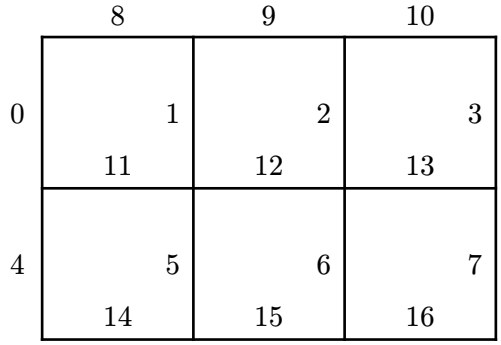


Figure 10. Indices are assigned to the grid lines of a 3×2 grid.

The indices of the grid lines surrounding a given cell can be directly computed from the cell's index i using the formulae shown in Figure 11.

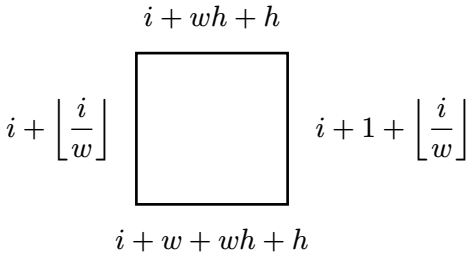


Figure 11. Computing indices of the grid lines surrounding cell i.
w and h denote the width and height of the grid.

3 Evaluation

The algorithm is evaluated using 1000 Slitherlink instances from Nikoli's *Fresh* series [10]. Slitherlink uses some simpler and some more complex constraints, and the results can be compared to those in [2]. Of the 1000 puzzles, 999 were solved by the algorithm, while 1 puzzle was too complex for the algorithm to solve with a search depth of 2. Larger search depths lead to extremely long execution times, so the last puzzle remains unsolved. Figure 12 shows execution times for these 999 instances. The reference implementation appears to be slightly slower than the recorded times in [2, sec. III B].

3.1 Generality

To evaluate the proposed constraints $\mathcal{C}^*$ based on the metric defined in Section 2.1, popular Nikoli puzzles are modeled using $\mathcal{C}^*$. The definitions of Sudoku and Slitherlink are adapted from [2, sec. III].

Sudoku An All Different constraint is added for every row, column, and 3×3 region.

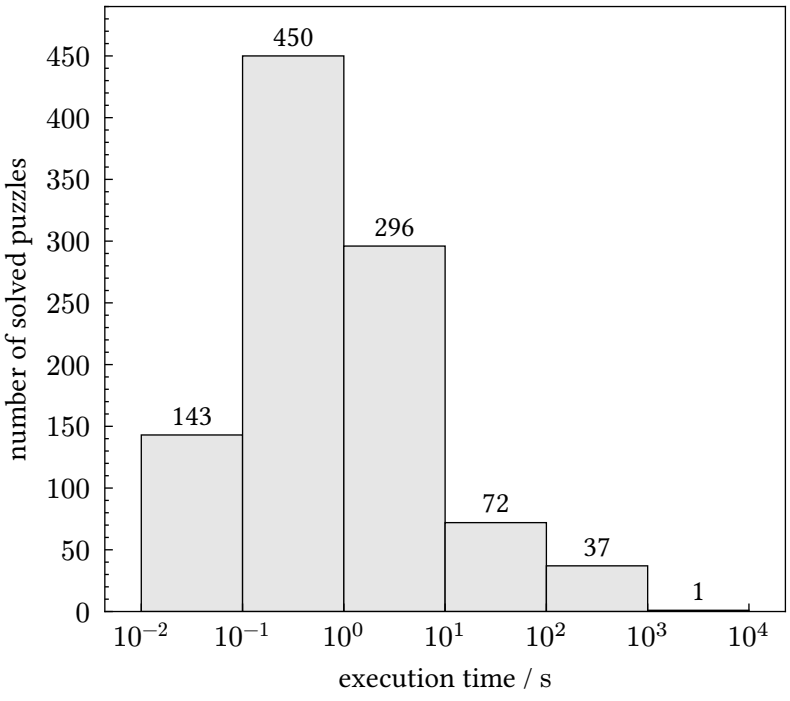


Figure 12. Execution times for Slitherlink puzzles from [10].
This figure only includes puzzles that were completely solved.

Slitherlink A grid line graph (see Figure 6) is used to model the puzzle, domains are initialized as $\{0, 1\}$, with 1 representing an edge, and 0 representing no edge.

A SUM EQUALS constraint is added per hint cell. The target sum Σ is the number in the cell, $\mathcal{X}$ contains the variables corresponding to the four surrounding edges. One EVEN and one LESS THAN (3) constraint are added per grid point (with $\mathcal{X}$ containing the 2 or 4 variables corresponding to the connecting edges), to ensure that the line does not branch off or cross itself. Finally, one CLOSED LOOP constraint is added.

Nurikabe The domains are initialized as $\{0, 1\}$, with 0 representing an empty cell, and 1 representing a filled cell. Domains of hint cells are initialized as $\{0\}$, because hint cells can not be filled.

To prevent filled squares, a LESS THAN (4) constraint is added for every 2×2 cell group on the board. A DISCONNECTED constraint over the polyomino graph of empty cells is added to ensure that no two hint cells are connected through an empty polyomino. One COMPONENT SIZE EQUALS constraint over the polyomino graph of empty cells per hint cell ensures that the size of each island is equal to the number it contains. Finally, a GRAPH CONNECTED constraint over the filled polyomino graph is added.

Akari [11] The domains are initialized as $\{0, 1\}$, with 0 representing an empty cell, and 1 representing a lamp. Domains of hint cells are initialized as $\{0\}$, because hint cells can not contain lamps.

A SUM EQUALS constraint is added per hint cell. The target sum Σ is the number in the cell, $\mathcal{X}$ contains the variables corresponding to the two or four horizontally and vertically adjacent cells. Areas containing exactly one lamp are computed by finding all polyominoes of empty cells in every row and every column:

$$A := \left\{ \text{polyominoes}_{\text{empty}}(a) \mid a \in \mathit{rows} \cup \mathit{cols} \right\}.$$

Additional SUM EQUALS (1) constraints are added per $a \in A$ to ensure no two lamps are "visible" from each other.

Hashiwokakero [12] A cell-connecting graph (see Figure 5) is used to model the puzzle, domains are initialized as $\{0, 1, 2\}$, with 0 representing no edge, and 1 and 2 representing a single and double edge, respectively.

A SUM EQUALS constraint is added per hint cell. The target sum Σ is the number in the cell, $\mathcal{X}$ contains the variables corresponding to the 2–4 neighboring lines. For empty cells, RELATION constraints are added to ensure that lines do not turn, cross, or branch off, and that lines stay single or double lines between "islands". Finally, a GRAPH CONNECTED and a TREE constraint over the cell-connecting graph are added.

Numberlink [13] A cell-connecting graph (see Figure 5) is used to model the puzzle, domains are initialized as $\{0, 1\}$, with 1 representing an edge, and 0 representing no edge.

A CONNECTED constraint over the cell-connecting graph is added for each pair of hint cells with the same number. An additional SUM EQUALS (1) constraint is added to each hint cell to ensure that only one path exists. For all empty cells, a LESS THAN (3) constraint is added, to ensure that the lines do not branch off or cross each other. Additional EVEN constraints can be added to improve performance.[4]

[4]These EVEN constraints are redundant, since the same continuity is already enforced by the CONNECTED constraint. Nonetheless, they are arguably directly derived from the puzzle rules [13].

Yajilin [14] A variable is added per grid cell. Their domains are initialized as $\{0, 2\}$, with 0 representing an empty cell, and 2 representing a filled cell. Domains of hint cells are initialized as $\{0\}$, because hint cells can not be filled. A cell-connecting graph (see Figure 5) is used to model the lines, its domains are initialized as $\{0, 1\}$, with 1 representing an edge, and 0 representing no edge.

A SUM EQUALS constraint is added per hint cell. The target sum Σ is the number in the cell, $\mathcal{X}$ contains the variables corresponding to the grid cells in the direction of the arrow. One EVEN and one LESS THAN (3) constraint are added per non-hint cell. They are defined over the variable corresponding to the cell and the grid variables corresponding to the edges connected to the cell, and ensure that the line does not branch off or cross itself, and that each cell is either part of the loop, or filled in. Finally, one CLOSED LOOP constraint is added.

Masyu [15] A cell-connecting graph (see Figure 5) is used to model the puzzle, domains are initialized as $\{0, 1\}$, with 1 representing an edge, and 0 representing no edge.

One EVEN and one LESS THAN (3) constraint are added per cell, to ensure that the line does not branch off or cross itself. SUM EQUALS constraints are added for every hint cell. On black circles, two SUM EQUALS (1) constraints are added, each linking two opposite edges together, ensuring that they do not form a straight line. Similarly, on white circles, four SUM EQUALS (1) constraints are added, each linking pairs of adjacent edges together, ensuring that they do not form a turn. The SUM EQUALS constraints also ensure that the loop passes through every hint cell. RELATION constraints are added for black hint cells, to ensure that the line does not make another turn immediately after the hint cell. A CONDITIONAL (LESS THAN (2)) constraint is added for every white hint cell, to ensure that the loop makes a turn immediately after the hint cell in at least one direction. Finally, one CLOSED LOOP constraint is added.

The generalized approach presented here does not work for all pencil puzzles. Some examples include Kakuro and Suraromu, whose elements can not be described using the model from Section 2, and Shikaku, Shakashaka, and Evolomino, whose rules can not be implemented using constraints from $\mathcal{C}^*$ in a way that closely resembles the rules.

3.2 Difficulty

[2, sec. II G] defines a difficulty metric

$$\operatorname{diff}(S) := \frac{V_0 + 4 \cdot V_1 + 9 \cdot V_2}{\sum_{D_i \in S} |D_i|},$$

where V_d is the number of variable instantiations during a Deduce step of depth d. Figure 13 compares this metric to Nikoli's difficulty rating. While there is a general trend towards higher values for puzzles with higher difficulty ratings, the difficulty metric is too noisy to draw any meaningful conclusions about the puzzle's difficulty.

A similar pattern can be observed with the execution time (Figure 14), the total number of agreements (Figure 15), the total number of Deduce steps (Figure 16), and the search depth required to solve the puzzle (Figure 17).

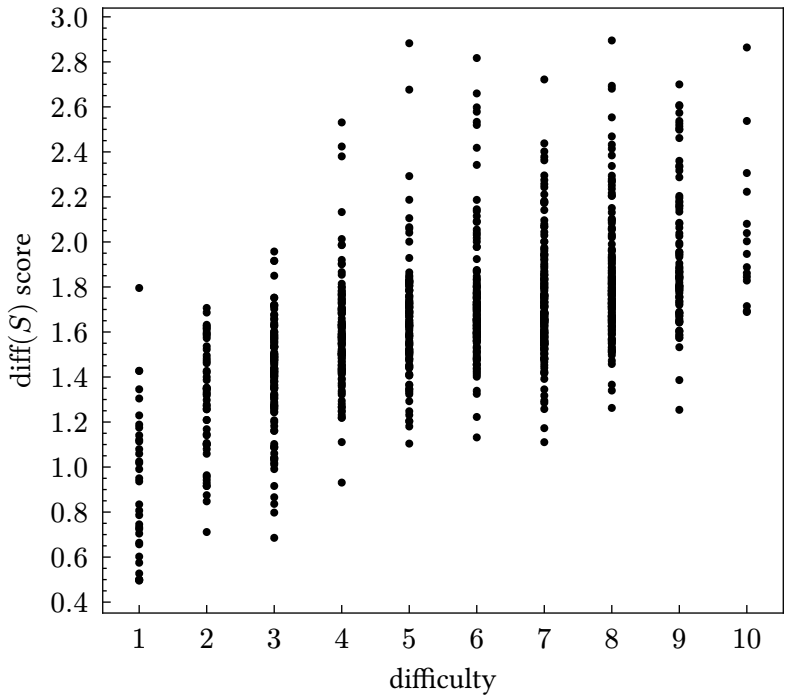


Figure 13. Calculated difficulty scores of the first 450 puzzles from [10] and the publisher's difficulty rating.

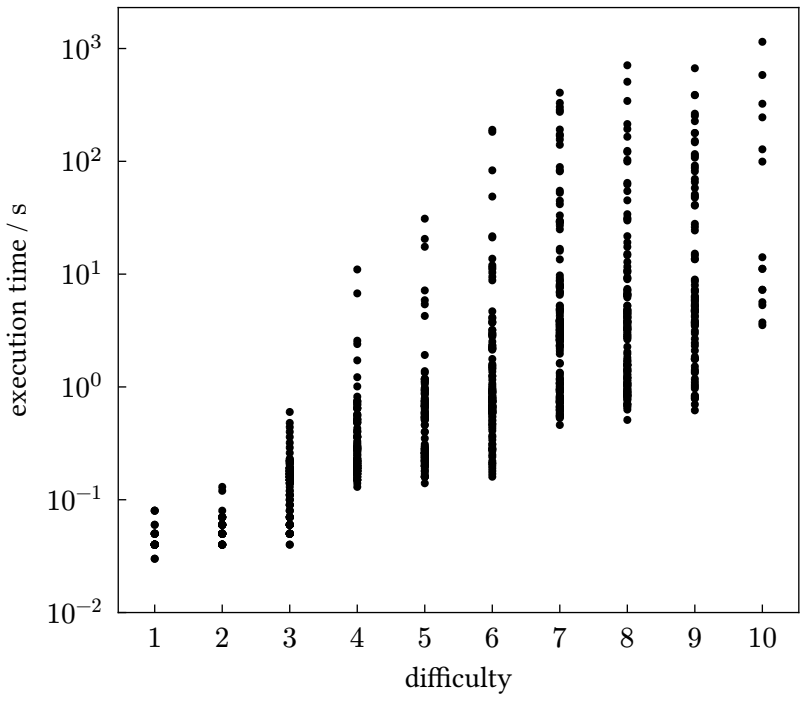


Figure 14. Execution times for Slitherlink puzzles from [10] and the publisher's difficulty rating.

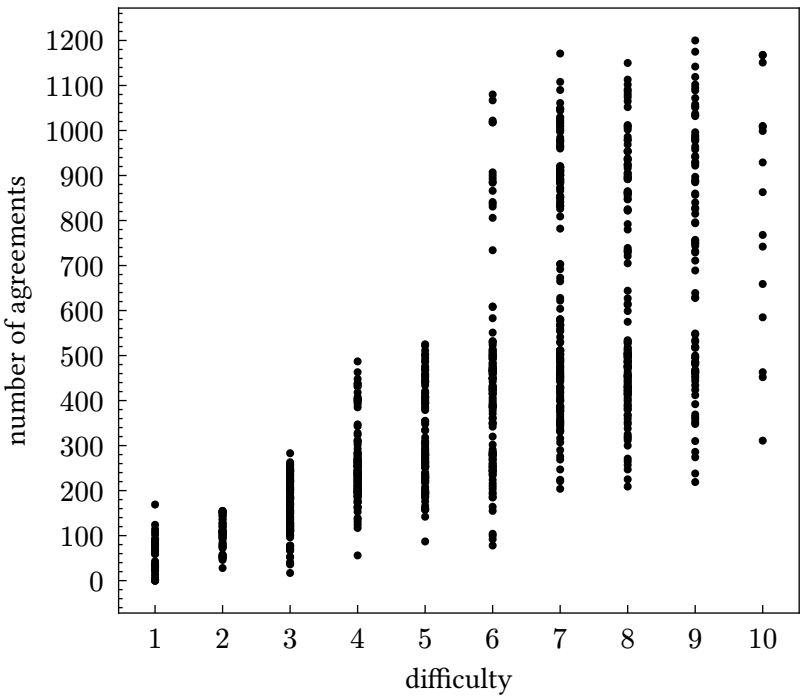


Figure 15. Number of agreements required to solve Slitherlink puzzles from [10] and the publisher's difficulty rating.

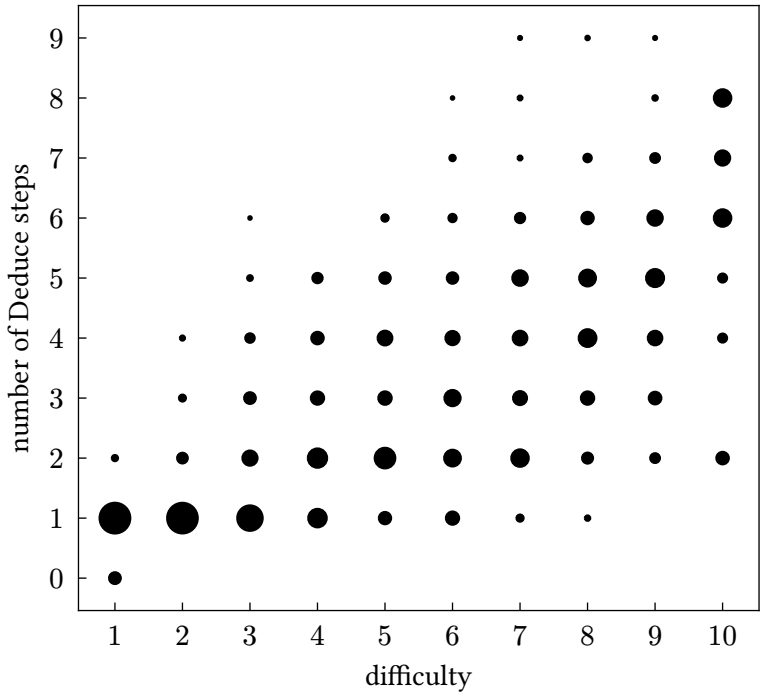


Figure 16. Number of Deduce steps required to solve Slitherlink puzzles from [10] and the publisher's difficulty rating. The sizes of the circles indicate the frequency relative to the number of puzzles with the same difficulty rating.

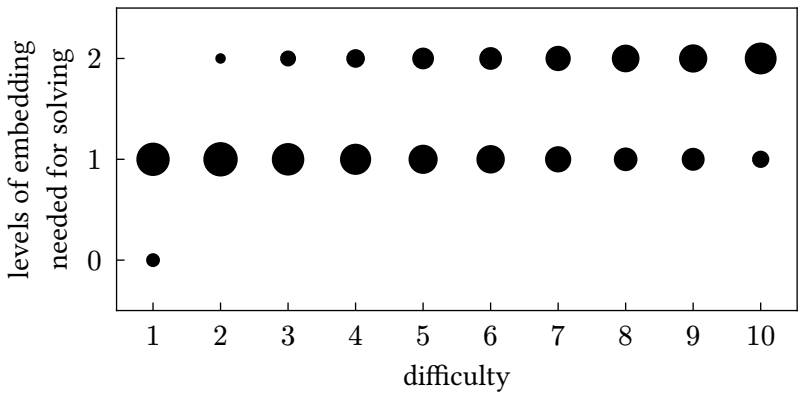


Figure 17. Levels of embedding required to solve Slitherlink puzzles from [10] and the publisher's difficulty rating. The sizes of the circles indicate the frequency relative to the number of puzzles with the same difficulty rating.

4 Discussion

Equation (3) in [2, sec. II B] defines the *Agreement* operation as

$$\left(\nexists v \in X_i^* : X_i \cdot v \Rightarrow X_j \cdot w\right) \Rightarrow X_j \leftarrow X_j \setminus w,$$

where $X_i \cdot v$ denotes instantiation of X_i to v, and X_i^* denotes the subset of remaining values available to a given variable X_i [2, sec. II A]. An implementation of the *Deductive Search* algorithm adhering to this equation might shave some w from X_j, *even though* X_j *was never fully initialized* during the shaving iterations, resulting in empty domains in solvable puzzles. Instead of checking if the instantiation of X_i to any v would lead to *instantiation* of X_j to w, the algorithm might check if the instantiation of X_i to any v *would not* lead to *elimination* of w from X_j^*:

$$\left(\nexists v \in X_i^* : X_i \cdot v \Rightarrow w \in X_j^*\right) \Rightarrow X_j \leftarrow X_j \setminus w.$$

Equation (2) in [2, sec. II B] also defines a second *Agreement* operation that works complementary to the one described in Section 1.1:

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

This operation works by directly setting X_j to w if the instantiation of X_i to *all* of its possible values results in X_j being set to w. This is redundant, because if instantiation of X_i to v leads to elimination of all $w' \in X_j^*, w' \neq w$, then all values w' are eliminated, leaving w as the only value for X_j. While it might improve runtime on some puzzles, this second agreement operation was not implemented for sake of simplicity.

4.1 Conclusion

The approach presented here can be used to model and solve a variety of pencil puzzles. Because of the absence of puzzle specific constraints, difficulty metrics derived from performed operations are only of limited value. Experiments indicate that Deductive Search with generalized constraints can solve puzzles in similar amounts of time as existing specialized approaches, but this is not necessarily the case for all puzzle types described in Section 3.1.

4.2 Future Work

The Closed Loop constraint is redundant, as its functionality is also enforced by the Graph Connected constraint, assuming Less Than (3) and Even constraints at every grid point. How exactly the solver's performance is affected by replacing Closed Loop constraints with Graph Connected constraints remains an open question.

The Sum Equals constraint does not always eliminate values that can be known to be incorrect. For example, consider a Sum Equals (10) constraint over domains $D_1 = \{3, 4, 5\}$ and $D_2 = \{5, 7\}$. The value 4 can be eliminated from D_1, because no possible variable assignment with sum 10 exists. The suboptimal nature of the implementation described in Section 2.1 has to be weighed against the potentially computationally expensive operation of calculating the sums of all possible combinations.

Similarly, some global constraints could benefit from additional rules for making proactive moves. Figure 18 shows a Slitherlink puzzle after a Deduce step with depth 0. To draw the remaining two lines, the global Closed Loop constraint has to be considered. Since the Closed Loop constraint can only rule out incorrect moves through panicking, which requires a Shaving operation, it "uses up" one level of the search tree. This also means that combining multiple global constraints requires a larger search depth, which becomes expensive very quickly.

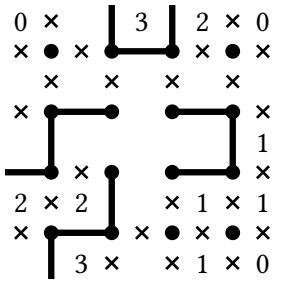


Figure 18. Detail of puzzle #301 from [10] after a single Simplification operation. To completely solve the puzzle, the global rule *lines form a single closed loop* [4] has to be considered, which requires one level of embedding.

As global constraints are more expensive to compute, performance might be improved by applying them only after local constraints have been exhaustively applied.

Generating Challenges

Deductive Search might serve as a useful starting point for generating new challenges for a given puzzle, using a similar method as described in [16]. First, a random solution is generated, by iteratively initializing any variable to a random value and then applying a Simplification. Hints are then removed from this generated solution until the resulting challenge would not have a single solution.

References

[1] Z. Butler, I. Bezáková, and K. Fluet, "Pencil puzzles for introductory computer science: An experience-and gender-neutral context," in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 2017, pp. 93–98.

[2] C. Browne, "Deductive Search for Logic Puzzles," 2013, pp. 1–8. doi: 10.1109/CIG.2013.6633649.

[3] J. Sesiano, *Magic Squares*. Springer, 2019.

[4] Nikoli Co., Ltd., "Slitherlink." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/slitherlink/

[5] T. Mantere and J. Koljonen, "Solving, rating and generating Sudoku puzzles with GA," in *2007 IEEE Congress on Evolutionary Computation*, 2007, pp. 1382–1389. doi: 10.1109/CEC.2007.4424632.

[6] I. Lynce and J. Ouaknine, "Sudoku as a SAT Problem.," in *AI&M*, 2006.

[7] J.-T. Tsai and P.-Y. Chou, "Solving Japanese puzzles by genetic algorithms," in *2011 International Conference on Machine Learning and Cybernetics*, 2011, pp. 785–788. doi: 10.1109/ICMLC.2011.6016787.

[8] C. Chang, Z. Fan, and Y. Sun, "A Difficulty Metric and Puzzle Generator for Sudoku," *UMAPJournal*, pp. 305–326, 2007.

[9] Nikoli Co., Ltd., "Nurikabe." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/nurikabe/

[10] フレッシュスリザーリンク. *Volume 10*. Tokio: Nikoli, 2016.

[11] Nikoli Co., Ltd., "Akari." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/akari/

[12] Nikoli Co., Ltd., "Hashiwokakero." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/hashiwokakero/

[13] Nikoli Co., Ltd., "Numberlink." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/numberlink/

[14] Nikoli Co., Ltd., "Yajilin." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/yajilin/

[15] Nikoli Co., Ltd., "Masyu." Accessed: July 19, 2025. [Online]. Available: https://www.nikoli.co.jp/en/puzzles/masyu/

[16] A. Chou and J. Kaashoek, "PuzzleJAR: Automated Constraint-based Generation of Puzzles of Varying Complexity," 2014.