Formulation of Sudoku Puzzle Using Binary Integer Linear Programming and Its Implementation in Julia, Python, and Minizinc

Article *in* Jambura Journal of Mathematics · June 2022

DOI: 10.34312/jjom.v4i2.14194

4 authors:

Fahren Bukhari
Bogor Agricultural University
18 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Sri Nurdiati
Bogor Agricultural University
94 PUBLICATIONS **264** CITATIONS
SEE PROFILE

Mohamad Khoirun Najib
Bogor Agricultural University
25 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Nandika Safiqri
Bogor Agricultural University
1 PUBLICATION **0** CITATIONS
SEE PROFILE

JAMBURA JOURNAL OF MATHEMATICS
Jambura J. Math. Vol. 4, No. 2, pp. 323–331, July 2022
Journal Homepage: http://ejurnal.ung.ac.id/index.php/jjom
DOI: https://doi.org/10.34312/jjom.v4i2.14194

Formulation of Sudoku Puzzle Using Binary Integer Linear Programming and Its Implementation in Julia, Python, and Minizinc

Fahren Bukhari[1], Sri Nurdiati[2*], Mohamad Khoirun Najib[3], Nandika Safiqri[4]

[1,2,3,4]*Department of Mathematics, Faculty of Mathematics and Natural Sciences, IPB University, Jl. Meranti, Kampus IPB Dramaga, Bogor 16680, Indonesia*
[*]*Corresponding author. Email: nurdiati@apps.ipb.ac.id*

ABSTRACT

Sudoku is a number puzzle game popular among people with various difficulty levels (easy, medium, hard, and extremely hard). Sudoku can be modeled as a linear programming problem in mathematics, particularly binary integer linear programming (BILP). Completing Sudoku using BILP is quite tricky because it requires many iterations. Therefore, this study aims to analyze the Sudoku problem using the BILP formulation and implement the problem using Julia, Python, and MiniZinc. Out of 15 cases for each difficulty level, Julia performs better than Python and MiniZinc based on computation time. Moreover, Sudoku with easy difficulty levels is solved with a longer computation time than the other three difficulty levels. The computation time for solving BILP is getting faster as the difficulty level of the Sudoku problem increases. This is because Sudoku problems with easy difficulty levels have more known values as clues and generate more constraints than other difficulty levels.

Keywords:
Julia; Linear Programming; Minizinc; Python; Sudoku

How to Cite:

F. Bukhari, S. Nurdiati, M. K. Najib, and N. Safiqri, "Formulation of Sudoku Puzzle Using Binary Integer Linear Programming and Its Implementation in Julia, Python, and Minizinc", *Jambura J. Math.*, vol. 4, No. 2, pp. 323–331, 2022, doi: https://doi.org/10.34312/jjom.v4i2.14194

1. Introduction

Sudoku is a logic puzzle game invented by Howard Garns in 1979 in the United States. Sudoku was first published as Number in Place in Dell Magazines. Sudoku became popular in Japan in the 1984 Monthly Nikolist newspaper by Maki Kaji as Suji wa dokushin ni kagiru, which can be translated as "digit must be single", or digits are limited to one occurrence [1].

Even though Sudoku is a number puzzle game, that doesn't mean that only people with math skills can play it. Sudoku can be played by anyone of all ages and occupations. Curiosity about solving Sudoku problems makes many people interested in playing it. In fact, Sudoku was once contested in an international television contest in which nine teams represented certain geographic areas [2].

e-ISSN: 2656-1344
Received: 7 May 2022 | Accepted: 9 June 2022 | Online: 25 June 2022

Sudoku has different difficulty levels, such as very easy, easy, moderate, hard, and extremely hard. In general, the difficulty of these puzzles depends on the number and location of clues [3], as shown in Table 1. The higher the Sudoku level, the more challenging the problems to solve.

Table 1. Difficulty level of Sudoku puzzles

Difficulty level	Number of clues	Minimum number of clues in each row and column
Very easy	≥ 46	5
Easy	$36-46$	4
Moderate	$32-35$	3
Hard	$28-31$	2
Extremely hard	$17-27$	0

These Sudoku puzzles can be modelled using a mathematical model as a linear programming problem. Linear programming is a mathematical method to obtain an optimal solution by maximizing or minimizing the objective function of the constraint function. Linear programming with an objective function with only two possible values, 0 or 1, is called binary integer linear programming (BILP) [4].

The most common Sudoku puzzles consist of a 9×9 grid of matrix, with nine 3×3 sub-grids. In addition to Sudoku with size 9×9, there is also Sudoku with size 4×4. In Sudoku, with a size of 9×9, the game's ultimate goal is to fill each grid with the numbers 1 to 9 so that each column, row, and sub-grid contains the numbers 1 to 9 in exactly one value [5, 6].

The Sudoku solution using linear programming will be difficult and time-consuming because it requires many iterations. For this reason, software assistance is needed that can solve problems more efficiently. Therefore, this research will use several programming languages, such as Julia, Python, and Minizinc, to solve linear programming problems formed from Sudoku puzzles. These three programming languages are open-source programs capable of solving linear programming problems.

This study aims to formulate a binary integer linear programming problem based on a Sudoku puzzle. Furthermore, several Sudoku cases from easy to extremely hard levels are solved using Julia, Python, and Minizinc; then, the computational time in these programming languages is calculated. Julia is a new dynamic and open-source programming language that is claimed to have high performance with ease of writing code. Therefore, in addition to the research objectives mentioned above, this study aims to introduce the Julia programming language to computational mathematicians in Indonesia. The research results are expected to provide additional insight into open-source programming languages that can be used to replace commercial linear programming solvers, such as Lingo, which indirectly supports IGOS (Indonesia Goes Open Source).

2. Methods

The research begins by collecting cases of Sudoku problems from easy to extremely hard levels. The Sudoku puzzle case to be solved is a sudoku puzzle obtained from the Sudoku-Classic Sudoku Puzzle android application version 1.23.0, downloaded from google playstore. From the four levels, 15 puzzles are taken each.

The first step is to formulate a binary integer linear programming (BILP) problem for a 9×9 Sudoku problem. After that, Sudoku problems from easy to extremely hard levels are solved using Julia 1.6.3, Python 3.10, and MiniZinc 2.5.5. The implementation process was carried out using a Lenovo IdeaPad 5 14ARE05 laptop with an AMD Ryzen 7 4800U processor and Radeon Graphics 1.80 GHz. We compare the computational time needed to solve Sudoku problems from easy to extremely hard levels from the results obtained.

2.1. Binary Integer Linear Programming

Integer linear programming (ILP) is a class of linear programming model optimization problems with the variable used in the form of integers, where the objective function is linear, and the constraint is a linear inequality [7]. If the variable must be an integer, the problem is called pure integer linear programming. If only some variables are integers, it is called mixed-integer linear programming (MILP). Meanwhile, ILP with a variable that must be 0 or 1 is called 0-1 ILP, also known as binary integer linear programming (BILP) [8]. Variables with a value of 0-1 usually represent a yes or no decision [9]. One method that can solve ILP problems is the Branch and bound method. This method is one of the methods to produce linear programming solutions that produce integer decision variables [10].

2.2. Julia, Python and Minizinc

Julia is a high-level, dynamic, and high-performance programming language [11]. Although classified as a dynamic programming language, Julia performance is almost comparable to static programming languages such as C/C++ and Fortran. Julia has a similar syntax to Matlab, so the transition process from Matlab users to Julia can be done easily. Julia is rapidly becoming a competitive language in data science and computational science [12, 13]. This research's linear programming problem will be solved using the JuMP package. JuMP (Julia for Mathematical Programming) is an open-source modelling language on Julia for expressing various optimization problems, such as linear, mixed-integer, quadratic, semi-definite, and non-linear programming, in a high-level algebraic syntax [14].

Python is a high-level programming language that is interpreter, interactive, and object-oriented. Python was developed by Guido Van Rossum in 1990 in Amsterdam to continue the ABC programming language [15]. Python programming language is easy to learn because the syntax used is clear and easy to understand [16]. Python allows users to write clear and logical syntax because Python has a complex structure of functions, classes, modules, nested code blocks, and packages. This study uses the PuLP package to solve linear programming problems. PuLP is an open-source tool for modelling linear programming in Python. Solutions in PuLP are obtained by calling other solvers, such as CBC and GPLK (free) or CPLEX, GUROBI, and MOSEK (commercial).

MiniZinc is an open-source programming language designed to solve optimization problems limited to real numbers and integers [17]. MiniZinc is a medium-level constraint modelling language that allows users to express optimization problems easily [18]. MiniZinc does not require third-party packages or applications to solve a given modelling problem, unlike Julia and Python. Minizinc already has several solvers built into it, such as Gecode, CBC, and findMUS.

3. Results and Discussion

3.1. Formulation of the Binary Integer Linear Programming (BILP) Problem

Definition 1. Sudoku is a popular number puzzle where the goal is to place the digits 1 to 9 on a 9×9 square or grid, with some of the digits already filled in as clues. The solution must meet the following rules:

1. Numbers 1 to 9 must appear in each 3×3 sub-grid
2. Numbers 1 to 9 must appear in each row
3. Numbers 1 to 9 must appear in each column.

Definition 2. Mathematically, the Sudoku problem of size 9×9 can be formulated into binary integer linear programming, namely as follows:

1. Index and parameters
   - $i \in \{1, 2, \ldots, 9\}$ indicates the row index in the Sudoku puzzle,
   - $j \in \{1, 2, \ldots, 9\}$ indicates the column index in the Sudoku puzzle,
   - $k \in \{1, 2, \ldots, 9\}$ indicates the grid value index in the i-th row and j-th column,
   - G is a set of some decision variables whose value is already known as a clue.
2. Decision variables

$$x_{ijk} = \begin{cases} 1, & \text{if the grid } (i,j) \text{ is } k \\ 0, & \text{otherwise} \end{cases} . \tag{1}$$

3. Objective function The objective function becomes irrelevant because each point satisfies the constraint will represent a solution to the Sudoku puzzle problem [19].

$$\min 0^T x_{ijk}, \quad \forall i, j, k. \tag{2}$$

4. Constraints
   (a) Each column contains exactly one integer number 1 to 9:

$$\sum_{i=1}^{9} x_{ijk} = 1, \quad \forall j, k. \tag{3}$$

   (b) Each row contains exactly one integer number 1 to 9:

$$\sum_{j=1}^{9} x_{ijk} = 1, \quad \forall i, k. \tag{4}$$

   (c) Each grid point contains exactly one integer number 1 to 9:

$$\sum_{k=1}^{9} x_{ijk} = 1, \quad \forall i, j. \tag{5}$$

   (d) Each sub-grid contains exactly one integer number 1 to 9:

$$\sum_{j=3q-2}^{3q} \sum_{i=3p-2}^{3p} x_{ijk} = 1, \quad \forall k \, \forall p, q \in \{1, 2, 3\}. \tag{6}$$

   (e) For decision variables that are already known as clues, the value of the decision variable is set to be 1:

$$x_{ijk} = 1, \quad \forall i, j, k \in G. \tag{7}$$

(f) Non-negativity and binary:

$$x_{ijk} = 1 \in \{0, 1\}. \tag{8}$$

Example 1. Given a Sudoku problem of size 9×9, as shown in Figure 1.

			2	4				1
							6	
						5	7	4
		3		8			1	
5		4						8
			7					
			6		9			
		8				6		
	7				4		9	2

Figure 1. An example of a 9×9 Sudoku problem

The BILP formulation of the Sudoku problem is the same as previously described, with details of the constraint function as follows:

Constraints (4a) and (4b) clearly say that each row and column of the Sudoku matrix is worth 1 to 9 without any repetition. This means that in the first row, the unfilled boxes cannot have a value of 1, 2, or 4. On the other hand, the unfilled boxes in the first column cannot have a value of 5.

Constraint (4c) says that each matrix element contains exactly one integer number 1 to 9. This is clear because a point in the Sudoku matrix has only exactly one value, which comes from the Sudoku meaning: "digit must be single".

Constraint (4d) says that each sub-grid contains exactly one integer number 1 to 9. The Sudoku matrix has several smaller sub-grids in it. For the 9×9 cases, there are nine sub-grids with a size of 3×3. In Figure 1, the sub-grid is represented by thick lines as edges. For example, for constraint (4d), there is an unfilled box in the sub-grid at the top-right (in blue); it cannot be 1, 4, 5, 6, or 7.

Constraint (4e) says that for decision variables that are already known as clues, the value of the decision variable is set to be 1. Based on Figure 1, the clue in the 1st row and 4th column is 2, so the formulation for this clue is

$$x_{142} = 1. \tag{9}$$

Thus, using the same approach, the constraints for each known value as a clue are as follows.

$$\begin{aligned} x_{142} = x_{154} = x_{191} = x_{286} = x_{375} = x_{387} = x_{394} &= 1 \\ x_{433} = x_{458} = x_{471} = x_{515} = x_{534} = x_{598} = x_{647} &= 1 \\ x_{746} = x_{769} = x_{838} = x_{876} = x_{927} = x_{964} = x_{989} = x_{992} &= 1 \end{aligned} \tag{10}$$

3.2. Implementation of BILP Problem using Julia, Python and Minizinc

With 15 cases each for easy, moderate, hard, and extremely hard difficulty levels, the BILP formulation is used to solve the problem using Julia, Python, and Minizinc (see Appendix A). The computation time of each programming language is calculated from an average of ten repetitions. Figure 2 below is the solution to the Sudoku problem given in Example 1.

6	5	9	**2**	**4**	7	3	8	**1**
4	3	7	5	1	8	2	**6**	9
2	8	1	9	3	6	**5**	**7**	**4**
7	6	**3**	4	**8**	2	9	**1**	5
5	9	**4**	1	6	3	7	2	**8**
8	1	2	**7**	9	5	4	3	6
1	2	5	**6**	7	**9**	8	4	3
9	4	**8**	3	2	1	**6**	5	7
3	**7**	6	8	5	**4**	1	**9**	**2**

Figure 2. Solution of the Sudoku problem in Figure 1

The case in Figure 1 is one of 15 cases at the extremely hard level because the number of clues given is only 22 numbers. The solution of the case in Figure 1 using Julia, Python and Minizinc is the same as Figure 2. Julia gave the fastest average time of 0.0405 seconds out of ten repetitions, while Python and Minizinc completed it in times 0.1172 and 0.2934 seconds.

In the same way, 15 cases for 4 difficulty levels are solved using Julia, Python, and Minizinc. Of the ten repetitions, the average computational time for each case is shown in Figure 3. The thick line in Figure 3 shows the average of each case solved on a given difficulty level. Meanwhile, the dashed line is the average per case of 10 repetitions.

Julia has the fastest computation time compared to the other two programming languages, while Minizinc has the longest computation time. However, both Julia, Python, and Minizinc were able to solve the binary integer linear programming problem of the Sudoku puzzle in no more than 0.5 seconds per execution. Moreover, in terms of writing syntax, Julia is simpler than Python and almost similar to Minizinc (see Appendix A). Thus, Julia has a simple syntax writing style and is almost similar to writing on paper but has faster performance than the other two programming languages.

Based on the difficulty level of the Sudoku problem solved, Julia gives a fairly stable average computation time of 0.025 seconds for all cases. However, two cases at the easy

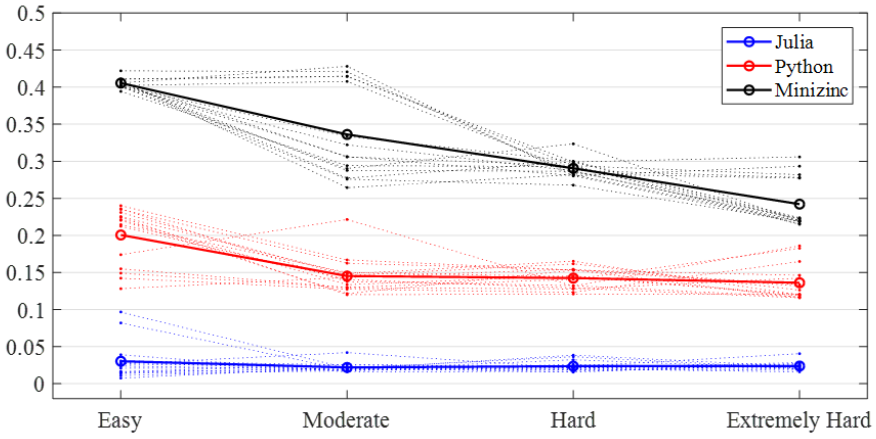

Figure 3. Comparison of the computation time of Julia, Python, and Minizinc to solve 14 Sudoku cases with four difficulty levels

level were completed in almost 0.1 seconds. In contrast to Julia, Python and Minizinc programming languages have the highest average computation time in the easy case. The computation time gets faster as the difficulty level of the Sudoku problem increases. The Sudoku problem with easy difficulty has more values known as clues, so the constraint function (4e) is also generated more than the other difficulty levels. The more constraint functions that must be met will affect the computation time generated by each programming language. Thus, the easier the Sudoku problem, the longer the computation time of the formulated BILP problem. This is different from the manual completion of Sudoku by humans. In the easy case, the more clues you give, the easier the empty squares in Sudoku will be to solve.

4. Conclusion

This research formulates the Sudoku problem into a mathematical equation, namely binary integer linear programming (BILP). Several cases are solved using this formulation in the programming languages Julia, Python, and Minizinc. Out of 15 cases for each difficulty level (easy, moderate, hard, and extremely hard), Julia performs better than Python and MiniZinc based on compute time. Furthermore, on the easy level, Sudoku is completed using BILP with longer computation time than the other three difficulty levels that are more difficult. The computation time for solving BILP problems is getting faster as the difficulty level of Sudoku questions increases. This is because Sudoku problems with easy difficulty have more known values as clues, so the resulting constraint function is also higher than other difficulty levels. This research can be developed by applying other algorithms, especially heuristic methods such as genetic algorithms and others, then comparing the results obtained with this research.

References

[1] D. B. Mishra, R. Mishra, K. N. Das, and A. A. Acharya, "Solving Sudoku Puzzles Using Evolutionary Techniques—A Systematic Survey," in *Advances in Intelligent Systems and Computing*, 2018, pp. 791–802, doi: http://dx.doi.org/10.1007/978-981-10-5687-1_71.

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

[3] H. Chel, D. Mylavarapu, and D. Sharma, "A novel multistage genetic algorithm approach for solving Sudoku puzzle," in *2016 International Conference on Electrical, Electronics, and*

Optimization Techniques (ICEEOT). IEEE, mar 2016, pp. 808–813, doi: http://dx.doi.org/10.1109/ICEEOT.2016.7754798.

[4] A. C. BartlettTimothy, P. Chartier, A. N. Langville, and T. D. Rankin, "An Integer Programming Model for the Sudoku Problem," in *J. Online Math. its Appl.*, vol. 8, no. 1, 2007.

[5] N. Kitsuwan, P. Pavarangkoon, H. M. Widiyanto, and E. Oki, "Dynamic load balancing with learning model for Sudoku solving system," *Digital Communications and Networks*, vol. 6, no. 1, pp. 108–114, feb 2020, doi: http://dx.doi.org/10.1016/j.dcan.2019.03.002.

[6] A. Zulaihah and A. Mardati, "Penggunaan Permainan Throwing Sudoku untuk Pengenalan Konsep Bilangan," in *Optimalisasi Peran Pendidikan dalam Membangun Karakter Anak untuk Menyongsong Generasi Emas Indonesia*, 2016, pp. 190–194.

[7] K. Genova and V. Guliashki, "Linear integer programming methods and approaches - A survey," in *Cybern. Inf. Technol.*, vol. 11, no. 1, 2011, pp. 13–25.

[8] M. F. Wardhana, *Penyelesaiaan Puzzle Sudoku Menggunakan Pemrograman Linear Integer*. Undergraduate Thesis: IPB University, 2014.

[9] H. P. Williams, "Logic and Integer Programming," in *International Series in Operations Research & Management Science*. London: Springer, 2009.

[10] S. D. Purba and F. Ahyaningsih, "Integer Programming Dengan Metode Branch and Bound Dalam Optimasi Jumlah Produksi Setiap Jenis Roti Pada Pt. Arma Anugerah Abadi," *Karismatika*, vol. 6, no. 3, pp. 20–29, 2020, doi: https://doi.org/10.24114/jmk.v6i3.22208.

[11] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, jan 2017, doi: http://dx.doi.org/10.1137/141000671.

[12] N. K. K. Ardhana, S. Nurdiati, M. K. Najib, and S. A. Mukrim, "Akurasi dan Efisiensi Solusi Persamaan Diferensial Biasa Dengan Masalah Nilai Batas Pada Julia dan Octave," *Jurnal Matematika UNAND*, vol. 11, no. 1, pp. 32–46, apr 2022, doi: http://dx.doi.org/10.25077/jmu.11.1.32-46.2022.

[13] K. Gao, G. Mei, F. Piccialli, S. Cuomo, J. Tu, and Z. Huo, "Julia language in machine learning: Algorithms, applications, and open issues," *Computer Science Review*, vol. 37, p. 100254, aug 2020, doi: http://dx.doi.org/10.1016/j.cosrev.2020.100254.

[14] I. Dunning, J. Huchette, and M. Lubin, "JuMP: A Modeling Language for Mathematical Optimization," *SIAM Review*, vol. 59, no. 2, pp. 295–320, jan 2017, doi: http://dx.doi.org/10.1137/15M1020575.

[15] J. Enterprise, *Otodidak Pemrograman Python*. Jakarta: PT Elex Media Komputindo, 2017.

[16] T. C. A.-S. Zulkhaidi, E. Maria, and Y. Yulianto, "Pengenalan Pola Bentuk Wajah dengan OpenCV," *Jurnal Rekayasa Teknologi Informasi (JURTI)*, vol. 3, no. 2, p. 181, jun 2020, doi: http://dx.doi.org/10.30872/jurti.v3i2.4033.

[17] K. Marriott, P. J. Stuckey, L. D. Koninck, and H. Samulowitz, *A MiniZinc Tutorial*, 2014.

[18] R. Caballero, P. J. Stuckey, and A. Tenorio-Fornés, "Two type extensions for the constraint modeling language MiniZinc," *Science of Computer Programming*, vol. 111, pp. 156–189, nov 2015, doi: http://dx.doi.org/10.1016/j.scico.2015.04.007.

[19] D. Assencio, "Solving Sudoku Puzzle with Linear Programming," 2017, url: https://diego.assencio.com/?index=25ea1e49ca59de51b4ef6885dcc3ee3b.

Appendix A

Program A1. Julia syntax to solve Sudoku problems

```
using JuMP
using GLPK
function solve_sudoku(puzzle)
    sudoku = Model(GLPK.Optimizer)
    variable(sudoku, x[i=1:9, j=1:9, k=1:9], Bin); # decision variable
    for i = 1:9, j = 1:9
        @constraint(sudoku, sum(x[i,j,k] for k in 1:9) == 1) # constraint (4c)
    end
    for ind = 1:9
        for k = 1:9
            @constraint(sudoku, sum(x[ind,j,k] for j in 1:9) == 1) # constraint (4b)
            @constraint(sudoku, sum(x[i,ind,k] for i in 1:9) == 1) # constraint (4a)
        end
    end
    for i = 1:3:7, j = 1:3:7, k = 1:9 # constraint (4d)
        @constraint(sudoku, sum(x[r,c,k] for r in i:i+2, c in j:j+2) == 1)
    end
    init_sol = puzzle
    for i = 1:9, j = 1:9
        if init_sol[i,j] != 0 # constraint (4e)
            @constraint(sudoku, x[i,j,init_sol[i,j]] == 1)
        end
    end
    optimize!(sudoku)
    x_val = value.(x)
    sol = zeros(Int,9,9)
    for i in 1:9, j in 1:9, k in 1:9
        if round(Int,x_val[i,j,k]) == 1
            sol[i,j] = k
        end
    end
    return sol
end
```

Program A2. Python syntax to solve Sudoku problems

https://github.com/Lakshmi-1212/Sudoku_Solver_LP/blob/main/Solver_LP.ipynb

Program A3. Minizinc syntax to solve Sudoku problems

https://github.com/buzzdecafe/minizinc/blob/master/sudoku.mzn