# Sudoku A Constraint Satisfaction Problem

Holger Findling, *Senior Member, IEEE*

**Abstract**—Research has shown that some NP complete problems with finite domains can be solved effectively using constraint reasoning. Constraint reasoning algorithms are capable of modeling optimization problems involving heterogeneous constraints and combinatorial search. The degree of difficulties involving Sudoku puzzles ranges from easy, medium, hard to evil; where hard and evil type problems require intelligent guessing and backtracking. This paper evaluates the implementation of a constraint reasoning algorithm capable of solving all models in the Sudoku puzzle space.

**Index Terms**— Constraint Satisfaction Problem, Constraint Reasoning, Constraint Propagation.

## 1 Introduction

Sudoku puzzles are special cases of Latin Squares. Additional restrictions are imposed on the Sudoku puzzle that requires 3x3 subgroups (squares) to contain the digits 1 through 9. Each row and column must also contain one instance of each numeric. The goal of the puzzle is to enter a numerical digit from 1 to 9 in each cell of the 9×9 grid. Each puzzle typically starts with a minimum of 16 preset numerics. Figure 1 shows an example of a Sudoku puzzle with 36 numerics given and its final solution.Solving a puzzle requires patience and logical ability. Any solution to a Sudoku puzzle is also a Latin Square.

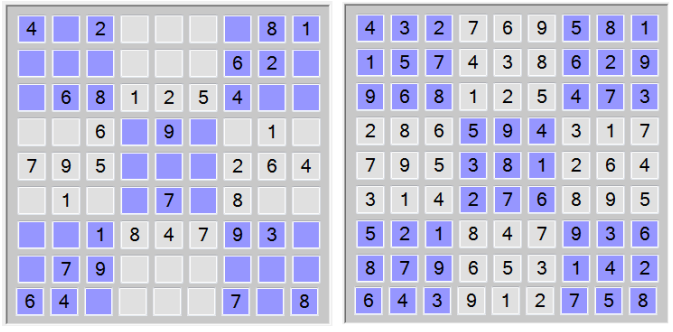


**Figure 1 – Sudoku Puzzle and Solution**

Leonard Euler developed the idea of ordering numbers in rows and columns in the 1780's and Howard Garns is credited with further developing the rules for the currently published Sudoku puzzles, which were first published in the late 1970's in the Math Puzzles and Logic Problems magazine by Dell Magazines. In 1984 Nikoli, Japan's leading puzzle creating company, discovered Dell's Number Place and decided to present them to their Japanese puzzle fans. The puzzles, which were first named Suuji Wa Dokushin Ni Kagiru, ("the numbers must be single") quickly became popular.

At the end of 2004 Wayne Gould, a retired Hong Kong judge as well as a puzzle fan and a computer programmer, visited London convincing the editors of The Times to publish the Sudoku puzzles. He had written a computer program that generates Sudoku puzzles of various difficulty levels. The Times published the puzzle November 12, 2004, which is the beginning of an enormous phenomenon that swiftly spread all over the world. [1]

Sudoku has become of great interest to the field of computational intelligence (CI) since the problem resides in the NP-Complete domain. Finding a polynomial solution for all models in this puzzle space would also provide a solution for all NP-Complete problems, and resolves the P = NP question [2]. Many CI algorithms [3] [4] [5] and mathematical approaches for solving the puzzle have been presented. Today, most AI researchers agree that Sudoku is a Constraint Satisfaction Problem (CSP) [6].

## 2 Constraint Satisfaction Problem

A Constraint Satisfaction Problem is formally defined by a tuple CSP (V, D, C), where

- V is a finite set of variables $\{v_1, v_2, \ldots, v_n\}$
- D is a domain for the variables $\{d_1, d_2, \ldots, d_n\}$
- C is a set of constraints $\{c_1, c_2, \ldots, c_n\}$

A solution to a CSP is discovered when an evaluation satisfies all constraints. An evaluation of the variables is a function from set V to the set of associated domains D, v: V→D [7].

Today, many variations to CSP algorithms exist, but typically most involve some form of search and variants of backtracking, and constraint propogation. Some of the algorithms include Simple Chronological Backtracking, Forward Checking, partial Look-Ahead, and full Look-Ahead [8]. When Backtracking is used do derive a CSP solution, the order in which variables are instantiated are important. Experiements have shown that the ordering of variables choosen for instantiation has a significant impact on running times.

Several heuristics have been developed for variable ordering, including the powerful search-rearrangement method [9]. This

heuristic selects the variables with the least amount of choices in their domain for instantiation first. A drawback is that the branches of the search tree are determined dynamically and computational times increase due to the additional processing. For the proposed Sudoku CSP algorithm a depth-first-search like approach is choosen instead. When a guessing step is required, the next variable for instantiation is choosen in a left-to-right and top-to-bottom order. The first value in the associated domain is selected. No additional considerations towards domain values are implemented.

Each cell in the grid V = $\{v_1..v_{81}\}$ is mapped to a domain gDom = $\{d_1..d_{81}\}$. Each row [1..9], column [1..9] and square [1..9] are also mapped to separate domains. The rules and constraints of the puzzle are inforced through domain propagation techniques.

## 3 SUDOKU CSP ALGORITHM

The Sudoku CSP algorithm is comprised of logical reasoning, guessing numbers using a depth-first-search like methodology, and chronological backtracking. The domains for the puzzle space are established using grid domains gDom[1..81], row domains row[1..9], column domains col[1..9], and square domains sqr[1..9]. The solution of the puzzle is stored in the puzzle grid [1..81]. The range of each domain is limited to integer 0 through 9; where 0 denotes "no number available" in the domain.

Figure 2 shows the recursive steps taking to solve the puzzle. Each iterative step inserts exactly one number into the puzzle grid [1..81] either by logical reasoning or guessing. The functions UpdateRowColumnDomain, UpdateSquareDomain, UpdateGridDomain, and Solve perform constraint propagation. The constraints imposed on the domains represent the rules of the puzzle. The function IsLastNumInDomain performs the logical reasoning to determine which number should be inserted next into the grid. Function Sort ensures the domains remain sorted in ascending order. When logical reasoning fails to determine the next number to be inserted into the grid, then function Think guesses the next number by selecting the first available number in a given domain. Since such a guess can be incorrect, a domain inconsistency can be created.

When domains become inconsistent the function Backtracking restores the puzzle state to the last known consistent state and removes the incorrectly guessed number from the associated domains. This process is repeated until the puzzle is solved.

Function Think shown in Figure 3 determines the next number to be inserted into the puzzle grid. It accomplishes the task by calling GetNextGridIndex which returns the first empty cell in the puzzle space. Function GetNextGridDomainValue returns the number to be inserted. Before inserting the number into the grid the puzzle space and state must be archived on the stack. Since the stack holds all puzzle states before a guess is implemented, it provides for chronolgical backtracking.

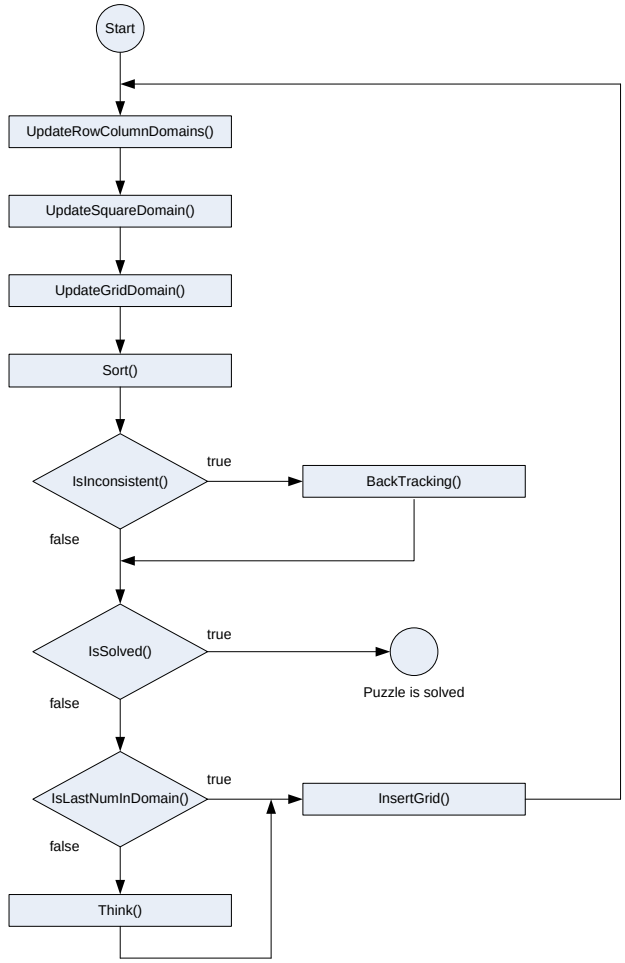


**Figure 2 – Function Solve ()**

The puzzle states are represented by the objects instantiated of class Puzzle. It contains the following data members:

- int grid[1..81]
- int gDom[81][1..9]
- int row[1..9][ 1..9]
- int col[1..9][ 1..9]
- int sqr[1..9][ 1..9]
- GridIndex
- DomainIndex
- GridValue

Variables GridIndex, DomainIndex, and GridValue contain the last values inserted into the puzzle space before it is archived on the stack. The information enables function Think to select "guess" the next number after a backtracking step is performed.

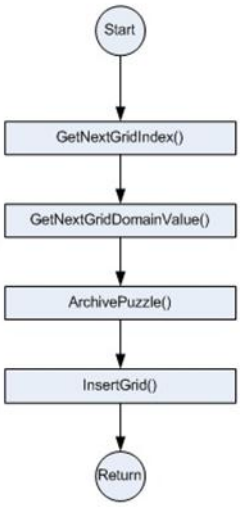


**Figure 3 – Function Think ()**

Figure 4 shows the function Backtracking which implements a simple chronological backtracking method. Function Restore is called to remove the last puzzle state stored on the stack and reinitializes the current state of the puzzle to the last known consistent state. The incorrect number guessed last is also removed from the associated domains. If the domain for the newly selected GridIndex contains more than one number, the first available number in the domain is inserted into the puzzle grid. However if only one number remains in the selected domain, a new iteration step is required to allow logic reasoning to select that number. In this case guessing is not required unless the new state proves inconsistent.

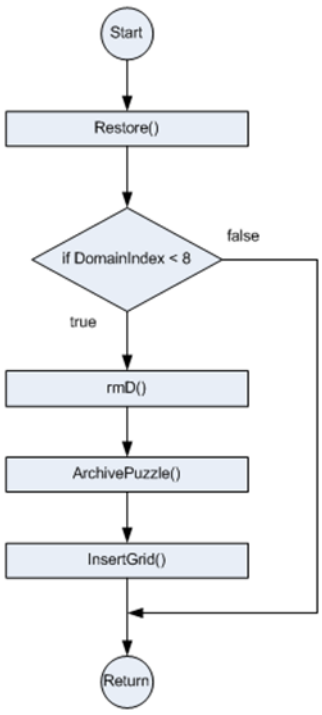


**Figure 4 – Function BackTracking ()**

## 4 TEST DATA

The performance of the Sudoku CSP algorithm was tested with four sets of puzzles organized in categories, easy, medium, hard, and evil. The easy and medium type puzzles which were selected to evaluate performance of the algorithm have an average of 33 numerics and 28 numerics initially given, respectively. The hard and evil type puzzles have an average of 26 numerics and 24 numerics initially given, respectively. Table 1 through Table 4 shows specifics for each puzzle.

**Table 1 - Easy Puzzles**

| Puzzle Type | Total Numbers Given |
|---|---|
| Puzzle Easy 1 | 36 |
| Puzzle Easy 2 | 35 |
| Puzzle Easy 3 | 36 |
| Puzzle Easy 4 | 35 |
| Puzzle Easy 5 | 35 |
| Puzzle Easy 6 | 33 |
| Puzzle Easy 7 | 32 |
| Puzzle Easy 8 | 33 |
| Puzzle Easy 9 | 31 |
| Puzzle Easy 10 | 32 |

**Table 2 - Medium Puzzles**

| Puzzle Type | Total Numbers Given |
|---|---|
| Puzzle Medium 1 | 30 |
| Puzzle Medium 2 | 30 |
| Puzzle Medium 3 | 30 |
| Puzzle Medium 4 | 27 |
| Puzzle Medium 5 | 27 |
| Puzzle Medium 6 | 26 |
| Puzzle Medium 7 | 29 |
| Puzzle Medium 8 | 26 |
| Puzzle Medium 9 | 27 |
| Puzzle Medium 10 | 29 |

The easy and medium puzzles can be solved using logic reasoning and do not require guessing. Therfore the number of iterative steps required should be 81 cells minus the number of cell values initially provided.

Hard and evil problems require a combination of logic resoning, guessing and backtracking. The number of guesses required to solve a particular puzzle is partially dependent on the number of numerics provided and their distribution in the puzzle space. Typically puzzles with less than thirty numerics given require some intelligent guessing. There are no known Sudoku puzzles with 16 or less preassigned numerics, because they would lead to more than one solution [10]. Figure 5

shows such a puzzle with zero numerics given and the first discovered solution.

**Table 3 – Hard Puzzles**

| Puzzle Type | Total Numbers Given |
|---|---|
| Puzzle Hard 1 | 28 |
| Puzzle Hard 2 | 27 |
| Puzzle Hard 3 | 28 |
| Puzzle Hard 4 | 26 |
| Puzzle Hard 5 | 26 |
| Puzzle Hard 6 | 28 |
| Puzzle Hard 7 | 26 |
| Puzzle Hard 8 | 27 |
| Puzzle Hard 9 | 22 |
| Puzzle Hard 10 | 25 |

**Table 4 – Evil Puzzles**

| Puzzle Type | Total Numbers Given |
|---|---|
| Puzzle Evil 1 | 23 |
| Puzzle Evil 2 | 23 |
| Puzzle Evil 3 | 26 |
| Puzzle Evil 4 | 25 |
| Puzzle Evil 5 | 25 |
| Puzzle Evil 6 | 26 |
| Puzzle Evil 7 | 24 |
| Puzzle Evil 8 | 27 |
| Puzzle Evil 9 | 27 |
| Puzzle Evil 10 | 23 |

## 5 RESULTS

The Sudoku CSP algorithm was tested with forty puzzles of type easy, medium, hard and evil. The test results show, Table 5 and Table 6, that easy and medium puzzles were solved using logic reasoning only. The average running times required to solve easy and medium puzzles is 9.7 milliseconds and 12.4 milliseconds, respectively.

Table 7 and Table 8 reflect the results for hard and evil puzzles. Both categories of problems required guessing and backtracking. The evil problems selected for this test required between 500 and 25,425 guess steps. Results show that the running times exponentially increase with the number of backtracking steps required. The average running times required to solve hard and evil puzzles is 23.9 milliseconds and 4.762 seconds, respectively.

A blank puzzle space without any numerics provided and its associated solution is shown in Figure 5. The computational time required to solve this puzzle is 469 milliseconds. 2289 guesses and 2230 backtracking steps were required to solve the puzzle. The solution reflects how the thinking process selects the first numbers available in the grid domains, resulting in the sequences {1, 2, 3, …, 9} and {4, 5, 6, …, 3} in the first two rows.

**Table 5 - Sudoku, Easy Puzzles**

| C++ | Iterative Steps | Logic Steps | Guess Steps | Backtracking Steps | Time sec |
|---|---|---|---|---|---|
| Puzzle Easy 1 | 45 | 45 | 0 | 0 | 0.0100005 |
| Puzzle Easy 2 | 46 | 46 | 0 | 0 | 0.0100006 |
| Puzzle Easy 3 | 45 | 45 | 0 | 0 | 0.0100005 |
| Puzzle Easy 4 | 46 | 46 | 0 | 0 | 0.0090005 |
| Puzzle Easy 5 | 46 | 46 | 0 | 0 | 0.0100006 |
| Puzzle Easy 6 | 48 | 48 | 0 | 0 | 0.0100006 |
| Puzzle Easy 7 | 49 | 49 | 0 | 0 | 0.0110006 |
| Puzzle Easy 8 | 48 | 48 | 0 | 0 | 0.0015600 |
| Puzzle Easy 9 | 50 | 50 | 0 | 0 | 0.0156000 |
| Puzzle Easy 10 | 49 | 49 | 0 | 0 | 0.0100006 |

**Table 6 - Sudoku, Medium Puzzles**

| C++ | Iterative Steps | Logic Steps | Guess Steps | Backtracking Steps | Time sec |
|---|---|---|---|---|---|
| Puzzle Medium 1 | 51 | 51 | 0 | 0 | 0.0110006 |
| Puzzle Medium 2 | 51 | 51 | 0 | 0 | 0.0110006 |
| Puzzle Medium 3 | 51 | 51 | 0 | 0 | 0.0110007 |
| Puzzle Medium 4 | 54 | 54 | 0 | 0 | 0.0110006 |
| Puzzle Medium 5 | 54 | 54 | 0 | 0 | 0.0110006 |
| Puzzle Medium 6 | 55 | 55 | 0 | 0 | 0.0120007 |
| Puzzle Medium 7 | 52 | 52 | 0 | 0 | 0.0110006 |
| Puzzle Medium 8 | 55 | 55 | 0 | 0 | 0.0156001 |
| Puzzle Medium 9 | 54 | 54 | 0 | 0 | 0.0156001 |
| Puzzle Medium 10 | 52 | 52 | 0 | 0 | 0.0156000 |

**Table 7 - Sudoku, Hard Puzzles**

| C++ | Iterative Steps | Logic Steps | Guess Steps | Backtracking Steps | Time sec |
|---|---|---|---|---|---|
| Puzzle Hard 1 | 65 | 58 | 11 | 4 | 0.0480027 |
| Puzzle Hard 2 | 112 | 105 | 12 | 8 | 0.0230014 |
| Puzzle Hard 3 | 66 | 60 | 9 | 3 | 0.0140008 |
| Puzzle Hard 4 | 111 | 102 | 17 | 14 | 0.0220012 |
| Puzzle Hard 5 | 209 | 188 | 37 | 23 | 0.0440025 |
| Puzzle Hard 6 | 53 | 51 | 2 | 0 | 0.0110007 |
| Puzzle Hard 7 | 102 | 93 | 15 | 8 | 0.0220013 |
| Puzzle Hard 8 | 83 | 81 | 4 | 2 | 0.0170010 |
| Puzzle Hard 9 | 103 | 96 | 11 | 5 | 0.0210012 |
| Puzzle Hard 10 | 83 | 78 | 7 | 2 | 0.0170009 |

**Table 8 - Sudoku, Evil Puzzles**

| C++ | Iterative Steps | Logic Steps | Guess Steps | Backtracking Steps | Time sec |
|---|---|---|---|---|---|
| Puzzle Evil 1 | 12841 | 11231 | 4079 | 4075 | 1.6000915 |
| Puzzle Evil 2 | 35418 | 32724 | 5666 | 5655 | 4.1372367 |
| Puzzle Evil 3 | 46216 | 43915 | 5551 | 5543 | 5.4033091 |
| Puzzle Evil 4 | 17455 | 15513 | 4493 | 4482 | 2.0871194 |
| Puzzle Evil 5 | 67162 | 61116 | 13501 | 13485 | 8.1034635 |
| Puzzle Evil 6 | 6104 | 5806 | 652 | 643 | 0.7260415 |
| Puzzle Evil 7 | 50611 | 44351 | 12792 | 12781 | 5.9443400 |
| Puzzle Evil 8 | 3994 | 3667 | 779 | 770 | 0.5000286 |
| Puzzle Evil 9 | 10526 | 10097 | 1019 | 1012 | 1.2680725 |
| Puzzle Evil 10 | 149942 | 138808 | 25425 | 25410 | 17.8570214 |

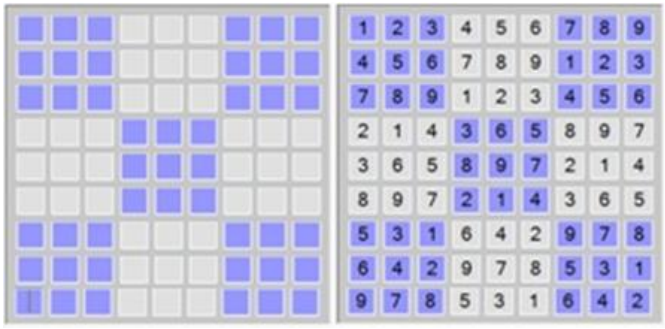

**Figure 5 – Blank Sudoku and Associated Solution**

## 5.1 Computational Complexity

The total number of possible models in the Sudoku puzzle space is 6,670,903,752,021,072,936,960. Eliminating symmetrically equivalent models reduces the space to 5,472,730,538 unique solutions. Since solving partially completed Latin squares reside in the NP-Complete domain [11], the Sudoku CSP algorithm also resides in this complexity class.

The logic reasoning steps and constraint propogation performed by the algorithm greatly reduces the computational costs associated with chronological backtracking. Although the easy and medium type problems are solved in polynomial time; hard and evil type problems require exponential long times to solve. The computational time differences between hard and evil problems are shown in Table 7 and Table 8. The difference between the two type problems is in the reduction of initial numerics provided.

Arto Inkala (AI), a Finnish mathematician, identified one of the most difficult Sudoku puzzles. It is known as Escargot in the AI circles, shown in Figure 6, and challenges most Sudoku algorithms. Puzzles like AI Escargot require exponentially long times to solve. If the thinking process requires visiting every permutation in the puzzle then the algoritm could take a very long time to render a solution.

**Figure 6 – AI Escargot**

The Sudoku CSP algorithm solved AI Escargot in 4.5422 seconds and required 18530 iterative steps, 17836 Logic Steps, 1386 Guess Steps and 1375 Backtracking Steps. In comparison to type Evil puzzles, we find Evil #10, Figure 7, more difficult to solve. Both, Evil #10 and AI Escargot have only 23 initial numerics given.

**Figure 7 – Evil #10**

## 6 Conclusion

We introduced in this paper a classical AI solution for constraint satisfaction problems. The Sudoku CSP algorithm is robust and handles all models in the puzzle space. Tests have shown that the complexity of Sudoku puzzles varies greatly depending on the initial number of numerics given. Easy and medium type puzzles can be solved in polynomial times; however, hard and evil problems require exponential running times.

The algorithm was designed to demonstrate academic basics of constraint satisfaction problems, backtracking, and constraint propogation. Puzzle types such as AI Escargot and Evil 10 challenge CSP algorithms, since they require large amounts of guessing and backtracking. Improvements to the algorithm can be made by implementing a search-rearrangement method. Selecting an unsolved cell in the grid with the least amount of domain choices reduces the probability of making an incorrect guess up to fifty percent.

## REFERENCES


[1] "Sudoku", Wikipedia, May 2011.

[2] Moraglio, A.; Togelius, J.; Lucas, S.; "*Product Geometric Crossover for the Sudoku Puzzle*", IEEE Evolutionary Computation, 2006. CEC 2006.

[3] Pacurib,J.A.; Seno, F.M.M.; Yusiong, J.P.T.; "Solving Sudoku *Puzzles Using Improved Artificial Bee Colony Algorithm*", IEEE Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference.

[4] Papdimitriou, "*Computational Complexity*", Addison-Wesley, ISBN: 0-201-53082-1.

[5] Zhe Chen, "*Heuristic Reasoning on Graph and Game Complexity of Sudoku"*, LATTIS, INSA, University of Toulouse.

[6] Lauren Aaronson, "*Sudoku Science*", IEEE Spectrum, Feb 2005.

[7] Kim Marriott, and Peter J. Stuckey, "*Programming with Constraints, An Introduction"*, The MIT Press, 1998 ISBN: 0-262-13341-5.

[8] Vipin Kumar, "*Algorithms for Constraint Satisfaction Problems: A Survey*", AI magazine, AAAI, 1992.

[9] Bitner, J., Reingold, E.M., "*Backtrack Programming Techniques*", ACM 18:651-655.

[10] I.Lynce and J. Quaknine, "*Sudoku as a SAT problem"*, Electronic Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, 2006.

[11] Charles J. Colbourn, "*The complexity of completing partial Latin squares*" Department of Computational Science, University of Saskatchewan, S7N OWO, Canada, 26 March 2002.



**Holger Findling** is a senior member of IEEE and received the BSEE degree from the University of Central Florida in 1986, the M.S.M. in Contract Management, 1989, and the MSCS degree, 1996, from the Florida Institute of Technology. He joined the Lockheed Martin Corporation in 1986 and worked large-scale development programs such as Automated Fingerprint Identification System (AFIS), DDG-Modernization, and DDG-1000. His research interest is in computational intelligence and algorithm design. He is teaching as an Adjunct Professor at the Orlando Graduate Center, Florida Institute of Technology.