Lisbon School of Economics & Management
Universidade de Lisboa

MASTER

MÉTODOS QUANTITATIVOS PARA A DECISÃO ECONÓMICA E EMPRESARIAL

MASTER´S FINAL WORK

PROJECT ELABORATED FOR OBTAINING THE MASTER'S DEGREE

SUDOKU SOLVER BASED ON HUMAN STRATEGIES

AN APPLICATION IN VBA

DANIELA MARQUES PAIS

SUPERVISION:
PROFESSOR DOUTOR FILIPE MANUEL GONÇALVES RODRIGUES

OCTOBER - 2022

ACKNOWLEDGMENTS

I would like to start by thanking my parents for their unconditional support throughout my existence and, in specific, during my academic journey. I am forever grateful to have you in my life!

The culmination of this cycle is also dedicated to my brother - the quiet guy who (ironically) is always one phone call away. Thanks for putting up with me all these years, for your help, for your patience.

This project would never have come to fruition without the human that backtracked all my mistakes. I was lucky enough to have your guidance throughout this experience.

Any error is of the entire responsibility of the cutest distraction ever, my last mile motivation … Ema.

GLOSSARY

CSPs - Constraint Satisfaction Problems

DL – Dancing Links

MFW – Master’s Final Work

MRV - Minimum Remaining Value

VBA - Visual Basic for Applications

Resumo

O Sudoku é um puzzle popularmente conhecido, com aplicações em diversas áreas que se estendem desde a Criptografia à Medicina. Por ser um problema NP-completo, a maior parte dos esforços para o resolver focam-se em heurísticas e não em métodos exatos. Exemplo destes últimos são as estratégias humanas. A proposta deste Trabalho Final de Mestrado (TFM) consiste no desenvolvimento de um Sudoku Solver, em VBA. O solver desenvolvido é um algoritmo de duas fases que incorpora estratégias humanas (Fase 1) e *backtracking* (Fase 2). A Fase 2 só é executada se, terminada a Fase 1, não for encontrada uma solução admissível.

Foi conduzida uma experiência computacional para testar a performance do solver para puzzles 9×9 de três níveis de dificuldade: fácil, moderado e difícil. Das 230 instâncias testadas, aproximadamente 55% foram resolvidas. O tempo máximo de resolução foi de 6,813 segundos, o tempo mínimo foi de 0,309 e a média do tempo total foi de 2,525 segundos.

PALAVRAS-CHAVE: Sudoku; Estratégias Humanas; *Backtracking*; *Dancing-Links*; Algoritmo X.

ABSTRACT

Sudoku is a popular puzzle, with applications in several areas ranging from Cryptography to Medicine. Because it is an NP-complete problem, most efforts to solve it focus on heuristics and not on exact methods. Examples of the latter are human strategies. The proposal of this Master's Final Work (MFW) is the development of a Sudoku Solver, in VBA. The developed solver is a two-phase algorithm that incorporates human strategies (Phase 1) and a backtracking procedure (Phase 2). Phase 2 is only executed if a feasible solution has not been found after Phase 1 ends.

It was conducted a computational experience to test the solver performance for 9×9 puzzles with three difficulty levels: easy, moderate, and hard. Among the 230 instances tested, approximately 55% were solved. The maximum running time was 6.813 seconds, the minimum time was 0.309, and the average total time was 2.525 seconds.

KEYWORDS: Sudoku; Human Strategies; Backtracking; Dancing-Links; Algorithm X.

TABLE OF CONTENTS

TABLE OF FIGURES

1. INTRODUCTION

Sudoku is a puzzle consisting of a $n \times n$ grid, where $n = \sqrt{m}$ and $n,\ m \in \mathbb{Z}^+$. The game begins with some numbers already in place (fixed cell). The objective is to fill the empty cells in the grid with integers from 1 to n in such a way that no number is repeated in each row, column, and mini-grid. A solution satisfying these conditions is called a feasible solution. A given sudoku might have several, none, or a unique feasible solution. As proven by McGuire et al. (2014), having at least 17 fixed cells is a necessary - but not sufficient – condition to ensure the uniqueness of the solution of any 9×9 puzzle.

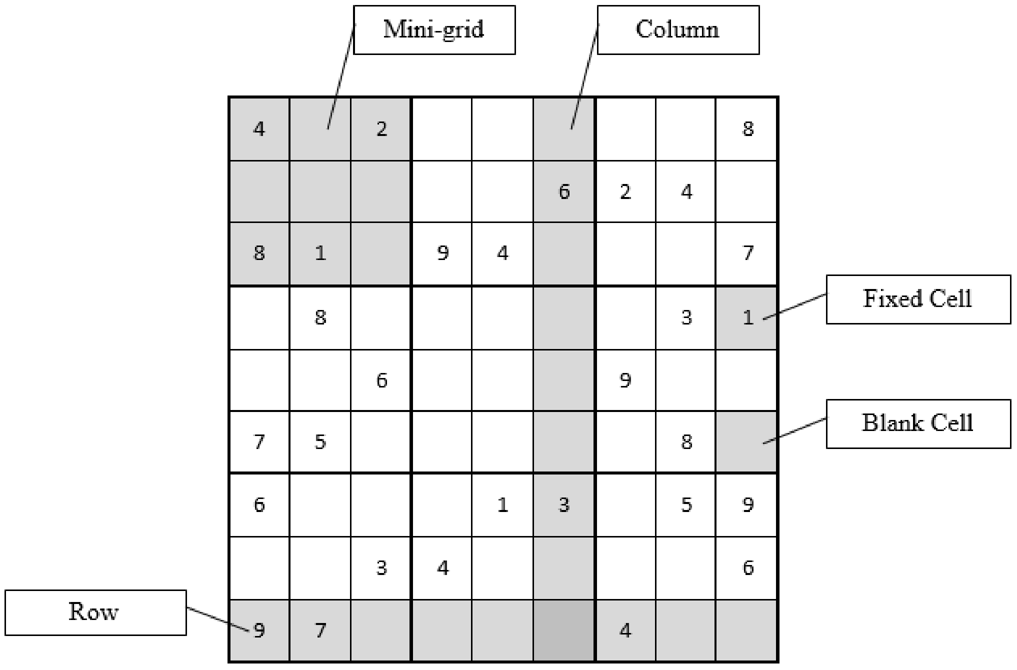


Figure 1: An example of a 9×9 Sudoku puzzle.

Sudoku was proved to be a NP-complete problem by Colbourn et al. (1984), which means that if the $P \neq NP$ conjecture is true, there is no polynomial-time algorithm that can solve all its instances. Hence, so far, the efforts to solve large-size sudokus focus mainly on developing and improving algorithms that are able to provide a solution close to a feasible one, without compromising neither on speed nor on generality[1]. Such algorithms include the genetic algorithm, simulated annealing, particle swarm optimization, ant colony, artificial bee colony, and variable neighbourhood search. [See Table 1 in Sevkli & Hamza (2019) for more information.] Furthermore, the difficulty of

[1] Generality is the algorithm's capacity to accommodate all possible inputs of the computational problem.

solving a sudoku puzzle is not only associated with the dimension of the grid (value of n) but also with the number of empty cells in the initial grid and the relative positioning of the fixed cells.

These puzzles can be modeled as constraint satisfaction problems (CSPs) that consist in assigning a value, within a finite domain, to each variable while ensuring that a set of restrictions is not violated. From an operations research perspective, the CSPs resulting from sudoku puzzles are similar to CSPs associated with different problems like assignment problems (assigning people to jobs, colors to a map, jobs to machines, etc.), protein folding (Strokach et al., 2020), and the ground-state problem (Ercsey-Ravasz & Toroczkai, 2012). Other applications include the fields of Cryptography (Rubinstein-Salzedo, 2018) and Steganography (Hong et al., 2008).

The wide range of practical applications resulting from sudoku puzzles combined with my interest for the game and the scarce exploration of exact algorithms to solve it were the most determinant factors for the choice of this Master's Final Work (MFW). This MFW proposes the development, in Visual Basic for Applications (VBA), of a sudoku solver mainly constructed on a human rule-based approach. The aim of this work is to generate feasible solutions while trying to mitigate the use of the far disclosed heuristics and minimize the time-consuming characteristic of these algorithms. The proposed solver will perform a backtracking algorithm, in case the human strategies executed reveal to be insufficient to reach a feasible solution. The backtracking algorithm choice is related to its efficiency for sudoku solving when compared to constraint programming as the results in Coelho and Laporte (2014) reveale.

The rest of the MFW is organized as follows. In Section 2, a summary of the exact algorithms used to solve sudoku puzzles is presented. In the subsequent section are introduced the human strategies intended to be implemented and it is detailed the functioning of Algorithm X. Section 4 deepens the way the solver algorithm procedures interact. The computational experience is reported and discussed in Section 5. Finally, Section 6 states the main conclusions of this work. Improvement points and future research directions are also discussed in this section.

2. LITERATURE REVIEW

The existing sudoku solvers are based on either exact or heuristic algorithms. The fundamental difference between them is that exact algorithms aim to provide a feasible solution to a given problem and not an approximation to it. For this reason, the heuristic algorithms - genetic algorithm, particle swarm optimization, ant colony optimization, and artificial bee colony optimization, just to mention a few - will not be the subject of study of this MFW. [For more details on this matter see Mishra et al. (2018).] Instead, the focus of this project will be on the cursory explored exact algorithms.

This section briefly describes prior works based on exact approaches related to the current study[2].

2.1 Human Strategies

"Human strategies" is a term used to describe a series of deductions that human players created to solve a sudoku puzzle. According to the difficulty of the puzzle, it might be required to perform strategies with different degrees of complexity to obtain a feasible solution. Some of these strategies are basic, such as naked single and hidden single – we will delve into these further ahead -, and others are more elaborate, like X-Wing and Swordfish. Nevertheless, the application of pure human strategies to solve these puzzles inherits the weaknesses of its creators: they might not be enough to find a feasible solution or take a lot of time to do so, especially as the search domain becomes much larger. Hence, the human strategy approach can be vastly improved with the conjunction of other techniques that mitigate its limitations.

Eppstein (2005) showed how his path and cycle finding algorithm complements the solvability capacity of basic human solver strategies. A bilocation graph of a partially completed sudoku puzzle is constructed: first, it is drawn a vertex for each blank cell of the grid; then if two vertices lie in the same row, column or mini-grid and if within that row, column, or mini-grid they are the only ones that can contain the integer x, such vertices are connected with an edge labelled with the integer x. The path or cycle formed

[2] Have in mind that there are other exact approaches to solve sudoku puzzles like, for example, constraint programming (Simonis, 2005).

either indicates the possible values to be placed in the vertices or enables the deduction of a unique value that can be placed in a particular vertex. It also points out the consequences of such placements for the remaining vertices of the chain.

Another method is brought by Deodhare et al. (2014) that presented a computation model to solve sudokus inspired in biological processes: the P system. A set of rules is applied randomly in an attempt to find a unique candidate to fill a blank cell. If it fails to do so, it tries to remove candidate options through human solver strategies until there is only one candidate left. At this point, if there are still several options to fill the cell, a *brute-force* algorithm is applied.

2.2 Backtracking

The typical backtracking algorithm visits the blank cells in a predefined order and places a number from the solution space of that particular cell on it, having in consideration the numbers on the already filled cells and the sudoku's rules. If the current number fails to satisfy the constraints of the problem, it is removed from the cell along with the subsequent choices of placements in other cells that derived from it. In such circumstance, another number is tried and this process is repeated until a feasible solution is found or it is concluded that no solution exists.

This procedure entails large running times because a huge tree of possibilities needs to be examined given that it runs through all elements of the domain. However, the general mechanism of doing and undoing can be improved through adaptations that have into account the specificities of the problem we are trying to solve. These might include: i) sequential comparisons in an appropriate step of the algorithm to avoid auxiliary data structures that require several accesses to the memory of the computer to update and downdate them; ii) allowing every branch of the tree to choose the next blank cell and the integer to place on it in any order (instead of the common left-right and top-bottom search in the grid), in an attempt to obtain a better pruning of the search tree. Next are mentioned two clever ways to deal with this in the specific case of sudoku solving.

Jana et al. (2015) applied a backtracking algorithm to columns instead of individual cells, reducing the number of variables from n^2 to n. They started by constructing a tree for the column that has the maximum number of filled cells. Such tree has the valid permutations of numbers for the blank cells. From there, they attempted to build a solution incrementally, column by column, and backtracking whenever necessary.

A similar way to reduce computational time and redundancy is to perform backtracking on mini-grids. Maji and Pal (2014) began by constructing a tree for each mini-grid containing all its valid permutations. Then, a valid option for a mini-grid is chosen and the numbers are placed. Next, the algorithm searches for permutations in the adjacent mini-grids' trees to see if it can find a valid one that respects the restrictions imposed by the new filled cells. If so, it repeats the process just described. Otherwise, it backtracks to the prior mini-grid and selects another valid option.

3. UNDERSTANDING THE SOLVER PHASES

The sudoku solver developed in this MFW is a two-phase algorithm that incorporates human strategies (Phase 1) and backtracking (Phase 2), which we shall go into further detail in the next section. For now, it will be explained their operating mechanisms and introduced key concepts.

3.1 Terminology

A sudoku puzzle consists of a $n \times n$ grid, where $n = \sqrt{m}$ and $n,\ m \in \mathbb{Z}^{+}$. As exemplified in **Figure 1**, each grid is formed by n^2 cells that can be aggregated in different components: row, column, and mini-grid. Each component must contain all integer numbers from 1 to n exactly once, and this defines the constraints of the problem.

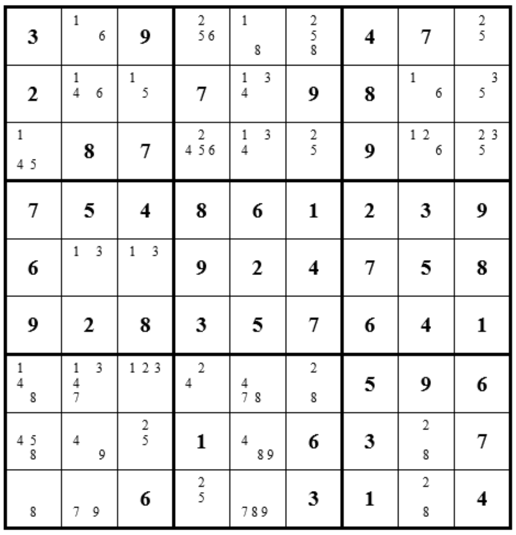
Rows, columns, and mini-grids are labelled with integer numbers from 1 to n in an ascending order, from top-bottom, right-left, and top-bottom and right-left, respectively. Thereby, cells are identified by a row-column coordination system: $[i, j]$ with $i, j \in \{1, 2, \dots, n\}$, where i represents the row number and j represents the column number. Cells initially filled with specific values are referred to as fixed cells and the remaining ones are named blank cells. The values that can be placed in each blank cell - satisfying the constraints of the problem - are called candidates.

3.2 Human Strategies

As stated in **Section 2.1**, human strategies are a series of deductions that human players created to solve a sudoku puzzle. After identifying the candidates for each blank cell, the human strategies described below can be employed:

1) **Naked Single:** for a specific cell, only one number fits. This number can be removed from the list of candidates of the other cells belonging to the same components (row, column, and mini-grid). See **Figure 2**.
2) **Naked Pair:** two cells sharing at least one component have the same set of two candidates. These numbers can be removed from the list of candidates of the cells that belong to the components that the two cells have in common. See **Figure 2**.

3) **Hidden Single (or Lone Ranger):** a number that is one of multiple candidates for a cell appears only once as a candidate in a row, column, or mini-grid. This number is the solution to that cell. See **Figure 2**.



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

Figure 2: Naked Single: in cell [9,1] only fits the number 8, then the number 8 can be removed from cells [7,1], [8,1], [9,5], and [9,8]. Naked Pair: cells [8,8] and [9,8] both have the numbers 2 and 8 as candidates, and they belong to the same mini-grid and column. This means that the number 2 can be eliminated from the candidates of cell [3,8]. Hidden Single: in cell [9,4] can be placed the number 5, because despite the fact that there are two candidates, for that mini-grid, it is the only cell where the number 5 can be positioned.

4) **Hidden Pair:** two cells sharing at least one component, have two numbers in common from their respective list of candidates. These numbers appear only in those cells within the limit of their common components. The other candidates can be removed from the referred two cells, leaving them with a naked pair. See **Figure 3**.

5) **Locked Candidate (or Pointing Pairs/Triplets):** a mini-grid has two/three occurrences of the same number and they are aligned on a single row or column. That number cannot fit in any other cell outside the mini-grid on that row or column. See **Figure 4**.

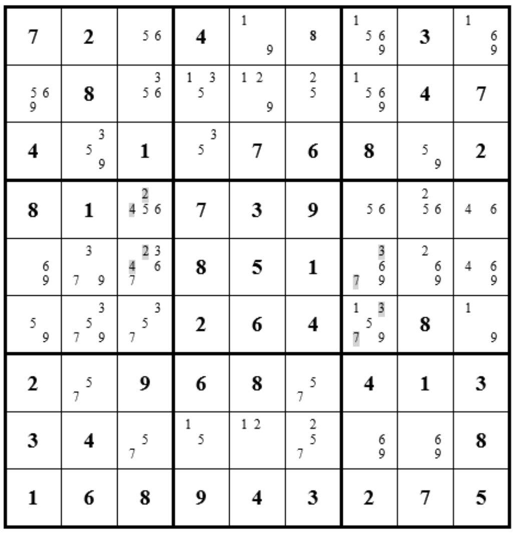


Figure 3: Hidden Pair: in mini-grid 6, the numbers 3 and 7 appear only in cells [5,7] and [6,7]. The numbers 6 and 9 can be eliminated from cell [5,7] and the numbers 1, 5, and 9 can be eliminated from cell [6,7]. Another hidden pair is highlighted in cells [4,3] and [5,3].

6) **Mini-grid/Line Reduction:** a column or row has the same number only in cells that belong to the same mini-grid. That number cannot fit in any other cell of that mini-grid. See **Figure 4**.

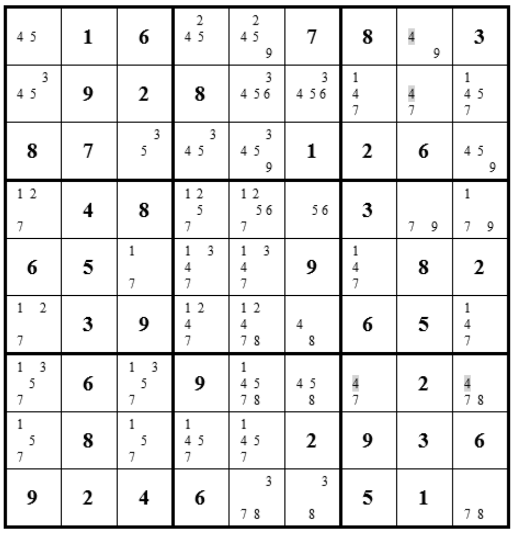


Figure 4: Locked Candidate: in mini-grid 9 the number 4 can only be placed in cells [7,7] and [7,9]. The number 4 can be removed from the cells of row 7 that do not belong to mini-grid 9. The number 4 can be eliminated from cells [7,5] and [7,6]. Mini-grid/Line Reduction: in column 8, the number 4 can either be placed in cell [1,8] or [2,8], that are both part of mini-grid 3. Thus, the number 4 can be excluded from the other cells of that mini-grid, namely cells [2,7], [2,9], and [3,9].

3.3 Backtracking with Algorithm X

Let us introduce some concepts related to backtracking. When dealing with backtracking algorithms, search trees are commonly used to display all possibilities of number placements depending on the sequential choices made. Each blank cell corresponds to a tree level. Every time a level is created, the candidates that can be positioned in the cell corresponding to that level are updated, regarding the selected candidates on the previous levels (**Figure 5**).

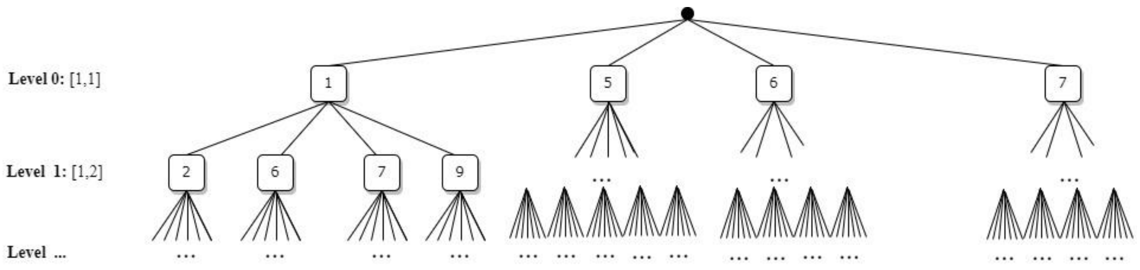


Figure 5: Example of a search tree for a simple backtracking algorithm trying to solve a sudoku puzzle. The initial candidates on cell [1,1] are {1,5,6,7} and on cell [1,2] are {1,2,6,7,9}. If the number 1 is placed on cell [1,1], then cell [1,2] loses one of its initial candidates. If instead, the number 5 had been placed on cell [1,1], then the five initial candidates of cell [1,2] would have remained on level 1.

In a broader outlook, the backtracking algorithms operating steps are:

1st) Start at level 0 by choosing a blank cell.

2nd) Place a valid number on the chosen blank cell.

3rd) Update the other blank cells candidates.

4th) Go to the next level:

→ If there are still blank cells:

— If a blank cell has no candidates, this means that at least one number was incorrectly placed in a previous blank cell (level). Delete the preceding placed numbers in reverse order (go back to prior levels) until reaching a level with more than one candidate. Then go back to the 2nd step, but this time choose a different valid number.

— If all blank cells have candidates, choose a new blank cell and go to the 2nd step.

→ If all cells are filled, then terminate.

An efficient backtracking algorithm is the Algorithm X (Knuth, 2019). To understand its mechanisms, let us reframe the sudoku problem.

Exact Cover Problem

The sudoku can be viewed as an exact cover problem. To better comprehend this concept and given the complexity of such endeavor, consider from now on the matrix below:

$$\begin{array}{c c} & \begin{array}{ccccccc} \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} & \boldsymbol{d} & \boldsymbol{e} & \boldsymbol{f} & \boldsymbol{g} \end{array} \\ A = & \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

Source: Knuth, 2019, p.64

For this particular case, the exact cover problem could be stated as follows: the objective is to cover all seven columns/items - named a, b, c, d, e, f, g - with disjoint rows/options – $\{c, e\}$, $\{a, d, g\}$, $\{b, c, f\}$, $\{a, d, f\}$, $\{b, g\}$, and $\{d, e, g\}$.

Every sudoku is an exact cover problem (see **Appendix A**) whose disjoint options are the possible positioning of numbers for each cell, that cover all constraints of the problem. In the next paragraph, these notions are detailed. To do so, have in mind that $r_i c_j \#_k$ with $i, j, k \in \{1, \dots, n\}$ corresponds to the cell with row i and column j, where it can be placed candidate k.

At most, there are n^3 disjoint options (or rows) - n^2 cells multiplied by n possible values to fill them. Therefore, if the cell $[i, j]$ is a fixed cell filled with number k, then the only option to that cell is simply $r_i c_j \#_k$, but if it is a blank cell, then the options are $r_i c_j \#_1$, $r_i c_j \#_2, \dots, r_i c_j \#_n$. The sudoku problem has $4 \times n \times n$ constraints (or columns/items) - 4 types of constraints: the first type ensures that each cell contains exactly one of n options to fill it and the other three types ensure that each row, column, and mini-grid contains one of each integer from 1 to n. To clarify to what correspond the 4 types of constraints, here is their representation using the previous notation:

- Cell constraints: $[1,1], \dots, [1, n], [2,1], \dots, [2, n], \dots, [n, 1], \dots, [n, n]$
- Row constraints: $r_1\#_1, \dots, r_1\#_n,\ r_2\#_1, \dots, r_2\#_n, \dots, r_n\#_1, \dots, r_n\#_n$
- Column constraints: $c_1\#_1, \dots, c_1\#_n,\ c_2\#_1, \dots,\ c_2\#_n, \dots, c_n\#_1, \dots, c_n\#_n$
- Mini-grid constraints: $m_1\#_1, \dots, m_1\#_n,\ m_2\#_1, \dots, m_2\#_n, \dots, m_n\#_1, \dots, m_n\#_n$

Since matrices of 0s and 1s used to represent exact cover problems are usually extremely sparse [particularly in the case of the sudoku problem where in each row there are four 1s and $(4 \times n \times n - 4)$ 0s], they can be transposed to a more constrict data structure – a matrix of nodes -, without losing information. The matrix of nodes is constructed by adding one node for each 1 in the binary matrix.

To solve the exact cover problem, the matrix of nodes will have to be manipulated. The coming subsection presents a technique that does this manipulation without loss of information and that is remarkably efficient when applied to backtracking algorithms.

Dancing Links

During a backtracking procedure, it is necessary to have a way of updating and downdating the data structure that supports the procedure efficiently. Therefore, instead of copying the information of all ancestors of the current node in the search tree every time a new level is entered (so that previous decisions can be reversed), Donald E. Knuth developed a much easier and faster technique. It is called dancing links (DL) and deletes/undeletes an item from a data structure without copying it, but only by modifying it. It works as follows:

Suppose there is a double-linked list, like the one in **Diagram 1**. As explained in Knuth (2019), let $L[x]$ and $R[x]$ be, respectively, the predecessor and the successor of node x. Node x can be deleted from the list by setting:

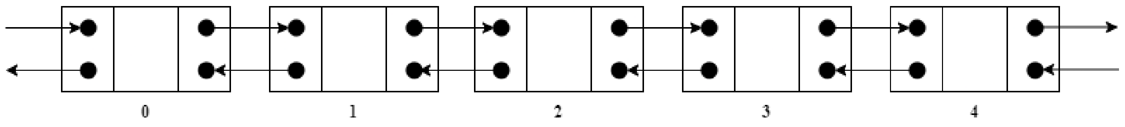
$$R[\,L[x]\,] \leftarrow R[x], \qquad L[\,R[x]\,] \leftarrow L[x],$$

and undeleted, through:

$$R[\,L[x]\,] \leftarrow x, \qquad L[\,R[x]\,] \leftarrow x,$$

after restoring the nodes eliminated since node x deletion, in reverse order.

Diagram 1: An example of a Dancing Links diagram.



Linked data structures are connected by pointers that represent the address of a location in memory, as illustrated in **Figure 6 (a)**. For example, imagine that we start with a linked data structure identical to the one in **Figure 6 (a)** and we delete node 3, and then node 1 – **Figure 6 (c)** captures these links transformations. To restore node 3, it would be necessary to first undelete node 1 - obtaining **Figure 6 (b)** -, and only then undelete node 3 – reestablishing the initial links of **Figure 6 (a)**.

x	$L[x]$	$R[x]$
0	4	1
1	0	2
2	1	3
3	2	4
4	3	0

(a)

x	$L[x]$	$R[x]$
0	4	1
1	0	2
2	1	4
3	2	4
4	2	0

(b)

x	$L[x]$	$R[x]$
0	4	2
1	0	2
2	0	4
3	2	4
4	2	0

(c)

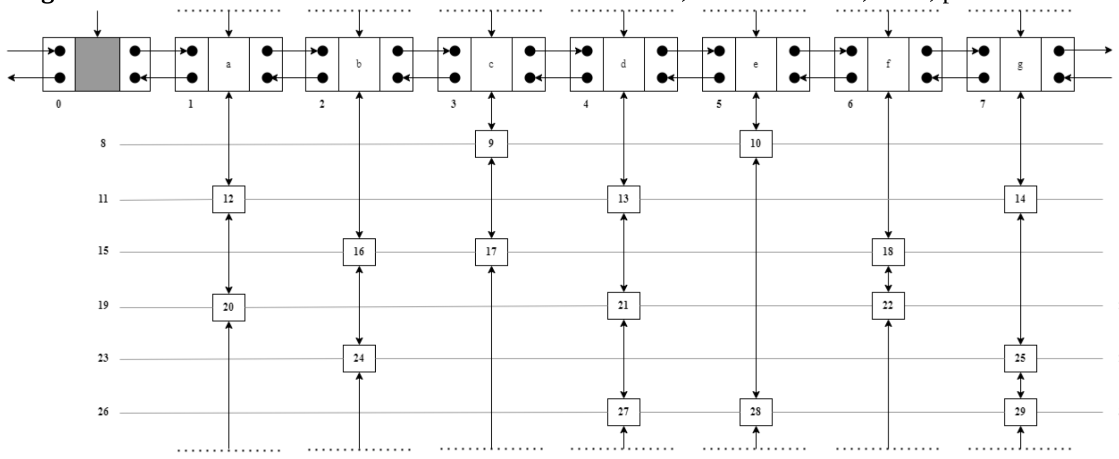
Figure 6: (a) Initial links of **Diagram 1**; **(b)** Links after deletion of node 3; **(c)** Links after deletion of nodes 3 and 1, in that order.

The main advantage of using a linked data structure as opposed to a continuously-allocated structure (such as arrays and matrices) is that changing pointers is easier and faster than moving items, particularly when dealing with large records. And backtracking algorithms require exactly that since they enumerate all solutions to a given set of constraints. Besides, and as clarified by Skiena (2008), the other benefit of linked data structures is that overflow occurs only when the memory is full.

Algorithm X's Data Structure

Algorithm X, by the same author of the DL technique (Knuth, 2019), visits all solutions to a given exact cover problem using a data structure that has embedded DL to easily add back nodes removed. Its data structure is a matrix of nodes that forms a doubly-linked list horizontally and vertically. **Diagram 2** represents a doubly-linked list which derives from matrix A (presented above).

Diagram 2: Double-linked list – the data structure for matrix A, based on Knuth, 2019, p.65.



The vertical entries (the header) correspond to the items (a, b, c, d, e, f, and g) – for the sudoku problem the header corresponds to the constraints previously mentioned in the **Exact Cover Problem Section** - and the horizontal entries correspond to the disjoint options. The nodes represent the current candidates which can be part of the solution (in **Diagram 2** these are nodes 9, 10, 12, 13, 14, 16, 17, 18, 20, 21, 22, 24, 25, 27, 28, and 29). The nodes of the header (numbered from 0 to 7) are connected with left and right pointers, and the remaining nodes are linked with up and down pointers. The header has an extra node (node 0), the root, which is reciprocally connected to the header last node (node 7) and it works as an entry point to the data structure. Each node of the header establishes also reciprocal links with the last candidate below it. In order to navigate the list in both directions, spacer nodes (nodes 8, 11, 15, 19, 23, 26, and 30) are inserted in the beginning and in the end of each horizontal entry.

Furthermore, the header's nodes have a size and name properties that indicate, respectively, the amount of nodes under them - the number of candidates left in the constraint, in case of the sudoku problem - and their denomination – the sudoku constraints denomination. The other nodes have a top property that tells what is the header of the vertical list of which they belong, with the exception of the spacer nodes, whose top field is represented with a negative integer. The way this data structure was constructed erases the inefficient random access to its elements that is common in linked lists, because it is not required to start from the root when searching for any node.

Algorithm X's Functioning

After constructing the doubly-linked list, the Algorithm X is executed. This algorithm is composed by the functions $cover$, $uncover$, $hide$, and $unhide$. We are now going to analyse their behaviour by using an example that intends to give an intuition of how they change the data structure.

Consider the **Diagram 2**. The list of available options comprises: $\{c, e\}$, $\{a, d, g\}$, $\{b, c, f\}$, $\{a, d, f\}$, $\{b, g\}$, and $\{d, e, g\}$. Assume that we want to cover item a. We choose option $\{a, d, g\}$ - the row that contains node 12 - to cover it. This option cannot be chosen again, so it is scratched from the list of options - the nodes of the row that contains node 12 will be "removed" (**Diagram 3**). The other options that cover item a can also be scratched, because they can no longer be chosen. In this case, there is only the option

$\{a, d, f\}$, so the nodes of the row that contains node 20 will be "removed" (**Diagram 4**). Since the option $\{a, d, g\}$ covers items d and g, this process of making disappear, among the remaining options of the list, the options that have these items will be repeated. Then, because there are still uncovered items, we choose one to cover and do it all over again.

Diagram 3: An intuitive illustration of function $hide(12)$ behaviour changes **Diagram 2**.

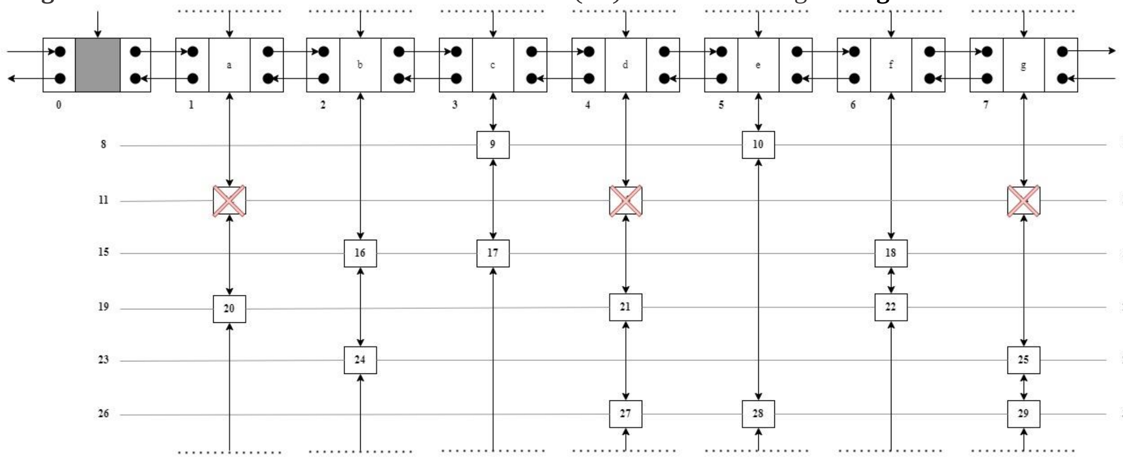
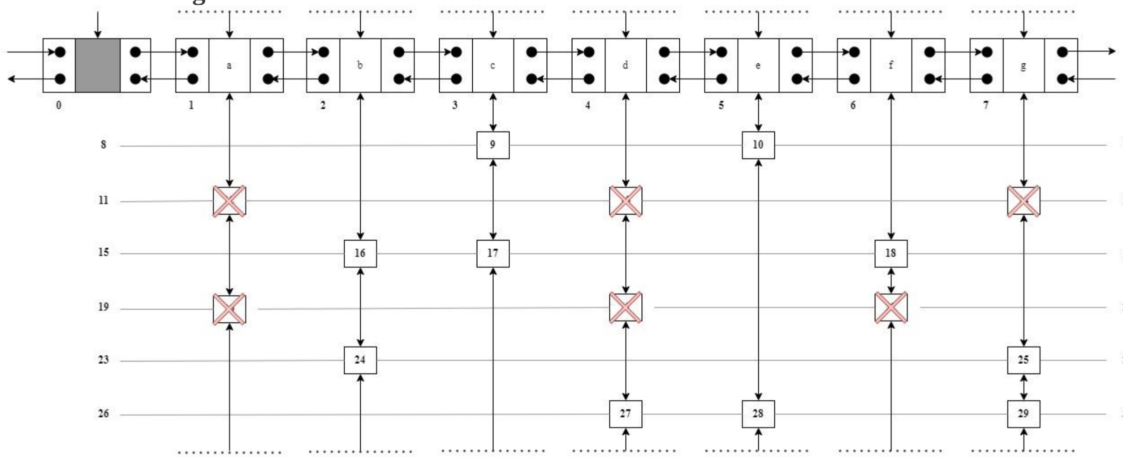


Diagram 4: An intuitive illustration of function $hide(20)$ behaviour, after the alterations caused in **Diagram 3**



The real changes that happen in the doubly-linked list of the example shown before are represented in **Diagram 5**, **Diagram 6**, and **Diagram 7**, in that order. To clarify these changes, we shall henceforward conduct an accurate description of the four functions. The cover and uncover functions change the horizontal links (**Diagram 7**). Inside these resides, respectively, the hide and unhide functions. When the hide function operates on a node x, it hides the disjoint option that contains node x (**Diagram 5** and **Diagram 6**), by changing the vertical links. This vertical links change consists of "removing" the nodes of the option that contains node x. The unhide function does the reverse: changes the vertical links by “inserting” the previously hidden nodes. After hiding the option that contains node x, the cover function forces to hide the options that have nodes below the nodes of the just-hidden option, which for this reason are not future viable options.

Diagram 5: $Hide(12)$ – the first vertical links changes to cover item a, originating from **Diagram 2**.

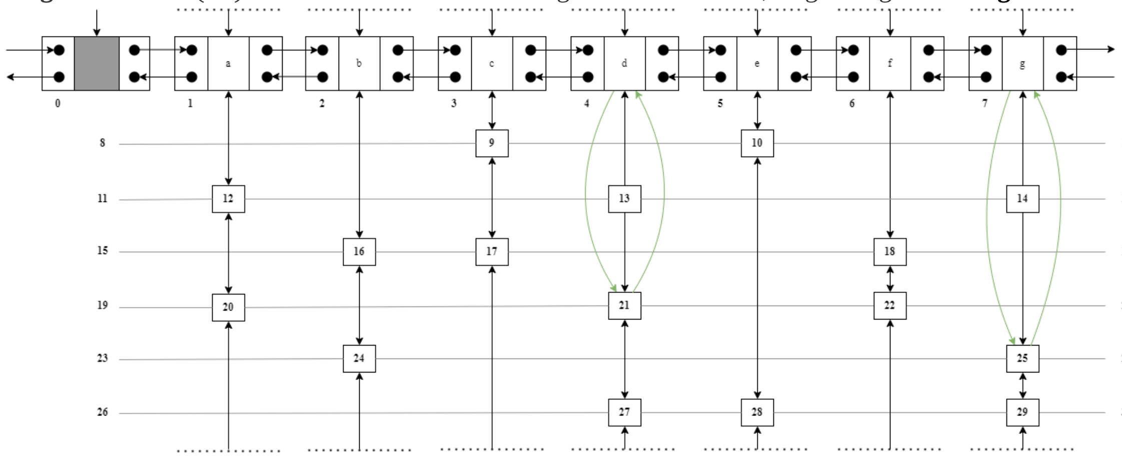


Diagram 6: $Hide(20)$ – the second vertical links changes to cover item a originating from **Diagram 5**.

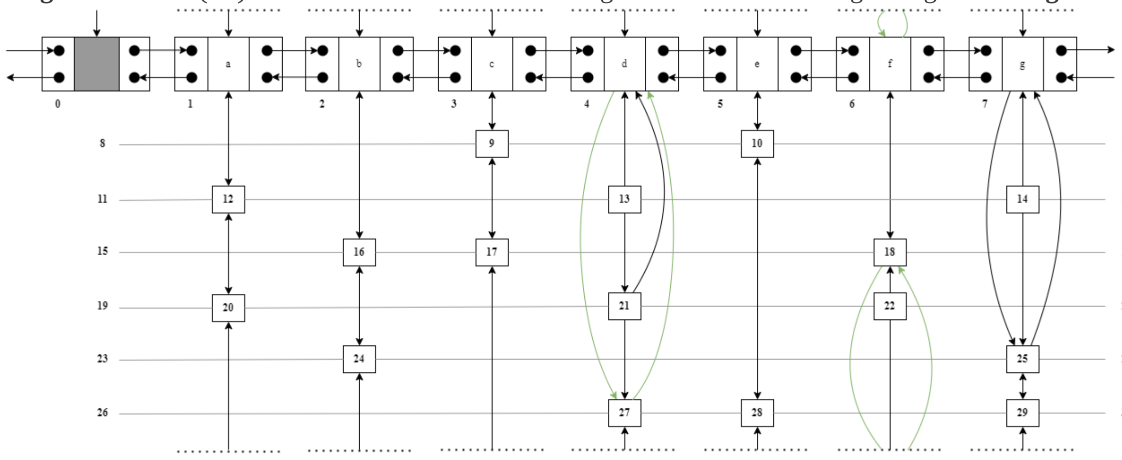
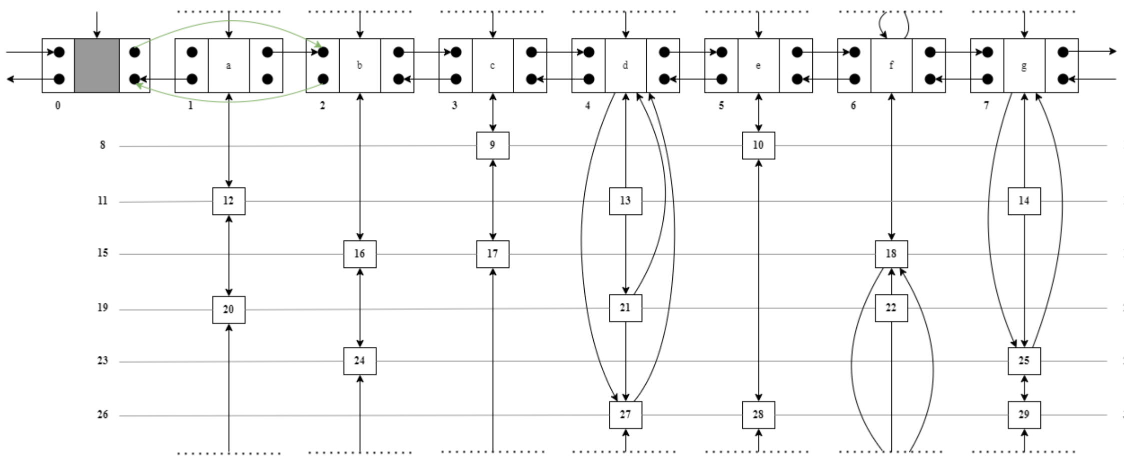


Diagram 7: Horizontal links changes to cover item a, originating from **Diagram 6**. The function $Cover(1)$, meaning to cover item a, is terminated.



The following is the terminology established for the next pseudocodes:

$\boldsymbol{R[x]}$ **= successor of node** $\boldsymbol{x}$ **U**$\boldsymbol{[x]}$ **= first node above** $\boldsymbol{x}$

$\boldsymbol{L[x]}$ **= predecessor of node** $\boldsymbol{x}$ **D**$\boldsymbol{[x]}$ **= first node below** x

$\boldsymbol{top(x)}$ **and** $\boldsymbol{size\ (x)}$ **properties are above described.**

In the cover function, the while cycle hides all the nodes of the option that contains the first node below item i (the item we intend to cover) - **line 2** of **Function 1**. After finishing the cycle, the horizontal links are altered.

Function 1: Pseudocode for the cover function

```
1:  $cover(i) =$
2:      $p \leftarrow D[i]$
3:      Do while $p \neq i$
4:          $hide(p)$
5:          $p \leftarrow D[p]$
6:      Loop
7:      $l \leftarrow L[i]$
8:      $r \leftarrow R[i]$
9:      $R[l] \leftarrow r$
10:     $L[r] \leftarrow l$
```

Note: Adapted from Knuth, 2019, p.66.

The uncover function does precisely the opposite (**Function 2**): first re-establishes the horizontal links and then, unhides the nodes.

Function 2: Pseudocode for the uncover function

```
1:  $uncover(i) =$
2:      $l \leftarrow L[i]$
3:      $r \leftarrow R[i]$
4:      $R[l] \leftarrow i$
5:      $L[r] \leftarrow i$
6:      $p \leftarrow U[i]$
7:      Do while $p \neq i$
8:          $unhide(p)$
9:          $p \leftarrow U[p]$
10:     Loop
```

Note: Adapted from Knuth, 2019, p.67.

The hide function (**Function 3**) goes through the nodes of the option of which node p belongs, and changes these nodes' vertical links. It is just like we saw before in **Diagram 5** to hide node $p = 12$: the vertical links of nodes 13 and 14 were altered, then there is a spacer (node 15), whose top property is smaller than zero (**lines 4** and **6** of **Function 3**), so the structure of control "If" knows that it arrived at the end of the option and changes the variable q to node 12 (**line 7** of **Function 3**); because q is back to the beginning of the option, the while cycle terminates. A similar process is used to unhide nodes (**Function 4**).

Function 3: Pseudocode for hide function

```
1.  $hide(p) =$
2.          $q \leftarrow p + 1$
3.          Do while $q \neq p$
4.              $x \leftarrow top\ (q)$
5.              $u \leftarrow U[q]$
6.              If $x < 0$ Then
7.                  $q \leftarrow u$
8.              Else
9.                  $d \leftarrow D[q]$
10.                 $D[u] \leftarrow d$
11.                 $U[d] \leftarrow u$
12.                 $size\ (x) \leftarrow size(x) - 1$
13.                 $q \leftarrow q + 1$
14.             End If
15.         Loop
```

Note: Adapted from Knuth, 2019, p.67.

Function 4: Pseudocode for unhide function

```
1:  $unhide(p) =$
2:          $q \leftarrow p - 1$
3:          Do while $q \neq p$
4:              $x \leftarrow top\ (q)$
5:              $d \leftarrow D[q]$
6:              If $x < 0$ Then
7:                  $q \leftarrow d$
8:              Else
9:                  $u \leftarrow U[q]$
10:                 $D[u] \leftarrow q$
11:                 $U[d] \leftarrow q$
12:                 $size\ (x) \leftarrow size(x) + 1$
13:                 $q \leftarrow q - 1$
14:             End If
15:         Loop
```

Note: Adapted from Knuth, 2019, p.67.

Procedure 1 gives an overview of how Algorithm X solves an exact cover problem via dancing links. Algorithm X starts by choosing the uncovered item i with the fewest number of nodes occurring in a column. If no option covers it and all viable combinations of options have been tried, there is no solution, and the algorithm terminates. Otherwise, it covers i, meaning that:

1st) It selects an option that covers i to be part of the solution

2nd) It hides all the options that cover i

3rd) It deletes item i from the list of uncovered items

4th) It covers, one at a time, each uncovered item $j \neq i$ that is part of the just-selected option.

This process is repeated until either:

- a feasible solution is reached (**line 8** of **Procedure 1** – all items are covered; the root node is connected to itself)
- the sequence of selected options does not lead to a feasible solution (**line 18** of **Procedure 1** – the node of the item that it is being covered is connected to itself; there are no option nodes below it) and it was not tried every viable combination of options.

In the second case, this process is reversed prior to a moment when a different option could have been chosen to be a part of the solution and this possibility (or branch of the search tree) has yet not been tried. It is important to have in mind that items are uncovered in the opposite order in which they were covered. The same principle applies to unhide options.

Check **Appendix B** to see how Algorithm X would solve the exact cover problem represented by matrix A, step by step.

Procedure 1: Pseudocode of Algorithm X

```
1:  [X1 – Initialize]
2:      Input the initial content of the Sudoku grid in a double-linked list like the one
        previously described.
3:      Let T be an array large enough to accommodate each node pointers of an option
        selected to be part of the solution.
4:      Let $x_0, \ldots, x_T$ be the node pointers for backtracking.
5:      Let Z be the last spacer address.
6:      $l \leftarrow 0$
7:  [X2 – Enter level $l$]
8:      If $R[0] = 0$ Then
9:          Visit the solutions that is $x_0 x_1 \ldots x_{l-1}$
10:         Go to X8
11:     End if
```

```
[X3 – Choose constraint $i$, not yet covered]
    Minimum Remaining Value (MRV) heuristic selects the first constraint $i$ with
    the fewest options.
[X4 – Cover $i$]
    $cover(i)$
    $x_l \leftarrow D[i]$
[X5 – Try $x_l$]
    If $x_l = i$ Then
        Go to X7
    Else
        $p \leftarrow x_l + 1$
        Do while $p \neq x_l$
            $j \leftarrow top(p)$
            If $j \leq 0$ Then
                $p \leftarrow U[p]$
            Else
                $cover(j)$
                $p \leftarrow p + 1$
            End If
        Loop
        $l \leftarrow l + 1$
        Go to X2
    End if
[X6 – Try again]
    $p \leftarrow x_l - 1$
    Do while $p \neq x_l$
        $j \leftarrow top(p)$
        If $j \leq 0$ Then
            $p \leftarrow D[p]$
        Else
            $uncover(j)$
            $p \leftarrow p - 1$
        End If
    Loop
    $i \leftarrow top(x_l)$
    $x_l \leftarrow D[x_l]$
    Go to X5
[X7 – Backtrack]
    $uncover(i)$
[X8 – Leave level $l$]
    If $l = 0$ Then
        Terminate
    Else
        $l \leftarrow l - 1$
        Go to X6
    End If
```

Note: Adapted from Knuth, 2019, p.67.

4. TWO-PHASE ALGORITHM

The developed solver is a two-phase algorithm that operates as follows: during Phase 1, it tries to solve the sudoku puzzle through human strategies; if after that a solution was not found, then Phase 2 starts and performs a backtracking procedure.

The combination of the two phases intends to merge their advantages. In the event of Phase 1 failing to find a solution, at least it may have eliminated candidates from some cells, pruning down the search space of Phase 2. In the worst-case scenario, the latter has to try all possible assignments on the initial blank cells until either a solution is found or the possibilities are expired.

The initialization of the Phase 1 algorithm consists of storing the trivial candidates for each blank cell by taking into account the fixed numbers and not including those in the list of candidates of cells that belong to the same row, column, or mini-grid. The pseudocode for each human strategy will not be detailed as there are several ways of implementing them and **Section 3.2** already provides an explanation of how the human strategies operate. Nevertheless, it was tried to use as little access to memory as possible, since it has an impact on the running time.

The Phase 1 algorithm (**Procedure 2**) goes through the human strategies – ordered from the more basic to the more complex ones –, one by one. It executes each human strategy repeatedly until no more candidates are eliminated. At this point, the algorithm moves to the next human strategy and proceeds the exact same way. After executing the last human strategy (the more complex one), the algorithm will return to the first human strategy (the more basic one) and do this all over again, until there are no more blank cells or no human strategy is capable of reducing cells' candidates.

Procedure 2: Pseudocode for Phase 1

```
1: At any moment, if $filled\ cells = n^2$, then the procedure terminates immediately.
2: [Beginning]
3:     **Do while** $filled\ cells < n^2$
4:         **Do**
5:             $Naked\ Single$
6:         **Loop until** $no\ naked\ single\ is\ found$
7:
8:         **Do**
9:             $Naked\ pair$
```

```
10:           Loop until $no\ naked\ pair\ is\ found$
11:
12:           Do
13:                $Hidden\ single$
14:           Loop until $no\ hidden\ single\ is\ found$
15:
16:           Do
17:                $Hidden\ pair$
18:           Loop until $no\ hidden\ pair\ is\ found$
19:
20:           Do
21:                Locked candidate
22:           Loop until $no\ locked\ candidate\ is\ found$
23:
24:           Do
25:                $Line\ reduction$
26:           Loop until $no\ line\ reduction\ is\ found$
27:
28:           If no candidates were eliminated Then
29:                Go to Line 32
30:           End if
31:      Loop
32: [Ending of human strategies]
```

To start Phase 2, it is necessary to transform the current sudoku - which hopefully has less candidates than the initial ones due to Phase 1 performance - into an exact cover problem and then convert it to a doubly-linked structure. Only then the Algorithm X operates (**Procedure 3**).

Procedure 3: Pseudocode for the Two-Phase Algorithm

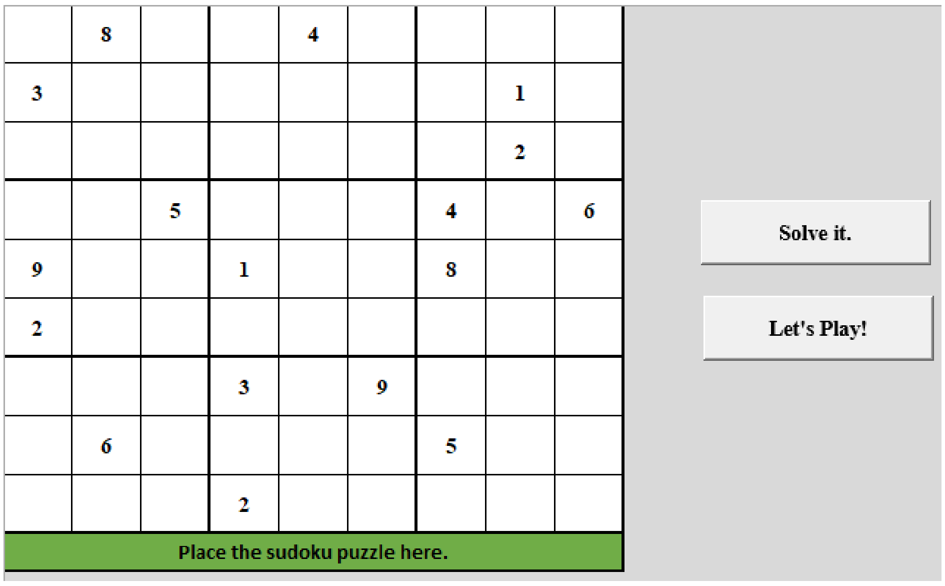
```
1:      Call Phase 1 algorithm
2:
3:      If $filled\ cells = n^2$ Then
4:           Terminate
5:      Else
6:           Call Algorithm X
7:      End If
```

5. COMPUTATIONAL EXPERIENCE

This section starts by describing what the application developed can do and how it interacts with the user. Afterwards, the solver's performance will be tested and the results will be interpreted.

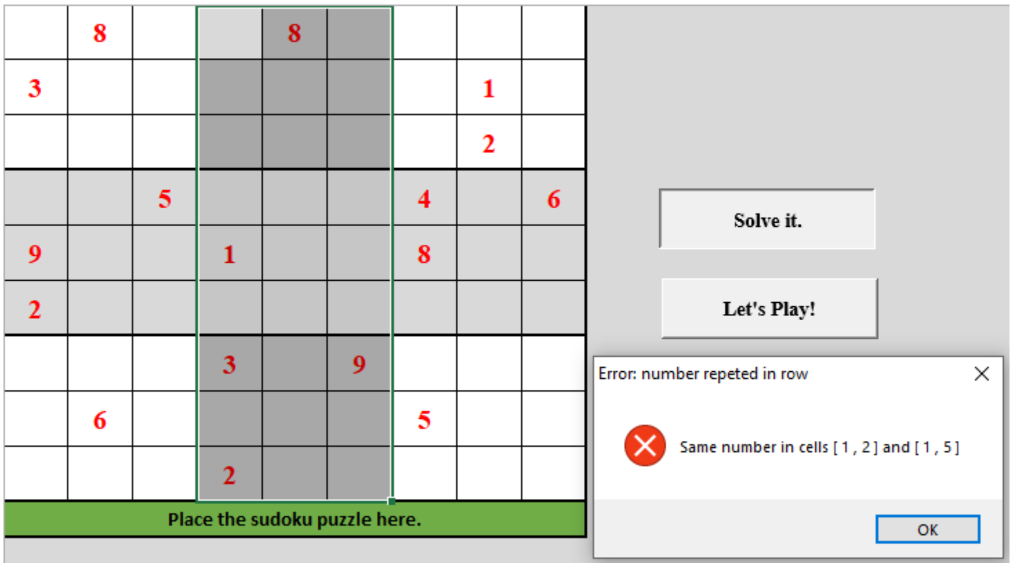
5.1 Application Functioning

The Sudoku solver implemented in Excel VBA opens a page with a 9 × 9 grid where the user can introduce a sudoku puzzle (**Figure 7**). Then, the user decides between seeing the puzzle solution (by pressing "Solve it.") or trying to find it himself (by pressing "Let's Play!"). Regardless of the choice, the grid is formatted – all cells are configured with the same font and size, the filled cells are colored red, and the mini-grids are alternately shaded with white and grey - and the program checks if the initial filled cells respect the problem constraints. If that is not the case, then the solver signals the error and stops running (**Figure 8**).



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

Figure 7: Sudoku application - home page.



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

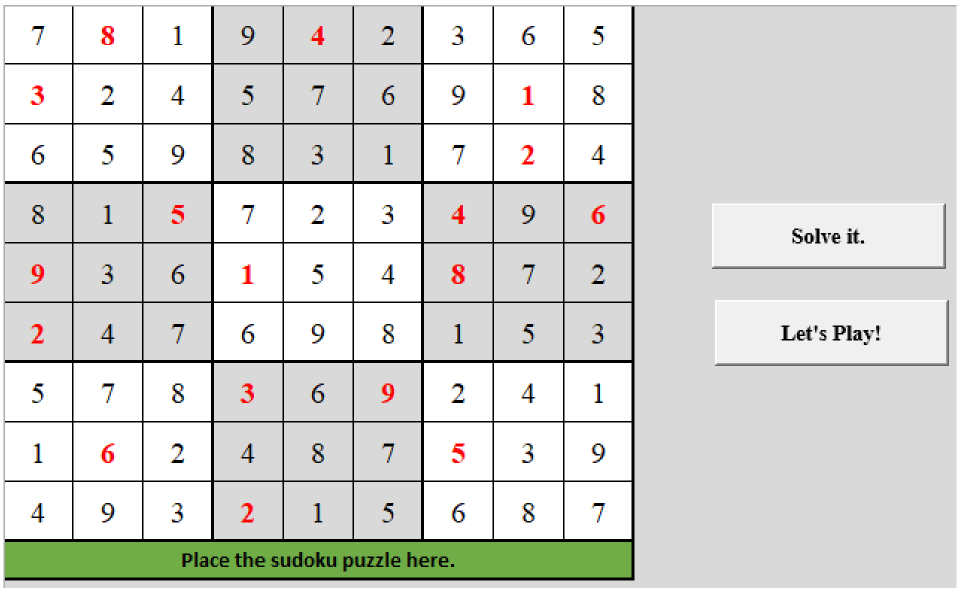
Place the sudoku puzzle here.

Error: number repeted in row

Same number in cells [1 , 2] and [1 , 5]

Figure 8: Sudoku application – initial problem constraints check.

If the user only wants to see the sudoku solution, the two-phase algorithm described in **Section 4** is executed and an outcome similar to the one seen in **Figure 9** is displayed.

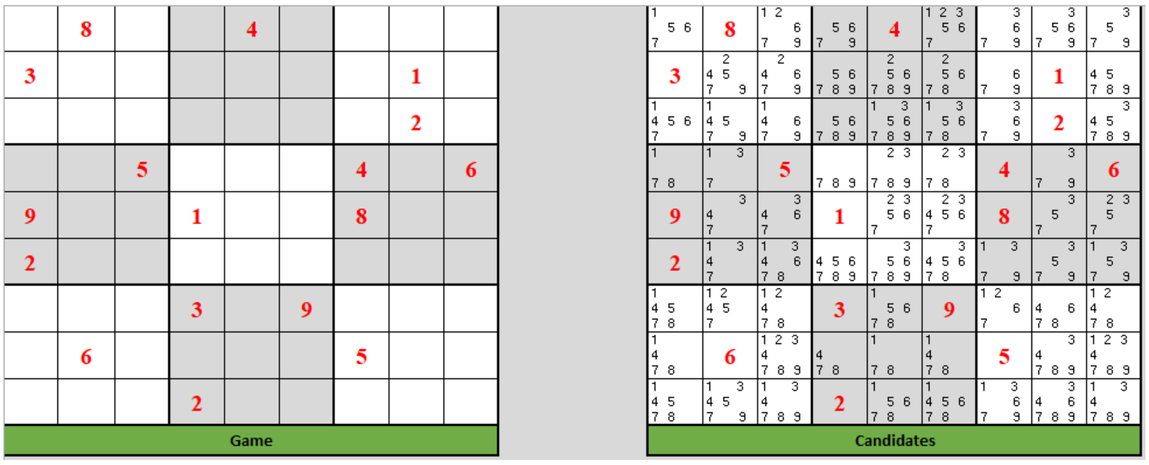


7	8	1	9	4	2	3	6	5
3	2	4	5	7	6	9	1	8
6	5	9	8	3	1	7	2	4
8	1	5	7	2	3	4	9	6
9	3	6	1	5	4	8	7	2
2	4	7	6	9	8	1	5	3
5	7	8	3	6	9	2	4	1
1	6	2	4	8	7	5	3	9
4	9	3	2	1	5	6	8	7

Place the sudoku puzzle here.

Figure 9: Sudoku Application - solution exhibited if button “Solve it.” is pressed.

If the user wants to play, the two-phase algorithm is also executed but the solution is not shown, it just serves to alert the player when a number is misplaced. After the solution is determined, a new sheet is opened, where two grids are presented (**Figure 10**). In the game grid (on the left), the user can select cells to place numbers, while the rest of the sheet is blocked. The candidates’ grid (on the right) shows, for each blank cell, the possible numbers that can be placed on it. This grid is automatically updated every time a cell in the game grid is filled or a human strategy is executed.



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

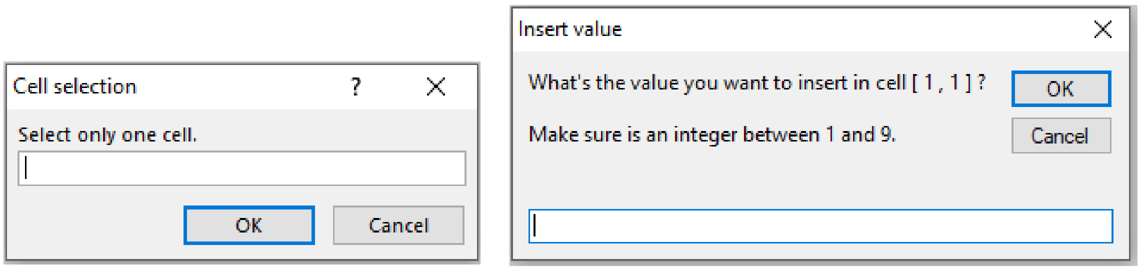
Game

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

Candidates

Figure 10: Sudoku application - game page.

The player is invited to select one cell (**Figure 11**). After doing it, the program verifies if only one cell was selected and if it was a blank cell (**Figure 12**). Granted that both these conditions are verified, the program authorizes to write the value the player wants to insert on the selected cell (**Figure 11**). Before placing the number in the game grid, it is confirmed if such number is an integer between 1 and 9 and if it is the correct number for the chosen cell. If these conditions do not hold, the solver provides three alternatives: to go through the same process of choosing a cell and filling it with a value, to quit the game, or to ask for a hint (**Figure 13**).



Cell selection

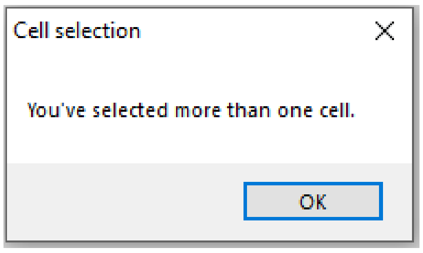
Select only one cell.

Insert value

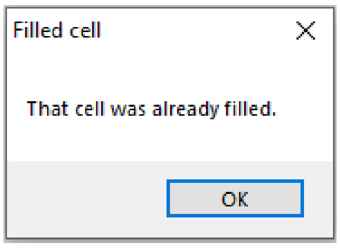
What's the value you want to insert in cell [1 , 1] ?

Make sure is an integer between 1 and 9.

Figure 11: Sudoku application – On the left is a box to input one cell coordinates and on the right is a box to enter a value on the previously selected cell.

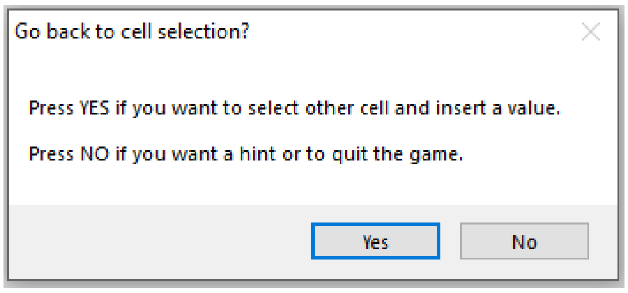

Cell selection

You've selected more than one cell.


Filled cell

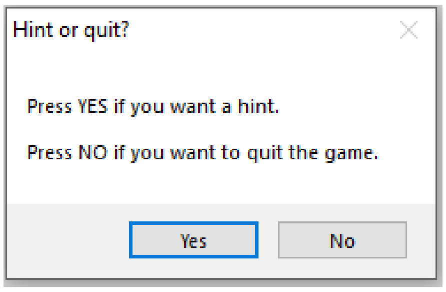
That cell was already filled.

Figure 12: Sudoku application – The two types of error message that might appear after inputting a cell coordinates.


Go back to cell selection?

Press YES if you want to select other cell and insert a value.

Press NO if you want a hint or to quit the game.

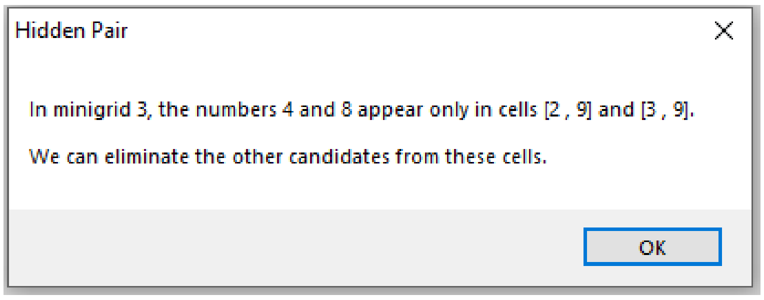

Hint or quit?

Press YES if you want a hint.

Press NO if you want to quit the game.

Figure 13: Sudoku application – On the left, a message box that allows to opt between selecting one blank cell, quitting the game or requesting a hint. If the button "NO" is pushed, the message box on the right is displayed so that the user can choose if he/she wants a hint or wants to quit the game.

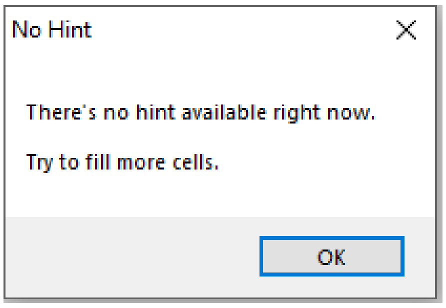
Every time a hint is requested, the human strategies are performed following the sequence presented in **Procedure 2**. When a human strategy leading to a reduction of candidates or to the definition of the value of a blank cell is identified, a message box explains the reasoning behind it and tells the changes caused in the candidates' grid (**Figure 14**). Then the candidates' grid is updated and the first message box of **Figure 13** pops-up again. Occasionally, the game grid might also be modified. When no human strategy can provide a new clue, the player is informed (**Figure 15**).


Hidden Pair

In minigrid 3, the numbers 4 and 8 appear only in cells [2 , 9] and [3 , 9].

We can eliminate the other candidates from these cells.

Figure 14: Sudoku application - example of a hint.


No Hint

There's no hint available right now.

Try to fill more cells.

Figure 15: Sudoku application – Message box that notifies the player for the absence of clues.

The solver terminates whenever the 81 cells are filled or the player decides to quit the game.

5.2 Experiments and Results

The algorithm presented in the previous section has been coded in VBA. All tests were performed using the same hardware and software setup for more reliable results, with only the Excel VBA running. The computer has a CPU Intel® Core™ i3-3120M @ 2.50 GHz and 4.00 GB of RAM.

Data Sources

To conduct the experiment, datasets from three different sources were used:

— the *"250 Sudoku"* published by Edigrama, a hobby book whose puzzles difficulty level for humans is scaled between 4 and 7 by this editor. For the experiment, a sample of 40 puzzles - 10 puzzles of each level – was used (**Appendix C**);

— the *Magictour-top95*[3], a list of 95 puzzles considered moderately hard. All the 95 puzzles were used in the experiment;

— the *HardestDatabase110626*[4], a collection of 376 puzzles claimed to be hard by different known rating programs. The first 95 puzzles were used in the experiment.

From now on, these datasets will be named Easy, Moderate, and Hard, respectively, as a reference to their difficulty rank. **Table 1** displays relevant information about the three datasets characteristics.

[3] Retrieved from: magictour.free.fr/top95 (Accessed: September 7th, 2022)

[4] Available at: http://forum.enjoysudoku.com/the-hardest-sudokus-new-thread-t6539.html#p65791 (Accessed: September 7th, 2022)

Table 1: Description of the datasets used for the computational experience.

	Easy	**Moderate**	**Hard**
Number of instances	40	95	95
Number of initial filled cells	[20, 28]	[17, 26]	[20, 23]
Average number of initial filled cells	24	21	21
Number of initial candidates	[170, 263]	[209, 313]	[228, 264]
Average number of initial candidates	217	264	248
Type of grid	9×9		

Experiments and Results

The main goal of this experiment is to analyse the performance of the two-phase algorithm for puzzles with distinct difficulty solving levels, circumscribed by the three datasets above. Several quantitative metrics - divided in global and group metrics- were collected. The global metrics provide a macro view of the solver performance. The group metrics allow a more thorough evaluation for each dataset, by providing insights about which phase of the two-phase algorithm was necessary to execute to solve the puzzles.

The global indicators (**Table 2**) proposed include the percentage of puzzles that the algorithm was able and unable to solve. For both situations, it is calculated the average percentage of candidates eliminated by human strategies. For the puzzles solved, it was also computed the average total time needed to get to a solution.

Table 2: Global indicators of the solver performance.

	Easy	**Moderate**	**Hard**
Percentage of solved puzzles	95	50.53	42.11
Average percentage of candidates eliminated during Phase 1 of solved puzzles	100	63.67	2.04
Average total time of solved puzzles (in seconds)	0.689	1.176	2.525
Percentage of unsolved puzzles	5	49.47	57.89
Average percentage of candidates eliminated during Phase 1 of unsolved puzzles	38.44	26.08	2.32

The takeaways provided by **Table 2** confirm some suspicions: the resolution speed of the two-phase algorithm is lower the easier the puzzles and the percentage of unsolved puzzles is higher the harder the puzzles – for the moderate and hard datasets the solver was unable to solve approximately 50% and 58% of the puzzles, respectively. The solver cannot find a solution only due to overflow. The overflow error occurs in Phase 2 when the solver tries to use more memory space than the one available. This happens because it calls the functions $cover$, $uncover$, $hide$, and $unhide$ recurrently. In general, the solver is fast for 9×9 puzzles – the highest total time of resolution of all 230 instances tested was 6.813 seconds and the lowest was 0.309 seconds. The maximum average total time across various difficulty levels is below 3 seconds (2.525 seconds).

The puzzles solved can be separated in three groups (**Table 3**):

— Group 1 - puzzles solved using exclusively human strategies

— Group 2 - puzzles solved with resort to backtracking only

— Group 3 - puzzles solved with a combination of both human strategies and backtracking.

Table 3: Unfolding of the percentage of solved puzzles.

	Easy	Moderate	Hard
Percentage of Group 1 puzzles	95	25.26	0
Percentage of Group 2 puzzles	0	0	16.84
Percentage of Group 3 puzzles	0	25.26	25.26
Percentage of solved puzzles	95	50.53	42.11

When applied alone, the human strategies were able to completely solve puzzles from the easy and moderate datasets: 95% of the easy dataset puzzles and 25.26% of the moderate ones. They provided, on average, a residual decrease of the initial number of candidates of a few puzzles of the hard dataset (**Table 2**). Besides that, the percentage gap of unsolved puzzles between the hard and moderate datasets is small ($57.89\% - 49.47\% = 8.42\%$) due to the Algorithm X efficiency. Algorithm X complemented the Phase 1 algorithm in 25.26% of the puzzles from both the moderate and hard datasets. The Phase 2 algorithm solved alone puzzles exclusively from the hard dataset (**Table 3**).

The following criteria were used to compare the solver performance at a group level. It was obtained, with respect to:

- Group 1, the average number of eliminated candidates via human strategies and the average time needed to fill all cells, per blank cell.
- Group 2, the average number of eliminated candidates through backtracking and the average time needed to fill all cells, per blank cell.
- Group 3, the average number of eliminated candidates per phase and the average total time the two-phase algorithm took to fill each blank cell.
- Every group, the average total time to reach a solution.

In relation to Group 1 - puzzles solved using exclusively human strategies -, information in **Table 4** reveals that the Phase 1 algorithm is fast. As expected, when comparing the moderate instances with the easy ones, the results show that, on average, the former need more total time to reach a solution, but that difference is not significant - on average, the human strategies solved the easy puzzles in less than 0.7 seconds and the moderate puzzles in about 1 second. It is important to recall that all instances of the moderate dataset have a superior number of initial candidates, which corresponds to the number of candidates eliminated in case of Group 1 puzzles. Despite this, the average cell filling time by blank cell of the moderate dataset is, on average, 40 times smaller.

Table 4: Group 1 indicators of the solver performance. Results concerning solved puzzles.

	Easy	**Moderate**	**Hard**
Average number of candidates eliminated	216	289	-
Average time needed to fill all cells, per blank cell (in seconds)	0.689	0.017	-
Average total time (in seconds)	0.689	1.038	-

When analysing the results of Group 2 – puzzles solved using exclusively Algorithm X -, it is reinforced the efficiency of the backtracking algorithm implemented (**Table 5**). In spite of the exhaustive search intrinsic to backtracking procedures and the average number of candidates eliminated - which is equal to the average number of initial candidates of the hard solved puzzles – being very high, the average cell filling time by blank cells is rather small.

Table 5: Group 2 indicators of the solver performance. Results concerning solved puzzles.

	Easy	**Moderate**	**Hard**
Average number of candidates eliminated	-	-	243
Average time needed to fill all cells, per blank cell (in seconds)	-	-	0.104
Average total time (in seconds)	-	-	6.183

The average cell filling time by blank cell for instances of Group 3 - puzzles solved with a combination of both human strategies and backtracking - is around the same magnitude as the prior (**Table 6**). For the moderate puzzles, the average percentage of candidates eliminated during Phase 1 and Phase 2 is, respectively, 27.34% and 72.66%. For the hard puzzles that percentage is 3.40% (an average of 8 candidates eliminated in Phase 1) and 96.60% (on average, 239 candidates eliminated in Phase 2), as can be examined below.

Table 6: Group 3 indicators of the solver performance. Results concerning solved puzzles.

	Easy	**Moderate**	**Hard**
Average number of candidates eliminated during Phase 1	-	73	8
Average number of candidates eliminated during Phase 2	-	183	239
Average time needed to fill all cells, per blank cell (in seconds)	-	0.105	0.108
Average total time (in seconds)	-	6.224	6.435

In some situations, across the three datasets, the solver would be unable to solve an instance and would solve another one that had the same (or similar) number of initial filled cells, initial candidates, and candidates after the human strategies being executed. There were also cases where the solver was able to solve instances with a superior number of remaining candidates after Phase 1 than the one existing in puzzles where the overflow error occurred. This can be explained by several factors: i) the relative positioning of the filled cells, ii) the values placed on the filled cells, iii) the vertical location of the correct option to be part of the solution of the first item that is covered after the double-linked list is constructed, and iv) the number of levels that the algorithm has to backtrack when hits a not feasible solution and how many times it repeats this backtracking process.

6. CONCLUSION

In this MFW it was proposed the implementation, in VBA, of a sudoku solver which has a two-phase algorithm embedded. Phase 1 performs human strategies and Phase 2 runs a backtracking procedure whenever necessary.

Any backtracking procedure is naturally computationally heavy. Therefore, it is crucial to use efficient data structures when implementing these kinds of algorithms. In this MFW, the support of the developed backtracking algorithm is a double-linked list, which is one of the most efficient (and complex) structures to implement backtracking algorithms. Moreover, the backtracking algorithm selected to be implemented, Algorithm X (created by Donald E. Knuth), is outstandingly efficient. A great contribution of this MFW is a thorough clarification of how Algorithm X operates and how to construct its data structure, which requires to transform a sudoku puzzle into an exact cover problem and only then into a matrix of nodes that forms a doubly-linked list horizontally and vertically.

The application created allows the user to see the solution of a particular 9×9 sudoku puzzle directly or to play the game by solving the sudoku step by step. In the last case, the player can request clues and insert values in cells. The application provides explanatory messages for values misplaced or invalid, for details of the logic behind the human strategies used, among others.

It was conducted a computational experience to test the performance of the two-phase algorithm. To do so, 230 instances were tested: 40 ranked as easy, 95 ranked as moderate, and 95 ranked as hard. The solver revelled to solve the puzzles fast, regardless of the level of difficulty of the sudoku. It solved 95% of the instances of the easy dataset, 50.53% of the moderate one, and 42.11% of the hard one. The solver was incapable to reach a solution for a substantial number of instances of the moderate and hard datasets, due to overflow errors that are innate to backtracking algorithms - especially when applied to problems (like the sudoku), which have a wide searching domain.

In addition, it was investigated the solver performance depending on the dataset difficulty ranking, for three different groups. It is important to point out the experimental findings for Group 1, because it provides insights particularly about the Phase 1 - the subject of interest - , which obtained positive performance results in the case of the easy and moderate instances solved. The solver successfully reached a solution, by executing exclusively human strategies, for 95% of the easy puzzles and 25.26% of the moderate puzzles. When analysing these percentages, it has to be highlighted that all instances of the moderate dataset have a superior number of initial candidates. Besides, while the average cell filling time by blank cell of the easy dataset is near 0.7 seconds, the one of the moderate dataset is close to zero – both encouraging results.

Our intentions for the future are the implementation of more human strategies and to test, using the same database, the impact of such addition on the solver performance. It would be interesting to also do the same experience, but this time with the human strategies sorted differently. Another potential future study would be to investigate the changes in the two-phase algorithm solver capacity if the human strategies were implemented inside other human strategies, instead of the current independent execution. For example, every time a locked candidate is found, the algorithm should explore if among the affected blank cells there are cells where the set of candidates was reduced to a single candidate (naked single), and then, it would continue to search for locked candidates.

REFERENCES

Coelho, L. C., & Laporte, G. (2014). A comparison of several enumerative algorithms for Sudoku. *Journal of the Operational Research Society, 65*, 1602-1610. doi:10.1057/jors.2013.114

Colbourn, C. J., Colbourn, M. J., & Stinson, D. R. (1984). The computational complexity of recognizing critical sets. In K. Koh, & H. Yap (Ed.), *Graph Theory Singapore 1983. Lecture Notes in Mathematics. 1073*, pp. 248-252. Springer, Berlin,Heidelberg. doi:10.1007/BFb0073124

Deodhare, D., Sonone, S., & Gupta, A. (2014). A Generic Membrane Computing-based Sudoku Solver. *2014 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, (pp. 89-99). doi:10.1109/ICICICT.2014.6781258

Eppstein, D. (2005). *Nonrepetitive Paths and Cycles in Graphs with Application to Sudoku.* Retrieved from arXiv:cs/0507053

Ercsey-Ravasz, M., & Toroczkai, Z. (2012). The Chaos Within Sudoku. *Scientific Reports, 2*(725). doi:https://doi.org/10.1038/srep00725

Hong, W., Chen, T.-S., & Shiu, C.-W. (2008). Steganography Using Sudoku Revisited. *2008 Second International Symposium on Intelligent Information Technology Application* (pp. 935-939). Shanghai: IEEE. doi:10.1109/IITA.2008.445

Jana, S., Maji, A. K., & Pal, R. K. (2015). A Novel Sudoku Solving Technique using Column based Permutation. *2015 International Symposium on Advanced Computing and Communication (ISACC)*, (pp. 71-77). doi:10.1109/ISACC.2015.7377318

Knuth, D. E. (1999). Dancing Links. In J. Davies, B. Roscoe, & J. Woodcock (Ed.), *Millennial perspectives in computer science: proceedings of the 1999 Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare* (pp. 187-214). Palgrave.

Knuth, D. E. (2019). *The Art of Computer Programming, Volume 4, Fascicle 5: Mathematical Preliminaries Redux; Introduction to Backtracking; Dancing Links.* (I. Pearson Education, Ed.) Addison-Wesley.

Maji, A. K., & Pal, R. K. (2014). Sudoku Solver using Minigrid based Backtracking. *2014 IEEE International Advance Computing Conference (IACC),* (pp. 36-44). doi:10.1109/IAdCC.2014.6779291

McGuire, G., Tugemann, B., & Civario, G. (2014). There Is No 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration. *Experimental Mathematics, 23*(2), 190-217. doi:10.1080/10586458.2013.870056

Mishra, D. B., Mishra, R., Das, K. N., & Acharya, A. A. (2018). Solving Sudoku Puzzles Using Evolutionary Techniques—A Systematic Survey. In M. Pant, K. Ray, T. Sharma, S. Rawat, & A. Bandyopadhyay (Eds.), *Soft Computing: Theories and Applications. Advances in Intelligent Systems and Computing* (Vol. 583). Springer. doi:10.1007/978-981-10-5687-1_71

Rubinstein-Salzedo, S. (2018). Zero-Knowledge Proofs. In *Cryptography. Springer Undergraduate Mathematics Series.* Springer, Cham. doi:10.1007/978-3-319-94818-8_16

Sevkli, A. Z., & Hamza, K. A. (2019). General variable neighborhood search for solving Sudoku puzzles: unfiltered and filtered models. *Soft Computing, 23*(15), 6585–6601. doi:10.1007/s00500-018-3307-6

Simonis, H. (2005). Sudoku as a Constraint Problem. In B. Hnich, P. Prosser, & B. Smith (Ed.), *Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, (pp. 13-27). Sitges (Barcelona).

Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer London. doi:10.1007/978-1-84800-070-4

Strokach, A., Becerra, D., Corbi-Verge, C., Perez-Riba, A., & Kim, P. M. (2020). Fast and Flexible Protein Design Using Deep Graph Neural Networks. *Cell Systems, 11*(4), 402-411. doi:10.1016/j.cels.2020.08.016

APPENDICES

Appendix A: Example of an exact cover problem matrix for a 4 × 4 sudoku puzzle - [1,1] is a fixed cell filled with the number 1 and [1,2] is a blank cell.

	[1,1]	[1,2]	…	$r_1\#_1$	$r_1\#_2$	$r_1\#_3$	$r_1\#_4$	…	$c_1\#_1$	$c_1\#_2$	$c_1\#_3$	$c_1\#_4$	$c_2\#_1$	$c_2\#_2$	$c_2\#_3$	$c_2\#_4$	…	$m_1\#_1$	$m_1\#_2$	$m_1\#_3$	$m_1\#_4$	…
$r_1c_1\#_1$	1	0		1	0	0	0		1	0	0	0	0	0	0	0		1	0	0	0	
$r_1c_2\#_1$	0	1		1	0	0	0		0	0	0	0	1	0	0	0		1	0	0	0	
$r_1c_2\#_2$	0	1		0	1	0	0		0	0	0	0	0	1	0	0		0	1	0	0	
$r_1c_2\#_3$	0	1		0	0	1	0		0	0	0	0	0	0	1	0		0	0	1	0	
$r_1c_2\#_4$	0	1		0	0	0	1		0	0	0	0	0	0	0	1		0	0	0	1	
…																						

Appendix B: Solving steps using Algorithm X to resolve the exact cover problem proposed in Knuth, 2019, p.64.

X1: $l = 0$	Initialize
X2:	Enter level $l = 0$
X3: Choose $i = 1$	Apply MRV: — Items a, b, c, e, and f have two option — Items d and g have three options Select the first item with less options (item a).
X4: $Cover(1)$ $Hide(12)$: "remove" (vertical re-link) node 13 and 14 $Hide(20)$: "remove" node 21 and 22 $x_0 = D[1] = 12$	Choose the option that contains node 12 (2^{nd} option) to cover item a. The 4^{th} option is no longer viable because it also covers item a and so it is hidden.
X5: $Cover(4)$ $Hide(27)$: "remove" node 28 and 29 $Cover(7)$ $Hide(25)$: "remove" 24 $l = 1$	The 2^{nd} option already covers item d, therefore option 6^{th} (at this point, the only option with a node under d) can no longer be selected. The same happens with item g and option 5^{th}.
X2:	Enter level $l = 1$
X3: $i = 2$	Apply MRV: — Items b, e, and f have one option — Item c has two options
X4: $Cover(2)$ $Hide(16)$: "remove" node 17 and 18 $x_1 = 16$	The only option remaining that covers item b is the 3^{rd} option (the one containing node 16).
X5: $Cover(3)$ $Hide(9)$: "remove" node 10 $Cover(6)$ $l = 2$	By choosing the 3^{rd} option to be part of the solution, items c and f are automatically covered. This means that the 1^{st} option can be dismissed since c was already covered.
X2:	Enter level $l = 2$
X3: $i = 5$	Apply MRV.
X4: $Cover(5)$ $x_2 = 5$	At this point, there is no other option available that covers item e. As a result, everything covered and hidden has to be undone in reverse order, until a point where it
X5:	
X7: $Uncover(5)$	
X8: $l = 1$	

X6: $Uncover(6)$ $Uncover(3)$ $Unhide(9)$: “insert” (restore vertical links) node 10 $i = 2$ $x_1 = 2$	can be chosen a different option to be part of the solution.
X5:	
X7: $Uncover(2)$ $Unhide(16)$: “insert” node 18 and 17.	
X8: $l = 0$	
X6: $Uncover(7)$ $Unhide(25)$: “insert” node 24 $Uncover(4)$ $Unhide(27)$: “insert” node 29 and 28 $i = 1$ $x_0 = 20$	Instead of choosing the 2^{nd} option to cover item a, which does not lead to a feasible solution, it is chosen the 4^{th} option (the one containing the node 20).
X5: $Cover(4)$ $Hide(27)$: “remove” node 28 and 29 $Cover(6)$ $Hide(18)$: “remove” node 16 and 17 $l = 1$	The 4^{th} option covers items a, d and f. Hence, 6^{th} and 3^{rd} options are excluded. The 2^{nd} option, that also covers items a, had been previously hidden, so we do not have to do anything about it.
X2:	Enter level $l = 1$
X3: $i = 2$	Apply MRV.
X4: $Cover(2)$ $Hide(24)$: “remove” node 25 $x_1 = 24$	The only option remaining that covers item b is the 5^{th} option (the one containing node 24).
X5: $Cover(7)$ $l = 2$	The 5^{th} option also covers item g.
X2:	Enter level $l = 2$
X3: $i = 3$	Apply MRV.
X4: $Cover(3)$ $Hide(9)$: “remove” node 10 $x_2 = 9$	The 1^{st} option is chosen to cover item c.
X5: $Cover(5)$ $l = 3$	The latter option also covers item e.
X2: $R[0] = 0$ $l = 0$	Are part of the solution the options that contain the nodes $x_0 = 20$, $x_1 = 24$ and $x_2 = 9$.
X8: Terminate.	

Appendix C: Samples of the "250 Sudoku" hobby book published by Edigrama, from levels 4 to 7.

The 81 cells of each sudoku puzzle are represented in a line and should be read as cells of a 9 × 9 grid from left to right and from top to bottom. For a particular cell position in the line: if it has a number, then the referred cell is a fixed cell whose value is that number; if it has a dot, then the referred cell is a blank cell.

Level 4 instances:

```
6.......87..259..4..9...1...2..8..1....3.5....4..6..5...4...8..8..914..52.......3
9...6...7.8.....3....219.....84719....1...4....48235.....654....2.....6.4...3...5
3.......25..8.3..6.9.....7...9.1.2....45298....1.3.5...4.....5.9..2.7..81.......4
...439...6.......1..9...5...21...79..6.198.2..85...41...3...1..7.......4...671...
...921....3.....7...6...5..26..4..98...296...79..3..61..9...6...1.....4....572...
9.......15.1...3.2...413....6.5.2.8..8.....2..7.3.9.1....976...7.6...2.84.......9
...615....1.....7...9.7.1..1.6...7.48..7.9..19.....2.8..5.2.8...3.....4....358...
...8.3....5..6..2...74.23..2...9...58.......45...2...7..61.98...1..8..9....7.5...
..3...7.....823....1.....5..9.3.2.4.3.8...9.6.7.1.6.8..4.....3....561.....6...8..
.8.165.4...6...7.....7.8...5...1...47...5...39...4...2...9.3.....2...3...1.624.5.
```

Level 5 instances:

```
3.......5..93.81.....6.9....7..8..3..26.7.84..4..9..2....9.2.....27.56..7.......2
...543...5.......6..2.9.3...5.4.7.2..7.....8..1.8.9.7...3.7.9..7.......1...634...
...759.....4...9..2...6...5.19...34....321....72...81.1...4...8..6...4.....982...
..56.87..49.....18....9....6..5.7..91.......27..1.4..6....6....92.....63..73.92..
...8.4.....9...7..12.....35.5..9..6.9..716..3.6..4..1.59.....46..3...5.....9.5...
.8..6..4.5...3...6...192.....1...3...6.3.4.2...2...7.....917...6...4...7.3..5..9.
.1.....9...59.84.....536...5..8.1..49.......64..2.7..1...642.....31.95...4.....2.
.6..3..4..7.....5.8...9...1..38.41....2...8....63.54..9...4...3.3.....9..4..6..1.
...2.6....4..1..6.1.......2..6.4.7...1256398...5.3.3..9.......8.7..3..4....7.9...
..2...1..8.9...7.6...927......1.5...36.....57...4.6......862...7.8...9.1..3...8..
```

Level 6 instances:

.32..4.....4..25........6..35..........7........6...2798..3..........4.15...8....

.....5.1.5...9..6.4.3.......7........9......8....46..3......15....93....8.24.....

...78........9...4492......5...68...8....5.3.........2...5..38...6....7...961....

...5...21..9.3....4.6.7.......71.....9......5.8......2.....9.....4..86..3.1...4..

.2....63.8......1....94...........736..........15...4..5.2......4..9........328.

...5..47.84.........5....6....17......3....982...5......9........6..43.......31.4

4...........5......2...3.6..3......6.4...1.3...79.....7.8...9..8...6..4.....2.5.

2...4...94.9.2.3.....1...5..6..........8...........472..2...9....1..8......5.7...

6........8...4..76....159....1.3.....32.9..........5.8.9.8........4.9.........1.5

...5.4....4..9..........2.6....31.....2...7.5..8.2....4......1.91.....5....6.2..9

Level 7 instances:

..42.9.8.6...........18..2....5.4...1.2............4.3.7........8..97.....3...51.

.......9...3.18.....5.6....72...5......8.3...4.....2...4......1......348...6.2...

......29....91....8.5.......76...........3..2.......743....58..28....9.....67....

68........1....9.....2.4......563........974.1......9.....7......5....32..2.8...6

...5......8..1.....6....2.8.59...8....17.4.........31.....3...57.54.........8..46

......27..3..8....91.............7.2...91...43..8...........5....2..4.....6..28.1

83...........2...........4..1.37.......8..4....5...9...2...1..338.....57.7...6...

...7..38.....9..5..17..3...5..........9...1.38..2.......4..1....9..47.........62.

1..62.......93..........2.9....9...723........4..1..3.......34...5..8.....1..67..

4...6.7.81...........92......9.........8.1..7..2....3..5......18.....5.4....36...