A Deductive Search-Based Solver for Kakuro Puzzles: Comparative Evaluation with Other Approaches and Difficulty Estimation

Bachelor's Thesis

Ayman Ibrahim Abdalla Arafa Abdeldayem
480661

Submitted on: 12.10.2025

Slightly revised version as of: 05.02.2026

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

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

Use of Generative AI Tools

- **ChatGPT**
  Provider: OpenAI
  Version/Model: GPT-5 (the free version may use different models depending on availability, usage conditions, and task type)
  Purpose: Used for Python scripting to analyze data and generate plots. Additionally employed for text formatting, preliminary language review, and partial assistance during development, including suggesting edits, generating code snippets, and creating initial documentation.

- **DeepL**
  Provider: DeepL SE
  Version/Model: DeepL Write
  Purpose: Used to enhance the clarity, grammar, and style of written text.

- **Gemini**
  Provider: Google DeepMind
  Version/Model: Gemini 2.5 Pro
  Purpose: Applied for final review (Deep Research function) to identify potential issues and ensure overall consistency.

Abstract

Kakuro is a logic puzzle in which players fill a grid with numbers to satisfy specified sums. Automated solvers for Kakuro vary widely in their algorithmic techniques and domains of application. This thesis presents a solver that emulates the reasoning process used by human solvers. Consequently, we formalize a set of techniques commonly used by Kakuro players and integrate them into Browne's Deductive Search (DS) method as a simplification strategy, falling back on a breadth-first search when simplification is insufficient. This design avoids deep embedding, aligning with human cognitive and computational limitations. DS is systematically compared with classical approaches, including backtracking and constraint programming. It demonstrates competitive runtimes relative to these solvers. Another advantage of this approach is that the solutions are based on logical strategies commonly employed by human solvers, thereby enhancing interpretability. Despite these advantages, it cannot determine whether a puzzle is unsolvable or handle instances with multiple valid solutions. We further explore an important application of DS: estimating puzzle difficulty using metrics collected during the solving process. Additionally, we introduce enhancements to refine difficulty estimation, including assigning costs to different strategies and incorporating the temporal order of the applied strategies. The results demonstrate that the estimated difficulty levels derived from DS strongly correlate with the official ratings provided by puzzle publishers, such as Nikoli.

Kurzfassung

Kakuro ist ein Logikrätsel, bei dem die Spieler Zellen mit Zahlen füllen, um vorgegebene Summen zu erreichen. Automatisierte Solver für Kakuro unterscheiden sich stark in ihren algorithmischen Techniken und Anwendungsbereichen. Diese Arbeit stellt einen Solver vor, der die menschlichen Lösungsstrategien nachbildet. Daher formalisieren wir einige Techniken, die von Kakuro-Spielern häufig eingesetzt werden, und integrieren sie als Simplifikationsmethode in den Deductive-Search-Ansatz (DS) von Browne. Reicht die Simplifikationsstrategien allein nicht aus, wird auf eine Breitensuche zurückgegriffen. Dieser Ansatz vermeidet tiefe Einbettungen und orientiert sich an den kognitiven und rechnerischen Beschränkungen des Menschen. DS wird systematisch mit klassischen Ansätzen wie Backtracking und Constraint Programming verglichen und zeigt dabei konkurrenzfähige Laufzeiten. Ein weiterer Vorteil des Ansatzes liegt darin, dass die Lösungen auf logischen Strategien basieren, die auch von menschlichen Spielern angewendet werden, was die Interpretierbarkeit erhöht. Allerdings kann der Solver weder feststellen, ob ein Rätsel unlösbar ist, noch mit Instanzen umgehen, die mehrere gültige Lösungen besitzen. Darüber hinaus untersuchen wir eine wichtige Anwendung von DS: die Schätzung des Schwierigkeitsgrads von Rätselinstanzen anhand von Metriken, die während des Lösungsprozesses gesammelt werden. Zudem werden einige Verbesserungen eingeführt, um die Schätzung zu verfeinern, darunter die Zuweisung von Kosten zu verschiedenen Strategien sowie die Berücksichtigung der zeitlichen Reihenfolge der angewandten Strategien. Die Ergebnisse zeigen, dass die mit DS ermittelten Schwierigkeitsgrade stark mit den offiziellen Bewertungen von Rätselverlagen wie Nikoli korrelieren.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Relevance of the Problem . . . 2
1.3 Literature Review . . . 2
1.4 Objective of the Thesis, Research Question . . . 3
1.5 Structure of the Thesis . . . 3

2 Kakuro **4**
2.1 Puzzle Description . . . 4
2.2 Rules . . . 4
2.3 Terminology . . . 5
2.4 Formal Representation of Kakuro as a Constraint Satisfaction Problem . . . 6
2.5 Human Solving Techniques . . . 7
2.5.1 Narrowing Domain (Propagation) . . . 7
2.5.2 Naked Singles . . . 10
2.5.3 Hidden Singles . . . 11
2.5.4 Naked Subsets . . . 11
2.5.5 Hidden Subsets . . . 12
2.5.6 Filter Incompatible Pairs . . . 12
2.5.7 Filter Incompatible Triples . . . 13
2.5.8 Divide-and-Conquer . . . 14
2.6 Summary of Human Solving Techniques . . . 15

3 Solving Approaches **17**
3.1 Backtracking Solver . . . 17
3.2 CP Solver . . . 19
3.3 Deductive Search-Based Solver . . . 20
3.4 Hybrid Solver: Forward Checking with Simplification . . . 25

4 Comparative Evaluation **26**
4.1 Methods . . . 26
4.1.1 Dataset . . . 26
4.1.2 Comparison Criteria . . . 26
4.2 Results . . . 27
4.2.1 Experimental setup . . . 27
4.2.2 Performance / Runtime . . . 27
4.2.3 Soundness and Completeness . . . 28
4.2.4 Transparency and Explainability . . . 29
4.3 Discussion . . . 31

5 Difficulty Estimation **32**
5.1 Methods . . . 32
5.1.1 Dataset . . . 32
5.1.2 Metrics . . . 32
5.2 Results . . . 35
5.3 Discussion . . . 37

6 Conclusion and Future Work **38**
6.1 Deductive Search . . . 38
6.1.1 Contributions . . . 38
6.1.2 Limitations and Future Research Directions . . . 38
6.2 Difficulty Estimation . . . 39
6.2.1 Contributions . . . 39
6.2.2 Limitations and Future Research Directions . . . 39

1 Introduction

1.1 Motivation

Logic puzzles, such as Sudoku and Kakuro, have gained popularity both as entertainment activities and as subjects of computer science research. Although many automated solvers for Kakuro exist [5, 4], most of these solving methods approach the problem purely as a combinatorial challenge, attempting to solve it systematically, often without incorporating domain-specific knowledge, such as the logical techniques applied by humans. Some solvers attempt to incorporate heuristic strategies to guide the solving process more efficiently, but these approaches can be unreliable due to randomization and may fail to find a solution for valid puzzles. Both approaches contrast with the methods actually used by human players to solve such puzzles.

Alternatively, if a solver applies strategies based on logical rules tailored specifically to the puzzle, as humans often do, it can simplify the solving process. This approach can support human understanding and reduce computational overhead compared to brute-force methods.

Furthermore, solving very difficult puzzle instances can be challenging, especially for beginners. While an expert might solve it completely through logical reasoning, a beginner may sometimes rely on trial and error. Similarly, if an automated solver does not implement enough logical strategies that can fully deduce the solution without search, it may still benefit from a hybrid approach, i.e., applying as much logic as possible and falling back on trial and error only when deterministic strategies are insufficient.

These puzzles are often categorized by difficulty level (e.g., easy, medium, or hard). This allows players to choose puzzles that match their skill level. Beginners can start with simple puzzles, while experts can challenge themselves with harder ones. However, an important question arises: How can puzzle creators automatically and meaningfully assess the difficulty of a puzzle?

Puzzles are, thus, an interesting area of study, not only because of their structural and mathematical aspects, such as the generation and validity of instances, but also because of their strong connection to human behavior. Topics such as handmade puzzles, human-like solvers, and difficulty estimation are important for enhancing the puzzle-solving experience.

1.2 Relevance of the Problem

Logic–arithmetic-based puzzles, such as Kakuro, can be modeled as constraint satisfaction problems (CSPs) [4]. These problems share a common objective: the task is to assign values to variables while respecting the given constraints. Therefore, solving methods developed for Kakuro can be adapted to a wide range of other combinatorial problems.

Additionally, this work benefits AI systems by encouraging more human-centric reasoning and logical inference rather than relying mostly on statistical approaches. This shift can be particularly useful in areas such as:

- **Tutoring**: where the goal is not only to reach a valid solution but also to support interaction between humans and systems by explaining reasoning steps and introducing relevant concepts, such as advanced strategies for solving puzzles.
- **Puzzle design**: where difficulty levels need to be carefully calibrated to match players' skills, since too easy puzzles may be boring while too difficult ones can be frustrating.

1.3 Literature Review

A fundamental approach for solving CSPs is backtracking [2]. This method is easy to implement. It explores all possible variable assignments systematically in a depth-first search. However, pure backtracking is generally inefficient for large or complex problem instances, as it performs an exhaustive search without incorporating any knowledge about the problem structure, leading to excessive computational overhead.

To address this inefficiency, several enhancements have been developed that incorporate knowledge of the problem structure into the search process. Forward checking, for example, involves propagating constraints after each assignment, which helps to reduce the search space and prevents infeasible assignments from being considered in subsequent steps [2].

Additionally, heuristic strategies, particularly those dependent on stochastic methods such as Monte Carlo techniques, have been employed to further optimize the search process. For instance, Cazenave [4] introduced a Monte Carlo-based approach to guide the exploration of the search tree more intelligently.

In his article, Helmut Simonis [9] presented an alternative approach, focusing on constraint programming (CP) methods. His work demonstrated improvements in constraint propagation and shaving techniques to reduce the need for search. However, the article includes very few Kakuro-specific strategies, focusing instead on general constraint-solving techniques from a solver-centric perspective.

Cameron Browne [3] proposed a hybrid method for solving logic puzzles. The approach combines logical simplification with depth-limited search: it first applies a suite of simplification strategies to simplify the puzzle as much as possible. When progress stalls, a breadth-first, depth-limited search is used as a fallback to solve difficult sections. The resulting state is then simplified again using logical strategies. This cycle of simplification and targeted search is repeated until a complete solution is found. A similar approach was explored by Pelánek [7] in his framework for Sudoku difficulty estimation.

1.4 Objective of the Thesis, Research Question

In this work, we propose a solver based on Deductive Search (DS) [3], designed to emulate human reasoning through logically-grounded steps. The goal is to bridge the gap between human techniques for solving Kakuro and automated methods by emphasizing interpretability. The approach accounts for human computational and cognitive limitations, such as the difficulty with exhaustive enumeration and deep recursive operations. Building on this foundation, the solver can be applied to estimate the difficulty of puzzle instances.

Therefore, the central research question is as follows: Can a solver that emulates human approaches to Kakuro puzzles reliably solve them while also providing a reasonable estimation of their difficulty?

1.5 Structure of the Thesis

- **Chapter 2: Kakuro** describes Kakuro puzzles and presents some logical rules and strategies commonly used by humans to solve them.
- **Chapter 3: Solving Approaches** discusses different algorithmic methods of solving Kakuro, including backtracking, constraint programming, the proposed DS-based solver, and a hybrid approach that combines backtracking with DS.
- **Chapter 4: Comparative Evaluation** compares the different solving approaches, highlighting the strengths and limitations of each method.
- **Chapter 5: Difficulty Estimation** focuses on difficulty estimation using DS. It discusses the metrics used and the reasoning behind them.
- **Chapter 6: Conclusion and Future Work** summarizes the findings, discusses limitations, and outlines potential directions for future research.

2 Kakuro

2.1 Puzzle Description

Kakuro (Japanese: カックロ), or Cross Sums, is a logic puzzle that consists of a 2-dimensional grid of cells. The cells can be categorized into two groups:

1. **Empty Cells:** These are typically white and empty. They should be filled with numbers between 1 and 9. A group of adjacent empty cells, arranged either horizontally or vertically, forms a *run* [5].
2. **Clue Cells:** These are usually black and contain one or two numbers, indicating the target sums for the adjacent horizontal or vertical runs. A number placed in the bottom-left side of the cell indicates a vertical sum (downward), and a number in the top-right side indicates a horizontal sum (rightward).

An example of Kakuro is shown in Figure 2.1.

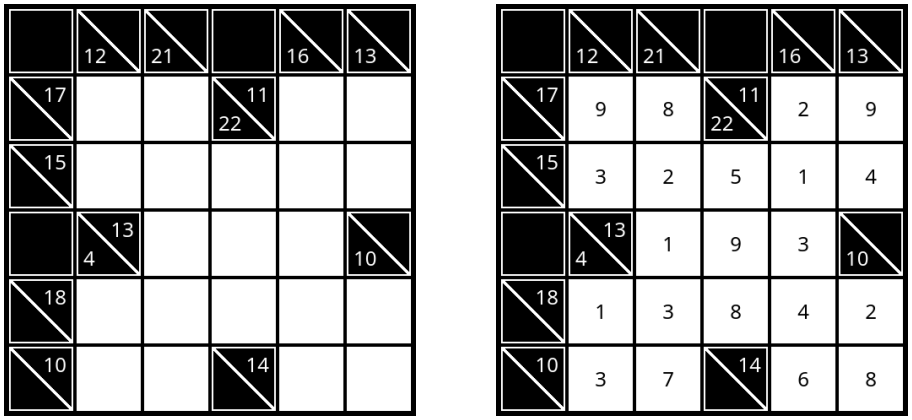

Figure 2.1: A Kakuro challenge (left) and its solution (right). Source: www.janko.at

2.2 Rules

Kakuro is similar to the well-known crossword puzzle. However, it is a Japanese pencil puzzle[6] that has some special characteristics: it is single-player, entirely based on mathematics, culturally independent, and language-neutral. It has the following simple rules:

- Empty cells can be filled with numbers from 1 to 9.
- Clue cells indicate the sum of numbers in the corresponding horizontal or vertical runs.
- No repeated numbers are allowed in a single run.

2.3 Terminology

Authors in the Kakuro literature often use different notations for the same concepts. For example, a sequence of connected empty cells associated with a clue may be called a *block* [9], a *sector* [1], or a *run* [5], which can lead to confusion. Moreover, in order to describe the puzzle more formally and to formulate solving strategies, it is helpful to have a unified terminology. Therefore, we introduce a set of terms as the foundation for the subsequent definitions and methods.

Let the Kakuro grid be denoted by K, where each cell is indexed by its row r and column c. We denote a cell by $k_{r,c} \in K$.

Cell Type

- $\mathrm{Type}(k) \in \{\texttt{EMPTY}, \texttt{CLUE}\}$.
- $\mathrm{Empty}(K) = \{k \in K \mid \mathrm{Type}(k) = \texttt{EMPTY}\}$ the set of all empty cells.
- $\mathrm{Clues}(K) = \{k \in K \mid \mathrm{Type}(k) = \texttt{CLUE}\}$ the set of all clue cells.

Empty Cells

For an empty cell k:

- $\mathrm{Value}(k) \in \{1, \ldots, 9\} \cup \{\bot\}$ assigned value (or $\bot$ if not assigned).
- $\mathrm{Domain}(k) \subseteq \{1, \ldots, 9\}$ candidate values that can be assigned to k.
- $\mathrm{HClue}(k) \in \mathrm{Clues}(K)$ the first clue cell left of the empty cell k.
- $\mathrm{VClue}(k) \in \mathrm{Clues}(K)$ the first clue cell over the empty cell k.

Clue Cells

For a clue cell c:

- **Initial state:**
  - $\mathrm{HSum}(c) \in \{3, \ldots, 45\} \cup \{\bot\}$ horizontal clue sum (or $\bot$ if none).[1]
  - $\mathrm{VSum}(c) \in \{3, \ldots, 45\} \cup \{\bot\}$ vertical clue sum (or $\bot$ if none).
  - $\mathrm{HRun}(c) = \{k_1, \ldots, k_h\}$ the set of empty cells right of c until a non-empty cell is encountered.
  - $\mathrm{VRun}(c) = \{k_1, \ldots, k_v\}$ the set of empty cells below c until a non-empty cell is encountered.

[1] The minimum $\mathrm{HSum}(c)$ or $\mathrm{VSum}(c)$ is 3, since a valid run must contain at least two cells, while the maximum is 45, corresponding to the sum of digits from 1 to 9, which is the largest possible run according to Kakuro rules.

- **Intermediate state in the solving process:**
  - $\text{RemainingHSum}(c) = \text{HSum}(c) - \sum_{\substack{k \in \text{HRun}(c) \\ \text{Value}(k) \neq \perp}} \text{Value}(k)$.
  - $\text{RemainingVSum}(c) = \text{VSum}(c) - \sum_{\substack{k \in \text{VRun}(c) \\ \text{Value}(k) \neq \perp}} \text{Value}(k)$.
  - $\text{RemainingHRun}(c) = \{k \in \text{HRun}(c) \mid \text{Value}(k) = \perp\}$.
  - $\text{RemainingVRun}(c) = \{k \in \text{VRun}(c) \mid \text{Value}(k) = \perp\}$.

Remark. Since Kakuro is symmetrical, strategies for horizontal and vertical runs are analogous. To avoid unnecessary duplication, we use the generic term *Run*, and for a run R, we use $\text{Remaining}(R)$ for its unassigned empty cells and $\text{RemainingSum}(R)$ for its remaining clue sum.

Operations

We also define some main operations that are repeated during the solving process:

- $\text{RecalculateDomain}(k)$: Reduces the domain of the empty cell k by recalculating it, as will be discussed in detail in Section 2.5.1.
- $\text{AssignAndPropagate}(k, v)$: Assigns the value v to the empty cell k, clears its domain, and propagates constraints to all cells in the same horizontal and vertical runs:
  1. $\text{Value}(k) \leftarrow v$.
  2. $\text{Domain}(k) := \emptyset$.
  3. $\forall k' \in \text{RemainingHRun}(\text{HClue}(k)) \cup \text{RemainingVRun}(\text{VClue}(k)) : \text{RecalculateDomain}(k')$
- $\text{Prune}(k, V')$: Removes a set of values $V' \subseteq \{1, \ldots, 9\}$ from the domain of k:

$$\text{Domain}(k) := \text{Domain}(k) \setminus V'.$$

2.4 Formal Representation of Kakuro as a Constraint Satisfaction Problem

A Kakuro puzzle can be modeled as a CSP defined by the tuple (X, D, C) [2], where:

- **Variables** $X = \{x_k \mid k \in \text{Empty}(K)\}$ correspond to the empty cells in the Kakuro grid. Each variable x_k represents the value to be assigned to the empty cell k.
- **Domains** $D = \{D_k \mid D_k \subseteq \{1, \ldots, 9\}\}$ specify the possible values for each variable x_k. Initially, $D_k = \{1, \ldots, 9\}$ for all k.
- **Constraints** C enforce the Kakuro rules on the variables:
  1. **All-different constraints:** For each run,

$$\text{AllDifferent}(\{x_k \mid k \in \text{HRun}(c)\}) \quad \text{and} \quad \text{AllDifferent}(\{x_k \mid k \in \text{VRun}(c)\}),$$

  meaning no repeated values in a run.

2. **Run sum constraints:** For each clue cell $c \in \text{Clues}(K)$ and its associated runs,

$$\sum_{k \in \text{HRun}(c)} x_k = \text{HSum}(c), \quad \text{if } \text{HSum}(c) \neq \bot,$$

$$\sum_{k \in \text{VRun}(c)} x_k = \text{VSum}(c), \quad \text{if } \text{VSum}(c) \neq \bot.$$

2.5 Human Solving Techniques

Humans typically solve Kakuro using a set of logical strategies to reduce the search space and gradually fill in the puzzle. Many of these strategies, such as Naked/Hidden Singles and Subsets, are adapted from Sudoku but modified to account for Kakuro's unique sum constraints.[2]

2.5.1 Narrowing Domain (Propagation)

Domain narrowing acts as a propagation step. It is applied to all empty cells through $\text{RecalculateDomain}(k)$ at the beginning of the solving process (initial propagation), and is re-invoked after each new assignment via $\text{AssignAndPropagate}(k, v)$ to update the domains of the affected cells.

Before applying any solving strategies, $\text{Domain}(k)$ of each empty cell $k \in \text{Empty}(K)$ can be systematically narrowed by pruning impossible candidates according to the following simple rules:

1. **No Repetitions:** Since numbers cannot repeat within the same run, all values already assigned to other cells in k's horizontal and vertical runs are pruned from $Domain(k)$:

$$\text{Prune}\Big(k,\ \underbrace{\{\text{Value}(k') \mid k' \in \text{HRun}(\text{HClue}(k)),\ \text{Value}(k') \neq \bot\}}_{\text{assigned in same horizontal run}} \cup \underbrace{\{\text{Value}(k') \mid k' \in \text{VRun}(\text{VClue}(k)),\ \text{Value}(k') \neq \bot\}}_{\text{assigned in same vertical run}}\Big)$$

An example is shown in Fig. 2.2.

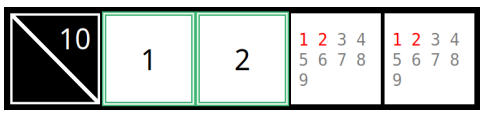

Figure 2.2: In this example, values 1 and 2 have already been assigned, so they must be removed from the domain of all other cells in the same run.

[2] Sources for solving techniques:
https://www.conceptispuzzles.com/de/index.aspx?uri=puzzle/kakuro/techniques,
https://hodoku.sourceforge.net/en/techniques.php,
https://theory.tifr.res.in/~sgupta/kakuro/algo.html.

2. **Lower-Upper Bounds:** The domain can be bounded by the clue sums and run lengths. For a run R containing k, define a lower bound (LB) and an upper bound (UB) based on the minimal and maximal possible sums of the other remaining unassigned cells:

   - **Initial-case bounds:** When no domain information is yet available, bounds can be computed from the clue sum $S = \mathrm{RemainingSum}(R)$ and run length $L = |\mathrm{Remaining}(R)|$ as:

$$\mathrm{LB} = \max\left(1,\ S - \sum_{i=1}^{L-1}(10-i)\right), \quad \mathrm{UB} = \min\left(9,\ S - \sum_{i=1}^{L-1} i\right)$$

   - **Intermediate bounds:** The domains of the cells have already been initialized and include all possible candidates:

$$\mathrm{LB}(k) = \max\left(1,\ \underbrace{\mathrm{RemainingSum}(R)}_{\text{Remaining sum including } k} - \underbrace{\mathrm{MaxSum}(\mathrm{Remaining}(R) \setminus \{k\})}_{\text{maximum sum of the other cells in the run (excluding } k)}\right),$$

$$\mathrm{UB}(k) = \min\left(9,\ \underbrace{\mathrm{RemainingSum}(R)}_{\text{Remaining sum including } k} - \underbrace{\mathrm{MinSum}(\mathrm{Remaining}(R) \setminus \{k\})}_{\text{minimum sum of the other cells in the run (excluding } k)}\right)$$

     The functions MinSum and MaxSum compute the minimal and maximal possible sums for a given set of empty cells in a run. We consider two alternative approaches for estimating these sum bounds:

     Approach 1 (cell-wise bounds): This approach computes bounds independently for each cell, ignoring the no-repetition constraint between cells:

$$\mathrm{MaxSum}_{\mathrm{cell}}(M) = \sum_{k\in M} \max \mathrm{Domain}(k), \quad \mathrm{MinSum}_{\mathrm{cell}}(M) = \sum_{k\in M} \min \mathrm{Domain}(k).$$

     Approach 2 (value-wise bounds): This approach respects the uniqueness of digits within a run by considering the union of all possible values and selecting distinct ones. Let

$$S_{\max} = \text{the set of the } |M| \text{ largest distinct values in } \bigcup_{k\in M} \mathrm{Domain}(k),$$

$$S_{\min} = \text{the set of the } |M| \text{ smallest distinct values in } \bigcup_{k\in M} \mathrm{Domain}(k).$$

     Then:

$$\mathrm{MaxSum}_{\mathrm{value}}(M) = \sum_{v\in S_{\max}} v, \quad \mathrm{MinSum}_{\mathrm{value}}(M) = \sum_{v\in S_{\min}} v.$$

     We compute both and select the tighter bound in each case.[3]

   The pruning can be performed as follows:

$$\mathrm{Prune}\Big(k,\ \underbrace{\{v \in \mathrm{Domain}(k) \mid v < \mathrm{LB}(k) \text{ or } v > \mathrm{UB}(k)\}}_{\text{out of bounds}}\Big)$$

[3] Both approaches do not necessarily yield the *optimal* values of MinSum and MaxSum. Finding the true optimum would require combinatorial optimization, which is deliberately avoided in this work to maintain consistency with the goal of emulating human-like solving.

An example is shown in Fig. 2.3.

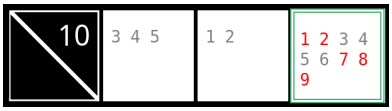

Figure 2.3: In this example, the highlighted cell has a lower bound of 3 because the maximum of the other cells is 7 (cell-wise bounds) and an upper bound of 6 because the minimum of the other cells is 4 (cell-wise bounds). Therefore, any value outside this range cannot form the required sum of 10.

3. **Magic Runs:** For a run of length L with clue sum S, certain special cases arise where the empty cells in the run have exactly one possible value combination. In these cases, we define D as the set of required values that realize the sum:

   - **Case 1** (the smallest sum fitting in L cells):

   $$S = \frac{L(L+1)}{2} \quad \Longrightarrow \quad D := \{1, 2, \ldots, L\}$$

   - **Case 2** (the second-smallest sum fitting in L cells):

   $$S = \frac{L(L+1)}{2} + 1 \quad \Longrightarrow \quad D := \{1, 2, \ldots, L-1, L+1\}$$

   - **Case 3** (the largest sum fitting in L cells):

   $$S = 10L - \frac{L(L+1)}{2} \quad \Longrightarrow \quad D := \{10 - L, 11 - L, \ldots, 9\}$$

   - **Case 4** (the second-largest sum fitting in L cells):

   $$S = 10L - \frac{L(L+1)}{2} - 1 \quad \Longrightarrow \quad D := \{9 - L, 11 - L, 12 - L, \ldots, 9\}$$

   - **Case 5** ($L = 8$): in this case, all values from 1 to 9 must be used except for a single value, which can be determined directly from the sum:

   $$L = 8 \quad \Longrightarrow \quad D := \{1, 2, \ldots, 9\} \setminus \{45 - S\}$$

   Then, we can prune any value not in D from the domain of any cell k in this run:

   $$\text{Prune}(k, \text{Domain}(k) \setminus D).$$

An example is shown in Fig. 2.4.

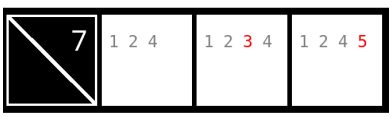

Figure 2.4: Magic Run: Case 2 for $L = 3$.

4. **Intersection:** Each empty cell k belongs to both a horizontal and a vertical run. The previous rules are applied independently to each run, yielding two candidate sets of values. The domain of k is then restricted to the intersection of these sets:

$$\text{Domain}(k) := \text{Domain}_H(k) \cap \text{Domain}_V(k)$$

where $\text{Domain}_H(k)$ is the set of values allowed for k by the horizontal run's constraints, and $\text{Domain}_V(k)$ is the set of values allowed by the vertical run's constraints.

An example is shown in Fig. 2.5.

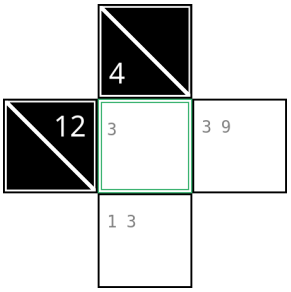

Figure 2.5: In this example, the highlighted cell can be 1 or 3 to fulfill the vertical run, but only 3 can fit in the horizontal run. Therefore, the value 1 can be removed by the Intersection rule.

2.5.2 Naked Singles

If an empty cell $k \in \text{Empty}(K)$ has exactly one possible value in its domain:

$$|\text{Domain}(k)| = 1$$

then that value must be assigned (assuming the instance is valid). Formally:

$$\text{AssignAndPropagate}(k, v) \quad \text{where} \quad \text{Domain}(k) = \{v\}.$$

An example is shown in Fig. 2.6.

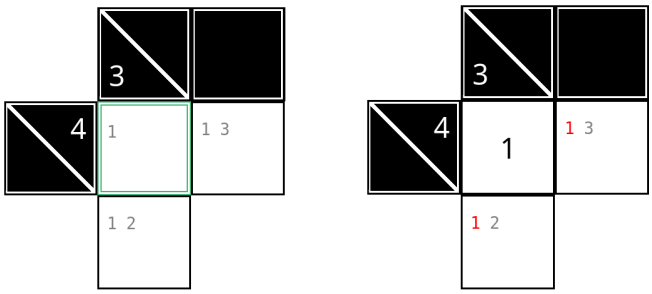

Figure 2.6: In this example, the highlighted cell has only one value in its domain, so it can be assigned, and the ones can be removed from the other 2 cells

2.5.3 Hidden Singles

Consider a run R with the number of remaining unassigned cells in R equals the number of distinct values appearing in their domains[4]:

$$|\text{Remaining}(R)| = \left| \bigcup_{k \in \text{Remaining}(R)} \text{Domain}(k) \right|,$$

and a value v appears in the domain of exactly one cell $k \in R$:

$$|\{k \in R \mid v \in \text{Domain}(k)\}| = 1,$$

then v must be assigned to k:

$$\text{AssignAndPropagate}(k, v).$$

An example is shown in Fig. 2.7.

Figure 2.7: In this example, the highlighted cell can only be assigned the value 1; otherwise, there will be an inconsistency in the other cells.

2.5.4 Naked Subsets

In a run R, suppose there exists a set of l cells

$$S \subseteq \text{Remaining}(R), \quad |S| = l$$

whose combined domains contain exactly l distinct values:

$$\left| \bigcup_{k \in S} \text{Domain}(k) \right| = l.$$

Then those l values must be placed in the cells of S, and can be removed from the domains of all other cells in $\text{Remaining}(R) \setminus S$:

$$\forall k' \in \text{Remaining}(R) \setminus S, \quad \text{Prune}(k', \bigcup_{k \in S} \text{Domain}(k)).$$

Additionally, let

$$\Sigma_S = \sum_{v \in \bigcup_{k \in S} \text{Domain}(k)} v$$

be the sum of the l values in the naked subset. The remaining cells must then sum to,

$$\Sigma_{R \setminus S} = \text{RemainingSum}(R) - \Sigma_S,$$

[4]Unlike Sudoku, Kakuro runs do not necessarily require all digits from 1 to 9 to appear. Therefore, Hidden Singles/Subsets apply only under this condition, making them more restricted than in Sudoku.

This new sum constraint can be stronger than the general sum constraint over all remaining cells (including those in S). It allows us to prune the domains of the cells in $\mathrm{RemainingRun}(R) \setminus S$ more effectively, by applying Narrowing Domain Rules again. An example is shown in Fig. 2.8.

Figure 2.8: In this example, the highlighted two cells have only two different values, so they must take these two values and be pruned from the domains of the other cells. Additionally, the value 6 can be removed, as it cannot form the sum 12 with any other value without repetitions.

2.5.5 Hidden Subsets

Consider a run R with the number of remaining unassigned cells in R equals the number of distinct values appearing in their domains:

$$|\mathrm{Remaining}(R)| = \left| \bigcup_{k \in \mathrm{Remaining}(R)} \mathrm{Domain}(k) \right|$$

and there exists a set V of l distinct values

$$V \subseteq \{1, \ldots, 9\}, \quad |V| = l$$

that appear only in l cells of R:

$$S = \{k \in \mathrm{Remaining}(R) \mid \mathrm{Domain}(k) \cap V \neq \emptyset\}, \quad |S| = l.$$

Then these l cells in S must take the l values from V, and all other values can be removed from their domains:

$$\forall k \in S, \quad \mathrm{Prune}(k, \{1, \ldots, 9\} \setminus V).$$

An example is shown in Fig. 2.9.

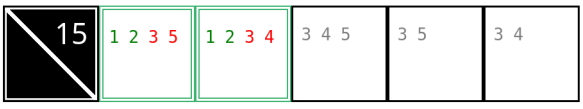

Figure 2.9: In this example, there are in total five distinct values across all domains, and the values 1 and 2 appear only in the highlighted cells. Therefore, these values must be assigned to them, and any other values in the highlighted domains can be removed.

2.5.6 Filter Incompatible Pairs

Consider a run R with exactly two unassigned cells k_a, k_b, and $S = \mathrm{RemainingSum}(R)$:

$$\mathrm{Remaining}(R) = \{k_a, k_b\}$$

We remove from $\mathrm{Domain}(k_a)$ any value v for which there is no complementary value $v' \in \mathrm{Domain}(k_b)$:

$$v' = S - v, \quad v' \neq v$$

Formally, for k_a:

$$\mathrm{Prune}\Big(k_a, \Big\{v \in \mathrm{Domain}(k_a) \mid \nexists v' \ : \ \begin{cases} v' \in \mathrm{Domain}(k_b) \\ v' \neq v \\ v' = S - v, \end{cases} \Big\}\Big)$$

and symmetrically for k_b. An example is shown in Fig. 2.10.

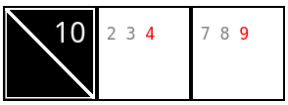

Figure 2.10: In this example, the run has two empty cells with sum 10. 4 can be removed from the first cell because its complement 6 is not in the second cell's domain, and 9 can be removed from the second cell because its complement 1 is missing in the first.

2.5.7 Filter Incompatible Triples

Analogous to Filter Incompatible Pairs, consider a run R with exactly three unassigned cells k_a, k_b, k_c, and $S = \mathrm{RemainingSum}(R)$:

$$\mathrm{Remaining}(R) = \{k_a, k_b, k_c\}$$

For each cell $k_x \in \{k_a, k_b, k_c\}$, we remove any value $v \in \mathrm{Domain}(k_x)$ for which there is no pair of distinct values

$$(v_1, v_2) \quad \text{with} \quad v_1 \in \mathrm{Domain}(k_y),\ v_2 \in \mathrm{Domain}(k_z),\ y \neq z \neq x$$

where:

$$v \neq v_1, \quad v \neq v_2, \quad v_1 \neq v_2, \quad v + v_1 + v_2 = S$$

Formally, for each k_x:

$$\mathrm{Prune}\Big(k_x, \Big\{v \in \mathrm{Domain}(k_x) \mid \nexists v_1, v_2 \ : \ \begin{cases} v_1 \in \mathrm{Domain}(k_y),\ v_2 \in \mathrm{Domain}(k_z) \\ v \neq v_1 \neq v_2 \\ v + v_1 + v_2 = S \end{cases} \Big\}\Big)$$

An example is shown in Fig. 2.11.

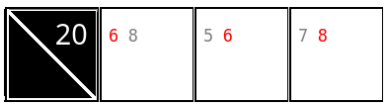

Figure 2.11: In this example, the run has three empty cells with sum 20. If the first cell contains 6, the other two must sum to 14 without repetition. Since this is not possible, 6 can be pruned. The same logic is applied to all cells in the run.

2.5.8 Divide-and-Conquer

This strategy is applicable to **visually identifiable patterns**: regions of the puzzle that appear as *islands*, connected to the rest of the puzzle through a single linking cell and bending immediately afterward. An example is shown in Fig. 2.12.

Formally, let $\mathcal{R}_v$ be a set of vertical runs and $\mathcal{R}_h$ a set of horizontal runs. Suppose their unions differ by exactly one empty cell:

$$\left|\left(\bigcup_{R\in\mathcal{R}_v} \text{Remaining}(R)\right) \Delta \left(\bigcup_{R\in\mathcal{R}_h} \text{Remaining}(R)\right)\right| = 1.$$

Let $k^\star$ be the unique cell in this symmetric difference ($k^\star$ belongs to exactly one of the two unions). Let C_v be the set of vertical clues governing the runs in $\mathcal{R}_v$, and C_h the set of horizontal clues governing the runs in $\mathcal{R}_h$. Compute the remaining sums:

$$S_v = \sum_{c\in C_v} \text{RemainingVSum}(c), \qquad S_h = \sum_{c\in C_h} \text{RemainingHSum}(c).$$

Then the value of $k^\star$ is determined by the absolute difference

$$v^\star = |S_v - S_h|.$$

$$\text{AssignAndPropagate}(k^\star, v^\star).$$

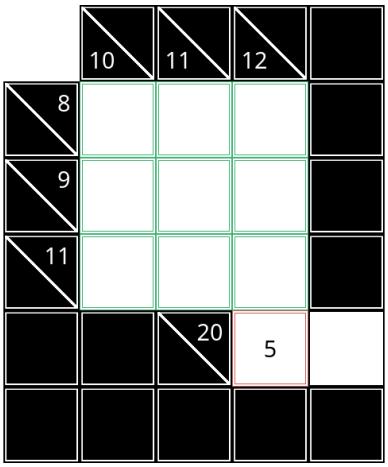

Figure 2.12: In this example, the red-highlighted cell is the unique element in the symmetric difference between the three vertical clues and the three horizontal clues governing the area. Its value can therefore be calculated directly from the difference of the clue sums: $(10 + 11 + 12) - (8 + 9 + 11) = 5$.

After applying this strategy, the different regions of the puzzle can be treated as two distinct Kakuro sub-puzzles. For this reason, the method is referred to as *Divide-and-Conquer*.

Note that the principle of Divide-and-Conquer can also be applied when the symmetric difference involves more than one cell (e.g., two or three), in combination with *Filter Incompatible Pairs* and *Filter Incompatible Triples*.

If the unions of $\mathcal{R}_v$ and $\mathcal{R}_h$ differ by $m \in \{2, 3\}$ empty cells, and these m cells all belong to one side of the symmetric difference (i.e., either only to the vertical group or only to the horizontal group), then the absolute difference of the clue sums specifies the total to which these cells must conform. Formally:

$$\left| \left(\bigcup_{R \in \mathcal{R}_v} \text{Remaining}(R) \right) \Delta \left(\bigcup_{R \in \mathcal{R}_h} \text{Remaining}(R) \right) \right| = m, \quad m \in \{2, 3\}.$$

Let $\{k_1, \ldots, k_m\}$ be the m cells in the symmetric difference. Then

$$k_1 + \cdots + k_m = |S_v - S_h|.$$

In this case, direct assignment of a single value is not possible; instead, the domains of the involved cells $\{k_1, \ldots, k_m\}$ can be reduced by excluding combinations that cannot achieve the derived required total.

2.6 Summary of Human Solving Techniques

Human solving techniques for Kakuro can be categorized into three main groups: *assignment techniques*, *propagation techniques*, and *domain reduction techniques*.

Assignment Techniques

These techniques assign values directly to cells.

- **Naked Singles**: Assign a value to a cell when its domain contains only a single possible value.
- **Hidden Singles**: Assign a value that must appear in a run and can occur in only one cell within that run.
- **Divide-and-Conquer Singles**: A special case where a cell's value can be determined directly if it is the only value in the symmetric difference between a set of vertical and a set of horizontal runs.

For these strategies, we assign the value and propagates the update to all directly affected cells within the same horizontal and vertical runs.

Propagation Techniques

These techniques refine the domains of unassigned cells after each assignment to reduce the search space.

- **No-Repetition Constraint**: Remove from a cell's domain any values already assigned in its horizontal or vertical runs.
- **Lower-Upper Bounds**: Restrict a cell's domain to values that could potentially satisfy the remaining sum of its run, considering minimal and maximal contributions from other unassigned cells.
- **Magic Runs**: For runs with only one valid combination of values, prune all numbers not included in this combination, leaving only the permutation to be determined.
- **Intersection**: For cells belonging to both a horizontal and vertical run, restrict the domain to values allowed by both runs by taking the intersection of candidate sets.

Domain Reduction Techniques

These techniques prune infeasible values from cell domains.

- **Naked Subsets**: When n cells share exactly n values, preserve these values for these cells and eliminate them from all other cells in the same run.
- **Hidden Subsets**: When n values must appear in a run and can occur only in n cells, eliminate all other values from these n cells.
- **Filter Incompatible Pairs/Triples**: Remove values that have no supporting values in other cells within short runs of two or three cells, respectively.
- **Divide-and-Conquer Subsets**: Combine the Divide-and-Conquer technique with filtering of incompatible pairs/triples when the symmetric difference involves more than one cell.

Note: Propagation techniques are not treated as standalone strategies; rather, they are applied immediately following an assignment to propagate constraints and update the domains of affected cells.

3 Solving Approaches

There are different algorithmic approaches to solving Kakuro puzzles, ranging from naive brute-force techniques to more advanced strategies. This chapter provides an overview of the implementation of some of these methods[1].

3.1 Backtracking Solver

Backtracking is a classical method for solving combinatorial problems. It explores possible assignments for empty cells in a depth-first search process recursively (Algorithm 1).

At each step, it picks an unassigned cell and iterates over its candidate values (Domain)[2]. Whenever a value is placed, the algorithm checks whether the assignment is still consistent with the corresponding horizontal and vertical clues.
If the assignment remains consistent[3], the solver proceeds recursively; otherwise, it backtracks by undoing the last assignment and testing the next value in the domain.

Algorithm 1: Kakuro: Backtracking

```
1  Function Backtrack(Board K):
2      if ∀k ∈ Empty(K) : Value(k) ≠ ⊥ then
3          return true                                   // All empty cells assigned
4
5      Pick k ∈ Empty(K) where Value(k) = ⊥      // Cells in Empty(K) are ordered row-wise
6      foreach v ∈ Domain(k) do
7          Value(k) ← v                                  // Assign value v to cell k
8          if k is consistent with HClue(k) and VClue(k) then
9              return Backtrack(K)
10
11         Value(k) ← ⊥                                  // Backtrack
12
13     return false                                      // No valid assignment found for this cell
```

[1] Git Repository: https://github.com/AEA1212/KakuroDS

[2] The domain can initially be defined as the full set of numbers from 1 to 9. However, we can apply one round of domain narrowing, as introduced in Chapter 2 (see Section 2.5.1), to reduce the search space before the backtracking process begins.

[3] Consistency checking ensures that the current assignment does not violate the puzzle constraints. In its simplest form, this check only detects violations once a run is fully assigned and the sum does not match or a duplicate value occurs. However, we implement early termination by verifying whether the remaining unassigned cells can still achieve the required sum. For example, if 3 cells remain not assigned and the remaining sum is 4, it is not possible to make a valid assignment since the sum of the three lowest distinct values (1, 2 and 3) cannot equal 4. In this case, we backtrack.

The concept of backtracking is simple and easy to implement. However, Kakuro is NP-Complete [8]; therefore, the time required to solve an instance can grow exponentially with the number of empty cells to be filled.

To address this issue, we use the well-known pruning technique, "*forward checking*", which involves propagating constraints after each assignment. The trick is to make use of Domain Narrowing Rules (see Section 2.5.1) by propagating the consequences of each assignment, i.e., by restricting the domains of all other cells in the same horizontal and vertical runs as the assigned cell. If an assignment is to be backtracked later, the effects of the propagation must also be undone.[4]

Forward checking helps to avoid exploring infeasible branches early in the search, as illustrated in Algorithm 2. This reduction may improve the solver's performance in practice.

Algorithm 2: Kakuro: Forward Checking Backtracking

```
1  Function ForwardCheck(Board K):
2      if ∀k ∈ Empty(K) Value(k) ≠ ⊥ then
3          return true                                   // All empty cells assigned
4
5      Pick k ∈ Empty(K) where Value(k) = ⊥
6      foreach v ∈ Domain(k) do
7          Value(k) ← v                                  // Assign value v to cell k
8          foreach k′ ∈ RemainingHRun(HClue(k)) ∪ RemainingVRun(VClue(k)) do
9              RecalculateDomain(k′)                     // Narrowing Domain rules (Sec. 2.5.1)
10
11         if ∀k ∈ Empty(K) Domain(k) ≠ ∅                // consistent
12          then
13             return ForwardCheck(K)
14
15         Value(k) ← ⊥                                  // Backtrack
16         foreach k′ ∈ RemainingHRun(HClue(k)) ∪ RemainingVRun(VClue(k)) do
17             undo RecalculateDomain(k′)                // Backtrack
18
19     return false                                      // No valid assignment found for this cell
```

[4] In our implementation, we take a naive approach by cloning the board before each assignment. This may cause overhead and could be optimized, for instance, by saving and restoring only the affected cells, but our design is constrained by earlier architectural choices.

3.2 CP Solver

Another approach for solving Kakuro puzzles is to express the problem as a *Constraint Satisfaction Problem* (see Section 2.4) and solve it using *constraint programming* solvers. Each empty cell is represented as a variable, and the puzzle rules are expressed as constraints on these variables: the all-different constraints and sum constraints for each run.

Unlike the backtracking approach, the CP solver does not require us to implement the search procedure manually. Instead, the solver applies pre-implemented techniques such as constraint propagation and intelligent branching heuristics to prune the search space and accelerate the solving process, as outlined in Algorithm 3.

we use two variants of solvers provided by Google OR-Tools:

- Original CP Solver[5]
- CP-SAT Solver[6]

Algorithm 3: Kakuro: CP(-SAT) Solver

Function `Solve`(*Board* K)**:**
  Initialize the model M // original CP-Solver or CP-SAT

  foreach $k \in \mathrm{Empty}(K)$ **do**
    Create variable $x_k \in \{1, \ldots, 9\}$ // Domain of each empty cell

  foreach $c \in \mathrm{Clues}(K)$ **do**
    if $\mathrm{HRun}(c) \neq \emptyset$ **then**
      Add constraint: $\textsc{AllDifferent}(\{x_k \mid k \in \mathrm{HRun}(c)\})$
      Add constraint: $\sum_{k \in \mathrm{HRun}(c)} x_k = \mathrm{HSum}(c)$
    if $\mathrm{VRun}(c) \neq \emptyset$ **then**
      Add constraint: $\textsc{AllDifferent}(\{x_k \mid k \in \mathrm{VRun}(c)\})$
      Add constraint: $\sum_{k \in \mathrm{VRun}(c)} x_k = \mathrm{VSum}(c)$

  Solve model M

  if *solution found* **then**
    return true // Solution exists
  else
    return false // No valid solution found

[5] https://developers.google.com/optimization/routing/original_cp_solver
[6] https://developers.google.com/optimization/cp/cp_solver

3.3 Deductive Search-Based Solver

We adapt the method demonstrated by Browne [3], applying a layered deduction method to progressively prune the domains of the cells in Kakuro puzzles without immediately resorting to search. The solver alternates between *levels of embedding* (LoEs) in a fallback-driven manner, attempting deductions at shallower levels first and only cascading to deeper levels when the current depth is insufficient to yield progress. During the deduction process, the status of a board K can be one of the following:

1. UNSOLVED: No solution has been proven or disproven yet.
2. SOLVED: A solution has been deduced.
3. CONTRADICTION: No possible solution exists (a domain of an empty cell is empty).
4. NON_DEDUCIBLE: No solution can be deduced with the current constraints and search depth.

It is noteworthy that the solver proceeds in a *strictly monotonic* manner: once a deduction is applied to the main board, it is never revoked [3]. Before going into this, let's define the following operations:

1. **Simplification:** the application of logical rules, as described in the Human Solver Techniques section (see Section 2.5), including arithmetic and pattern-based strategies. Simplification prunes inconsistent values, and assigns forced values wherever possible.

Algorithm 4: Kakuro: Simplification Strategies

```
1  Function SIMPLIFY(Board K):
2      while ΔK ≠ ∅ (as long as simplified) do
3          if PROCESSNAKEDSINGLES(K) > 0 then
4              continue                                   // Begin from the top again
5          if FILTERINCOMPATIBLEPAIRS(K) > 0 then
6              continue
7          if DIVIDEANDCONQUER(K) > 0 then
8              continue
9          if FILTERINCOMPATIBLETRIPLES(K) > 0 then
10             continue
11         if PROCESSHIDDENSINGLES(K) > 0 then
12             continue
13         if PROCESSNAKEDSUBSETS(K, 2 [naked doubles]) > 0 then
14             continue
15         if PROCESSHIDDENSUBSETS(K, 2 [hidden doubles]) > 0 then
16             continue
17         if PROCESSNAKEDSUBSETS(K, 3 [naked triples]) > 0 then
18             continue
19         if PROCESSHIDDENSUBSETS(K, 3 [hidden triples]) > 0 then
20             continue
21         if PROCESSNAKEDSUBSETS(K, 4 [naked quadruples]) > 0 then
22             continue
23         if PROCESSHIDDENSUBSETS(K, 4 [hidden quadruples]) > 0 then
24             continue
25     return K
```

Notice that the strategies in Algorithm 4 are ordered in ascending order of their rated difficulty. The solver does not proceed to a harder strategy until no further progress can be achieved from the easier ones. This cycle is repeated until no strategy can achieve additional progress.[7]

Observing the human solving process shows that solvers tend to start with short runs within the same strategy. Therefore, we apply each strategy to the smallest applicable run first.

2. **Shaving:** This operation is performed in a breadth-first, depth-limited search. a value from a cell's domain is *hypothetically* assigned, and then the consequences are propagated using `SIMPLIFY` [3].

   Two outcomes are possible from the hypothetical assignment:

   - If the assignment and its following simplification does not cause inconsistency, the value remains in the domain of the cell, but we still can not assign it, because the correctness is not yet guaranteed, since the search is not necessarily complete.
   - If the assignment leads to a contradiction or inconsistency (e.g., the domain of an unassigned cell becomes empty), the value can be immediately pruned from the cell's domain.

   Formally:

$$\Big(\exists v \in \mathrm{Domain}(k) : \mathrm{AssignAndPropagate}(k, v) \Rightarrow \bot\,(\mathsf{CONTRADICTION})\Big) \implies \mathrm{Prune}(k, \{v\})$$

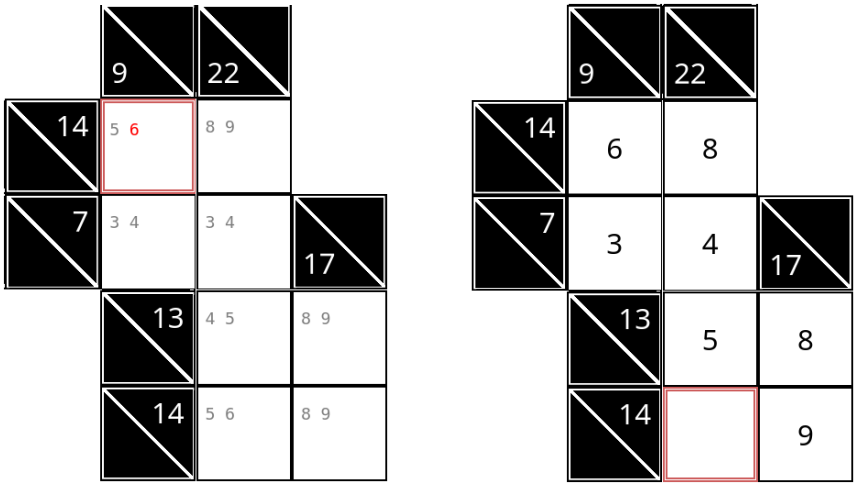

Figure 3.1: If the value 6 is assigned to the highlighted cell in the left board, simplification produces the situation on the right, in which the red-highlighted cell has an empty domain. This proves that 6 is an invalid assignment and must be pruned from the domain.

[7] This ordering is not based on any evidence, but rather on our judgment of cognitive effort; a precise evaluation and validation of the difficulty of each strategy are left to future work. Also, the size of the subset strategies being performed is a hyperparameter. We think that subset strategies involving more than 4 cells would be very complicated for players to consider cognitively.

3. **Agreement:** This strategy [3] is applied *simultaneously* with shaving and collects the effects of hypothetical assignments on other cells. It can be used in two ways:

   a) **Agreement on assignment:** If trying all values in the domain of a cell k forces a specific value v' to be assigned to another cell k', we can assign v' to k' directly. Formally:

$$\Big(\forall v \in \text{Domain}(k) : \text{AssignAndPropagate}(k, v) \Rightarrow (\text{Value}(k') \leftarrow v')\Big) \\ \implies \text{AssignAndPropagate}(k', v')$$

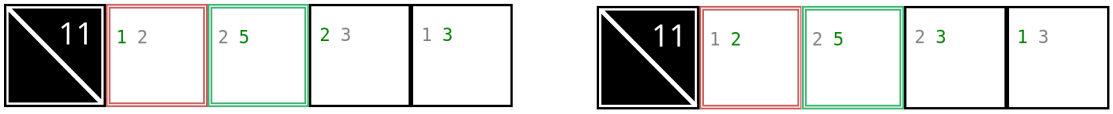

Figure 3.2: In this example, all values in the domain of the red-highlighted cell *agree* on the value 5 for the green-highlighted cell, this means every value assigned to the red cell forces the green cell to be assigned the value 5 after applying `SIMPLIFY`.

The value 2 could also be pruned from the domain of the green-highlighted cell using **Shaving**, but in our implementation, we process **Shaving** and **Agreement** simultaneously, so we process Agreement on the red-highlighted cell before we start Shaving at the green-highlighted one.

   b) **Agreement on pruning:** If the assignment of every value v in the domain of a cell k results in a value v' always missing in the domain of a cell k', then v' can be pruned from k'. Formally:

$$\Big(\forall v \in \text{Domain}(k)\ \exists v' \in V' : \text{AssignAndPropagate}(k, v) \Rightarrow (\text{Value}(k') \leftarrow v')\Big) \\ \implies \text{Prune}(k', \text{Domain}(k') \setminus V')$$

Figure 3.3: In this example, assigning any value to the red-highlighted cell forces the green-highlighted cell to take a value other than 4. Therefore, the value 4 can be safely pruned from its domain.

The following pseudocode (Algorithm 5) illustrates how the function `DEDUCE` operates, applying **Shaving** and **Agreement** at a given LoE [3].

Algorithm 5: Kakuro: Deduction

```
1  Function DEDUCE(Board K, depth):
2
3      foreach k ∈ Empty(K) do
4          if Value(k) = ⊥ ∧ Domain(k) ≠ ∅ then
5              agreement ← ∅                                   // map: cell ↦ union of assignments
6
                // shaving and collect information for agreement
7              foreach v ∈ Domain(k) do
8                  K′ ← clone(K)                               // hypothetical board to try value v
9                  K′ ← AssignAndPropagate(k, v)
10                 SIMPLIFY(K′)
11
12                 if STATUS(K′) = CONTRADICTION then
13                     K ← Prune(k, {v})                       // shaving in the original board
14                     SIMPLIFY(K)
15                 else
16                     if depth > 1 then
17                         K′ ← DEDUCE(K′, depth − 1)           // next LoE
18
                        // Collect agreement info (⊥ if unassigned)
19                     foreach k′ ∈ Empty(K′) do
20                         agreement(k′) ← agreement(k′) ∪ Value_K′(k′)
21
                // Apply agreement from collected info
22             foreach k′ ∈ Empty(K) : do
23                 if ⊥ ∉ agreement(k′)                        // all assignments of k agree
24                 then
25                     if |agreement(k′)| = 1                  // 1. agreement on assignment (forced value)
26                     then
27                         K ← AssignAndPropagate(k′, agreement(k′))
28                         SIMPLIFY(K)
29                     if |agreement(k′)| > 1                  // 2. agreement on pruning (possible values)
30                     then
31                         K ← Prune(k′, Domain(k′) \ agreement(k′))
32                         SIMPLIFY(K)
33
                // Early escape if depth > 1 and update occurred
34             if depth > 1 ∧ ΔK ≠ ∅ then
35                 return K
36     return K
```

To apply the deduction process, DS function begins with a round of `SIMPLIFY` at 0-LoE, applying deterministic strategies to reduce the domains of cells.

If the puzzle remains unsolved, the solver proceeds to `DEDUCE` at 1-LoE. Within `DEDUCE`, both **Shaving** and **Agreement** are applied, followed by simplification steps to further refine the board through assignments or pruning of values.If progress stalls at 1-LoE, the solver escalates to 2-LoE, recursively applying deduction.

This process continues iteratively until either no further changes occur ($\Delta K = \emptyset$) or the board is fully solved, as shown in Algorithm 6 [3].

Algorithm 6: Kakuro: Deductive Search-Based Solver

```
1  Function DS(Board K):
2
3      K ← SIMPLIFY(K)                                          // 0-LoE
4
5      while ΔK ≠ ∅ and STATUS(K) = UNSOLVED do
6          while ΔK ≠ ∅ and STATUS(K) = UNSOLVED do
7              K ← DEDUCE(K, 1)                                 // 1-LoE
8
9          if STATUS(K) = UNSOLVED then
10             K ← DEDUCE(K, 2)                                 // 2-LoE
11
12     if STATUS(K) = UNSOLVED then
13         return NON_DEDUCIBLE
14
15     if STATUS(K) = SOLVED then
16         return SOLVED
```

3.4 Hybrid Solver: Forward Checking with Simplification

Another solving technique for Kakuro is a hybrid approach that combines *backtracking* with *DS*. At the beginning of each recursive call of `SimplifyAndForwardCheck` (Algorithm 7), we apply `SIMPLIFY` to prune domains before continuing the backtracking process.

This approach is equivalent to DS for instances solvable at 0-LoE. For instances requiring deeper deduction, instead of performing shaving and agreement as in DS, we perform a backtracking with forward checking using `SIMPLIFY` for propagation. This allows constraints to propagate more efficiently than in a standard forward checking solver (see Algorithm 2)

Algorithm 7: Kakuro: Forward Checking with Simplification

```
1  Function SimplifyAndForwardCheck(Board K):
2      SIMPLIFY(K)
3
4      if ∀k ∈ Empty(K) Value(k) ≠ ⊥ then
5          return true                                   // All empty cells assigned
6
7      Pick k ∈ Empty(K) where Value(k) = ⊥
8      foreach v ∈ Domain(k) do
9          Value(k) ← v                                  // Assign value v to cell k
10         foreach k′ ∈ RemainingHRun(HClue(k)) ∪ RemainingVRun(VClue(k)) do
11             recalculateDomain(k′)                     // Narrowing the Domain rules (Sec. 2.5.1)
12
13         if ∀k ∈ Empty(K) Domain(k) ≠ ∅                // consistent
14          then
15             return SimplifyAndForwardCheck(K)
16
17         Value(k) ← ⊥                                  // Backtrack
18         foreach k′ ∈ RemainingHRun(HClue(k)) ∪ RemainingVRun(VClue(k)) do
19             restoreDomain(k′)                         // Backtrack
20
21
22     return false                                      // No valid assignment found for this cell
```

4 Comparative Evaluation

4.1 Methods

In this chapter, we compare all Kakuro solvers from chapter 3:

1. **Backtracking Solver (BT)**
2. **Forward Checking Solver (FC)**
3. **OR-Tools Original CP Solver (CP)**
4. **OR-Tools CP-SAT Solver (CP-SAT)**
5. **Deductive Search-Based Solver (DS)**
6. **Hybrid Solver: Forward Checking with Simplification (FC-Simplify)**

4.1.1 Dataset

In our experiments, we used instances from **Janko.at**[1] and **Nikoli**[2], extracted from multiple book collections, including *Fresh*, *Penpa*, and *Tokoton*. In total, we considered 2,656 instances, varying in size and difficulty.

4.1.2 Comparison Criteria

The comparison criteria include:

- **Performance / Runtime:** Each solver is run on all instances with a fixed time limit to ensure reasonable runtime.
- **Soundness, Completeness, and Refutational Completeness:** evaluate whether a solver:
  - always returns correct solutions (*soundness*).
  - can find all valid solutions (*completeness*).
  - can correctly detect unsolvable instances (*refutational completeness*)
- **Transparency and Explainability:** The extent to which the solver's reasoning process and decisions can be understood and explained.

[1] https://www.janko.at/Raetsel/Kakuro/index.htm
[2] https://www.nikoli.co.jp/en/puzzles/kakuro/

4.2 Results

4.2.1 Experimental setup

All experiments were carried out on a computer equipped with an AMD Ryzen 7 5800H CPU (Mobile) and running Ubuntu 24.04 LTS. All solvers were executed in a single-threaded mode. The solvers were implemented in C++20 and built using CMake 3.28 with g++ 13.3. For the CP solver, the Google OR-Tools library (v9.11.4210) was used. C++ was chosen for its efficiency, comprehensive standard library, and robust debugging support through GDB.

4.2.2 Performance / Runtime

Table 4.1 summarizes runtime performance across all solvers. It reports the number of solved instances, overall solving rate (using a 5-second timeout per instance), and basic statistics of solver runtimes.

Table 4.1: Performance comparison of solvers over the full dataset.

Solver	Solved	(%)	Mean Time (s)	Median Time (s)	Min Time (s)	Max Time (s)
BT	2294	86.37	0.3301	0.0331	0.00009	4.88
FC	2522	94.95	0.2941	0.0954	0.00104	4.93
CP	2656	100.00	0.0010	0.0002	0.00003	1.27
CP-SAT	2656	100.00	0.0021	0.0017	0.00031	0.03
DS	2655	99.96	0.0041	0.0024	0.00014	1.19
FC-Simplify	2655	99.96	0.0036	0.0024	0.00015	0.79

Note: The reported statistics include only instances that were solved within the 5-second timeout.

Table 4.1 provides an aggregated overview, but it does not capture the distribution of runtimes across individual instances. Figure 4.1 shows the cumulative number of solved instances as a function of runtime, which provides a clearer picture of the performance of each solver.

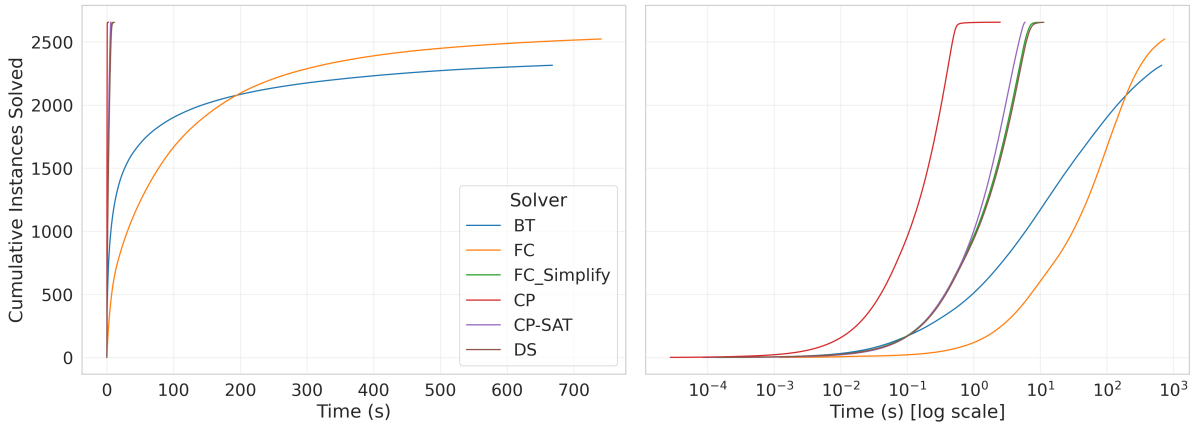

Figure 4.1: Cumulative number of instances (sorted in ascending order by runtime) solved as a function of cumulative runtime for different solvers. A linear scale for an overview (left), and a logarithmic scale on the x-axis to highlight differences in runtime more clearly (right).

4.2.3 Soundness and Completeness

Table 4.2: Number of instances per solver: correct, incorrect, unable to solve, and timed out.

Solver	correct solution	incorrect solution	unable to solve	timed-out (> 5 s)
BT	2294	0	0	362
FC	2522	0	0	134
CP	2656	0	0	0
CP-SAT	2656	0	0	0
DS	2655	0	1	0
FC-Simplify	2655	0	0	1

All tested instances from the dataset are solvable, and all except one has a unique solution. The exception is a puzzle from Janko that has two valid solutions, and it is the one that could not be fully solved using DS method (Table 4.2).

BT, FC, and FC-Simplify were unable to complete all instances within the time limit. However, by design, these solvers are capable of solving all solvable instances, including those with multiple valid solutions, given enough time.

Soundness: All proposed methods never return an incorrect solution, so we can consider them to be *sound*.

Completeness: DS method is not complete for the instances with multiple solutions, as it cannot deduce any of the valid solutions. It can only work with valid problems that have a unique solution [3]. In contrast, BT, FC, and FC-Simplify are designed to return the first valid solution. They can be extended to perform a full search to enumerate all valid solutions. CP and CP-SAT likewise can solve any solvable instance (with unique or multiple valid solutions). Therefore, all methods except DS can be considered *complete*.

Refutational Completeness: BT, FC, FC-Simplify, CP, and CP-SAT can detect unsolvable instances. By executing a complete search, each of these solvers is guaranteed to conclude that no solution exists. For DS, the situation is more subtle: if a direct inconsistency is derived (e.g., an empty domain arises during reasoning), the method can correctly report that the puzzle is invalid. However, if the reasoning reaches a `NON_DEDUCIBLE` state under a given limited depth, such as 2-LoE, the solver cannot determine whether the instance is unsolvable or merely beyond its deductive ability. In that sense, DS lacks refutational completeness.

4.2.4 Transparency and Explainability

BT and FC are classical depth-first search-based approaches. Since the search space grows **exponentially** with the size of the problem, it can quickly become too large and infeasible for humans to track. In Kakuro, the depth of the search tree corresponds to the number of empty cells. Although FC-Simplify makes use of simplification strategies, it is still fundamentally a backtracking method. Consequently, it can produce deep branches that are difficult for humans to track.

CP and CP-SAT benefit from the formalism of constraint programming. They allow some degree of introspection through solver logs. However, the level of explainability remains largely **technical** and **solver-centric** rather than human-centric. While these solvers are capable of employing very powerful solving techniques and providing detailed statistics about the strategies applied, they primarily focus on efficiency and optimality rather than on interpretability and alignment with human limitations.

In contrast, the DS method is designed to emulate human reasoning. Its deductive rules correspond to recognizable logical steps, enabling a natural mapping between solver actions and human-understandable explanations. As shown in Figure 4.2, most instances from different sources are solvable at 0-LoE by applying the implemented logical strategies in the Simplification step, without the need for any search.

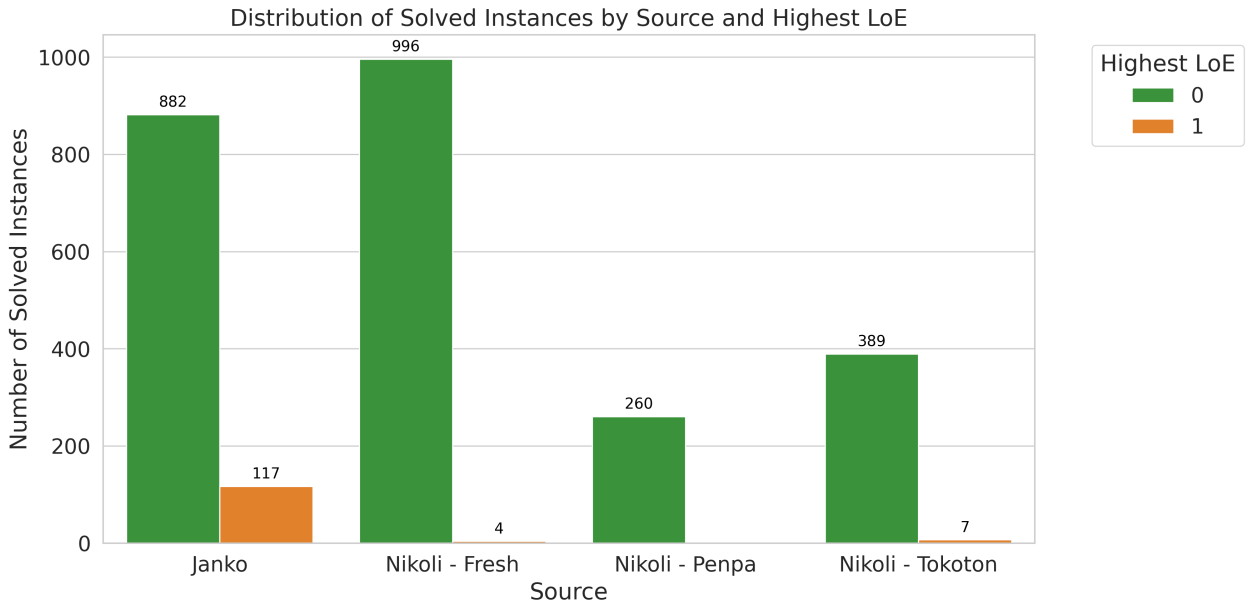

Figure 4.2: Number of solved instances by source and highest LoE reached (DS solver).

Moreover, the distribution of the total number of steps taken by the DS solver is shown in Figure 4.3. Compared to backtracking, the number of steps (applied strategies) remains limited and does not grow exponentially. Even for very large instances (e.g. 990 empty cells), the total number of applied strategies increases linearly with the number of empty cells, as shown in the figure 4.4.

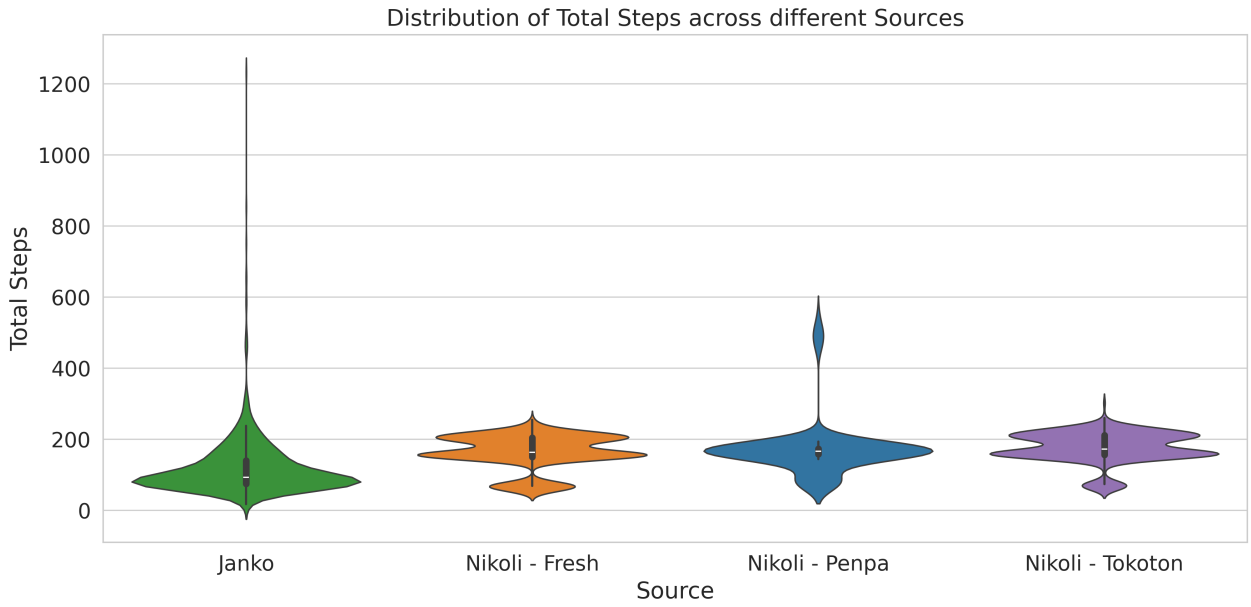

Figure 4.3: Distribution of total steps required to solve instances by the DS solver across different sources.

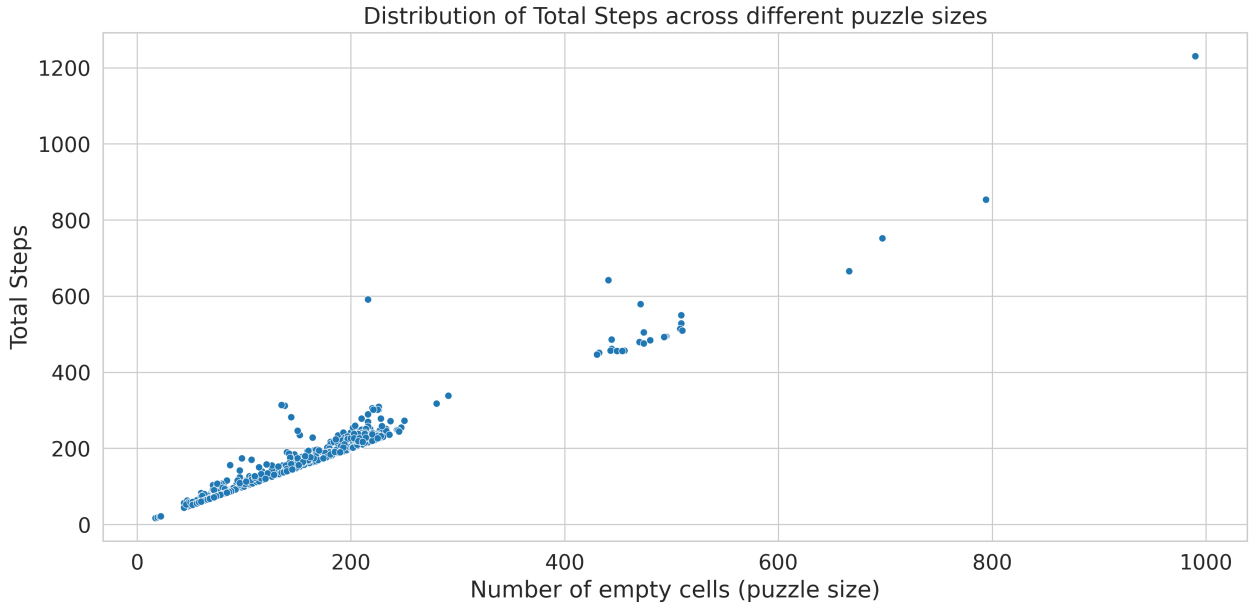

Figure 4.4: Distribution of total steps required to solve instances by the DS solver across different sizes.

4.3 Discussion

The results highlight the differences in terms of performance, completeness, and explainability of the Kakuro solvers. CP and CP-SAT are the fastest, benefiting from sophisticated architectures and advanced optimization techniques. The DS solver is also fast and lightweight; relying solely on basic logical strategies, it can handle large puzzles within a reasonable time. In contrast, classical search-based solvers, such as BT and FC, struggle with larger puzzles due to exponential growth in the search space. The hybrid solver FC-Simplify, however, gains a significant advantage from the simplification strategies, achieving an efficiency close to that of DS.

All solvers are sound and produce only correct solutions. Completeness varies: BT, FC, FC-Simplify, CP and CP-SAT are complete, whereas DS, given sufficient LoEs, is only complete for the puzzles with unique solutions. However, DS can be extended to handle puzzles with multiple solutions by customizing it so that it first deduces all forced cells using its deductive rules. Then, only the unsolvable region can be solved by using any of the other solvers. This hybrid strategy preserves the explainability of DS for uniquely determined parts, while making DS complete for all solvable instances, like the other solvers.

Explainability is also an exclusive strength of the deductive solver, providing step-by-step explanations that are understandable to humans. For this reason, we also implemented a web interface that parses solver logs and provides both visual and textual views for each solving step of a given instance.[3]

In summary, CP solvers are ideal when speed and completeness are critical, classical search methods are suitable only for small instances, and the DS solver is most effective when explainability is important, such as for educational or verification purposes.

[3] https://github.com/AEA1212/KakuroDS/tree/main/WebInterface

5 Difficulty Estimation

Puzzle instances vary greatly in how challenging they are for human solvers. Some can be solved through simple reasoning, while others require deeper inference and greater cognitive effort. To account for this variation, we aim to assign each puzzle a difficulty score based on the types of reasoning strategies involved and how complex they are to apply.

In this chapter, we use the DS Solver as an analytical tool to approximate human-perceived difficulty by analyzing the solver's deduction steps and their logical complexity.

5.1 Methods

5.1.1 Dataset

In our experiments, we used the same puzzle instances as in Chapter 4 to evaluate the difficulty measures. Some relevant information regarding the difficulty of these instances is summarized below:

- **Janko.at:** 1000 instances ranging in difficulty from level 2 to 8, plus an additional extremely hard level (*schwer*), which we treat as level 9 for consistency.
- **Nikoli collections:** extracted from multiple book collections, including:
  - **Fresh 1-10:** 1000 instances, levels 1 to 10, Some instances at the beginning were unclassified; for consistency, we assign them level 0.
  - **Penpa 2011-2023:** 260 instances, levels 1 to 4.
  - **Tokoton 1-4:** 396 instances, levels 1 to 32.

Note that the difficulty ratings for **Janko.at** instances are based on the average solving time by players, while for **Nikoli Tokoton**, the ratings are based on the estimated solving time.

5.1.2 Metrics

To estimate puzzle difficulty, we consider the sequence of steps taken by the DS Solver to deduce a solution. We record the steps in *chronological order*, assuming that at each step the solver applies the easiest possible strategy at the shallowest possible LoE. Then, we use the following metrics of each step to approximate the cognitive effort required to solve the instance:

- **Strategy:** The solving strategy applied in the step (associated with a cost, which depends on the type of the strategy).
- **LoE:** The depth at which the step is applied. Greater depth indicates higher cognitive effort.

- **Number of pruned values:** The reduction of the search space resulting from the step.
- **Instance size:** The total number of candidate values across all empty cells in the initial puzzle state, used as a normalization factor.

We present three methods for estimating puzzle difficulty, where each method extends the previous one:

1. **Browne-Difficulty**: A baseline measure of difficulty based on DS [3].
2. **Strategies-Cost-Difficulty**: Extends Browne-Difficulty by assigning a cost to each solving strategy.
3. **Causal-Cost-Difficulty**: Further extends Strategies-Cost-Difficulty by incorporating the temporal order of solving steps.

Browne-Difficulty:

Browne [3] proposed this formula to calculate difficulty of instance K:

$$\text{diff}_{Browne}(K) = \frac{S_0 + 4\,(S_1 + V_1 + A_1) + 9\,(S_2 + V_2 + A_2)}{\sum_{k \in \text{Empty(K)}} |\text{Domain}(k)|}$$

where S_n, V_n, and A_n are the number of updates due to simplification, shaving, and agreement, respectively, at n-LoE. The denominator serves as a normalization factor by the total number of domain values.

However, there is some ambiguity in Browne's description. Specifically, the meaning of *updates* is not entirely clear, as Browne mentions both the number of values pruned from domains and the number of assignments made interchangeably. We interpret it as referring to domain pruning.

Strategies-Cost-Difficulty

Browne's approach treated *Simplification* as a single operation, resulting in eliminating some values from the domains of the cells, considering only the LoE and the number of eliminated values from `SIMPLIFY`. A more fine-grained approach is to assign each strategy a *cost* (or weight) corresponding to the cognitive effort required to apply it [7].

Mathematically, the total weighted contribution of simplification at n-LoE can be expressed as a sum over all applied strategies:

$$S_n = \sum_{\text{strategy} \in \text{Strategies}} S_{n,\text{strategy}} \cdot \text{cost}(\text{strategy}),$$

where $S_{n,\text{strategy}}$ denotes the number of values pruned by the given strategy at level n, and $\text{cost}(\text{strategy})$ is its associated cognitive cost.

As reference values for cost, we use the following assignments in Table 5.1.

Table 5.1: Reference costs for each strategy.

Strategy (Simplification)	**Cost**
Easy	1.0
NakedSingle	
FilterIncompatiblePairs	
DivideAndConquerSingle	
DivideAndConquerSubset	×size factor
Medium	1.5
FilterIncompatibleTriples	
HiddenSingle	
NakedSubset	×size factor
HiddenSubset	×size factor
Strategy (Deduction)	**Cost**
Shaving	2.0
AgreementAssignment	2.0
AgreementPruning	2.0

For subset strategies, the base cost is multiplied by a size factor $\in [1, 2]$: Pair = 1.33, Triple = 1.667, Quadruple = 2.0.

Causal-Cost-Difficulty

While the cost-based difficulty extends Browne's formula and differentiates between the cognitive effort needed to apply each strategy, by assigning costs to them, it still treats the difficulty as a *flat sum* of weighted updates. This ignores the order in which updates occur during the solving process.

To address this, we define a *causal-cost-difficulty* measure, which incorporates both the strategy costs and the sequential dependencies of updates. The key idea is that each solving step contributes to difficulty not only through its immediate effect (number of values pruned, cost of the strategy, and embedding depth), but also through its causal position in the solving chain.

Formally, the causal difficulty is computed incrementally as follows:
Let $\alpha > 1$ denote the *causality factor*, which controls how much past solving effort carries forward into subsequent steps.

$$\text{current\_diff}_0(K) = 0,$$
$$\text{current\_diff}_i(K) = \text{current\_diff}_{i-1}(K) \cdot \alpha \; + \; \text{contribution}(s_i).$$

where the contribution of step s_i is

$$\text{contribution}(s_i) \; = \; w_{\text{LoE}(s_i)} \; \cdot \; \text{pruned}(s_i) \; \cdot \; \text{cost}(s_i) \; \cdot \; \text{size\_factor}(s_i).$$

where $w_{\text{LoE}(s_i)} \in \{1, 4, 9\}$ is the weight corresponding to the LoE in Browne's formula for step s_i.

Finally, we normalize as in Browne-Difficulty:

$$\text{diff}_{Causal}(K) = \frac{\text{current\_diff}_n(K)}{\sum_{k \in \text{Empty(K)}} |\text{Domain}(k)|},$$

This formulation extends the *Strategies-Cost-Difficulty* by preserving the *temporal order* of solving. Earlier steps causally influence later steps through the causality factor α.

To evaluate the proposed methods, we measure the correlation between the estimated difficulty using each method and the given difficulty levels from the original source (Janko/Nikoli). We use **Spearman correlation** rather than Pearson correlation, since our interest lies in the **rank-order** between predicted and given difficulty rather than assuming a linear relationship.

5.2 Results

Figure 5.1 shows the Spearman correlation between **Browne-Difficulty** and the difficulty levels assigned to each instance grouped by the source (constant), and likewise for **Strategies-Cost-Difficulty** (constant). It also illustrates how the correlation for **Causal-Cost-Difficulty** changes with the **causality factor** α.

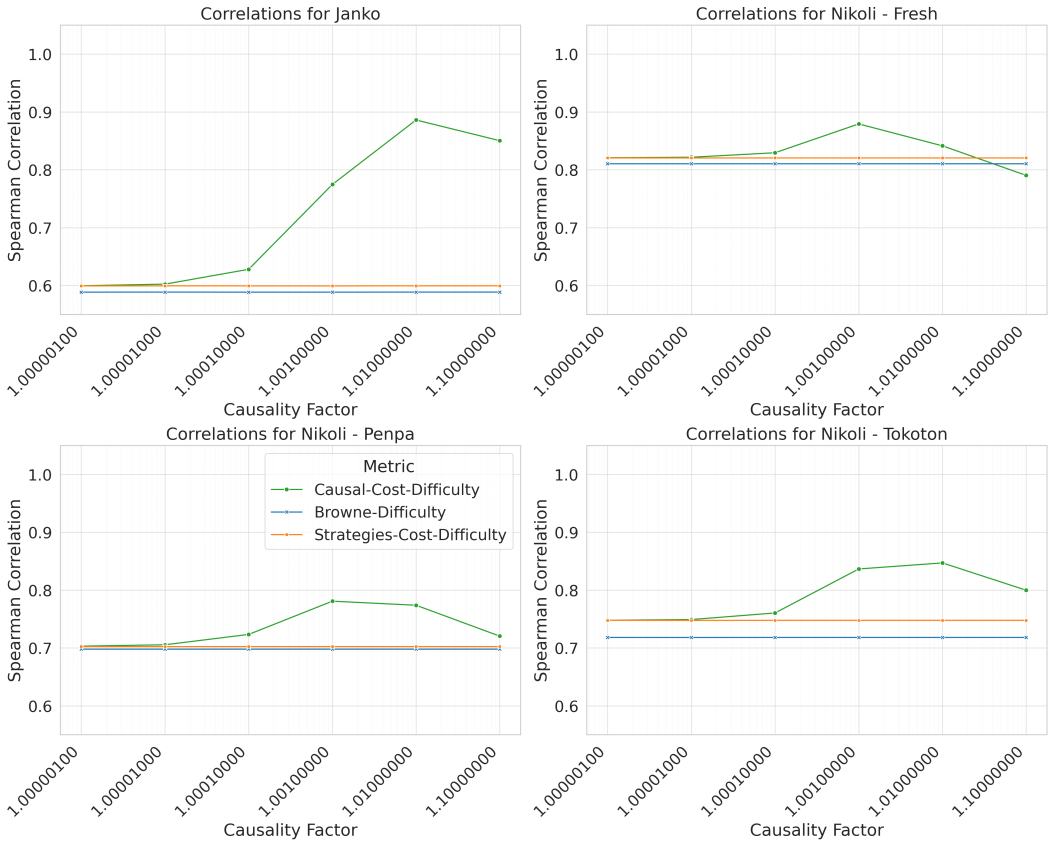

Figure 5.1: Spearman correlation between predicted difficulty and human ratings as a function of the causality factor α.

From Fig. 5.1, we can observe that **Cost-Difficulty** achieves slightly better correlation for both Janko and Nikoli instances. Furthermore, using **Causal-Cost-Difficulty** with a reasonable causality factor (e.g., $\alpha = 1.001$) enhances the correlation for all sources, as shown in Table 5.2.

Source	**Browne-Difficulty**	**Strategies-Cost-Difficulty**	**Causal-Cost-Difficulty** ($\alpha = 1.001$)
Janko	0.59	0.60	0.77
Nikoli - Fresh	0.81	0.82	0.88
Nikoli - Penpa	0.70	0.70	0.78
Nikoli - Tokoton	0.72	0.75	0.84

Table 5.2: Spearman correlations for different difficulty metrics across base sources.

While correlations provide an overview, we also examine the distribution of predicted difficulty across given levels. Figure 5.2 shows box plots of difficulty estimates across all sources. This visualization highlights how well the metrics separate puzzles of different given levels.

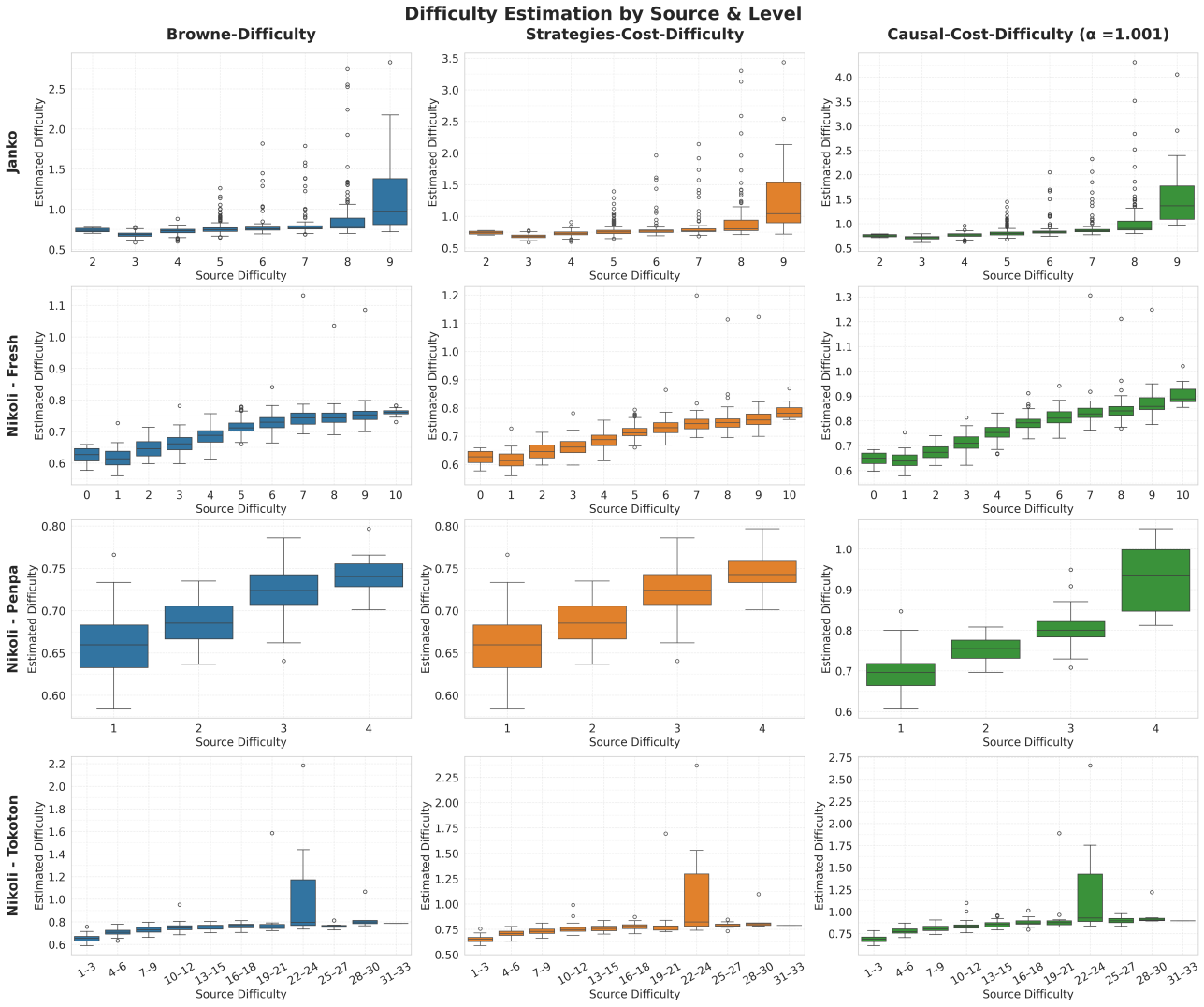

Figure 5.2: Difficulty distribution across levels for different sources.

From Fig. 5.2, we observe that **Causal-Cost-Difficulty** improves the separation of estimated difficulty levels, making the distributions more distinct across levels.

5.3 Discussion

The results show that Browne's baseline formula for difficulty estimation already correlates well with the difficulty levels assigned by the puzzle creators. This indicates that it can be reliably used as an automated method for assessing puzzle difficulty.

Incorporating **strategy costs** slightly improves the baseline Browne metric. However, the greatest improvement comes from introducing the **causality factor**, which accounts for the order in which strategies are applied during the solving process.

Furthermore, the metric is independent of the puzzle source, as it is based entirely on the strategies applied during solving, with features for each strategy reflecting its cognitive weight. Therefore, it can serve as a unified framework for Kakuro difficulty estimation.

In this framework, several hyperparameters were fixed to reasonable values, including the weights for each LoE (w_{LoE}), the costs of individual strategies, their ordering in the `SIMPLIFY` function, and the causality factor (α). These parameters could be further optimized to make the difficulty estimation more accurate.

6 Conclusion and Future Work

6.1 Deductive Search

6.1.1 Contributions

In this thesis, we applied Browne's DS method [3], which emulates human reasoning, to the well-known Kakuro puzzle.

We defined and formalized a set of simple strategies that can be applied by humans to solve such puzzles. A key observation is that `SIMPLIFY` plays a central role in this approach. If effective logical simplifications can be applied, many puzzles can be solved entirely at 0-LoE without the need for search or embedding points. Even when search is necessary, strong simplification significantly improves the efficiency of the propagation. This effect is directly reflected in the performance of deduction mechanisms such as shaving and agreement within `DEDUCE`.

6.1.2 Limitations and Future Research Directions

A critical point in the DS solver is that it propagates updates exhaustively, rather than limiting them to the local region around the update. This raises a counter-argument to the assumption of alignment with human reasoning: while each propagated update may be cognitively simple for humans to follow, the total number of propagation steps can be far greater than what a human would typically perform. On the other hand, restricting propagation to align with human reasoning more closely could reduce solver effectiveness and force deeper reasoning at higher LoEs. This trade-off between cognitive limitations of humans and solving efficiency is an important aspect that could be explored further.

In **shaving**, to disprove a candidate value, we assign it hypothetically and propagate constraints using `SIMPLIFY` until no further progress is possible. However, this process can be excessive for values that are difficult to disprove. A possible improvement is to limit the number of simplification loops applied to each candidate and then revisit them incrementally. This approach is analogous to the way the LoE increases only when the current level fails to yield further deductions. In other words, simplification within shaving could be applied in a breadth-first manner, gradually increasing the allowed number of loops as needed.

An open question is the degree to which the solver truly correlates with human reasoning. In this work, our research was limited by the lack of data on how humans actually solve Kakuro puzzles. For example, which regions they focus on first or which strategies they prioritize. These data would allow for a more precise evaluation of how closely DS aligns with human solving behavior.

It is also notable that DS is highly efficient in terms of runtime, despite not being explicitly optimized for this purpose. Further optimization could improve its speed and resource usage, presenting a promising direction for future work.

6.2 Difficulty Estimation

6.2.1 Contributions

In this work, we adapted Browne's difficulty estimation framework [3] to estimate the difficulty of Kakuro puzzles across a wide collection of instances collected from different sources and varying in difficulty levels. We also introduced two extensions to this method (*Strategies-Cost-Difficulty* and *Causal-Cost-Difficulty*). Both demonstrated improvements over the original approach.

6.2.2 Limitations and Future Research Directions

Our evaluation was based on analyzing the distribution of predicted difficulty levels as well as measuring the correlation with the levels provided by the puzzle creators. However, it is important to note that such provided levels by the puzzle creators can be subjective as well. An empirical evaluation through user studies would therefore be a valuable future contribution, providing stronger validation of difficulty estimation methods.

The costs assigned to the strategies (see Table 5.1) were classified subjectively into two categories: *easy* and *medium*. In addition, we introduced a size factor $\in [1, 2]$ for subset strategies, depending on whether they involve pairs, triples, or quadruples. We believe that refining these cost assignments, for example, by defining more distinct classes, adjusting the weighting scheme, and reordering the strategies within `SIMPLIFY` respectively, could further improve the accuracy of difficulty estimation.

List of Figures

2.1 Kakuro challenge and solution 4
2.2 Example for No Repetitions 7
2.3 Example for Lower-Upper Bounds 9
2.4 Example for Magic Run 9
2.5 Example for Intersection 10
2.6 Example for Naked Single 10
2.7 Example for Hidden Single 11
2.8 Example for Naked Subset 12
2.9 Example for Hidden Subset 12
2.10 Example for Incompatible Pair 13
2.11 Example for Incompatible Triple 13
2.12 Example for Divide-and-Conquer Strategy 14

3.1 Example for Shaving 21
3.2 Example for Agreement on assignment 22
3.3 Example for Agreement on pruning 22

4.1 Solver performance over time 27
4.2 Instances solved by source and LoE (DS solver) 29
4.3 Required steps to solve instances by source (DS solver) 30
4.4 Required steps to solve instances by the number of the empty cells of the instance (DS solver) 30

5.1 Effect of causality factor on correlation 35
5.2 Difficulty distributions across sources 36

List of Algorithms

1 Kakuro: Backtracking 17
2 Kakuro: Forward Checking Backtracking 18
3 Kakuro: CP(-SAT) Solver 19
4 Kakuro: Simplification Strategies 20
5 Kakuro: Deduction 23
6 Kakuro: Deductive Search-Based Solver 24
7 Kakuro: Forward Checking with Simplification 25

List of Tables

4.1 Performance comparison of solvers over the full dataset. 27
4.2 Number of instances per solver: correct, incorrect, unable to solve, and timed out. . . 28

5.1 Reference costs for each strategy. 34
5.2 Spearman correlations for different difficulty metrics across base sources. 36

Bibliography

[1] Denis Berthier. *Pattern-Based Constraint Satisfaction and Logic Puzzles*. Lulu Publishers, 2012.

[2] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research*, 119(3):557–581, 1999.

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

[4] Tristan Cazenave. Monte-carlo kakuro. In *Advances in Computer Games*, pages 45–54. Springer, 2009.

[5] Ryan P Davies, Paul A Roach, and Stephanie Perkins. The use of problem domain information in the automated solution of kakuro puzzles. *IAENG International Journal of Computer Science*, 37(2), 2010.

[6] Huw Lloyd, Matthew Crossley, Mark Sinclair, and Martyn Amos. J-pop: Japanese puzzles as optimization problems. *IEEE Transactions on Games*, 14(3):391–402, 2021.

[7] Radek Pelánek. Difficulty rating of sudoku puzzles: An overview and evaluation. *arXiv preprint arXiv:1403.7373*, 2014.

[8] Oliver Ruepp and Markus Holzer. The computational complexity of the kakuro puzzle, revisited. In *International Conference on Fun with Algorithms*, pages 319–330. Springer, 2010.

[9] Helmut Simonis. Kakuro as a constraint problem. *Proc. seventh Int. Works. on Constraint Modelling and Reformulation*, 2008.