TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Valerii Holovko 177231

ONLINE MULTIPLAYER SUDOKU GAME WITH HINTS GENERATOR

Master's thesis

Supervisor: Eduard Petlenkov
professor

Tallinn 2019

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Valerii Holovko 177231

ONLINE MITMEMÄNGIJA SUDOKU MÄNG VIHJETE GENERAATORIGA

Magistritöö

Juhendaja: Eduard Petlenkov
Professor

Tallinn 2019

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Valerii Holovko

30.04.2019

Abstract

In this paper the following topics are covered:

- Analysis, application and modification of existing techniques and algorithms for Sudoku generation;
- Analysis, application and modification of existing techniques and algorithms for Sudoku complexity definition;
- Analysis of suitable network types for developing a multiplayer game;
- Analysis of matchmaking principle and rooms organization.

As the output of this paper the investigation of a multiplayer Sudoku game developing is expected.

This thesis is written in English and is 47 pages long, including 7 chapters and 22 figures.

List of abbreviations and terms

POM	Project object model
P2P	Peer-to-peer
MVC	Model View Controller

Table of contents

List of figures

1 Introduction

Sudoku is a popular puzzle with numbers. Sudoku is a 9th order Latin square. The game field is a 9x9 square, divided into smaller squares with a side of 3 cells. Thus, the entire playing field consists of 81 cells. At the beginning of the game, they are filled with some numbers (from 1 to 9), since the empty playing field does not make sense. The complexity of this game depends on the number of empty cells [2].

The purpose of this work is to create a multiplayer Sudoku game analysing the existing sudoku solving algorithms and multiplayer games structures.

The objective of this work is to implement the following things:

1) Algorithm of the field generation depending on the complexity level of the game:
   - Developing the field complexity checker on the basis of existing techniques and modifying them;
   - Applying existing techniques and algorithms for field generation and comparing it to the alternative hand-made ones.
2) Hints generator – 5 hints available per player.
3) Verification of the correctness of the game result.
4) Introducing multiplayer option:
   - Comparing networks structures benefits;
   - Choosing the most appropriate network type;
   - Mastering the matchmaking concept and rooms organization.
5) Developing a user-friendly user interface for the game.

2 A simple sudoku field generation algorithm

2.1 An algorithm of solving Sudoku from the human being point of view

Of course, I started with the algorithm. Solving the next and very simple Sudoku, I tried to determine the steps that the machine must take to achieve the result. In order I will immediately list those moves that allow solving a simple Sudoku [4]:

1) For each unopened (empty) cell, it is necessary to determine the digits that could be there. Those. they should include numbers that are not among the set values either in the current line, or in the current column, or in the current block. See the figure 1.
2) Go through all the unopened cells for a single value in the cells of possible values. These numbers are depicted with a green background. It is clear that there can be no other value for this cell. Therefore, we open this cell, assigning to it this unique value. After that we go through all the cells of possible values in the cells of the current row, column, block (I called these cells "related") and delete the single value found from these cells.
3) Now we will look for unique values in all undiscovered cells. Unique values are those values that are found in cells of possible values of a row or column or block once. These numbers are depictedwith a yellow background. Again, these unique values can only be set for the cell that contains them. Therefore, we set the value of this cell and delete this value from the cells of possible values of "related" cells.
4) Actually, for a simple Sudoku this is all, i.e. perform clauses 2 and 3 until all cells are open.

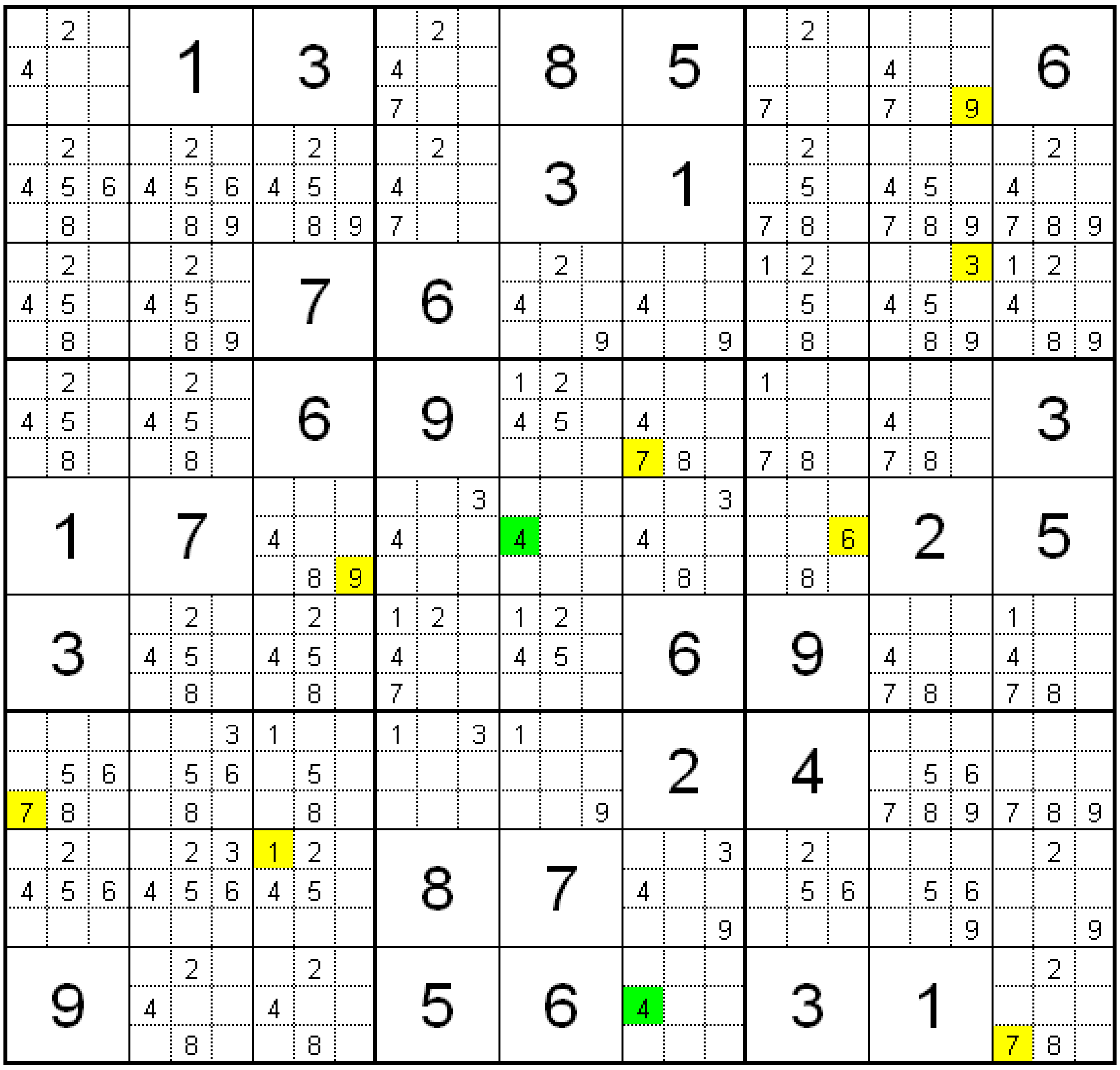


Figure 1. An algorithm of solving Sudoku from the human being point of view [1].

2.2 What to do when there's no unique decision?

We consider the following, fairly complex Sudoku, in which from the very beginning there are no unique values in the cells of possible values. It is the Sudoku that I worked out in order to generate steps of the algorithm, which are described below.

When it can be seen that there is no further unique decision, the most logical way to start is with cells, containers of the least number of possible values, ideally 2 values. Some of these values will necessarily be present in the solved puzzle. Therefore, we assign the first value from the container of possible values to the cell value [5].

This is how the mechanism for finding the expected continuation of the search is implemented. After finding the cell with the minimum number of possible values, we form an array of int type values, nine by nine in dimension, into which we write the values of open cells. In this array, the cell with which we have just “worked” already has a certain value (the first value from the container of possible values). This array is an input parameter for the Core class constructor. Based on the values from this array everything necessary for a solution is found. For all undiscovered cells, the containers of possible values are filled, while for the cell for which we made an assumption, the value is already set [5].

If as a result of the next iteration of the searching for single and unique values cycle those were not found, then [7]:

- Find the cell with the minimum number of values in the container of possible values;
- Based on the values of open cells, as well as the first value from the container of possible cell values found in the previous subparagraph, form an array “assumption” of values of type int, dimension 9x9;
- A cycle of searching for single and unique values is started. This happens until method returns a solution in the form of an array of values of type int, 9x9 in dimension.

2.3 Field generation method description

1) To make a basis field (grid):

   The grid must obey the rules of Sudoku. We place in the first line 1, 2...8, 9 and in the lines below we shift the numbers 3 positions to the left, i.e. 4, 5...2, 3 and 7, 8...5, 6. Next, moving to the next area vertically we shift the previous area by 1 position to the left. As a result, this field should be composed, it is the basic one:

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

Figure 2. Basic field.

2) Shuffle the mesh:

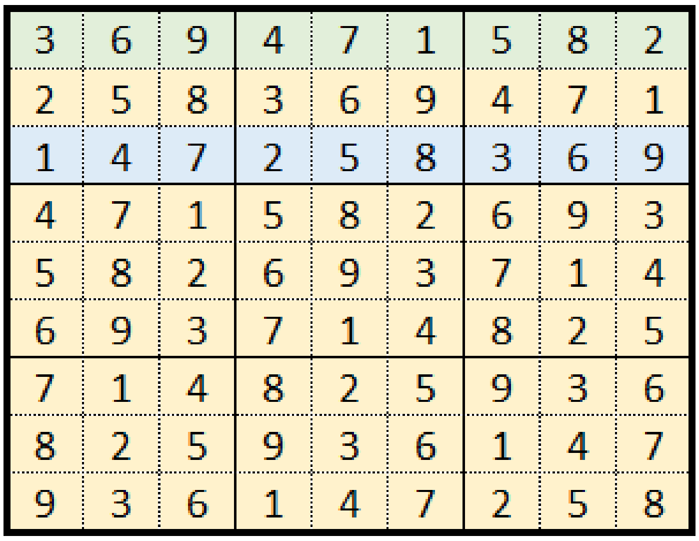
There are several types of permutations, after which the Sudoku table will remain in a valid state satisfying its constraints. These include:

- Transposition of the entire table - columns become rows and vice versa:

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

Figure 3. Transposed field.

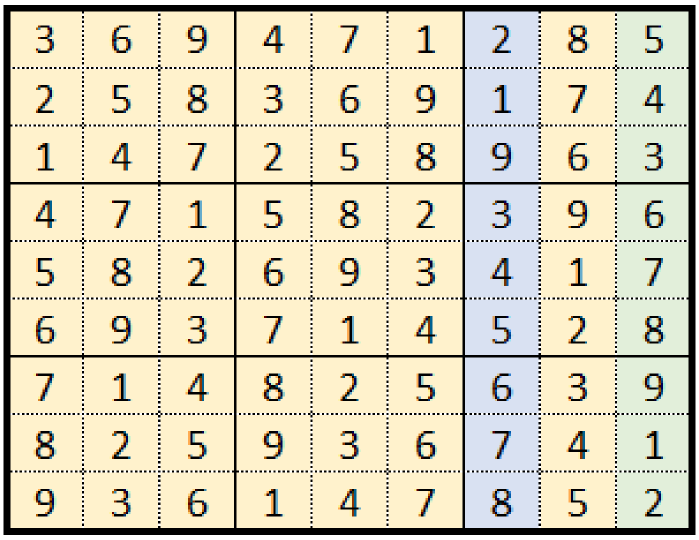
- Swap of two strings within one square:



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

Figure 4. Field with swapped rows.

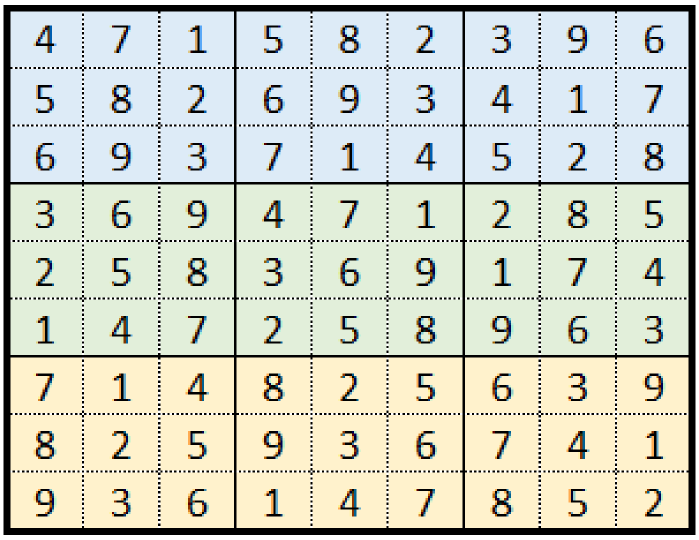
- Swap of two columns within one square:



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

Figure 5. Field with swapped columns.

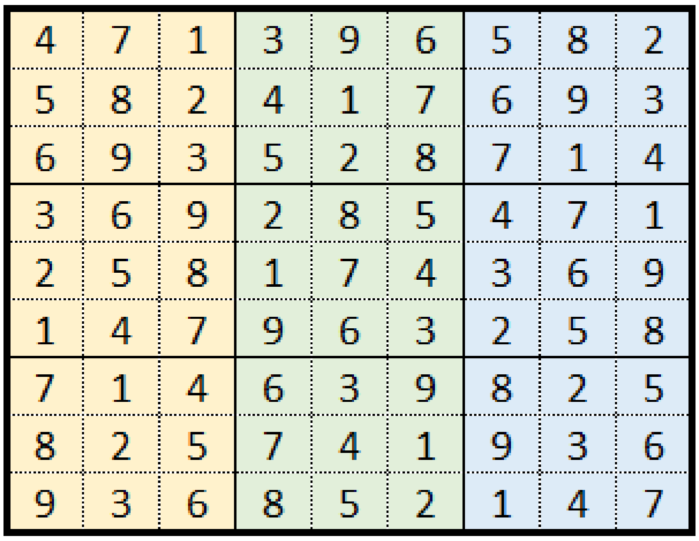
- Swap of two squares horizontally:



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

Figure 6. Field with the blocks swapped horizontally.

- Swap of two squares vertically:



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

Figure 7. Field with the blocks swapped vertically.

Now, in order to get a random combination, it is enough to run the permutation functions in a random order. That provides sufficient level of randomness [3].

3) Erasing the cells' content:

After we got the result, we need to state the problem exactly in this sequence so that we can guarantee the uniqueness of the solution. And this is the most complicated part. The questions is how many cells can be removed to have one and unique solution of Sudoku remained? This is one of the important factors on which Sudoku difficulty depends.

In total, there are 81 cells in Sudoku. Usually, the difficulty is distributed in this way:

1) When there are 30-35 “tips” on the field, it’s considered as an easy one;
2) When there are 25-30 “tips” on the field, it’s considered as an average one;
3) When there are 20-25 “tips” on the field, it’s considered as a complex one.

This is a decision made considering large amount of Sudoku real examples. There are no definite rules for Sudoku complexity. It’s possible to compose a complex one with 30 tips and an easy one with 22, for instance.

So, there are two ways we can proceed:

1) Random approach - you can try to erase 50-60 random cells, but there’s no guarantee that Sudoku could be solved. For example, if there are just 3 lines filled (27 cells);
2) Randomly with a simple constraint approach — for example, you can take a certain number N as a limit, so that N rows and columns could be empty. Taking N = 0 - for light levels, N = 1 - medium, N = 2 – difficult.

So, it’s time to erase the cells (each cells’ probabilities are equivalent, so we have 81 cells that can be erased, so we will check everything with brute force attack):

The steps are the following:

1) Choose a random cell N
2) Mark N viewed
3) Remove N
4) Calculate solutions. If it is not unique, then return value of N.

The output will be the most difficult of the possible Sudoku options for this mixing. The variable that evaluates complexity reflects the number of remaining elements.

First, we need to consider the requirements for the Sudoku field generation that are to be satisfied [4]:

1) The field should be composed in a proper way, i.e. there should be only one possible correct solution.

2) There should be a possibility to generate fields of various complexity, so that it could fit people of different skill levels.

3) The difficulty level should be approximately the same in frames of a certain difficulty level that is set by a player, i.e. the difficulty shouldn't depend on how 'lucky' the player is with the generated field.

4) The algorithm should work quickly.

The algorithm for generating the field is basically working in the following way:

1) The field should be generated in a random way that satisfies all constrains of Sudoku.

2) The random cells of the field should be picked and their symbols should be removed.

3) After removing the symbol from the selected cell, the algorithm should check how many possible solutions of the current Sudoku field there are. If there more than on unique solution, the value should be returned back to the selected cell.

As a result, at this stage, we have a so-called 'irreducible field', i.e. a field that has no more cells which could be deleted so that there's only one and unique solution left in the game.

At this stage we have a field, but there's still no information about how complex this Sudoku actually is.

For step 3 there's a need in a solver and this solver has to be really fast as it's going to be executed for each selected cell many times to generate one single field. When the solver is implemented, it’s actually much easier to compose a field at the step 1.

We need to modify it so that it could work with an empty field and so that it could make random decisions when possible.

3 Algorithm X and ‘Dancing Links’

3.1 Algorithm overview

The construction of Sudoku is ideal to refer it to the problem of constraint satisfaction. The main aspects in this approach are the following: it is not required to distinguish among different digits to paste in a cell, so there’s less to code and it’s a chance to implement Donald Knuth’s ‘Dancing Links’ algorithm. This is a backtracking algorithm which allows to eliminate choices which were set by the constraints very fast and it also allows to put them back very quickly [6] [7] [8].

There’s an example of solving a smallest latin square, which a sudoku field actually is, a two by two latin square. This example is provided to explain the problem of constraint satisfaction.

This two by two square is just the same as sudoku field but has no blocks. As it is a two by two square, there are only digits like 1 and 2 to be paste in its cells.

It’s obvious that there’re only 2 solutions. At first, it is required to construct a matrix of constraints and variants of digits locations. The first are represented by columns of a matrix and the seconds are represented by raws.

As the result I have twelve constraints in case of this particular latin square (as there are 2 rows and 2 columns, i.e. 4 constraints per every of 3 different ones):

1) 1-4: Paste a digit in every cell;
2) 5-8: Paste each digit in any cell of every raw;
3) 9-12: Paste each digit in any cell of every column.

In total there are eight placement variants. In fact, any digit may be put in any cell. It’s expressed in a matrix below:

Choice	**Constraint**											
	A number in column				**The number**				**The number**			
	1		**2**		**1**		**2**		**1**		**2**	
	and row				**must appear in row**				**must appear in column**			
	1	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**
1 at (1,1)	■				■				■			
2 at (1,1)	■						■				■	
1 at (1,2)		■				■			■			
2 at (1,2)		■						■			■	
1 at (2,1)			■		■					■		
2 at (2,1)			■				■					■
1 at (2,2)				■		■				■		
2 at (2,2)				■				■				■

Figure 8. Matrix of possible digits locations [8].

There are 64 ways of putting a number in total (rows) and 64 constraints (columns) and it’s the smallest possible field four by four. For nine by nine field these numbers increase up to 729 ways of putting a number (rows) and 324 constraints (columns) [8].

The reason behind using this matrix is that each solution of the square corresponds to the set of rows of the matrix so that each constrained is covered only once. In the chosen set of rows there’s only one black square in each column. This may be considered as an NP-hard exact cover problem [10] [8].

There are the rows that correspond to the of the above-mentioned square. Every column has one and only black square.

1 at (1,1)	■				■				■			
2 at (1,2)		■						■			■	
2 at (2,1)			■				■					■
1 at (2,2)				■		■				■		

Figure 9. Matrix of the possible solutions [8].

3.2 Algorithm X application

In order to find an existing solution to a square, it is required to apply a backtracking algorithm. A set of rows may be built incrementally and the solution will be the following [8]:

1) It's required to choose a constraint that is not satisfied. It is a column that has no black squares in the rows in the solution set. If there are no constraints that are not satisfied left, the solution set is full. That is a random solution. If there's a need to find all the solutions, this particular one should be saved and then we need to backtrack once again and make a different choice this time.
2) Now we need to chose a row that would satisfy the constraint. That means the one that has a black square in the previously selected column. If such a row doesn't exist, then it means there's no further solution with this set of choices and we must backtrack once again to stage where we are to choose a row and make a different choice this time.
3) If the selected row satisfies the constraint, that needs to be added to the solution set.
4) At this step we need to erase the rows that satisfy the constraints (i.e., the ones that are satisfied by the row that was chosen previously). That means we need to erase all the rows that had black squares exactly in the positions in which the selected raw has, i.e. in the same columns.
5) At this step we need start all over again and jump to the first step.

The verity of the method is not influenced by the column selection at step 1. This constraint choice is not vital, but the difference is in how fast the algorithm works and it’s influenced by the above-mentioned choice. The conclusion is to select the constraint that would be satisfied with the least number of rows. This way the search would be less branched and the speed would be much greater.

The verity of the method is not influenced by the row selection at step 2 either. In case there are lots of solutions come out while backtracking all the rows that are satisfying the constraints, it influences the sequence in which the results are worked out.

1 at (1,1)	■	🟩	🟩	🟩	■	🟩	🟩	🟩	■	🟩	🟩	🟩
2 at (1,1)	■						■				■	
1 at (1,2)	🟥	■				■			■			
2 at (1,2)	🟥	■						■			■	
1 at (2,1)	🟥		■		■					■		
2 at (2,1)	🟥		■				■					■
1 at (2,2)	🟥			■		■				■		
2 at (2,2)	🟥			■				■				■

Figure 10. Selecting an unsatisfied constraint column (highlighted in red) and selecting a row that satisfies this constraint (highlighted in green).

2 at (1,1)	🟧						■				■	
1 at (1,2)		■				■			🟧			
2 at (1,2)		■						■			■	
1 at (2,1)			■		🟧					■		
2 at (2,1)			■				■					■
1 at (2,2)				■		■				■		
2 at (2,2)				■				■				■

1 at (1,1)	🟧				🟧				🟧			

Figure 11. Erasing the rows (highlighted in blue) that satisfy at least one of the constraints that are satisfied by the selected row in the set of solutions (orange squares) and moving the selected row to the set of solutions (highlighted in green).

2 at (1,2)		■		🟥				■			■	
2 at (2,1)			■	🟥			■					■
1 at (2,2)				■		■				■		
2 at (2,2)				■				■				■

1 at (1,1)	■			🟥	■				■			

Figure 12. Selecting an unsatisfied constraint column (highlighted in red) and selecting a row that satisfies this constraint (highlighted in green).

2 at (1,2)	🟦	■	🟦	🟦	🟦	🟦	🟦	🟧	🟦	🟦	■	🟦
2 at (2,1)	🟦	🟦	■	🟦	🟦	🟦	■	🟦	🟦	🟦	🟦	🟧
1 at (2,2)	🟦	🟦	🟦	🟧	🟦	■	🟦	🟦	🟦	■	🟦	🟦

1 at (1,1)	■				■				■			
2 at (2,2)	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧	🟩	🟩	🟩	🟧

Figure 13. Erasing the rows (highlighted in blue) that satisfy at least one of the constraints that are satisfied by the selected row in the set of solutions (orange squares) and moving the selected row to the set of solutions (highlighted in green).

1 at (1,1)	■		🟥		■				■			
2 at (2,2)			🟥	■				■				■

Figure 14. Selecting an unsatisfied constraint column (highlighted in red).

There are no rows that could satisfy this constraint. In this case, we need to backtrack, i.e. go to the previous step and make a different choice.

2 at (1,2)		■		🟥				■			■	
2 at (2,1)			■	🟥			■					■
1 at (2,2)	🟩	🟩	🟩	■	🟩	■	🟩	🟩	🟩	■	🟩	🟩
2 at (2,2)				■				■				■

1 at (1,1)	■			🟥	■				■			

Figure 15. Selecting the same unsatisfied constraint column (highlighted in red) and selecting another row that satisfies this constraint (highlighted in green).

2 at (1,2)		■						■			■	
2 at (2,1)			■				■					■
2 at (2,2)				🟧				■				■

1 at (1,1)	■				■				■			
1 at (2,2)				🟧		🟧				🟧		

Figure 16. Erasing the rows (highlighted in blue) that satisfy at least one of the constraints that are satisfied by the selected row in the set of solutions (orange squares) and moving the selected row to the set of solutions (highlighted in green).

The last two rows are added next. There are added to the set of solutions one by one and the sequence doesn't make any difference anymore. The are no more possible incorrect choices left.

2 at (1,2)		■						■			■	
2 at (2,1)			■				■					■
1 at (1,1)	■				■				■			
1 at (2,2)				■		■				■		

Figure 17. The final set of solutions.

3.3 'Dancing Links' technique overview

Dancing Links is the technique for efficient implementing Donald Knuth's Algorithm X [11]. Usually, exact cover problems the constraints amount which are satisfied by every single row is very small. In the nine by nine field, there are 729 ways of putting a number (rows) and 324 constraints (columns) in the matrix. In total, there are 236196 cells on the field and just a small, in comparison, number of cells is actually occupied – just 2916 [8].

There is a need to go through the constraints (columns) at step two and step four in a very fast way as well as through rows at step four. I need to create a linked list for each column and for each row. In order to remove cells from the corresponding columns at step four there's a need to use a double linked list. In comparison to the usual linked list structure, each node contains an extra pointer that is usually called 'previous pointer' along with the usual pointer and data stored [13].

In my case, the linked list's entities, i.e. cells, actually have four pointers:

1) To the cells from below;
2) From above;
3) To the left;
4) To the right.

In order to select the better column to go through I need some data structure to store the constraints (columns) that has a counter of non-empty cells.

There are column headers on the top (A, B, C…) with the number in each of them. This number represents the number of filled cells in the column. Also, there's a header in the beginning of the list, it's an additional cell. It's needed so that it would be possible to locate the headers that were not yet mentioned, i.e. the constraints that are yet to satisfy.

The whole idea is to eliminate the column from the given matrix altogether with each row that actually intersects this column. This is done at the steps one and four. It's implemented there.

So, it can be clearly seen on the Fig. 18 that the connection between cells were redirected and all the cells marked with the cross icons are hence eliminated from the given matrix. This operation is performed after covering the first column A.

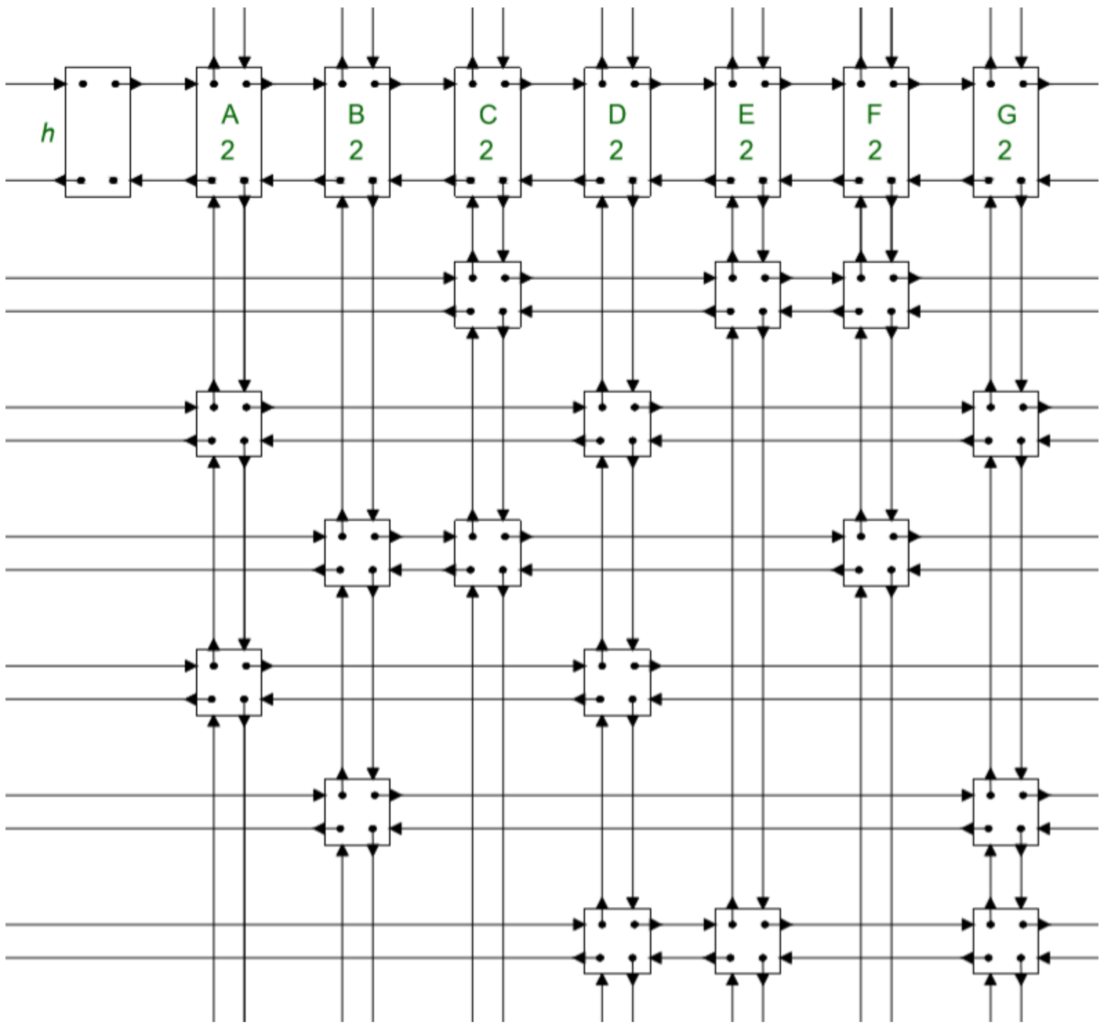


Figure 18. The look of the initial matrix [9].

The very top pros of this approach is that the link to the header of the column is always stored and is never lost. This way, it's always possible and quite easy to backtrack and make a different choice. According to Knuth, this operation is called 'uncovering the column' [12].

It's worth to pay attention and check whether the reverse is performed in the correct order, i.e. exactly in the same sequence as the columns were actually removed.

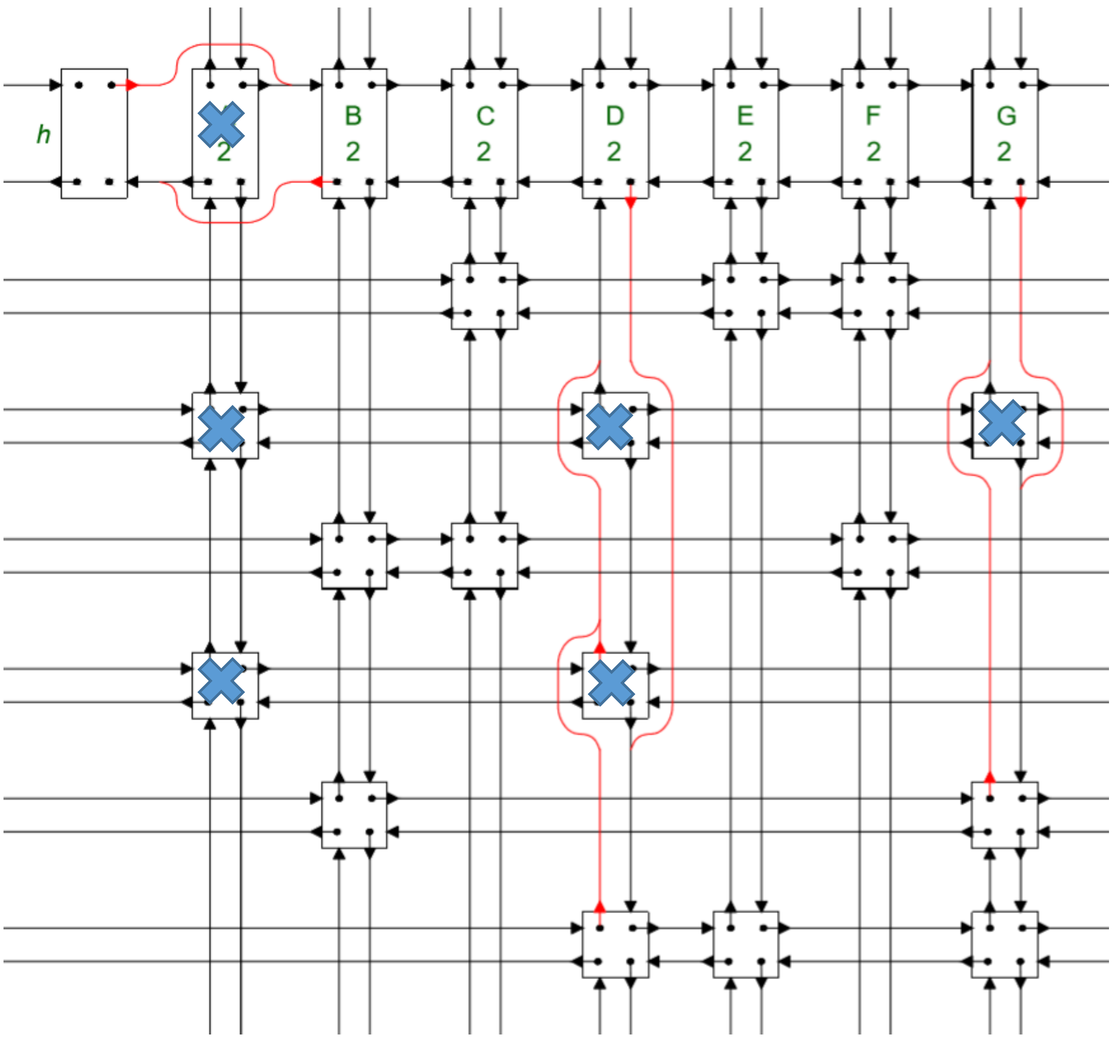


Figure 19. The look of the matrix after covering the column (constraint) [9].

3.4 'Dancing links' application

It's required for the algorithm to operate in 2 options [8]:

1) It needs to compose an arbitrary already correctly filled sudoku field. When the field is composed, the generator stops. The rows should be chosen in the arbitrary order when generating. The very first row may be filled in an arbitrary way with the unique number from one to nine – this way we save time.
2) To perform an analysis of whether there is more than one unique solution of the field. When the algorithm happens to find another solution, it should stop immediately. Otherwise, it should stop when the whole structure is analyzed and there's a conclusion that there's only one unique solution to this field.

The approach that implies a selection of the row with the fewest number of fewest ‘satisfied’ rows has shown itself as the most compelling one rather than the one analyzing the very left remaining column.

4 Difficulty distribution

4.1 Design overview

The grading is based on the human rules that are applied to solve the task. There’s a list of the ‘rules’ that is known by the grader and is ranked in accordance to the difficulty level of this move. So, basically, the main point is to actually solve the task by applying those rules one by one in the ‘difficulty’ order [19].

First, the rule with the smallest difficulty index is applied. If it’s impossible to put a number in a cell applying only this single rule, the next one with the greater difficulty index is applied. In other words, the difficulty index of each conclusion made to completely solve a cell is the difficulty index of the easiest rule that is sufficient to make the conclusion [19].

Each time when there’s a number put in a cell, there’s a mark with the difficulty index of the rule applied to put this number in a cell added as well. There’s a counter that updates each time when the rule of the greater difficulty is applied. This counter stores the greatest index of the difficulty that was achieved when solving the task following the programmed rules [8].

The difficulty of the sudoku is actually defined by the index that is stored in the counter and thus by the ‘hardest’ rule that was applied to solve it.

The hints generator is supposed to work with the same technique. Once a player requests a hint, the game finds the simplest choice and fills the cell.

4.2 ‘Human rules’ description

There are various techniques for solving Sudoku. I need to include in the rules list only ones that are rather simple, i.e. that do not require a person who is solving it to take a paper and make multiple ‘guessings’ with a pencil. These ones would rather make it not an electronic Sudoku version already because it would require third-party items besides your device [8].

Here’s the list of the techniques according to Simon Armstrong that I assume to be the useful ones when estimating the realistic difficulty:

1) Last cell: there are eight already-filled cells in a row, column or a block and hence it’s obvious what the last digit is.

2) Sole Candidate: it is often the case that a cell can only possibly take a single value, when the contents of the other cells in the same row, column and block are considered [16].

3) Unique Candidate: if a cell is the only one in a row, column or block that can take a particular value, then it must have that value [16].

4) Locked Set: if two cells in the same row, column or block have only the same two candidates, then those candidates can be removed from other cells in that row, column or block [16].

5) Hidden Subset: this technique is very similar to naked subsets, but instead of affecting other cells with the same row, column or block, candidates are eliminated from the cells that hold the hidden subset [16].

6) Naked Subset: same as Locked Set, but there are three cells in the same row, column or block have only the same three candidates, then those candidates can be removed from other cells in that row, column or block [16].

The listed techniques are ordered in accordance to their difficulty. It’s useful to set the index in a range between 0 and 100 so that there could a more precise difficultness evaluation be done.

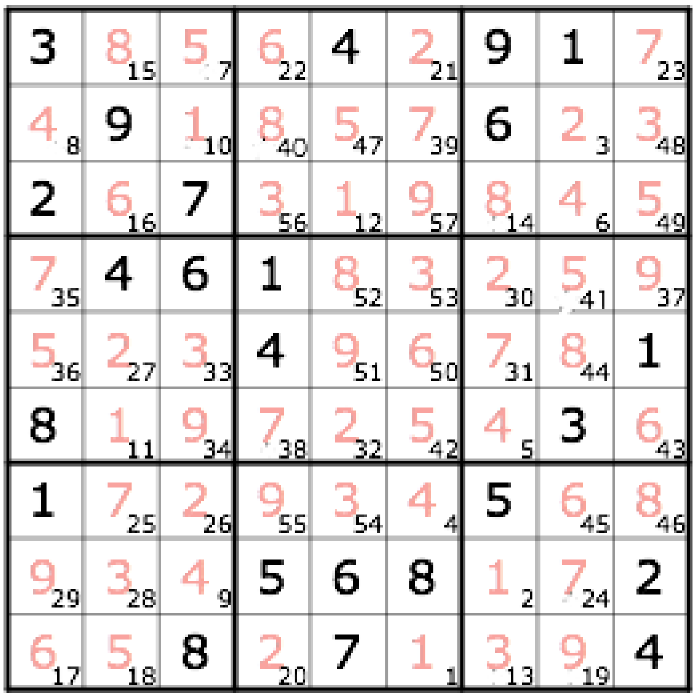
The indexing of the difficulty of the position must be evaluated by the following steps for the evaluation to be more realistic [19]:

1) All the cells that can be ‘solved’ by applying human rules at the current step must be counted.

2) If there are more than one technique must be applied at the same time in order to make a conclusion (or the same rule, but several times), that should be assumed as a single ‘step’ in frames of the decision difficulty. In order to evaluate the

difficulty index for such a decision, the sum of the indexes of the rules applied must be calculated. This approach is not that precise, but is considered to be a trade-off solution between having all these combinations listed as the ready-made techniques.

3) I need to take all the decisions which index is higher than the current minimum one. The user is not going to look for the hardest solution at each step, but for the simplest one.



3	8_{15}	5_{7}	6_{22}	4	2_{21}	9	1	7_{23}
4_{8}	9	1_{10}	8_{40}	5_{47}	7_{39}	6	2_{3}	3_{48}
2	6_{16}	7	3_{56}	1_{12}	9_{57}	8_{14}	4_{6}	5_{49}
7_{35}	4	6	1	8_{52}	3_{53}	2_{30}	5_{41}	9_{37}
5_{36}	2_{27}	3_{33}	4	9_{51}	6_{50}	7_{31}	8_{44}	1
8	1_{11}	9_{34}	7_{38}	2_{32}	5_{42}	4_{5}	3	6_{43}
1	7_{25}	2_{26}	9_{55}	3_{54}	4_{4}	5	6_{45}	8_{46}
9_{29}	3_{28}	4_{9}	5	6	8	1_{2}	7_{24}	2
6_{17}	5_{18}	8	2_{20}	7	1_{1}	3_{13}	9_{19}	4

Figure 20. An example of how the decisions are indexed.

5 Multiplayer concept

5.1 Client-Server networks

The multiplayer in the games may be carried out in various ways and these strategies may be divided into two subgroups which are called authoritative and non-authoritative.

Usually, the authoritative group is represented by the client-server architecture build. In the client-server architecture the main processor is the server – one and single unit that manages all the processes in the game. Each player is connected to the server as a client and there is a steady exchange of the data between them. It’s worth noticing with each data exchange there’s a replica of the application state is created on the client side [14].

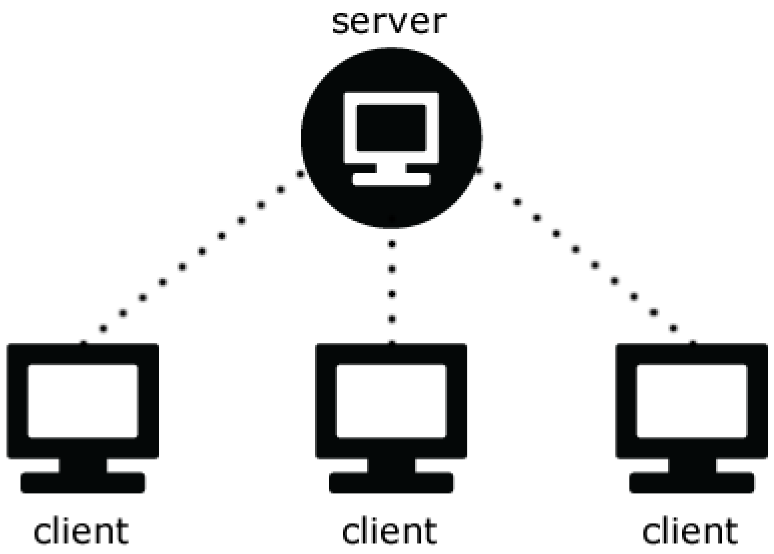


Figure 21. Example of the authoritative approach: Client-Server architecture [14].

First, it’s required to understand the following notions [15]:

- Game host: the physical machine that hosts the game;
- Host server: the application that allows the data transmission and connections between multiple clients;
- Client: a replica of the game state that is received from the server.

For example, if there’s an action like putting some number to a certain sudoku cell performed, then the current updated state of the game is sent to the server. Then server analyses whether this data is satisfactory and then updates the game state on the server side in accordance to the information it received from one of the clients. After that, other clients receive the updates from the server in its turn and update their current local replicas of the game states in accordance to the information they receive.

5.2 Peer-to-Peer networks

In its turn, the non-authoritative group is represented by the peer-to-peer architecture build. In the peer-to-peer architecture there’s no main processor machine. Each peer is actually controlling it game state itself. All the processes are handled in the local machines. In this case, there is a steady exchange of the data between each peer. In case of the straightforward exchange of the data there’s no entity that checks the data sent for the correctness as the server was in this role and hence, the peers are just assuming that the data is correct [14] [20].

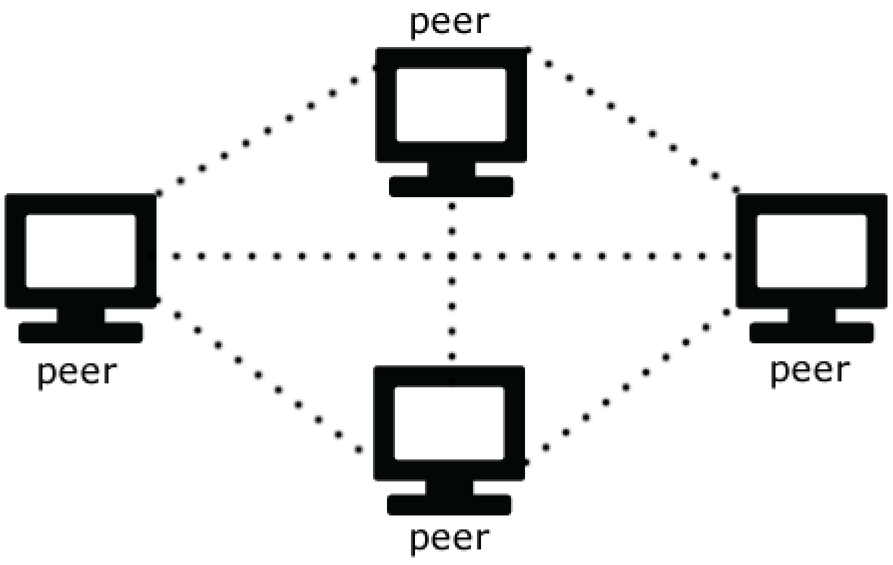


Figure 22. Example of the non-authoritative approach: Peer-to-peer architecture [14].

The authoritative approach is much more protected as the server completely controls the state of the game according to the rules set and can just simply ignore the suspicious data package. As the data security is vital, I assume P2P approach to be the least appropriate solution especially when it comes to cheating.

As the main benefit of this approach, I mention the fact that there is no separate server required, so the cost of maintenance is significantly cheaper.

Unfortunately, if the host (one of the peers that simulates the server) decides to leave the game, the whole game shuts down. So, this is rather a bad side if there's a need to create a multiplayer game with rooms for more than two players.

Also, there is speed up of the connection as the latency is cut in half since the players communicate with each other, there's only one interaction performed. Also, it's possible to distribute the upload bandwidth load [21].

5.3 Peer-to-peer vs Client-Server networks

On the other hand, there's still an open question of the possibility to replace the data packages sent by the clients and hence a way for the players to cheat. There's still a reduction comparing to the Peer-to-peer technology as the check may be implemented on the client side who is acting as a host. If the abuse actions come from the non-host client, then it can be quickly detected when the data comes to his opponent who is acting as a host in the game [21].

Even though, I assume that the security level in case of a P2P connection is quite enough since there's no in-game purchases possible and there's no storage of any personal data of the players. The main disadvantage that forced me to use a client-server approach rather than P2P is that once a host leaves the game for whatever reason (because of the bad connection or just decides to quit the game), the game stops.

I would need to code a host server which comes up with the game. When one of the clients starts the game, it may either host the game or join the existing one. This is supposed to be decided by the game. The host server instance is created once on there is player that is going to host the game. After that, the players that are to be play the role of clients are to search for the existing instance of the host server and to send the request to the connect to it. This way the matchmaking room is created [21].

It's not a problem if the game has only 2 players in the room – one host and one usual client. But if I want to build a solution that allows, for instance, 4 people solving the task together at the same time, that raises a problem as of there're still 3 players left, the game can be continued with no problem from the game essence point of view, but if the one who has left the game would be a host off the game, the game will stop immediately.

Also, as all the functions are handled on the server side in case of a client-server approach, that means that there will be really few updates of the application that a client should install. Most of the updates should only be carried out on the server side, not the client one.

After a research I decide to use a client-server architecture type for the sudoku multiplayer.

6 Project structure

6.1 Project management tools

As nowadays there are lots of services and ready-made frameworks, I would consider using a project build tool that helps installing all the dependencies in the project in an automated simple way. I would consider using which defines how packages for .NET are created, hosted, and consumed, and provides the tools for each of those roles and have preinstalled set of plugins which all NuGet-based project feature which provides a uniform build system [17].

Also, NuGet can provide [17]:

- Managing dependencies in a project tree in an easy way. It takes care of all 'down-level' dependencies;
- Maintains a simple reference list of the packages upon which a project depends, including both top-level and down-level dependencies. That is, whenever you install a package from some host into a project, NuGet records the package identifier and version number in the reference list;
- Provides the tools developers need for creating, publishing, and consuming packages.

Unfortunately, NuGet doesn't have convenient in-built testing tools, so I end up with xUnit unit testing tool.

From already-existing multiplayer frameworks, I decided to try Photon. That's a networking engine and multiplayer platform. It has lots of ready-made solutions for matchmaking, creating and handling rooms and handling clients' connections [18].

6.2 Patterns

I decide using MVC patter for this multiplayer project. MVC pattern defines the interaction between view, controller and model layers in the project. Also, I do need extra service and data storage layers.

7 Summary

As the result of the paper analysis, application and modification of existing techniques and algorithms for Sudoku generation was performed. The application of Donald Knuth's Algorithm X and Dancing Links technique for applying it is the reasonable choice having its advantages.

Also, there was the analysis of Sudoku complexity was performed and it turns out that there is no universally accepted approach. The definition of complexity is performed by trying to solve the generated field with non-machine algorithms, the human ones.

The reasonable choice of network type seems to be a Client-Server approach as it offers complete control over the packages sent by clients, hence cheating is least possible. Taking into account current aim, it is not that vital to reduce expenses for keeping the server by spreading the load over the clients by applying P2P network type.

The output of the work is a developed multiplayer Sudoku game.

References

[1] Sudoku JAVA Solution [Internet source] – URL: https://sites.google.com/site/sudokujavasolution/algorithm#TOC---

[2] What is Sudoku? [Internet source] – URL: http://www.sudoku-space.com/sudoku.php

[3] Habr: Алгоритм генерации судоку [Internet source] – URL: https://habr.com/ru/post/192102/

[4] G. Zambon, Sudoku Programming with C: Apress, 2015.

[5] Mathematics and Sudokus: Solving Algorithms [Internet source] – URL: http://pi.math.cornell.edu/~mec/Summer2009/meerkamp/Site/Solving_any_Sudoku_II.html

[6] Algorithm X in 30 lines [Internet source] – URL: https://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html

[7] O. A. Pestov, Учебный модуль: Перебор с возвратом: Kirov, 2015 [Internet source] – URL: http://kuimova.ucoz.ru/modul_7-perebor_s_vozvratom.pdf

[8] Zendoku puzzle generation – Gareth Rees [Internet source] – URL: http://garethrees.org/2007/06/10/zendoku-generation/

[9] GeeksForGeeks: Exact Cover Problem and Algorithm X – Atul Kumar [Internet source] – URL: https://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-2-implementation-dlx/

[10] Wikipedia: Exact cover [Internet source] – URL: https://en.wikipedia.org/wiki/Exact_cover

[11] Wikipedia: Knuth's Algorithm X [Internet source] – URL: https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X

[12] Wikipedia: Dancing Links [Internet source] – URL: https://en.wikipedia.org/wiki/Dancing_Links

[13] GeeksForGeeks: Doubly Linked List [Internet source] – URL: https://www.geeksforgeeks.org/doubly-linked-list/

[14] Evantotuts+: Building a Peer-to-Peer Multiplayer Networked Game – Fernando Bevilacqua [Internet source] – URL: https://gamedevelopment.tutsplus.com/tutorials/building-a-peer-to-peer-multiplayer-networked-game--gamedev-10074

[15] Gaffer On Games: What Every Programmer Needs To Know About Game Networking – Glenn Fiedler [Internet source] – URL: https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/

[16] SadMan Software: How to Solve Sudokus - An Index of Sudoku Solving Techniques [Internet source] – URL: http://www.sadmansoftware.com/sudoku/solvingtechniques.php

[17] Microsoft: An introduction to NuGet [Internet source] – URL: https://docs.microsoft.com/en-us/nuget/what-is-nuget

[18] Photon: Matchmaking Guide [Internet source] – URL: https://doc.photonengine.com/en-us/realtime/current/lobby-and-matchmaking/matchmaking-and-lobby#play_with_your_friends

[19] Andrew C. Stuart, Sudoku Creation and Grading: 2007 [Internet source] – URL: http://www.sudokuwiki.org/Sudoku_Creation_and_Grading.pdf

[20] Research Gate: Peer-to-Peer Architectures for Massively Multiplayer Online Games: A Survey: Amir Yahyavi, Bettina Kemme, McGill University [Internet source] – URL: https://www.researchgate.net/publication/262233529_Peer-to-Peer_Architectures_for_Massively_Multiplayer_Online_Games_A_Survey

[21] Research Gate: P2PSE - A Peer-to-Peer Support for Multiplayer: elipe J. Vilanova, Carlos Eduardo B. Bezerra, The Federal University of Rio Grande do Sul, Brazil [Internet source] – URL: https://www.researchgate.net/publication/228692972_P2PSE_-_A_Peer-to-Peer_Support_for_Multiplayer_Games

Appendix 1 – Program source code

```
namespace Sudoku.Model
{
	public class CompleteField
	{

		public List<Cell> Field { get; set; }
		public int FieldNumber { get; set; }
		public FieldStatus Status { get; set; }

		public FullBoard()
		{
			Field = new List<Cell>();
			Status = FieldStatus.Normal;
		}
	}
}
```

Figure 1. CompleteField.cs listing

```
namespace Sudoku.Data
{
	public class fieldContainer : IFieldContainer
	{
		private const string fieldComposition = "FieldComposition.xml";
		private const string savedGame = "SavedGame.xml";

		readonly private string fieldCompositionPath = string.Empty;
		readonly private string savedGamePath = string.Empty;

		public fieldContainer ()
		{
			fieldCompositionPath                          =
Path.Combine(AppDomain.CurrentDomain.GetData("APPBASE").ToString(),
fieldComposition);
			savedGamePath                                 =
Path.Combine(AppDomain.CurrentDomain.GetData("APPBASE").ToString(),
savedGame);

	}
}
```

Figure 2. A part of FieldContainer.cs listing

```
namespace Sudoku.Controller
{
	public class ComposingController : Controller
	{
		private IFieldTemp fieldTemp;
		private IFieldService fieldService;

		public  ComposingController(IFieldTemp  fieldTemp,  IFieldService
fieldService)
		{
			fieldTemp = temp;
			fieldService = service;

			ViewData["FieldSize"] = Constants.FieldSize;
			ViewData["BlockSize"] = Constants.BlockSize;
		}

		public ActionResult ComposeField()
		{
			CompleteField  field  =  new  CompleteField()  {  Field  =
fieldService.ComposeNewField() };
			return View("BuilderView", field);
		}

		[HttpGet]
		public ActionResult SaveField(CompleteField field)
		{
			int fieldNumber = field.Field;
			fieldTemp.SaveField(field.Field, ref fieldNumber);
			field.Field = fieldNumber;
			field.Status = fieldService.GetFieldStatus(field.Field);
			TempData["Field"] = field;
			return RedirectToAction("SaveField");
		}

		[HttpGet]
		public ActionResult SaveField()
		{
			return                                View("BuilderView",
(CompleteField)TempData["Field"]);
         }
		public ActionResult UpdateField(CompleteField field)
        {
		field.Status = fieldService.GetFieldStatus(field.Field);
		return View("BuilderView", field);
	}
	}
}
```

Figure 3. ComposingController.cs listing

```
            public ActionResult NewGame()
            {
                  CompleteField  field  =  new  CompleteField()  {  Field  =
fieldService.fieldComposition() };
                  int fieldNumber;
                  field.LoadNewPuzzle(field.FieldList, out fieldNumber);
                  field.fieldNumber = fieldNumber;
                  return View("GameView", field);
            }

            public ActionResult UpdateGame(CompleteField field)
            {
                  field.Status                                           =
fieldService.GetFieldStatus(field.FieldList);
                  return View("GameView", field);
            }

            public ActionResult ResetGame(CompleteField field)
            {
                  field.Field = fieldService.fieldComposition();
                  fieldLoader.ReloadPuzzle(field.Field, field.fieldNumber);
                  return View("GameView", field);
            }

            public ActionResult SaveGame(CompleteField field)
            {
                  fieldSaver.SaveGame(field.Field, field.fieldNumber);
                  field.Status                                           =
fieldService.GetFieldStatus(field.FieldList);
                  return View("GameView", field);
            }

            public ActionResult LoadGame()
            {
                  CompleteField  field  =  new  CompleteField()  {  Field  =
fieldService.fieldComposition() };
                  int fieldNumber;
                  fieldLoader.LoadSavedPuzzle(field.Field, out fieldNumber);
                  field.Status = fieldService.GetPuzzleStatus(field.Field);
                  field.fieldNumber = fieldNumber;
                  return View("GameView", field);
            }
```

Figure 4. A part of MatchController.cs listing

```
namespace Sudoku.Service
{

	public class Cell
	{
		public int X { get; set; }
		public int Y { get; set; }
		public int BlockNumber { get; set; }
		[Range(1,       Constants.BoardSize,       ErrorMessage       =
Constants.CellErrorMessage)]
		public int? Value { get; set; }

		public Cell()
		{
			Value = null;
		}
	}
}
```

Figure 5. A part of Cell.cs listing

```
namespace Sudoku.Controllers
{
	public class StartController : Controller
	{
		public ActionResult Start()
		{
			return View();
		}
	}
}
```

Figure 6. StartController.cs listing

```
            private Random random = new Random();
            private IFieldContainer fieldContainer = null;

            public FieldLoader(IFieldContainer fieldContainer)
            {
                  fieldContainer = fieldContainer;
            }

            public   void   LoadNewField(List<Cell>   cellList,   out   int
fieldNumber)
            {
                  fieldNumber = GetRandomFieldNumber();
                  LoadField (fieldNumber, cellList);
            }

            public void ReloadField(List<Cell> cellList, int fieldNumber)
            {
                  LoadField(fieldNumber, cellList);
            }

            public void LoadField(List<Cell> cellList, out int fieldNumber)
            {
                  XDocument savedGame = FieldRepository.LoadGame();
                  XElement                x                        =
savedGame.Descendants("Field").FirstOrDefault();
                  fieldNumber = (int)x.Element("Number");
                  LoadCellListFromFieldXElement(x, cellList);
            }

            private int GetRandomFieldNumber()
            {
                  XDocument fieldSetupDoc = fieldContainer.LoadField();

                  var                fieldNumberList                  =
fieldComposition.Descendants("Field").Select(b                       =>
(int)b.Element("Number")).ToList();
                  int fieldConfigCount = fieldNumberList.Count();
                  int randomFieldIndex = random.Next(fieldConfigCount);
                  return fieldNumberList[randomFieldIndex];
            }

            private void LoadField(int fieldNumber, List<Cell> cellList)
            {
                  XDocument fieldComposition = fieldContainer.LoadField();

                  XElement x = fieldComposition.Descendants("Field").First(b
=> (int)b.Element("Number") == fieldNumber);
                  LoadCellListFromFieldXElement(x, cellList);
            }
```

Figure 7. A part of FieldLoader.cs listing

```
	public class FieldSaver : IFieldSaver
	{
		private IFieldContainer fieldContainer = null;

		public FieldSaver(IFieldContainer fieldContainer)
		{
			fieldContainer = fieldContainer;
		}

		public void SaveGame(List<Cell> cellList, int fieldNumber)
		{
			XDocument savedGame = InitializeSavedGame();


		private  void  OverwriteField  (List<Cell>  cellList,  XDocument
fieldComposition, int fieldNumber)
		{
			XElement                  cellsXElement                  =
fieldComposition.Descendants("Field").First(b  =>  (int)b.Element("Number")  ==
fieldNumber).Element("Cells");
			cellsXElement.ReplaceWith(CreateCellsXElement(cellList));

			fieldContainer.SaveField(fieldComposition);
		}

		private        int        FindNextAvailableFieldNumber(XDocument
existingFields)
		{
			return    existingFields.Descendants("Field").Max(b     =>
(int)b.Element("Number")) + 1;
		}

		private  static  void  SaveField(List<Cell>  cellList,  XDocument
saveXDoc, int fieldNumber,
						Action<XDocument> saveMethod)
		{
			saveXDoc.Element("FieldComposition").Add(
				new XElement("Field",
				  new XElement("Number", fieldNumber),
				  new XElement("Difficulty"),
				  CreateCellsXElement(cellList)));

			saveMethod(saveXDoc);
		}

		private static XElement CreateCellsXElement(List<Cell> cellList)
		{
			return new XElement("Cells",
				cellList.Select((c, i) => c.Value.HasValue ?
									new
XElement("Cell",   new   XAttribute("index",   i),   new   XAttribute("value",
c.Value.Value)) :
							null));
		}
```

Figure 8. A part of FieldSaver.cs listing