Uppsala University

High Performance Programing

1TD062 62013

# Project - report

*Authors:*
Csongor Horváth

March 15, 2023

# Introduction

In my individual project of the course I chose to work on the large scale sudoku solver. I found this task the most interesting, because I was always interested in logical puzzles as sudoku and similar. Also I have an interest in Linear and Integer Programming, so it would be interesting to compare run time performances for solving large scale sudokus with MIP solver, which is quit efficient for solving sudoku problems in my experience. But these result will be only cited from outer source and not implemented by myself.

This problem is also widely studied up to this day and motivated several thesis project such as [2] and several papers such as [1], [3], which are studying the problem with an IP and a brute-force point of view.
In my project I implemented the given algorithm, which is a brute force sudoku solver with back tracking. The basic of the problem is to fill out a partially filled table of size $(n^2 \times n^2)$ with the numbers $(1, \dots n^2)$,so that no row, column or $n \times n$ distinct square has same numbers in it.
In the implementation the back tracking means basically that, we stop continuing one filling out of the table if with the last added number to it it already has duplicates of a number in a sufficient region.

Since it's been a long time goal of mine to write a code, which generates Sudoku, which have only one solution, therefore I also added this aim to my project. So the outline of my project: First I created a straight forward implementation of the given pseudo code correction it's errors and tested it's correctness. This algorithm can only determinate if the starting table has a solution or not. Then I added an extra feature, which can determinate weather a table have more than one solution. I have done it by adding a global variable which stores how many times the program reached a correct solution and only exit the recursion if it is the second time it is reached a good full table, otherwise it is keeps running. And finally I wrote two different algorithms to generate sudoku tables with the above criteria.

After I finished with this I started the optimization of my code. First I did my serial optimization of the code, then I made a parallel implementation. The parallel implementation is differ a lot from the original, because due to the recursive call structure it is impossible to do simple parallel implementation.

So these are the things in the following report about my project.

## Original code

First show the code for the simply task of the original pseudo code of an algorithm which can decide about a partially filled table if it can be completed to a full "sudoku" table. The main part of my task will be to optimize code based on this. This code is work fine, to call it we need a `table` int pointer to a list with length of $N^4$, this represent the table of size $N^2 \times N^2$ with the conventional indexing. `N` is represent in the call of the `solve_table` function the same $N$ value as I am referring to $n$ in the introduction. The `head` int pointer shows a list with the indexes of empty cells from the index 1 and `idx` is the number of unfilled cells.

Listing 1: Original C implementation of the algorithm

```
int validate_table(int* table, int id, int N){
    int NN=N*N; int x = id/(NN); int y = id%(NN);
    for(int i=0;i<NN;i++){
            if(table[x*NN+y] == table[i*NN+y] && x != i)
            return 0;}
    for(int i=0;i<NN;i++){
            if(table[x*NN+y] == table[x*NN+i] && y != i)
            return 0;
            }
    for(int i=-(x%N); i<N-(x%N);i++){
        for (int j = -(y%N); j  < N-(y%N); j++){
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)]
                && (i!=0 || j!=0))
            return 0;}}
    return 1;}
int solve_table(int* table, int N, int* head, int idx){
        if(idx==0){return 1;}
        int id=head[idx];
        for(int i=1; i<N*N+1;i++){
            table[id]=i;
            if (validate_table(table, id, N))
                if(solve_table(table, N, head, idx-1))
                    return 1;}
        table[id]=0;
        return 0;}
```

## Sudoku generation

Now I would like to introduce my methods for generating a $9 \times 9$ Sudoku with only one solution and the problems I encountered during it. First I had to came up with a method for generating a starting table, which don't already hurts the conditions. Therefore I made a function which files up a starting table with random numbers and each cell is filled based on a random number with a given probability. I used 5% of probability, because when I gave more than 10% of probability to a cell to be filled I mostly got bad starting tables. Now this is barely any number to start with, but I used this version. I also noticed that by default the algorithm will iterate through the empty cells cells by cells and row by row and since my starting table is mostly empty it will give boring and executable results. Therefore I introduced a shuffle to the list of the empty cells indexes.

Here I encountered my first problem. Because the time without shuffling the time for finding one solution was always very quick due to the spare starting table, but with shuffling the time to find one solution already vary a lot more than anything else I encountered during this task. E.g. for $9 \times 9$ table with ordered list it was too small to measure, but after shuffling the list well it could go up to more than 5 minutes just to find the first solution.

Therefore I will need to consider carefully what time measurements I want to use. And to be able to compare the run times I will have to fix some starting tables and order of shuffle with them or use large spare starting tables with ordered list of empty cell. Otherwise times would be incomparable. But this way I will lose the generality of the timing and it will be showing the time to solve very specific tasks.

Now back to my Sudoku generation code. First I implemented a version which was unreliable therefore I will only introduce the basic idea and I won't show my code for it here. The idea was that through the recursion I can continue running the function until I find a second solution. During this time I can make a note of it which one was the highest level of recursion where a change is happened. So if I only leave the fields in the lower part of recursion empty, than the solution must be unique. In theory it is working fine, but due to the usage of shuffle the possible high number of loops in the inner recursion can take too long time (more than 10min). But in some case it runs fast($< 1s$). In this cases it gives a correct result and mostly the number of empty field was vary between 30-35. But the shuffles when more empty field would occur takes too much time. As a note I mention that for these generation I only used small number of shuffling.

Than I made a second somewhat more reliable algorithm for generating sudokus with unique solution. Here the idea was that even with small number of shuffles the algorithm which was running until it's found the second solution in the recursion could take too long time. So I wanted an implementation which only runs till the first solution is found but still usable to generate sudokus. So after finding the first solution I stopped the algorithm this time and started leaving out the last some elements of from the list of the originally empty cells. So I set them back to being

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

(a)

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

(b)

Figure 1: Generated sudokus

empty and checked if I can find more than one filling out of the table. When I reached the point of having more than one solution I took a step back and got a table sufficient to my needs. To try and free up even more cells I add a number of reshuffles at the point where failing emptying the next cells without getting more possible solution. This way I achieved a more reliable generating algorithm, which can be still pretty unreliable based on the random shuffling with run times. Also this way I could achieve tables with up to 50 empty cells with less than 5s run time on more occasion. So I present some self generated sudoku below in figure 1 and figure 2.

As this generative method is based on completely on the above presented `solve_table` function. I only modified the the `if(idx==0)` case in it and called it multiple times as explained above and as can be seen in the code in the appendix (`new_gen.c`). And the most time consuming part of the generation is done by this function, so the optimization of the run time for my sudoku generator is equivalent with the optimization of the original code or it's modified version presented in listing . Therefore in the next parts I will only concentrate to optimizing that part of my code for generating sudokus. And the submitted C code only contains that and the time measurement experiments for it as the point of this project is the code optimization procedure. And therefore the whole generative code can be found only in the appendix in the report.

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

(a)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | 7 | | |
| | 7 | 1 | 5 | | | | 6 | |
| | | | | | | 4 | 1 | |
| | | | | 8 | 3 | 1 | | |
| | 5 | 4 | 6 | 2 | 1 | | | |
| 1 | 3 | | | 5 | | 2 | | 6 |
| 5 | 1 | | 4 | 3 | | | | 2 |
| | | | 2 | | | | | |
| | 2 | | | 7 | 5 | 6 | | 3 |

(b)

Figure 2: Generated sudokus

## Serial code optimization

### Experimental design

As I already explained in the previous section the run time vary a lot if I use different starting tables and different shuffles of the list of the empty cells. Note that this measurement is more precise if I include more cases, so my 3 test cases are a sum of multiple run times with similar length and similar characteristics. So in the submitted code I use a binary *input* file containing the starting setup as a starting table and a shuffled list of the empty fields and the number of empty fields in $2 * N^4 + 1$ int number.

I set up a number of test cases and made time measurements using a bash script to run my program for all test file and sum the blocks of the test cases. Using sum of similar test cases ensures stability of the run times, so the small variances due to random effects are become less significant.
Now I would like to introduce the test cases. First I wanted to have a case for larger table. As the run times with empty starting cells and no shuffle are typically fast I tested it, and the run time for $N = 4$ is still very small, but for $N = 5$ it was too long ($> 30s$), so I don't waited until it finished. But the $N = 4$ case this way the run time too small, so I filled up with the first row with a shuffled version of the numbers and saved this as my first test case.
For the second test case I wanted to have some realistic problem, so I copied real life sudokus from the Internet and used an unshufled list. This test case is the least reliable as the sum of the run times are still very small and the individual test cases run time can vary a lot due to small execution time. I still choosing this as a test case because this could be a real life application of the problem. The others are more made up, but for optimization purpose I will consider the other 2 measurements more.
My third test case is the largest one. Here there is a few celled filled in to start with and a small amount of shuffle in the list is applied. I tried to chose cases where the run time was not too small and not too long. This is the larges test case with most variance. This is the sum of 8 run time.

To make my time measurement reproducible all my code and the binary files connection to this project can be found in the following Github link: `https://github.com/horvathcso/HPP-proj`.

So now first I present time measurements with the original code I will add 3 number as run time every time in the following order. One for the $16 \times 16$ table, one for the $9 \times 9$ sudokus without shuffle and one with the $9 \times 9$ tables with shuffle. As previously explained these time measurements are sums.
For time measurements I used the university Linux systems with Intel(R) Xeon(R) CPU E5520 2.27GHz CPU. Without any optimizations the run times were 1.2858s, 0.0135s and 32.5652s. The code for this version can be found under the name *origin_test.c* in the appendix.

Now I would like to explain a bit more in details the code of my experiment. It contains multiple function, the one of which didn't really used for the running of the experiment, but much more were used during testing and during setup of the experiment are not in the submitted version, so I don't speak about them. The only exception is the `print_table` function which is only not used for making the bash script much simple to sum the run time results. There is the standard `get_wallsecond` function used for time measurement. There is a `read_data` function used to read the starting position of the experiments from the above described binary files. The original design was capable of running each solver multiple times for longer and more stable run times for this I used a `cpy_table` function to copy existing setups. Note that in the presented time measurements this loop length is set to 1. And there is a `check` function for checking initial and final table. And then there is a version of `solve_table` and `validate_table` function with possibility to search for multiple solution. Note that all test case is run only for the first found solution. But due to the setup of the algorithm it is equivalent to optimize to first found solution or second found solution.
In the `main` the set up look like. I read the data from file to variable, check if it already hurts validation. Then I run the experiment in a for cycle possibly multiple times, but in the presented measurements always one time. I measure this solving procedure. Then I check the result and give the run time in second to the console output.
I also wrote a bash script to run all my time measurement for one code. The file name and the flags was changed in the bash script for different measurements, but an example can be seen in the appendix.

The runtime measurements can be found in the table at the end of this section.

## Strength reduction techniques

The *strength reduction* seems mostly useless in our case as I don't use much arithmetic operator to start with. In the main part `solve_table` and `validate_table` the only arithmetic operation outside the ++ in the for loops are `int x = id/(NN); int y = id%(NN);`. And these can be only avoided if the list containing the empty indexes would store already the pair of indexes, which would indicate working with

a list of structure which can raise other problems in the process. It may result in slight time improvement, but when I tested it I don't find significant time difference therefore I stick to my original implementation with these arithmetic. Another modification I could make is that the second one of this two calculation can be done by subtracting the first result multiplied by the length of the rows from `id`. As multiplication and subtractions are faster than modulo, but it also don't had any effect on my measurements. But I am not implementing this due to no run time improvement.
Another addition to strength reduction is that I can reach speed up by using constant variables where possible. Thus speeding up the arithmetic functions. So a change I can try out is declare the `N,NN` variables globally as constant variables instead of simply variable and moving them as function argument as they are predetermined in each cases. For this I have declared in the code the N,NN values as I can't change a const variable. The run time this way would slightly improve as can be seen in our two more significant test case (times: 1.7683, 0.0165, 37.5256). I still leaving out this due to it leads to functional loss as the size of the table this way can't be added in run time, but defined in compiler time. And in my point of view this small improvement don't worth it. But if one would optimize their code for fixed $N$, than it can be a good idea to add.
I also tried to only set each `N,NN` variable `const`, but this didn't resulted any improvement.

## Function inlineing and the restrict key word

The next thing I can try is use function inlineing. This works by adding `static inline` to the functions. This is important to the most called functions, but I can add it to every function as I am not intended to call them from other files, so the static declaration is not a problem. Maybe the functions are too big, so the compiler ignores the inlineing or just the effect is not significant, but I couldn't see any significant improvements in run times.

Additionally I am use the restrict key word for the functions argument for pointers so it is clear that they are referring to different memory segments. This can be done only at `sole_table` where there is two integer pointer. This seemingly improved the run times slightly for the first two case as it is showed in table 1. Therefore this is the first long term modification to my code.

## Loop optimization and branches

Loop optimization can come from loop unrolling and loop fusion. The most significant change I am sure should be helpful is that I can merge the first two for loop in the `validate_table` function easily. Moreover to improve performance using one if statement is enough inside the one for loop to get the same result. And the original order of the `and` statement in the if was the better one as $x! = i$ almost never fails. And the order of the new `or` is insignificant. But I can't leave the branch out of the loop as that is the only point of the loop.
For the other loop I could also merge it to the one loop, but due to splitting the index to row column it would be inefficient. So instead I can try and speed up it

by defining `x%N` and `y%N` outside the loops, so the only calculated once and not in every for loop.
Note that when I combined this with the restrict I left out the static inlineing because it somehow made run time slower with the modified for loops. Also combining the loop modification with restrict didn't seems to result further improvement from simply using one of them.
Also note that I could try to speed up a bit more by loop unrolling, but the next part I will start using compiler optimization, which is much better at it for this code.

## Compiler optimization

In the end I most likely will use the compiler flag `-Ofast -march=native` , but first I tried out the weaker optimization flag which results can be seen in table 1. As it can be seen the most improvement reached by techniques implemented already in `-O1`, but `-O2,-O3` also make further speed up. Interestingly `-Ofast` flag is not really better than `-O3`. And `-march=native` makes also no real improvement. But I would suggest using the strongest flag, which is `-Ofast -march=native`. This is supposedly the fastest and also it seems to be the fastest from my measurements.

### Vectorization, caching and memory usage

It would be also possible to play with the vectorization and caching to gain speed. For caching the main advantage would come from the table to be fitted in the cache memory in a good way to speed up the system for checking. This is not a possible task, because if we can't fit the whole table in the cache, then a cache line will not contain the correct row and column at the same time anyway. But it is possible that I did not chose the best storage with a flat list of length $N^4$ for performance purposes. E.g. it could be stored as list of list, but in my opinion that will lead to worse performance due to more cache misses. For real life application if we would need further speed up, then we could use analysis tools to find out the exact number of cache misses with different underlying storage structures.

An other speed up would come from vectorization. Note that the auto vectorization is enabled from `-O3`, but I used `-fopt-info-vec-all` to check that I guessed it correctly that with the given problem I don't get any vectorized loop. Note that I couldn't find out how I could modify my code that it could use auto vectorizatoin for the loops, because I only using loops to check complicated things with if statements. Which can't be vectorized.

A bug could be in the program to contain memory leaks and insufficient memory usage. Therefore I used `valgrind` for each executable to test for memory leak in the programs and I could not find errors with it in my final versions of codes.

## Additional not usable optimization techniques

We also learned about the fact that avoiding small function calls and using pure functions can result in speed up of the program. In my problem I don't see how I could use thees techniques, but for other applications they can be considered to use.

*The final code after serial optimization can be found in the appendix under name final_test.c*

| | $16 \times 16$ | sudokus | $9 \times 9$ shuffled |
|---|---|---|---|
| original | 1.8198 | 0.0173 | 43.6123 |
| only inline | 1.7667 | 0.0166 | 42.9265 |
| inline+restrict | 1.3423 | 0.0109 | 30.7505 |
| loop merge | 1.2620 | 0.0108 | 30.3068 |
| loop + inline + restrict | 1.8994 | 0.0160 | 41.3627 |
| restrict + loop | 1.2 | 0.0094 | 26.2176 |
| `-O1` | 0.7226 | 0.0066 | 13.1006 |
| `-O2` | 0.6532 | 0.0052 | 11.2535 |
| `-O3` | 0.5926 | 0.0039 | 8.9598 |
| `-Ofast` | 0.5873 | 0.0045 | 8.7815 |
| `-Ofast -march=native` | 0.5461 | 0.0045 | 8.3814 |

Table 1: Time measurements

## Parallel optimization

It is a complicated task to make our code parallel. The problem is caused by the fact that we need to make the for part of the `solve_table` function to make parallel, but it is not possible easily for more reason. First there is an if return statement inside, which is not supported by the parallelization framework. Secondly I don't want to step back to the true brute-force method, but I want to use backtracking, so I want to finish all parallel process once a return value is reached in the for loop. Thus I will use a shared variable instead of return statement and instead of return I will have to do a break from the parallel loop to be able to use parallelization.
First I tested whats happens if I make the `validate_table` function parallel, but it doesn't achieve any speed up, only significant decreasement in run time performance. I did the implementation in openmp which resulted in the below code. Most likely it makes the performance poorer, because the cycles are short and most of the time they broke sooner with the original implementation due to a return statement, this way it is doing at least $NN$ testing before returning.

```
int validate_table(int* table, int id, int N){
    int NN=N*N;
    int x = id/(NN); int y = id%NN;
    int esc=0;

    #pragma omp parallel for
    for(int i=0;i<NN;i++){ // row col
        if((table[x*NN+y] == table[i*NN+y] && x != i) || (
           table[x*NN+y] == table[x*NN+i] && y != i))
            {
            esc=1;
            }
    }

    if(!esc){
    int xn=x%N; int yn=y%N;
    #pragma omp parallel for
    for(int i=-(xn); i<N-(xn);i++)
        for (int j = -(yn); j < N-(yn); j++)
        {
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)] && (i
               !=0 || j!=0))
            {
            esc=1;
            }
        }
    }
    if(esc){return 0;}
    else return 1;
}
```

Therefore I need to make the loop in the `sole_table` function parallel to get more speed up. It should we possible to do it with openmp's cancel directive, but in my previous experiment the pthreads are more efficient in the end due to lower level implementation, so I will do the implementation of this in pthread. The idea is that I should be able to do breaks from threads. Also note that it, we have finite resources, so it is not efficient to try to make all levels for parallel only the upper levels.

Now I want to introduce the idea behind my parallel implementation. A pseudo code of the new implementation can be found below:

```
int solve_table(){
        if(no empty cell){
            table_found()
        }
        if(table_found){
            pthread_exit()
        }

        if(is_multithread())
        {
            call_new_subfunctions_in_new_thred()
        }
        else{
            original_serial_implementation()
        }
}
```

So the basic idea is that if a condition is valid, then I will do the recursive function calls in new threads. This condition in my implementation is based on the currently used threads number, which I store in a global variable and I edit it with `mutex`. And otherwise I just use the original serial implementation in the threads. Note that this implementation gives the possibility that if the threads created at the higher stage of recursion stops and we still couldn't find a solution, then an other thread in a random deeper (in the recursion) state would split up to more thread. This case most likely to happen when I am given an unsolvable problem. The code can be found in the appendix as *pthread_test.c*.

Note that due to the above structure that threads are created a bit randomly due to the randomness in their scheduling and the fact that I am scheduling more thread at the same time in the program as there is in the computer. So the run time of the parallel algorithms are vary a lot. For some case even 100 times changes occur between the running of the same executable with same parameters (e.g: 20 run for *input_3v3*: max: 0.4321, min: 0.0072). But even in the more stable cases a 5 times difference in run time is occurring between the different executions. Therefore it is not really possible to give sufficient run time data for the parallel case.

Also note that because it is not predefined what threads will be created and in which order they are executed. So in some cases the parallel run time becomes

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

(a)

Figure 3: Unsolvable sudoku

longer then the serialized version. Most likely in cases when the solution is found early in the tree of the recursive calls (only leaf where the validation don't fail), which is originally visited with a DFS order. We could avoid this with a more sophisticated implementation, where the number of called thread don't exceed the number of computer threads at any time.
The only time measurement which is stable is an unsolvable sudoku. Therefore I used the following table to time measurement. I also made timings for the previous cases, but as I explained for them the run time vary a lot, so I won't present anything for them here. On the other hand when I used a table which is unsolvable, then in any case I should iterate through all the possibilities, so the timing will become stable for the parallel case. As it is hard to find unsolvable sudokus, so I only present time measurement with one table (Figure 3).

As it is presented in Table 2, the run times are similar as previously for the serialized tests. With the parallel version without optimization flags we can see that it runs 4-5 times faster. And with optimization it is still makes a run time 3 times faster. It is logical that the serial code is better optimized, so it is not a surprise that with optimization we gain a bit less than without. Also this improvement is as expected due to the finite resources of the computer. With this I conclude the parallelization part. As it is seen it can achieve the expected speed up with unsolvable puzzles. And it will achieve a much unstable run time with solvable ones. But overall it can be a good practice to use for higher performance.

| | test case |
|---|---|
| original | 0.5518 |
| final_test | 0.4836 |
| `-O1` | 0.1768 |
| `-O2` | 0.1888 |
| `-O3` | 0.1528 |
| `-Ofast` | 0.1506 |
| `-Ofast -march=native` | 0.1557 |
| parallel no flag | 0.1037 |
| parallel `-Ofast` | 0.0550 |

Table 2: Time measurements

## Summary

Overall in my project I implemented and optimized a given algorithm. Show modification how it can be used to decide if a sudoku has multiple solution. With the help of this I presented two different sudoku generation algorithm.

As the project main goal I optimized the serial code first without compiler optimization, then I investigated what further improvement can be achieved with optimization flags. I went through most of the learned method even if they couldn't be applied in this case and investigated their impact if they were applicable. Finally I introduced a major change to the algorithm which made it possible to use multiple threads. Then I investigated the reason why the run times of the parallel version can vary a lot for my previous test cases and searched for a more stable test case, where I looked at the improvements achieved with parallelization.

As a conclusion I would like to cite a result for an other investigation comparing different solutions for sudoku solver. As a founding in [4], the rule based solver seems to be faster then our implemented algorithms for solving real life sudokus (which has exactly one solution). Though they are harder to scale in many cases. It also found that the Boltzmann machine can solve sudokus much solver then the previously mentioned methods. While in an other article [5] I find that the backtracking algorithm is faster than the rule based. So I would guess that the run time comparison is based on the exact implementations and the given test cases. This article additionally contains measurements for the constraint based solution with integer programming, which it found much solver than the other methods. So overall I would conclude that the backtracking algorithm is quite useful for solving real sudoku problem and one of the fastest algorithm for this task.

## Appendix

### Code: *new_gen.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec /
     1000000;
  return seconds;
}

void create_table(int* table, int NN, int* head, int* idx
   )
// TO DO
{
    int i,j,rnd;
    for(i = 0; i < NN; i++)
        for(j=0; j < NN; j++){
            rnd= rand()%100;
            if(rnd<5)//Not empty
            {table[i*NN+j] = rand() % (NN) + 1;}
            else{ // empty
                table[i*NN+j] = 0;
                head[(*idx+1)]=i*NN+j;
                (*idx)++;}}
}

void print_table(int* table, int NN)
{
    int i,j;
    for(i = 0; i < NN; i++){
           for(j=0; j < NN; j++)
            if(table[i*NN+j]==0){printf("␣&␣");}
            else printf("%d&␣",table[i*NN+j]);
        printf("\n");}
}

void write_table(int* table, int NN){
    FILE *fp = fopen("table.txt", "w");
    if (fp) {
        int i,j;
        for(i = 0; i < NN; i++){
            for(j=0; j < NN; j++)
                if(j!=NN-1)
                fprintf(fp, "%d␣\t",table[i*NN+j]);
                else
                fprintf(fp, "%d",table[i*NN+j]);
            fprintf(fp,"\n");}
        fclose(fp);
    }
}

int validate_table(int* table, int id, int N){
    int NN=N*N;
    int x = id/(NN);
    int y = id%(NN);
    for(int i=0;i<NN;i++){
            if(table[x*NN+y] == table[i*NN+y] && x != i)
            return 0;}

    for(int i=0;i<NN;i++){
            if(table[x*NN+y] == table[x*NN+i] && y != i)
            return 0;
            }

    for(int i=-(x%N); i<N-(x%N);i++){
        for (int j = -(y%N); j  < N-(y%N); j++)
        {
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)] &&
               (i!=0 || j!=0))
            return 0;
        }
    }
    return 1;
}

int start_validation(int* table, int N){
    int NN=N*N;
    for (int i = 0; i < NN; i++)
    {
        for (int j = 0; j < NN; j++)
        {
                for (int k = j; k < NN; k++) // row
                   column
                {
                    if (table[i*NN+j]!=0 && table[i*NN+j
                       ]==table[i*NN+k] && j!=k)
                        return 0;
                        if(table[j*NN+i] !=0 && table[j*NN+i
                            ]==table[k*NN+i] && j!=k)
                             return 0;
                    }
              }
         }
         //block validation
         for (int i = 0; i < N; i++)
         {
              for (int j = 0; j < N; j++)
              {
                   for (int k = 0; k < N; k++)
                   {
                        for (int l = k+1; l < N; l++)
                        {
                             if (table[N*(i*N+k/N)+(j*N+k%N)]!=0
                                 && table[N*(i*N+k/N)+(j*N+k%N)]==
                                 table[N*(i*N+l/N)+(j*N+l%N)])
                             {return 0;}
                        }
                   }
              }
         }
         return 1;
    }


    int nr_sol;

    int generate_table(int* table, int N, int* head, int idx,
         int first){
              if (!first){if(idx==0){if(nr_sol==1){ nr_sol++;
                  return 1;} else{nr_sol++;
                   printf("\n");return 0;}}}
              else{if(idx==0)return 1;}
              int id=head[idx];
              for(int i=1; i<N*N+1;i++){
                   table[id]=i;
                   if (validate_table(table, id, N)){
                        if(generate_table(table, N, head, idx-1,
                            first))
                        {
                             return 1;
                        }
                   }
              }
              table[id]=0;
```

```
        return 0;
}

void shuffle_sub(int* new, int i, int n){
int j = rand() % (n-i) + i+1;
    int tmp = new[i];
    new[i] = new[j];
    new[j] = tmp;
}
void shuffle(int* old, int* new, int n){
    for (int i = 1; i <= n; i++) //copy
    {
        new[i]=old[i];
    }
    for (int i = 1; i < n; ++i)
    {
        srand(time(0));
        for (int j = 0; j < 1; j++)
        {
            shuffle_sub(new, i, n);
        }

    }
}
void reshuffle(int* list, int idx, int n){
    for (int i = idx; i < n; ++i)
    {
        srand(time(0));
        for (int j = 0; j < 2; j++)
        {
            shuffle_sub(list, i, n);
        }

    }
}

void cpy_table(int* t1, int*t2, int NN){
    for (int i = 0; i < NN*NN; i++)
        t2[i]=t1[i];
}

void new_gen(int* table, int N, int* head,  int* p_idx){
    int NN=N*N; nr_sol=0; int idx= (*p_idx);
    while ((! start_validation(table, N)))
       // generate starting table untill pass validation
    {
        idx=0;
        create_table(table,NN,head, &idx);
    }

        //shuffle
        int* head_shuffle = (int*)malloc((idx+1)*sizeof(
           int));
        shuffle(head,head_shuffle,idx);


        int r=generate_table(table, N,head_shuffle, idx,
           1);
        print_table(table,NN);
        if(r){                                  // if
           original has solution
            nr_sol=1; idx=0;
            int* table_cpy=(int*)malloc(NN*NN*sizeof(int)
               );


            for (int k = 0; k < 10; k++){
                while (nr_sol==1)
                    {nr_sol=0; idx++;
                                                        //
                       reset variable values
                        printf("Iteration␣with␣%d␣empty␣
                           cell\n",idx);
                         cpy_table(table,table_cpy,NN);
                         for (int i = 0; i < idx; i++)
                                             // set empty
                             places
                             table_cpy[head_shuffle[i
                                +1]]=0;
                         generate_table(table_cpy, N,
                           head_shuffle, idx,0);

                    }
                nr_sol=1; idx--;
                printf("Reshuffle");
                reshuffle(head_shuffle,idx,(*p_idx));
            }
            free(table_cpy);
            for (int i = 0; i < idx-1; i++)
                                // set empty places
                table[head_shuffle[i+1]]=0;
            printf("\nResult␣containing␣%d␣empty␣cell\n",
               idx);
            print_table(table,NN);
        }
        else
           //if no solution, but not detected with
           start_validation
        {
            create_table(table,NN,head, &idx);
            new_gen(table,N,head,&idx);
        }
        *p_idx=idx;
        free(head_shuffle);
}

void check(int* table, int N, int idx){
    nr_sol=0; int* head=(int*)malloc((idx+1)*sizeof(int))
       ; int cnt=0;
    for (int i = 0; i < N*N; i++)
        for (int j = 0; j < N*N; j++)
            if(table[i*N*N+j]==0){cnt++; head[cnt]=i*N*N+
               j;}
    double start =get_wall_seconds();
        generate_table(table, N,head,cnt,0);
    double end =get_wall_seconds();
    if (nr_sol==1)
        printf("Generation␣succesful\n");
    else
        printf("Generation␣failed!␣%d\n",nr_sol);
    printf("\nThe␣checking␣took:␣%lfs\n", (end-start));
    free(head);
}

int main(int argc, char* argv[]) {
    srand(time(0));

    if(argc != 3) {
        printf("Please␣give␣2␣argument:␣N␣(number␣of␣
           elements␣to␣sort),␣save_table␣(bool).\n");
        return -1;
    }
    int N = atoi(argv[1]);
    int NN=N*N;
        const int savetable = atoi(argv[2]);

    //Create table and table operations
    int* table=(int*)malloc(NN*NN*sizeof(int));
    int* head=(int*)malloc(NN*NN*sizeof(int));
    int idx=0;
    create_table(table,NN, head, &idx);
    if(savetable){write_table(table,NN);}
    printf("\n");


    double s=get_wall_seconds();
        new_gen(table,N,head, &idx);
    double e=get_wall_seconds();
        printf("\nGeneration took %lf\n", (e-s));
    check(table,N, idx);




    free(head);
    free(table);
}
```

Code: ***origin_test.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>


static double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec /
     1000000;
  return seconds;
}

void read_data(int* data, const char* filename, int NN){
    FILE *file;
        file=fopen(filename,"rb");
        fread(data,(2*NN*NN+1)*sizeof(int),1,file)!=0;
        fclose(file);
}

void cpy_table(int* t1, int*t2, int NN){
    for (int i = 0; i < NN*NN; i++)
        t2[i]=t1[i];
}

void print_table(int* table, int NN)
{
    int i,j;
    for(i = 0; i < NN; i++){
        for(j=0; j < NN; j++)
            if(table[i*NN+j]==0){
                printf("␣&␣");}
            else
                printf("%d&␣",table[i*NN+j]);
        printf("\n");}
}

int check(int* table, int N){
    int NN=N*N;
    for (int i = 0; i < NN; i++)
        for (int j = 0; j < NN; j++)
                for (int k = j; k < NN; k++) // row
                   column
                {
                    if (table[i*NN+j]!=0 && table[i*NN+j
                       ]==table[i*NN+k] && j!=k)
                        return 0;
                    if(table[j*NN+i] !=0 && table[j*NN+i
                       ]==table[k*NN+i] && j!=k)
                        return 0;
                }
    //block validation
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                for (int l = k+1; l < N; l++)
                {
                    if (table[N*(i*N+k/N)+(j*N+k%N)]!=0
                       && table[N*(i*N+k/N)+(j*N+k%N)]==
                       table[N*(i*N+l/N)+(j*N+l%N)])
                    {return 0;}
                }
    return 1;
}

int validate_table(int* table, int id, int N){
    int NN=N*N;
    int x = id/(NN);
    int y = id%NN;
    for(int i=0;i<NN;i++){
        if(table[x*NN+y] == table[i*NN+y] && x != i)
            return 0;
        }

    for(int i=0;i<NN;i++){
        if(table[x*NN+y] == table[x*NN+i] && y != i)
            return 0;
        }

    for(int i=-(x%N); i<N-(x%N);i++)
        for (int j = -(y%N); j  < N-(y%N); j++)
        {
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)] &&
               (i!=0 || j!=0))
                return 0;
        }
    return 1;
}

int nr_sol;
int solve_table(int* table, int N, int* head, int idx,
   int first){
        if (!first){
            if(idx==0){
                nr_sol++;
                if(nr_sol==1)
                    return 1;
                else
                    return 0;}}
        else{
            if(idx==0)
                return 1;}

        int id=head[idx-1];
        for(int i=1; i<N*N+1;i++){
            table[id]=i;
            if (validate_table(table, id, N))
                if(solve_table(table, N, head, idx-1,
                   first))
                    return 1;
        }
        table[id]=0;
        return 0;
}


int main(int argc, char* argv[]) {
    // read args
    if(argc != 3) {
        printf("Please␣give␣2␣argument:␣N␣(if␣table␣side␣
           length␣is␣N*N),␣test_file␣(binary␣starting␣
           data␣for␣problem␣size␣N).\n");
        return -1;
    }
    int N = atoi(argv[1]);
    int NN=N*N;
        const char *filename=argv[2];

    // Init data
    int* data=(int*)malloc((2*NN*NN+1)*sizeof(double));
    read_data(data, filename, NN);
    int* table=data;
    int* head=data+NN*NN;
    int idx=head[NN*NN];
    if(!check(table,N)){
        printf("Invalid␣table␣to␣start!");
    }
    else{
```

```
    // Run test
    int* table_cpy=(int*)malloc(NN*NN*sizeof(int));
    double start=get_wall_seconds();
        for (int i = 0; i < 10; i++)
        {

            nr_sol=0;
            cpy_table(table,table_cpy,NN);
            solve_table(table_cpy, N, head, idx, 0);
        }
    double end=get_wall_seconds();


    // Check result
    if (check(table_cpy, N))
    {
        printf("%lf\n",(end-start));
    }
    else{
        print_table(table_cpy,NN);
        printf("Generation␣failed");
        printf("%lf\n",(end-start));
    }
    free(table_cpy);
    }

    free(data);
}
```

Code: *final_test.c*

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <omp.h>

static inline double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec /
     1000000;
  return seconds;
}

static inline void read_data(int* data, const char*
   filename, int NN){
    FILE *file;
        file=fopen(filename,"rb");
        fread(data,(2*NN*NN+1)*sizeof(int),1,file)!=0;
        fclose(file);
}

static inline void cpy_table(int* t1, int*t2, int NN){
    for (int i = 0; i < NN*NN; i++)
        t2[i]=t1[i];
}

static inline void print_table(int* table, int NN)
{
    int i,j;
    for(i = 0; i < NN; i++){
        for(j=0; j < NN; j++)
            if(table[i*NN+j]==0){
                printf("␣&␣");}
            else
                printf("%d&␣",table[i*NN+j]);
        printf("\n");}
}

static inline int check(int* table, int N){
    int NN=N*N;
    for (int i = 0; i < NN; i++)
        for (int j = 0; j < NN; j++)
                for (int k = j; k < NN; k++) // row
                   column
                {
                    if (table[i*NN+j]!=0 && table[i*NN+j
                       ]==table[i*NN+k] && j!=k)
                        return 0;
                    if(table[j*NN+i] !=0 && table[j*NN+i
                       ]==table[k*NN+i] && j!=k)
                        return 0;
                }
    //block validation
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                for (int l = k+1; l < N; l++)
                {
                    if (table[N*(i*N+k/N)+(j*N+k%N)]!=0
                       && table[N*(i*N+k/N)+(j*N+k%N)]==
                       table[N*(i*N+l/N)+(j*N+l%N)])
                    {return 0;}
                }
    return 1;
}

int validate_table(int* table, int id, int N){
    int NN=N*N;
    int x = id/(NN); int y = id%NN;

    // row column
    for(int i=0;i<NN;i++)
        if((table[x*NN+y] == table[i*NN+y] && x != i) ||
           (table[x*NN+y] == table[x*NN+i] && y != i))
            return 0;

    //region
    int xn=x%N; int yn=y%N;
    for(int i=-(xn); i<N-(xn);i++)
        for (int j = -(yn); j  < N-(yn); j++)
        {
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)] &&
               (i!=0 || j!=0))
                return 0;
        }
    return 1;
}

int nr_sol;
int solve_table(int* __restrict table, int N, int*
   __restrict head, int idx, int first){
        if (!first){
            if(idx==0){
                nr_sol++;
                if(nr_sol==1)
                    return 1;
                else
                    return 0;}}
        else{
            if(idx==0)
                return 1;}

        int id=head[idx-1];
        for(int i=1; i<N*N+1;i++){
            table[id]=i;
            if (validate_table(table, id, N))
                if(solve_table(table, N, head, idx-1,
                   first)){
                    return 1;
                }
        }
        table[id]=0;
        return 0;
}


int main(int argc, char* argv[]) {
    // read args
    if(argc != 3) {
        printf("Please␣give␣2␣argument:␣N␣(if␣table␣side␣
           length␣is␣N*N),␣test_file␣(binary␣starting␣
           data␣for␣problem␣size␣N).\n");
        return -1;
    }
    const int N = atoi(argv[1]);
    const int NN=N*N;
        const char *filename=argv[2];

    // Init data
    int* data=(int*)malloc((2*NN*NN+1)*sizeof(double));
    read_data(data, filename, NN);
    int* table=data;
    int* head=data+NN*NN;
    int idx=head[NN*NN];
    if(!check(table,N)){
        printf("Invalid␣table␣to␣start!");
    }
    else{
    // Run test
    int* table_cpy=(int*)malloc(NN*NN*sizeof(int));
    double start=get_wall_seconds();
        for (int i = 0; i < 10; i++)
        {

            nr_sol=0;
            cpy_table(table,table_cpy,NN);
            solve_table(table_cpy, N, head, idx, 0);
        }
    double end=get_wall_seconds();


    // Check result
    if (check(table_cpy, N))
    {
        printf("%lf\n",(end-start));
    }
    else{
        print_table(table_cpy,NN);
        printf("Generation␣failed");
        printf("%lf\n",(end-start));
    }
    free(table_cpy);
    }

    free(data);
}
```

**Code (bash):** ***tim.sh***

```
#!/bin/bash

gcc *_test.c -fopenmp -o test
echo "4x4␣table:"
./test 4 input_4_v1
res1=$(./test 3 input_3_sud_ord_2)
res2=$(./test 3 input_3_sud_ord_3)
res3=$(./test 3 input_3_sud_ord_4)
echo "Sudokus:"
echo "$res1+$res2+$res3" |bc

res4=$(./test 3 input_3_78_6)
res5=$(./test 3 input_3_79_6)
res6=$(./test 3 input_3v1)
res7=$(./test 3 input_3v2)
res8=$(./test 3 input_3v3)
res9=$(./test 3 input_3vv1)
res10=$(./test 3 input_3vv2)
res11=$(./test 3 input_3vv3)
echo "Shuffled␣3"
echo "$res4+$res5+$res6+$res7+$res8+$res9+$res10+$res11"
   |bc
```

Code: ***pthread_test.c***

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>

struct arg_struct
{
    int* table;
    int N;
    int* head;
    int idx;
    int* idxs;
};

pthread_mutex_t lock;
int solve_table(int* __restrict table, int N, int*
   __restrict head, int idx, int * idxs);

static inline double get_wall_seconds() {
  struct timeval tv;
  gettimeofday(&tv, NULL);
  double seconds = tv.tv_sec + (double)tv.tv_usec /
     1000000;
  return seconds;
}

static inline void read_data(int* data, const char*
   filename, int NN){
    FILE *file;
        file=fopen(filename,"rb");
        fread(data,(2*NN*NN+1)*sizeof(int),1,file)!=0;
        fclose(file);
}

static inline void cpy_table(int* t1, int*t2, int NN){
    for (int i = 0; i < NN*NN; i++)
        t2[i]=t1[i];
}

static inline void print_table(int* table, int NN)
{
    int i,j;
    for(i = 0; i < NN; i++){
        for(j=0; j < NN; j++)
            if(table[i*NN+j]==0){
                printf("␣&␣");}
            else
                printf("%d&␣",table[i*NN+j]);
        printf("\n");}
}

static inline int check(int* table, int N){
    int NN=N*N;

    for (int i = 0; i < NN; i++)
        for (int j = 0; j < NN; j++)
                for (int k = j; k < NN; k++) // row
                   column
                {
                    if (table[i*NN+j]!=0 && table[i*NN+j
                       ]==table[i*NN+k] && j!=k)
                        return 0;
                    if(table[j*NN+i] !=0 && table[j*NN+i
                       ]==table[k*NN+i] && j!=k)
                        return 0;
                }
    //block validation
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                for (int l = k+1; l < N; l++)
                {
                    if (table[N*(i*N+k/N)+(j*N+k%N)]!=0
                       && table[N*(i*N+k/N)+(j*N+k%N)]==
                       table[N*(i*N+l/N)+(j*N+l%N)])
                    {return 0;}
                }
    return 1;
}

int validate_table(int* table, int id, int N){
    int NN=N*N;
    int x = id/(NN); int y = id%NN;

    for(int i=0;i<NN;i++) // row col
        if((table[x*NN+y] == table[i*NN+y] && x != i) ||
           (table[x*NN+y] == table[x*NN+i] && y != i))
            return 0;

    int xn=x%N; int yn=y%N;
    for(int i=-(xn); i<N-(xn);i++)
        for (int j = -(yn); j  < N-(yn); j++)
        {
            if(table[x*NN+y] == table[(x+i)*NN+(y+j)] &&
               (i!=0 || j!=0))
                return 0;
        }
    return 1;
}

int solution=0; int nr_thread=0; int * final_table;
void* thread_func(void* arg){
    struct arg_struct *args = (struct arg_struct *)arg;
    solve_table(args->table,args->N,args->head,(args->idx
       )-1, (args->idxs));

    pthread_mutex_lock (&lock);
    nr_thread--;
    pthread_mutex_unlock (&lock);
    pthread_exit(NULL);
}


int solve_table(int* __restrict table, int N, int*
    __restrict head, int idx, int* idxs){
        if(idx==0){
            final_table=(int*)malloc(N*N*N*N*sizeof(int))
               ;
            cpy_table(table,final_table,N*N);
            solution=1;
            return 1;
        }
        if(solution==1){
            pthread_exit(NULL);
        }

        int id=head[idx-1];

        if(nr_thread<8)
        {
            pthread_t threads[N*N];
            int cnt=0;
            int * table_cpy[N*N];
            struct arg_struct args[N*N];
            for(int i=1; i<N*N+1;i++){
                table[id]=i;
                if (validate_table(table, id, N) &&
                    solution==0){

                     table_cpy[cnt] = (int*)malloc(N*N*N*N
                        *sizeof(int));
                     cpy_table(table,table_cpy[cnt], N*N);
                     args[cnt].table=table_cpy[cnt]; args[
                        cnt].N=N; args[cnt].head=head;
                        args[cnt].idx=idx; args[cnt].idxs=
                        idxs;
                     pthread_create(&threads[cnt], NULL,
                        thread_func, (void *)&args[cnt]);
                     cnt++;
                     pthread_mutex_lock (&lock);
                         nr_thread++;
                     pthread_mutex_unlock (&lock);
                 }

             }
             for(int t=0 ; t<cnt; t++){
                 pthread_join(threads[t],NULL);
                 free(table_cpy[t]);}

             if(solution==1)
                 return 1;
             else{
                 table[id]=0;
                 return 0;
             }
         }
         else{ //serial
             for(int i=1; i<N*N+1;i++){
                 table[id]=i;
                 if (validate_table(table, id, N))
                     solve_table(table, N, head, idx-1,
                        idxs);
                     if(solution==1){
                         return 1;
                     }
             }
             if(solution==1)
                 return 1;
             table[id]=0;
             return 0;
         }
}
```

```c
int main(int argc, char* argv[]) {
    pthread_mutex_init(&lock,NULL);
    // read args
    if(argc != 3) {
        printf("Please␣give␣2␣argument:␣N␣(if␣table␣side␣
           length␣is␣N*N),␣test_file␣(binary␣starting␣
           data␣for␣problem␣size␣N).\n");
        return -1;
    }
    const int N = atoi(argv[1]);
    const int NN=N*N;
        const char *filename=argv[2];

    // Init data
    int* data=(int*)malloc((2*NN*NN+1)*sizeof(double));
    read_data(data, filename, NN);
    int* table=data;
    int* head=data+NN*NN;
    int idx=head[NN*NN];
    if(!check(table,N)){
        printf("Invalid␣table␣to␣start!");
    }
    else{
    // Run test
    int* idxs=(int*)malloc(idx*sizeof(int));
    double start=get_wall_seconds();
            solve_table(table, N, head, idx, idxs);
            if(solution==1)
                table=final_table;
    double end=get_wall_seconds();


    // Check result
    if (check(table, N))
    {
        printf("%lf\n",(end-start));
    }
    else{
        //print_table(table,NN);
        printf("Generation␣failed");
        printf("%lf\n",(end-start));
    }
    }
    pthread_mutex_destroy(&lock);
    free(data);
}
```

## 1 References

[1] Andrew C Bartlett, Timothy P Chartier, Amy Langville, Timothy D Rankin , An integer programming model for the sudoku problem. , *Researchgate*, (2008), https://www.researchgate.net/publication/228615106_An_integer_programming_model_for_the_sudoku_problem

[2] Fiorella Grados, Aref Mohammadi, A REPORT ON THE SUDOKU SOLVER. *Bachelor's Essay at dept. Computer Science at Royal Institute of Technology (KTH)*, (2013) https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand13/Group1Vahid/report/Aref-Fiorella-KexJobb-sist.pdf

[3] Jad Matta, Brute Force Approach Algorithm for Sudoku Brute Force Approach Algorithm for Sudoku Brute Force Approach Algorithm for Sudoku , *Researchgate*, DOI:10.13140/RG.2.2.36113.17761 https://www.researchgate.net/publication/ 337935582_Brute_Force_Approach_Algorithm_for_Sudok
_Algorithm_for_Sudoku_Brute_Force_Approach_Algorithm_for_Sudoku

[4] PATRIK BERGGREN, DAVID NILSSON, A study of Sudoku solving algorithms *,Bachelor's Essay at dept. Computer Science at Royal Institute of Technology (KTH)*, https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand12/Group6Alexander/final/Patrik_Berggren_David_Nilsson.report.pdf

[5] Martin Kondor, The Cure Of Cancer & Sudoku Solver Algorithms, https://medium.com/intuition/the-cure-of-cancer-sudoku-solver-algorithms-cac95da70121