

Solving Shakashaka with Constraint Logic Programming

OMMITTED

OMMITTED

Abstract. Logic puzzles are a natural sandbox for learning constraint programming. Shakashaka is a logic puzzle game popularized by the Japanese publisher Nikoli, and has been studied using integer programming. We formulate the properties of Shakashaka in terms of constraint programming, and compare our results, to the extent possible, with previous work. We show the linear time complexity of our approach in a subset of Shakashaka puzzles.

Keywords: Constraint logic programming · Logic puzzle · Integer programming · Prolog · Shakashaka

1 Introduction

Logic puzzles have a number of attractive characteristics as a target domain for researching constraint programming. Humans have trivialized the language component of puzzles, and have difficulty understanding its logic. Computers on the hand are the reverse, having no problem exploiting a puzzle’s logic, but failing to easily cope with the language.-[10].

Logic puzzles have rules describing attributes that should be verified while solving the puzzle. A famous example of a logic puzzle is Sudoku: its properties are that each row, column, and box contains numbers from 1 to 9 exactly once. These properties can be formulated as formal constraints, and with constraint logic programming we can create a program which can find one or more solutions to a puzzle, or to check if a given configuration is a valid solution [9].

Shakashaka, also known as Proof of Quilt, is a logic puzzle created by Japanese publisher Nikoli¹ – the same company that brought Sudoku as we know it to the mainstream. All instances of Shakashaka consist of $m \times n$ rectangular grids of unit squares, where each square can either be white or black, with black squares potentially being numbered, from 0 to 4. The objective of the game is to fill all white squares of the grid with a pattern of triangles such that all of the white areas form rectangular shapes. Figure 1 shows an example of a Shakashaka puzzle and its solution.

This paper is structured as follows. In Section 2 we provide a brief overview of the history and significance of logic puzzles and how they have been tackled using constraint programming. In Section 3 we formulate the Shakashaka puzzle in terms of constraints. In Section 4 we present experimental results using a solver implemented with SICStus Prolog. Finally concluding in Section 5.

¹ <https://www.nikoli.co.jp/en/puzzles/shakashaka.html>

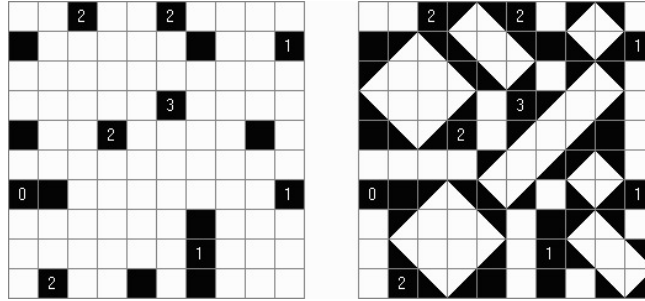


Fig. 1. A Shakashaka instance (left) and its solution (right)

2 Solving logic puzzles with constraints

One's aim in solving logic puzzles is to determine a solution by using the various clues and restrictions defined by the puzzle's properties. Several techniques have been used to solve puzzles throughout the years, such as iterative algorithms [2], optimal algorithms [13], integer programming [4] and [12], and so on. In this paper, constraint programming is used to solve the Shakashaka puzzle.

Logic puzzles are declarative, i.e., they can be solved without any world knowledge other than the rules of the puzzle and logical deduction techniques [9]. Logic programming is, therefore, an appropriate language for stating search and constraint problems. Its relational form (facts and predicates) trivializes the definition of constraints while its non-determinism removes the need for implementing our own search methods.

Constraint programming has a long tradition in artificial intelligence, processing itself has been present in systems related to constraint solving such as [7]. Solving puzzles is a natural sandbox in the constraint programming community [3, 16]. Popular examples include the N-Queens problem, the Zebra puzzle and, naturally, Sudoku [11, 15], among many others.

The puzzle solving community's interest in this practice naturally lead to the forming of organized play. An international federation² hosts puzzle solving contests around the world. Many conferences are organized in the fields of constraint solving and logic programming³. Logic puzzles are also commonly used for teaching (constraint) logic programming in universities around the world [16].

3 Formulating Shakashaka






In this section, the rules of Shakashaka puzzles are thoroughly described. We then formulate a model of the puzzle as a constraint satisfaction problem.

² <http://www.worldpuzzle.org/>

³ <http://www.constraintsolving.com/conferences>

3.1 Overview

As briefly mentioned in Section 1, Shakashaka puzzles consist of $m \times n$ grids of m columns and n lines, where each square of the grid may be either white or black. Black squares may be numbered, from 0 to 4.



An instance is considered *solved* when every connected white region in the grid forms a rectangle (either axis-aligned or rotated by 45°). In order to reach this end state, the solver can fill any white square in the grid with a triangular pattern () such that each white area forms a rectangular shape. We refer to these four patterns as *half-squares*, as they are white squares filled with black right isosceles triangles. One may also leave the white squares blank ()

A concise summary of Shakashaka rules can be written as follows:

- Half-squares can only be placed on empty cells, i.e., white square cells.
- A number in a black square (ranging from 0 to 4) indicates how many half-squares must be placed in the respective square's Von Neumann neighborhood (there must be n half-squares orthogonally adjacent to a black square labeled with a number n).
- Black squares that are not numbered bear no restriction on the number of half-squares in their vicinity.
- In any valid solution, each white-connected region must form a rectangular shape.

In order to solve a Shakashaka puzzle, we start by representing the grid as a $m \times n$ matrix of m lines and n columns:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ a_{2,1} & \dots & & & \\ a_{3,1} & & \dots & & \\ \dots & & & \dots & \\ a_{m,1} & & & & a_{m,n} \end{bmatrix} \quad (1)$$

From a player's perspective, there are, at any moment, five (5) possible ways one can interact with any white square (): either by filling it with one of the half-squares or by keeping it blank, indicated henceforth by .

3.2 Modeling with Constraints

We can represent each of the five possible placement options with five $m \times n$ matrices. Each element of each matrix will be a boolean value, signaling whether that particular position is filled by the type associated with the matrix. We can now denote each position of these matrices by their index i and j , where $1 \leq i \leq m$ e $1 \leq j \leq n$.

In theory, a constraint satisfaction problem model can be pre-processed to ensure that any choices made will lead to solutions, without the need to backtrack (*backtrack-free search* [8]). However, the conventional machinery for ensuring backtrack-free search adds additional constraints, which may require an impractical amount of space. On the other hand, there are approaches that focus on removing redundant values by specifying forbidden compound labels [5]. Following this approach, in our formulation below constraints (b) and (c) try to prune as much as possible the variable domains, aiming to reach no backtracks.

Given the decision variables stated above, we can now place all of the constraints needed to solve any Shakashaka puzzle.

- (a) Ensure that, for every non-black square position (i, j) , the sum of all five matrices must be one, that is, there can be no overlap of the five available types of squares that can be placed on a white square:

$$\blacksquare_{i,j} + \blacktriangleleft_{i,j} + \blacktriangleright_{i,j} + \blacksquare_{i,j} + \square_{i,j} = 1 \quad (2)$$

For black squares, the sum must be zero.

- (b) Ensure that half-squares always form 90° angles with the borders, avoiding 45° angles which lead to incorrect solutions. This needs to be done by examining the adjacency of each type of half-square and the edges of the puzzle. For instance, for any position $(1, j)$ (the top edge of the puzzle) we know that half-squares of types \blacktriangleleft and \blacktriangleright cannot be placed, as they would invalidate any solution by creating 45° angles with the edge of the puzzle. Applying this logic to all the other possibilities, we get:

$$\begin{cases} \blacktriangleleft_{1,j} + \blacktriangleright_{1,j} = 0 \\ \blacktriangleleft_{i,1} + \blacktriangleright_{i,1} = 0 \\ \blacktriangleleft_{m,j} + \blacktriangleright_{m,j} = 0 \\ \blacktriangleleft_{i,n} + \blacktriangleright_{i,n} = 0 \end{cases} \quad (3)$$

- (c) In order to avoid 45° angles with black squares in position (i, j) , we get:

$$\blacktriangleleft_{i+1,j} + \blacktriangleleft_{i+1,j} + \blacktriangleleft_{i-1,j} + \blacktriangleleft_{i-1,j} + \blacktriangleleft_{i,j+1} + \blacktriangleleft_{i,j+1} + \blacktriangleleft_{i,j-1} + \blacktriangleleft_{i,j-1} = 0 \quad (4)$$

- (d) Black squares numbered k must have exactly k orthogonally adjacent half-squares:

$$\blacktriangleleft_{i+1,j} + \blacktriangleleft_{i+1,j} + \blacktriangleleft_{i-1,j} + \blacktriangleleft_{i-1,j} + \blacktriangleleft_{i,j+1} + \blacktriangleleft_{i,j+1} + \blacktriangleleft_{i,j-1} + \blacktriangleleft_{i,j-1} = k \quad (5)$$

- (e) Ensure that bounded white shapes always form rectangular shapes. In Figure 2 we can see that rectangular shapes formed by half-squares can be ‘expanded’ outward in any direction, as seen in Figure 2.a) to Figure 2.b).

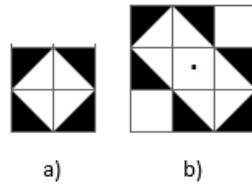


Fig. 2. A bounded white rectangular shape a) that can be expanded correctly in one direction b).

We ensure that a half-square needs to be followed by another of the same type in a given diagonal direction, forming 180° , or closed off by a half-square of its compatible type, i.e., one that, when adjacent, forms a 90° angle. In the case of \blacktriangleleft , this can be expressed by:

$$\blacktriangleleft_{i,j} \leq \blacktriangleleft_{i+1,j+1} + \blacktriangleleft_{i,j+1} \quad (6)$$

However, this constraint does not handle the case where a white space is necessary to maintain a rectangular shape when two half-squares of type \blacksquare , for instance, are placed next to each other horizontally (as in the case of Figure 2.b)). We can solve this with the following expression:

$$\blacksquare_{i,j} + \blacksquare_{i+1,j+1} \leq \square_{i,j+1} + 1 \quad (7)$$

- (f) The constraints so far ensure the construction of valid rectangular shapes of any kind through the use of half-squares. However, Figure 3 shows a case where the solver hitherto produces an invalid solution. To fix this, we must ensure white spaces are inside of a rectangular shape. We arrive at this constraint:

$$\square_{i,j} + \square_{i,j+1} + \square_{i+1,j} \leq \square_{i+1,j+1} + \blacksquare_{i+1,j+1} + 2 \quad (8)$$

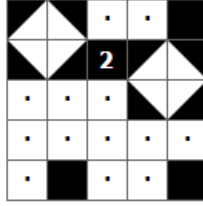


Fig. 3. An invalid solution considering constraints (a) to (e).

- (g) So far, the solver may still produce nested rectangles. An example of an invalid solution produced by the solver without this constraint is seen in Figure 4.

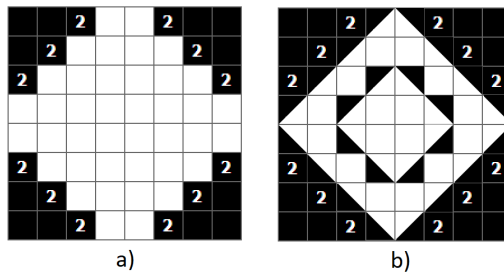


Fig. 4. An instance of a puzzle a) with an invalid solution b) considering constraints (a) to (f).

Two rectangles are nested if one properly contains another. This problem is due to the lack of constraints in half-squares of the same type that are placed diagonally adjacent but not aligned with each other. Suppose that (i, j) and $(i + 1, j + 1)$ are both,

for example, \blacksquare . In this case, these two half-squares will never form a rectangular shape, as seen in Figure 4.b). To avoid nesting for this instance, we can use the following constraint:

$$\blacksquare_{i,j} + \blacksquare_{i+1,j+1} \leq 1 \quad (9)$$

4 Model comparison and evaluation

Our solver was developed in SICStus Prolog and its `clpfd` (*Constraint Logic Programming Over Finite Domains*) library [14]. In this section we evaluate how the solver deals with increasing puzzle complexity, particularly under different labeling options⁴.

By default, SICStus Prolog uses the up labeling option, that is, it explores the domain in ascending order. We decided to use the down labeling option as it improves solving time by approximately 12,5% and reduces the number of backtracks needed to reach a solution. We ran the solver through 10 puzzles⁵ with results consistently under 350 milliseconds. The results are shown in Table 4. Puzzle 11 is a special “huge” puzzle⁶.

| Puzzle | Size | White spaces | Constraints | Backtracks | Time (ms) | Time (ms) [6] |
|--------|-------|--------------|-------------|------------|-----------|---------------|
| 1 | 10x10 | 76 | 1085 | 0 | 16 | 20 |
| 2 | 10x10 | 77 | 1360 | 0 | 27 | 30 |
| 3 | 10x10 | 82 | 1344 | 0 | 29 | 30 |
| 4 | 18x10 | 131 | 1935 | 0 | 50 | 70 |
| 5 | 18x10 | 156 | 3232 | 0 | 58 | 90 |
| 6 | 18x10 | 144 | 2794 | 0 | 47 | 70 |
| 7 | 24x14 | 297 | 6680 | 1 | 115 | 210 |
| 8 | 24x14 | 295 | 6907 | 8 | 121 | 190 |
| 9 | 36x20 | 645 | 15085 | 296 | 349 | 840 |
| 10 | 36x20 | 632 | 14288 | 149 | 366 | 910 |
| 11 | 50x50 | 2087 | 47628 | 127628 | 9745 | — |

Table 1. Experimental results (average of 5 runs per puzzle).

The last column in Table 4 shows the time obtained with another Shakashaka solver, developed by Demaine et al. [6]⁷, with the exception of the last puzzle. While it is not possible to establish a direct comparison between the solvers, we can analyze the running time increase as the puzzles scale in size. We observe that our solver, in regards to the number of white spaces, has time complexity $O(n)$, whereas the solver by Demaine et al. [6] has time complexity $O(2^n)$. This relationship is visible in Figure 5.

⁴ For all the results obtained, we used an Intel Core i5 processor 1.6GHz with 8 GB of memory running macOS Mojave (10.14.3).

⁵ Obtained from <http://nikoli.com/en/puzzles/shakashaka/>

⁶ Obtained from <https://www.puzzle-shakashaka.com/> in their *Special Monthly Shakashaka* from March 1st, 2019.

⁷ The solver used in [6] is SCIP 3.0.0 [1] (Binary: Windows/PC, 32bit, cl 16, intel 12.1: statically linked to SoPlex 1.7.0, Ipopt 3.10.2, CppAD 20120101.3). The machine used had an Intel Core2 Duo P8600@2.40 GHz with RAM 4GB on Windows Vista Business SP2.

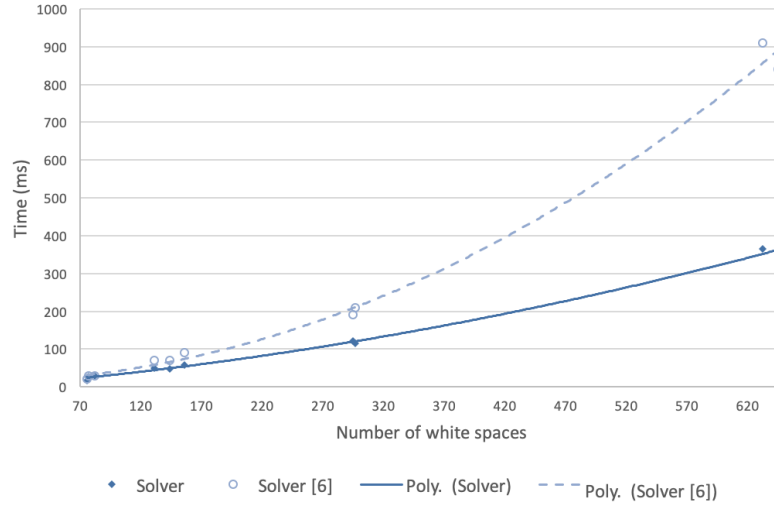


Fig. 5. Performance comparison between solvers in regards to the number of white spaces.

The number of backtracks increases very significantly with puzzles of greater dimensions, which have a large density of black squares near the center.

We also produced a rudimentary puzzle generator which creates artificial Shakashaka puzzles of the type seen in Figure 6, where n indicates the number of non-numbered black squares in the first half of the first line. Parameter n determines the size of the puzzle in terms of the number of cells in the board. All of these puzzles have a unique solution.

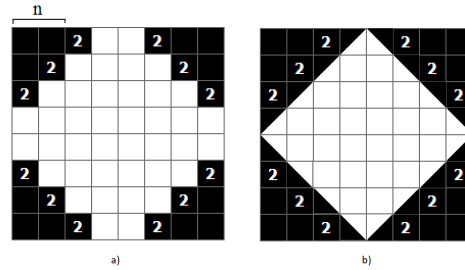


Fig. 6. Artificial Shakashaka puzzle a), with its unique solution b).

Producing this sort of puzzles for $n \in [1..50]$, we noticed that the solver required no backtracks at all to solve them. Figure 7 shows the time taken to find a solution for each puzzle, depending on the total number of cells present in the board.

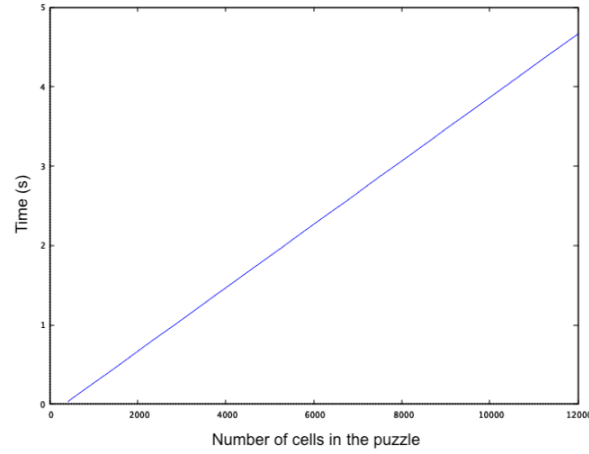


Fig. 7. Relation between number of cells and elapsed time for puzzles of the type illustrated in Figure 6.

5 Conclusions and Future Work

In this paper we described the implementation of an efficient solver for Shakashaka puzzles through a formal definition of the problem in terms of constraints, and using constraint logic with integer programming.

The main problem faced was minimizing the number of backtracks through careful constraint definitions, with the primary goal of achieving backtrack-free solutions. This was possible through several domain reductions of the variables used and the use of compound labeling, provided by the SICStus Prolog environment. The result is a solver that can find solutions with very few backtracks, for puzzles of up to 25×25 . The solver presents a quasi-linear relationship between the number of white spaces present in the puzzle and the time taken to find a solution. In regards to the solution proposed by Demaine et al. [6], we can observe that our solution scales much better, in comparison, presenting a linear time complexity, as opposed to polynomial.

For future work we aim to develop a generic Shakashaka puzzle generator that produces puzzles with unique solutions. This will enable us to more thoroughly test the performance of our approach, analyze its time complexity and optimize it accordingly.

References

1. Achterberg, T.: Scip: solving constraint integer programs. *Mathematical Programming Computation* **1**(1), 1–41 (Jul 2009)
2. and, and: Solving japanese puzzles with logical rules and depth first search algorithm. In: 2009 International Conference on Machine Learning and Cybernetics. vol. 5, pp. 2962–2967 (July 2009). <https://doi.org/10.1109/ICMLC.2009.5212614>

3. Apt, K.: Principles of Constraint Programming. Cambridge University Press, New York, NY, USA (2003)
4. Bartlett, A., Chartier, T.P., Langville, A.N., Rankin, T.D.: An integer programming model for the sudoku problem. *Journal of Online Mathematics and its Applications* **8**(1) (2008)
5. Beck, J.C., Carchrae, T., Freuder, E.C., Ringwelski, G.: Backtrack-free search for real-time constraint satisfaction. In: Wallace, M. (ed.) *Principles and Practice of Constraint Programming – CP 2004*. pp. 92–106. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
6. Demaine, E.D., Okamoto, Y., Uehara, R., Uno, Y.: Computational complexity and an integer programming model of shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* **E97.A**(6), 1213–1219 (2014). <https://doi.org/10.1587/transfun.E97.A.1213>
7. Fikes, R.E.: Ref-arf: A system for solving problems stated as procedures. *Artificial Intelligence* **1**(1), 27 – 120 (1970). [https://doi.org/https://doi.org/10.1016/0004-3702\(70\)90003-2](https://doi.org/https://doi.org/10.1016/0004-3702(70)90003-2), <http://www.sciencedirect.com/science/article/pii/0004370270900032>
8. Freuder, E.C.: A sufficient condition for backtrack-free search. *J. ACM* **29**(1), 24–32 (Jan 1982). <https://doi.org/10.1145/322290.322292>, <http://doi.acm.org/10.1145/322290.322292>
9. Graber, M., Felgentreff, T., Hirschfeld, R.: Solving interactive logic puzzles with object-constraints. In: *Reactive and Event-based Languages and Systems (REBELS)* (2014)
10. Lev, I., MacCartney, B., Manning, C., Levy, R.: Solving logic puzzles: From robust processing to precise semantics. In: *ACL 2004: Second Workshop on Text Meaning and Interpretation*. pp. 9–16. Association for Computational Linguistics, Barcelona, Spain (Jul 2004), <http://www.aclweb.org/anthology/W04-0902>
11. Lynce, I., Ouaknine, J.: Sudoku as a SAT problem. In: *International Symposium on Artificial Intelligence and Mathematics, ISAIM 2006*, Fort Lauderdale, Florida, USA, January 4–6, 2006 (2006), <http://anytime.cs.umass.edu/aimath06/proceedings/P34.pdf>
12. Mingote, L., Azevedo, F.: Colored nonograms: An integer linear programming approach. In: Lopes, L.S., Lau, N., Mariano, P., Rocha, L.M. (eds.) *Progress in Artificial Intelligence*. pp. 213–224. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
13. Pacurib, J.A., Seno, G.M.M., Yusiong, J.P.T.: Solving sudoku puzzles using improved artificial bee colony algorithm. In: *2009 Fourth International Conference on Innovative Computing, Information and Control (ICICIC)*. pp. 885–888 (Dec 2009). <https://doi.org/10.1109/ICICIC.2009.334>
14. SICStus: Clpfd intro - sicstus prolog. <https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/CLPFD-Intro.html>, accessed: 2019-02-30
15. Simonis, H.: Sudoku as a constraint problem. In: *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*. pp. 13–27 (2005)
16. Szeredi, P.: Teaching constraints through logic puzzles. In: Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Váncza, J. (eds.) *Recent Advances in Constraints*. pp. 196–222. Springer (2004)