

Solving Killer Sudoku using Dancing Links

Yalin Dogrusoz

1 Abstract

Sudoku is a logic puzzle that has been extensively study in recreational mathematics and computer science. This paper focuses on Killer Sudoku, a variant that combines traditional Sudoku rules with additional arithmetic “cage” constraints. The investigation uses of Donald Knuth’s Algorithm X, implemented with the Dancing Links (DLX) data structure—a 2-dimensional doubly linked list designed to efficiently solve the exact cover problem—to apply Killer Sudoku to the exact cover problem by representing cage constraints as choices and implements a Killer Sudoku solver. The implementation confirms that DLX provides a highly efficient method for solving both classical and Killer Sudoku instances, with run-times on test puzzles showing practical viability; the implemented Sudoku solver can be used to generate all solutions to a given problem, and hence also confirm that a given puzzle only has one unique solution as desired. Beyond Sudoku, these findings highlight the adaptability of exact cover formulations to broader constraint satisfaction problems.

2 Introduction

Sudoku, a Japanese game with international appeal, was invented in 1979 and gained popularity very quickly. A standard Sudoku grid consists of $n^2 \times n^2$ cells, most often 9×9 , partitioned into $n \times n$ sub-grids called *blocks*, with n^2 blocks in total. Each individual cell in the grid must be filled with a *single digit* from 1 to n^2 such that the grid satisfies the following rules:

- No two cells in the same row contain the same digit.
- No two cells in the same column contain the same digit.
- No two cells in the same block contain the same digit.

Sudoku puzzles start with some cells already containing numbers; these filled cells are called *clues*, and they ensure that there is a unique solution to each Sudoku puzzle and give the solver a starting point. Figure 1 shows an example Sudoku puzzle and its completed solution.

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

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

Figure 1: A Sudoku puzzle with clues and its unique solution [15].

It is desirable to develop and run an algorithm capable of solving a given Sudoku puzzle efficiently; this allows the determination of a solution to a given puzzle quickly, as well as verifying that the puzzle has a unique solution as desired. From both a recreational and computational perspective, uniqueness is crucial: a puzzle with multiple solutions lacks the logical challenge that makes Sudoku engaging.

This paper will give technical preliminaries in section 3 on the mathematics behind Sudoku and the algorithms and data structures developed to efficiently solve it, and in section 4 develop and implement an algorithm to efficiently solve Killer Sudoku, a variant of Sudoku using the ideas introduced in section 3. Section 5 shortly gives results, evaluations and implications of this paper.

3 Technical Preliminaries

3.1 Killer Sudoku

Killer Sudoku is a variant of the normal Sudoku puzzle. All rules of Sudoku apply, and alongside, there are two additional constraints. The irregular shapes among cells outlined with dashed lines in figure 2 – called *cages* – introduce two additional constraints; first, the entries in the cells within a cage must sum to the number labeled; second, no two cells in the same cage should share the same number (even if they are in different rows, columns, and blocks). Consequently, Killer Sudoku usually requires much fewer clues to be given as digits, as the cages themselves are also clues that limit the possible solutions.

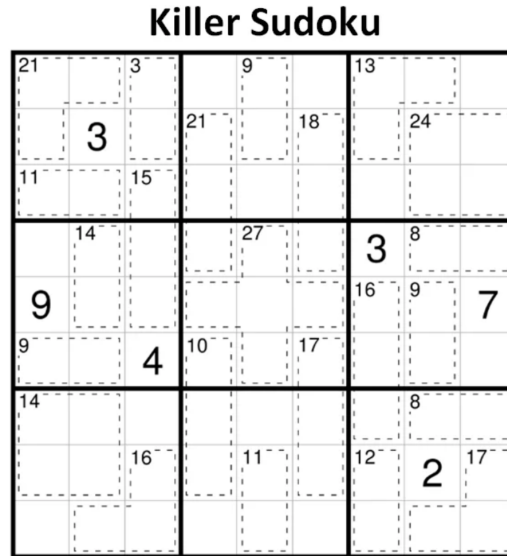


Figure 2: Example Killer Sudoku Puzzle [14].

3.2 Sudoku as a Graph Coloring Problem

Graph coloring, also referred to as vertex coloring, is the problem of assigning colors to vertices such that no two adjacent vertices have the same color. If a proper coloring is produced using at most k colors, that is called a k -coloring. [9]

The constraints of a Sudoku puzzle—that certain cells cannot share the same digit—can be represented as a *graph coloring* problem. Each cell corresponds to a vertex, and edges connect cells that must contain different digits. Figure 3 shows an example of a Sudoku solution interpreted as a graph.

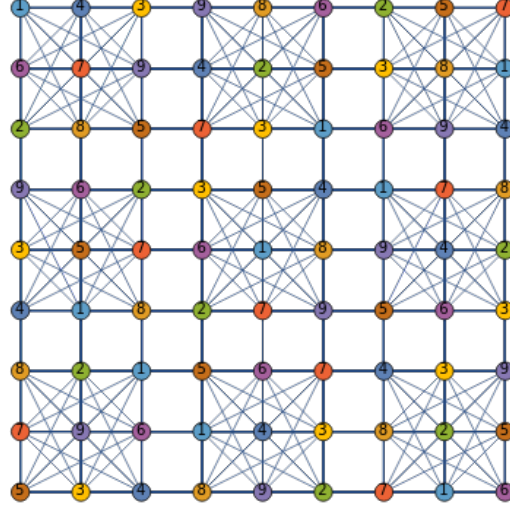


Figure 3: Sudoku grid interpreted as a graph coloring problem. Although not explicitly shown, every vertex in the same row are adjacent to each other, and every vertex in the same column are adjacent to each other [4].

3.3 Mathematical Representation

For an $n^2 \times n^2$ Sudoku grid, let $\{i, j\}$ be the vertex in the i -th row and j -th column. The pair of edges $\{a, b\} \{i, j\} \in E(G)$ if and only if at least one of the following holds:

- $a = i$
- $b = j$
- $\left\lceil \frac{a}{n} \right\rceil = \left\lceil \frac{i}{n} \right\rceil$ and $\left\lceil \frac{b}{n} \right\rceil = \left\lceil \frac{j}{n} \right\rceil$.

Each vertex is adjacent to $3n^2 - 2n - 1$ others: $n^2 - 1$ other cells in the same row, $n^2 - 1$ other cells in the same column and $(n - 1)^2$ other cells in the same block, excluding the ones already counted. [11] Thus, solving Sudoku corresponds to finding a proper n^2 -coloring of this graph.

3.4 Computational Complexity

The problem of solving a partially filled Sudoku puzzle is NP-complete(nondeterministic polynomial-complete), meaning it belongs to a class of problems that are both in NP and as hard as any problem in NP. Here, “nondeterministic” refers to a model of computation in which the machine, when faced with multiple choices, is assumed to always pick the correct one that leads to a solution: a theoretical model not possible to implement with current technology. [5] For Sudoku, given a completed grid, it’s easy to check that it satisfies all the rules, so verifying a solution takes only polynomial time. But actually finding that solution may require an exponential runtime, which is what makes it computationally hard. Yato and Seta showed that this difficulty is representative of all NP problems by reducing Sudoku

to 3-SAT, a well-known problem proven to be NP-complete. [20] However, Sudoku’s NP-completeness doesn’t mean that it is totally unfeasible to solve a 9×9 Sudoku. Heuristics and algorithms can create efficient solutions, with Algorithm X solving a “hard” 9×9 Sudoku in 10 seconds. [18]

Consequently, determining the number of solutions to a Sudoku puzzle—that is, finding the number of valid n^2 -colorings in a partially n^2 -colored graph—is #P-complete, signifying that it is even harder than NP-complete problems in terms of counting complexity. [20]

There exist two primary methods used for counting the number of solutions to a Sudoku puzzle; one algorithmic and one theoretical. The former method is simply running an algorithm—the most efficient being Algorithm X, discussed in Section 3.5—that finds a solution until all possible solutions are found. The latter uses chromatic polynomials of graph theory to find the number of valid colorings: the number of solutions.

3.4.1 Theoretical Counting: Chromatic Polynomials

The chromatic polynomial $P(G, k)$ expresses the number of k -colorings in a graph G as a polynomial which is a function of k . The chromatic polynomial can be obtained with the deletion contraction recurrence: for any edge e of a loop-free graph G ,

$$P(G, k) = P(G - e, k) - P(G/e, k).$$

Here G/e refers to the contraction of edge e : the removal of the edge and merging of its two endpoints, decrementing both the edge and vertex count. [11] Following through this recursive definition of the chromatic polynomial for smaller Sudoku puzzles like the 4×4 Sudoku is relatively easy and allows the determination of the number of solutions. The recurrence can be repeated until the graph reduces to either of the elementary graphs

- K_n having $P(K_n, k) = k(k-1)(k-2) \cdots (k-n+1)$ or
- Edgeless graph with n vertices having $P(G, k) = k^n$.

In theory, using the recurrence until every graph reduces to one of the above could take up to 2^e reductions where e is the number of edges—56 for the 4×4 Sudoku—allows the determination of the chromatic polynomial. The number of 4-colorings in a fully empty Sudoku grid was found to be 288 using chromatic polynomials computed using software: a task that wasn’t relatively computationally intensive [11]. This can also be thought of combinatorially, verifying the number 288; for more, refer to [16].

Yet, for larger graphs like the 9×9 Sudoku, the number of edges grow exponentially producing a chromatic polynomial of degree 81 (the degree of the polynomial is always equal to the order of the graph) becomes unreasonable with current technology and only lower and upper bounds can be determined.

3.4.2 Asymptotic Notation

To analyze algorithm performance, asymptotic notation is used as follows:

- **Big O (O)** – Describes an *upper bound* on the algorithm’s running time. For the “worst-case” input, the algorithm will never take more than this much time to run.

Formally, let $f(n)$ and $g(n)$ be real-valued functions defined on an unbounded subset of positive integers such that $f(n) = O(g(n))$.

There must exist $c > 0$ and $n_0 \in \mathbb{Z}^+$ such that $0 \leq f(n) \leq cg(n) \forall n \geq n_0$.

The values of c and n_0 must be fixed for function f and must not depend on the value of n . [3]

- **Average Case** – The expected time complexity over all possible inputs; an average case.
- **Big Omega (Ω)** – Describes a *lower bound* on the algorithm’s running time. For the “best-case” input, the algorithm will still take at least this much time to run.

Formally, let $f(n)$ and $g(n)$ be real-valued functions defined on an unbounded subset of positive integers such that $f(n) = \Omega(g(n))$.

There must exist $c > 0$ and $n_0 \in \mathbb{Z}^+$ such that $0 \leq cg(n) \leq f(n) \forall n \geq n_0$.

The values of c and n_0 must be fixed for function f and must not depend on the value of n . [2]

3.5 Algorithm X and dancing links (DLX)

The simplest yet most inefficient algorithm to find a solution to a Sudoku problem, one or all, would be an exhaustive and greedy one which tries all combinations and backtracks whenever a constraint is not satisfied.

A better approach to solving this NP-complete problem is a different heuristic algorithm found by Donald Knuth: Algorithm X. Algorithm X applies Sudoku to an *exact cover problem*. [12]

3.5.1 Exact Cover Problem

The exact cover problem is a problem in which there are various subsets given, and the solution is to pick a set of those subsets such that the union of each subset creates all 1s in the whole row. For example, in the given binary matrix (2-dimensional data structure consisting of only 1s and 0s) below each row represents a subset.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Here, subsets R_1 , R_4 and R_5 can be selected as a solution to the exact cover problem. [12]

3.5.2 Sudoku as an Exact Cover Problem, Algorithm X

When applying Sudoku to an exact cover Problem, there exist n^6 combinations (r, c, d) representing the row, column and digit of all possible entries, as each of r, c and d have possible choices from 1 to n^2 . Furthermore, every combination satisfies exactly 4 constraints in Sudoku:

- Digit d in row r
- Digit d in column c
- Digit d in block $\left\lceil \frac{r}{n} \right\rceil, \left\lceil \frac{c}{n} \right\rceil$
- Only one digit in cell $\{r, c\}$.

When the set of n^6 combinations (n^2 rows, n^2 columns, n^2 digits) are represented as rows and the set of $4n^4$ constraints each of them satisfies is represented as columns having entry 1 if the constraint is satisfied by that combination and 0 if not, the binary matrix formed can be an exact cover problem to find the solution to the Sudoku problem.

For example, the row $(1, 1, 1)$ would satisfy 4 constraints:

- Digit 1 in row 1
- Digit 1 in column 1
- Digit 1 in block $\{1, 1\}$
- Only one digit in cell $\{1, 1\}$.

Hence, for the row representing this combination, 1s would be filled in columns representing the above constraints. In the selected group of subsets for the exact cover problem, no other row should have 1s in the same columns, as that would result in conflicts such as there being two 1s in row 1.

Formally, a set of n^4 rows R_1, R_2, \dots, R_{n^4} is selected from the set of n^6 possible rows in the exact cover binary matrix. If the set R_1, R_2, \dots, R_{n^4} satisfies

$$R_1 \cup R_2 \cup \dots \cup R_{n^4} = \vec{1}$$

and $\forall i, j \in \{1, 2, \dots, n^4\} \mid i \neq j$

$$R_i \cap R_j = \vec{0},$$

where $\vec{1}$ and $\vec{0}$ are vectors representing a string of 1s and 0s respectively, a valid Sudoku solution has been found.

Algorithm X [12] solves the exact cover problem for Sudoku following the steps below describing a heuristic trial and error approach:

1. Create a binary matrix A representing each combination of row, column and digit as a row, and columns as constraints.
2. Select the rows given as clues, and disregard the conflicting rows (remove all rows that have 1s in the same set of columns as those in already selected rows).

3. Deterministically, choose a column c with the least 1s (Any column would work, but the least 1s gives the least options: a heuristic).
4. Nondeterministically, choose a row r such that $A[r, c] = 1$ (row r satisfies the constraint of column c).
5. Include r in the partial solution, adding it to the group of subsets selected.
6. For each j such that $A[r, j] = 1$,
 Delete column j from A (remove all other columns satisfying the same constraint).
 For each i such that $A[i, j] = 1$,
 Delete row i from matrix A (remove all rows that satisfy the same constraint).
7. Repeat steps 3-6 until one of the below cases arise:
 - Case 1: A is empty: the problem has been successfully solved and the group of subsets chosen satisfy the exact cover problem and hence solve the Sudoku. Stop.
 - Case 2: In step 4, no r can be chosen such that $A[r, c] = 1$. This means that there is no solution with the currently added sets of rows (subsets). Backtrack one step and repeat.
 - Case 3: The algorithm exhausted all possible backtracking paths and the matrix never became empty. There are no solutions to the given Sudoku. Stop.

3.5.3 Dancing Links

While Algorithm X provides a logical strategy to solve exact cover problems, it requires frequent removal and restoration of rows and columns from a sparse¹ binary matrix, which are operations that are inefficient if performed on typical 2D arrays. More specifically, adding or removing an element to an $m \times n$ matrix—a 2D array with m rows and n columns—has time complexity $O(m \times n)$ as (in the worst case) the removal or addition of an element requires all other $m \times n$ elements to be shifted; [7] also, new arrays need to be created and copied onto, creating memory inefficiency. To address this, Donald Knuth introduced a specialized data structure for this task which he called *dancing links* – often abbreviated as DLX – to implement Algorithm X in a way that enables fast and efficient backtracking. [10]

In a doubly linked list, each element of the list has pointers to others (each element holds the address of adjacent elements), which allows iteration through the list. These pointers are commonly represented as arrows in figures, and endpoints in the linked list are implemented using null pointers (pointers pointing to nothing). For doubly linked lists, the addition or deletion of elements only require constant—that is, $O(1)$ —time reducing time complexity significantly compared to a 2D array implementation. [6]

The operations of *adding* and *removing* an element essentially can be done by *unlinking* in linked lists. A 1-dimensional linked list is shown for simplicity below:

Operators $L[x]$ and $R[x]$ respectively define what the left-pointer (in the backwards direction) of element x is pointing to, and what the right-pointer is pointing to.

Then, *unlinking* b would correspond to doing the two following operations:

¹A sparse matrix is one that has a majority of 0 entries

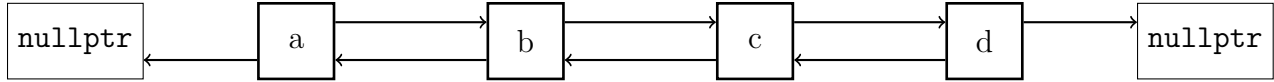


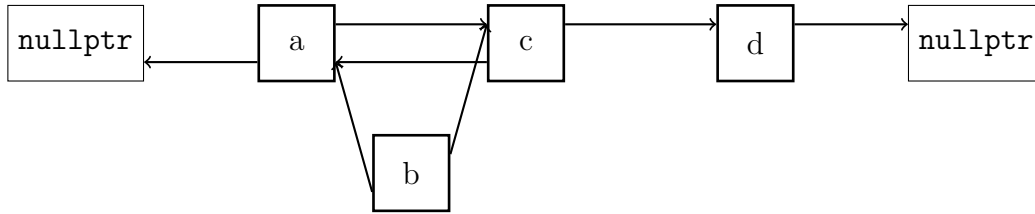
Figure 4: Doubly linked list example

$L[R[b]] = L[b]$ (This points c 's left-pointer to a)

$R[L[b]] = R[b]$ (This points a 's right-pointer to c).

Observe that b has nothing pointing to it (essentially removed), but b 's own left and right pointers still point at a and c respectively.

The resulting 1-dimensional doubly linked list after unlinking b is shown below:

Figure 5: Doubly linked list after unlinking b

Since b still points at a and c , it can be added back to the doubly linked list by performing operations

$R[L[b]] = b$ (This restores a 's right-pointer to b)

$L[R[b]] = b$ (This restores c 's left-pointer to b)

which would result in the original linked list being restored. [12]

DLX uses a 2-dimensional doubly linked list where traversal can happen left, right, up, and down. An example 2-dimensional doubly linked list is shown below in figure 6. Here, the second dimension counterpart of $L[x]$ and $R[x]$ would be $U[x]$ and $D[x]$ for pointers pointing up and down respectively.

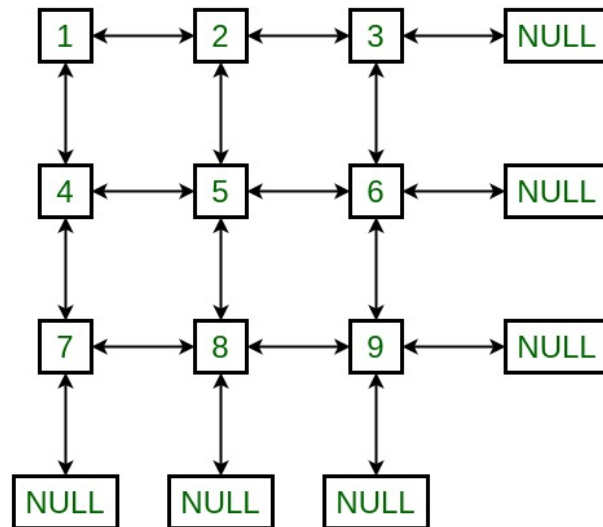


Figure 6: 2-Dimensional Doubly Linked List [8]

As shown in Figure 7, the Sudoku puzzle applied to an exact cover problem can be represented on a 2D doubly linked list (DLX) by having columns for constraints and rows for every “option” of combinations. Every entry of 1 in the binary matrix corresponds to an element in the DLX data structure, with every 1 in the same column being linked by vertical pointers up and down, and every 1 in the same row being linked by horizontal pointers left and right. Instead of having null pointers at the end of each column and row, pointers just link back to the start of the column or row, completing the cycle. Due to these cyclical or circular structures that form a ring on each row and circular, the doubly linked list is *toroidal*.

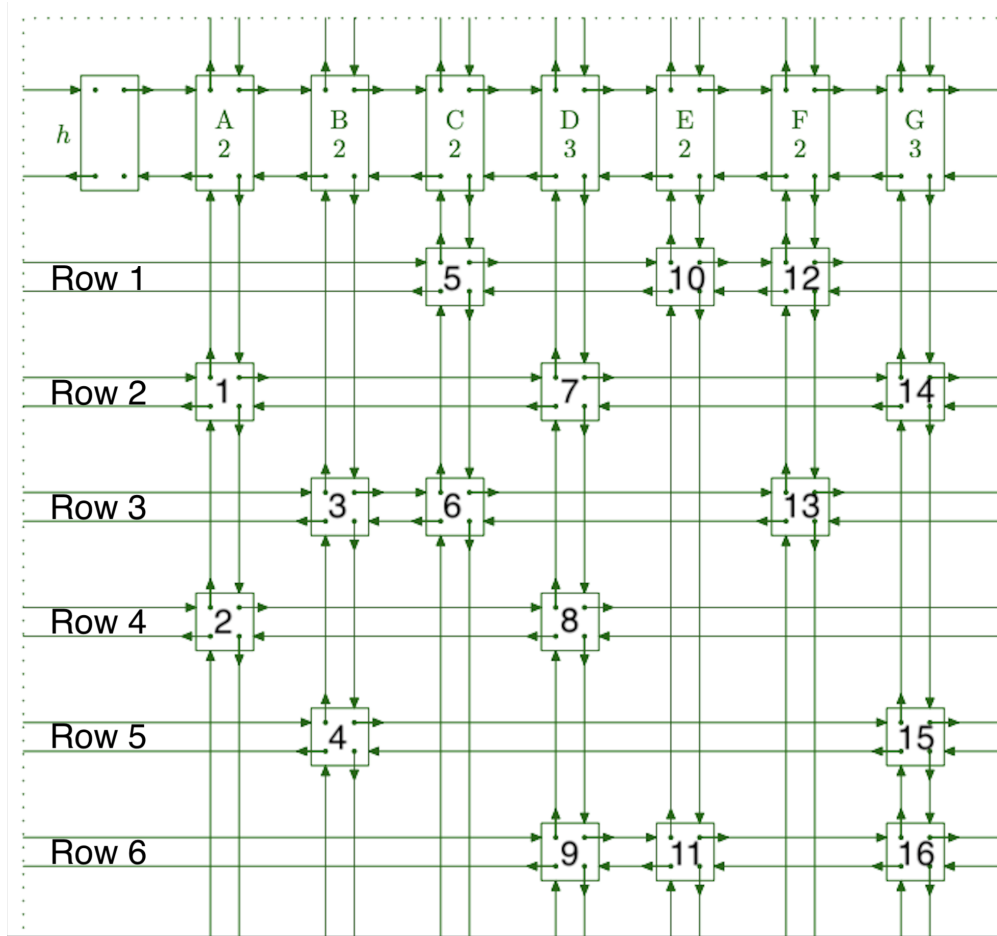


Figure 7: Dancing links (DLX). Note that rows and columns are labelled, and the numbers below the column labels represent the number of elements in that column, currently [12].

The application of Algorithm X to DLX illustrated by the example above is as follows:

1. Column *A* is chosen randomly out of the set of columns with the least number of elements.
2. One of the rows that has an element in column *A* must be selected to satisfy the constraint. Nondeterministically, the fourth row from the top (containing elements 2 and 8) is chosen to satisfy the constraint represented by column *A*.
3. Since no other row that satisfies the constraint represented by column *A* should be chosen, we traverse vertically (following the pointer) from column *A* and row 4 (element 2) to element 1 and delete that row (row 2) since that row satisfy the same constraint and hence won't be added together with row 4. More generally, after picking a certain row with an element in the selected column, other rows in that column are traversed to vertically, and those rows are deleted by traversing horizontally through the whole row until wrapping back to the beginning.
4. Along with the selected row—row 4—constraint (column) *D* is also satisfied with ele-

ment 8 being selected alongside element 2.

5. Since we satisfy column D with element 8, all other rows that have elements in column D must also be deleted the same way as done in 3. Hence, only rows 1, 3, and 5 remain after the addition of row 2 to the solution set and deletion of rows 4 and 6.
6. The non-satisfied column with the least number of elements is now column G , with only element 15 remaining to satisfy it. Hence, row 5 is added to the solution set and columns B and G are now satisfied due to elements 4 and 15 respectively.
7. Since B and G are satisfied now, all other rows with elements at B or G are not to be picked. Hence, row 3 is also deleted from the doubly linked list.
8. There remains only row 1 to satisfy the remaining constraints. Row 1 is added. The solution set is checked for if it is a valid solution. That is, if

$$R_1 \cup R_3 \cup R_7 = \vec{1}$$

and $\forall i, j \in \{1, 3, 7\} \mid i \neq j$

$$R_i \cap R_j = \vec{0},$$

This is satisfied, and hence rows 1, 3, and 7 yield a solution.

Note that another reason why DLX is a better way to store the Sudoku Puzzle as an exact cover Problem rather than a 2D array is that it is memory-efficient, as it only has to create elements for the 1s in the binary matrix and the 0s are not represented. Since for large Sudoku puzzles, all except a few entries in a row are 0s and the binary matrix is therefore largely sparse, this is highly advantageous.

For a 9×9 Sudoku, every row— (r, c, d) combination— satisfies exactly 4 constraints:

- A single digit at (r, c) ,
- Digit d in row r ,
- Digit d in row c ,
- Digit d in block $\left\lceil \frac{r}{n} \right\rceil, \left\lceil \frac{c}{n} \right\rceil$.

Since there are a total of 324 columns representing constraints, and that each row only has 4 elements, there is a considerable amount of sparsity, creating inefficiency in Sudoku's case for solving the exact cover problem without a specialized data structure.

4 Solving Killer Sudoku

The techniques of Algorithm X and Knuth's DLX can be applied to devise an efficient algorithm that solves (and perhaps counts solutions) to Killer Sudoku as well. Killer Sudoku's additional set of constraints—that is, the sum constraint in each cage—must be represented as a binary constraint to represent as an exact cover problem.

KenKen, also known as Mathdoku, is a puzzle which is similar to Killer Sudoku. In KenKen, the following rules apply:

- No two cells in the same row share the same number.
- No two cells in the same column share the same number.
- The arithmetic operation given in each cage must be satisfied. [17]

KenKens can have all 4 basic operations in cages instead of just the sum in Killer Sudoku. For example, for a cage of size two, it might ask that $\frac{a}{b} = 2$ for some elements a and b . To solve KenKens, most often a backtracking method without many heuristics called Constraint Programming (CP) is used. However, for Sudoku, heuristic methods like the DLX was found to be better than exhaustive backtracking methods like CP [1].

Hence, this paper will aim to apply Killer Sudoku to an exact cover problem, allowing it to be both memory-efficiently and time-efficiently solved by Algorithm X when using Knuth's DLX data structure. Although a CP method would be easier to implement with code, DLX outperforms this method with its advantages in time and memory efficiency.

Applying Killer Sudoku to DLX is challenging as its additional constraints need to be represented as binary values, since DLX solves the exact cover problem which works with binary matrices. In order to do this, the row and column constructions need to be revised. Instead of representing each row—the options of selection—as a combination of a digit in a row and column, each row will represent a valid entry across a whole *cage*. Since in Killer Sudoku, conventionally, cages are no bigger than 4-5 cells, the permutation of cell entries will not exceed computational ability and create major time inefficiency. Larger cells aren't desirable for real-world puzzles as humans solving them would also have to consider an immense number of permutations, making the puzzle neither feasible nor enjoyable.

To explain the cage-oriented method, consider this instance: a 9×9 Sudoku has a 2-cell sized cage at cells $\{1, 1\}$ and $\{1, 2\}$, with a sum constraint to add up to 6. Then the possible combinations of digits that could be placed in these two cells are pairs 1, 5 and 2, 4 which could be permuted across the two cells to produce 4 possibilities; note that 3, 3 is not a possibility because of the requirement that two cells in the same cage can not share the same number. Then, since 4 permutations exist, 4 rows are created corresponding to the solution of that cage, one of whom will be picked for the partial solution.

To address the columns in this new representation, the old constraints as following still apply.

- Digit d in row r
- Digit d in column c
- Digit d in block $\left\lceil \frac{r}{n} \right\rceil, \left\lceil \frac{c}{n} \right\rceil$
- Only one digit in cell $\{r, c\}$

Additionally, we add new columns for each cage (1 for each). A row entry that satisfies the conditions of cage i (every row will at least satisfy 1 cage's conditions as the rows are created accordingly), that row will have a 1 in the column corresponding to the i -th cage constraint in the binary matrix.

For instance, the permutation of having the digit 2 in $\{1, 1\}$ and 4 in $\{1, 2\}$ for the above example would satisfy the following constraints:

- Digit 2 in row 1
- Digit 4 in row 1
- Digit 2 in column 1
- Digit 4 in column 2
- Digit 2 in block $\{1, 1\}$
- Digit 4 in block $\{1, 1\}$
- Only one digit in cell $\{1, 1\}$
- Only one digit in cell $\{1, 2\}$
- Satisfies the sum constraint of cage 1.

Note that unique digits across a cage do not have to be checked as non-unique pairs of digits like $\{3, 3\}$ are not added in the first place, as only unique combinations of entries in a cage are calculated. Also, if there are any cells that aren't within a cage, these can be simply represented as an (r, c, d) combination as before, and not satisfy any of the column constraints.

This way of representing the Killer Sudoku problem with its additional constraints allows it to be turned into a binary matrix of boolean constraints, and applied onto the exact cover problem which can be efficiently solved with Knuth's DLX. The exact cover problem will have $4n^4 + c$ columns (constraints), where c is the number of cages, and as many rows as the sum of the number of all possible permutations for each cage or non-cage cell.

For an arbitrary cage of size k , the number of options (rows) is bounded by $O(\binom{n^2}{k} \times k!)$. This is because there are n^2 digits 1 to n^2 to choose combinations from—to add up to the required sum—and $k!$ different ways to permute the combination of unique digits.

5 Results

5.1 Implementation

The Java code to the implementation of DLX can be found on github.com/yalindogrusez/sudoku.

In the implementation above, the algorithm is coded to stop after finding a maximum of 10 solutions even when there are more in order to prevent excessive memory usage and consequently hinder a runtime error.

Furthermore, the implementation was successful and efficient, after tests were ran with solutions being found under 1 second consistently, making it practically viable. Below is a screenshot of the terminal for a sample Killer Sudoku encoded in the Java code and ran with the 6 results found:

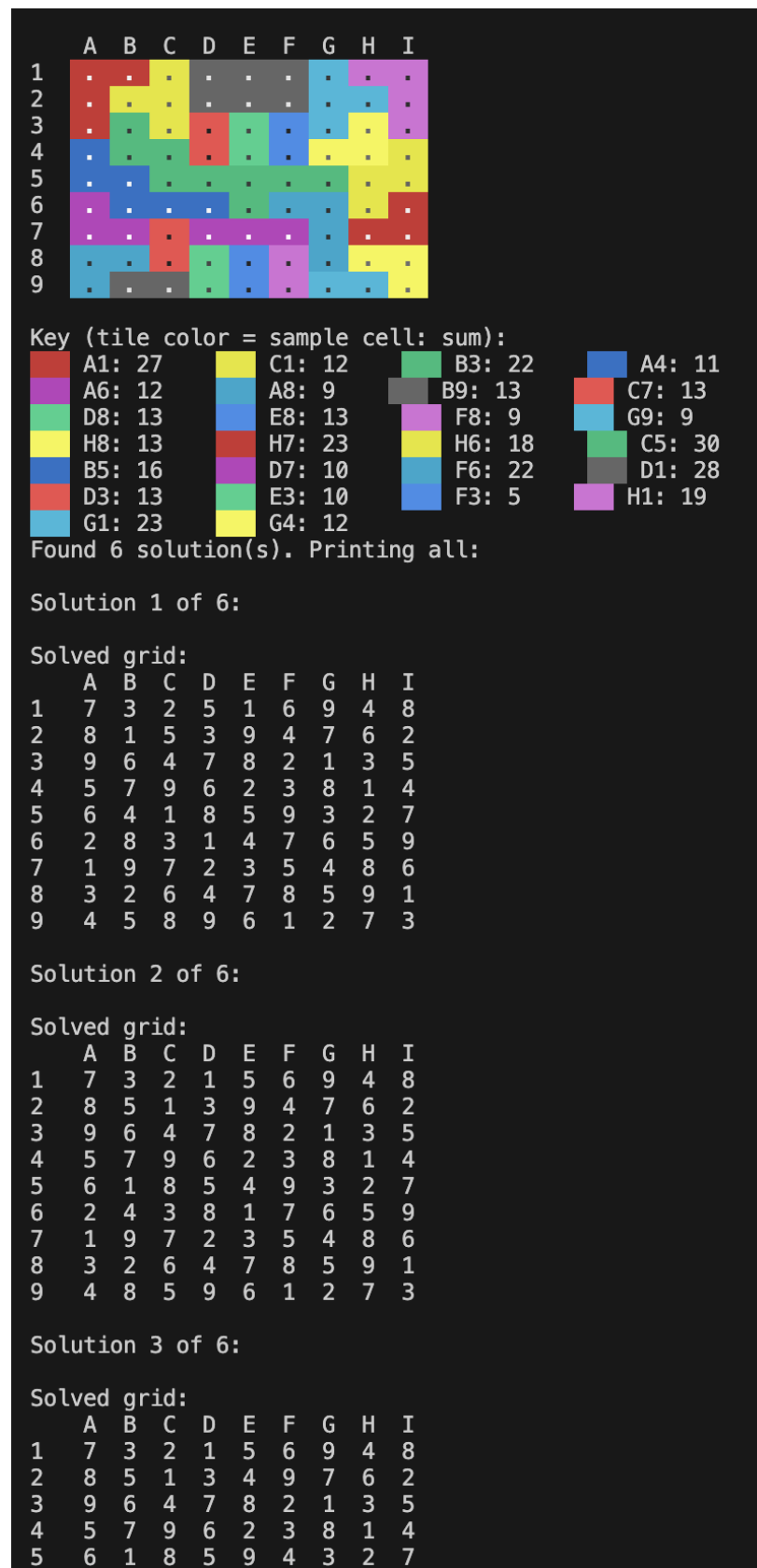


Figure 8: Example Run with Puzzle and its Solutions.

This implementation recursively finds all permutations for cage sums, and implements heuristics to reduce inefficiencies. Perhaps, for future reference, finding only combinations in increasing order and then reordering them to produce all permutations will be more computationally efficient.

Furthermore, the core *search()* function also works recursively by calling the *cover()* and *uncover()* functions with the partial solution to cover a column and backtrack to uncover it respectively, checking at each step for a dead end to backtrack or correct solution to stop.

5.2 Implications

While this study focused on Killer Sudoku, the framework has broader implications. Variants of Sudoku such as Diagonal Sudoku, Samurai Sudoku or Jigsaw Sudoku can be handled by adding further constraint columns, demonstrating the adaptability of Knuth's DLX.

More generally, this work illustrates how exact cover formulations and efficient data structures can extend to a wide range of constraint satisfaction problems, from recreational puzzles to practical domains such as scheduling or optimization. [19] Future research may also explore hybrid approaches that blend DLX with heuristic or human-like solving strategies, both for efficiency and for educational use in computer science and algorithms.

References

- [1] A. Bhattarai, D. Uprety, P. Pathak, S. N. Shrestha, S. Narkarmi, and S. Sigdel. A study of Sudoku solving algorithms: Backtracking and heuristic. <https://arxiv.org/abs/2507.09708>, 2025. arXiv:2507.09708 [cs.LO].
- [2] P. E. Black. Ω . <https://xlinux.nist.gov/dads/HTML/omegaCapital.html>, 2018. Dictionary of Algorithms and Data Structures [online], ed. Paul E. Black. Accessed: 2025-08-06.
- [3] P. E. Black. Big-O notation. <https://www.nist.gov/dads/HTML/bigOnotation.html>, 2019. Dictionary of Algorithms and Data Structures [online], ed. Paul E. Black. Accessed: 2025-08-06.
- [4] A. O. Duran and W. R. Inc. Solving Sudoku puzzles with graph theory. <https://community.wolfram.com/groups/-/m/t/2983903>, 2025. Accessed: 2025-07-16.
- [5] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.
- [6] A. Gautam. Exploring time complexity of doubly linked list operations. <https://programiz.pro/resources/dsa-doubly-linked-list-complexity/>, 2024. Accessed: 2025-08-06.
- [7] GeeksforGeeks. Time and space complexity of 1-d and 2-d array operations. <https://www.geeksforgeeks.org/dsa/>

- time-and-space-complexity-of-1-d-and-2-d-array-operations/, 2023. Accessed: 2025-08-06.
- [8] GeeksforGeeks. Construct a doubly linked list from 2d matrix. <https://www.geeksforgeeks.org/dsa/construct-a-doubly-linked-list-from-2d-matrix/>, 2025. Accessed: 2025-08-06.
- [9] GeeksforGeeks. Graph coloring applications. <https://www.geeksforgeeks.org/dsa/graph-coloring-applications/>, 2025. Accessed: 2025-08-08.
- [10] M. Harrysson and H. Laestander. Solving Sudoku efficiently with dancing links. <https://www.diva-portal.org/smash/get/diva2:770655/FULLTEXT01.pdf>, 2025. Accessed: 2025-07-23.
- [11] A. M. Herzberg and M. R. Murty. Sudoku squares and chromatic polynomials. *Notices of the American Mathematical Society*, 54(6):708–717, 2007. Accessed: 2025-08-06.
- [12] D. E. Knuth. Dancing Links. <https://arxiv.org/abs/cs/0011047>, 2000. arXiv:cs/0011047 [cs.DS].
- [13] D. E. Knuth. Stanford lecture: “dancing Links”. https://www.youtube.com/watch?v=_cR9zD1vP88, 2018. Presented via Stanford Online. Accessed: 2025-08-05.
- [14] C. U. Math Explorers’ Club. Killer Sudoku. <https://sudoku247online.com/killer-sudoku/>, 2025. Accessed: 2025-08-05.
- [15] C. U. Math Explorers’ Club. The math behind Sudoku. <https://pi.math.cornell.edu/~mec/Summer2009/Mahmood/Home.html>, 2025. Accessed: 2025-07-16.
- [16] C. U. Math Explorers’ Club. The math behind Sudoku: Four-by-four case. <https://pi.math.cornell.edu/~mec/Summer2009/Mahmood/Four.html>, 2025. Accessed: 2025-07-16.
- [17] M. R. McManus. How kenken puzzles work. HowStuffWorks, 2011. <https://entertainment.howstuffworks.com/puzzles/kenken-puzzles.htm>.
- [18] Ouked. sudoku_solver. https://github.com/ouked/sudoku_solver, 2022. Accessed: 2025-08-06.
- [19] H. Strømnes. Parallelizing algorithm x for solving exact cover problems, 2023. Bachelor’s Thesis.
- [20] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(5):1052–1060, 2003. Accessed: 2025-08-06.