High Performance Programming (course 1TD062)
Uppsala University – Spring 2024
Sudoku Solver

Oskar Nylin

4th April 2024

1 Sudoku

Sudoku is a game based on logic, where the player is supposed to fill in all blank spaces on a partially filled board. Classic Sudoku consists of a 9x9 grid, where at least 17 cells are filled and only one solution exist (otherwise it is not considered legitimate). A number (1-N) can only exist in one place, for each row, column and box. [3]

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

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

[2]

Figure 1: Example of how a Sudoku looks before/after it is solved

By solving hundreds of Sudoku's on my own, have I developed a few strategies on how a board can be solved. Four of which have I successfully implemented into this solver. My main focus, when solving a Sudoku, is to eliminate possibilities. Possibilities are all numbers that can be assigned to a cell without breaking any rules.

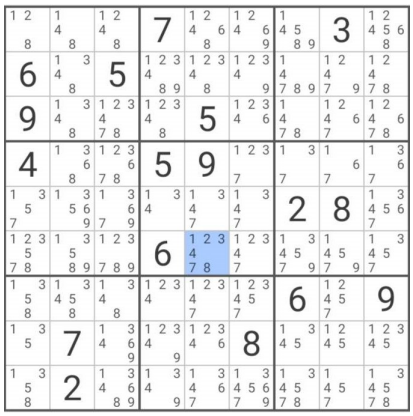


Figure 2: Initial board with all possibilities

The first, and easiest, strategy to eliminate possibilities is of course to find all cells where only one possibility exists. Such an occasion is very rare on a initial board, but can occur more frequently once a few cells have been filled.

Second strategy is quite similar to the first. Instead of searching for cells where only one number can exist, I search for boxes where one number can only exist in one of the boxes cells.

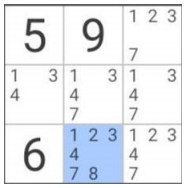


Figure 3: Example of box where number 8 can only exist in one cell

The third strategy is Pairs. A pair is two numbers that can only be assigned in the same two cells, thus eliminating all other possibilities that theoretically could be assigned to the same two cells.

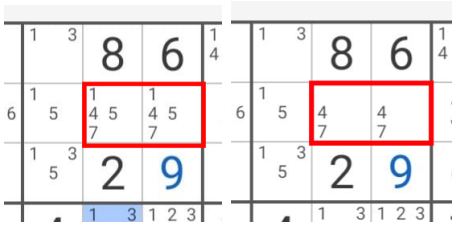


Figure 4: Possibilities in a box, before/after a pair is found

Now from the previous image can we also see that two lines are formed, one vertical and one horizontal, that eliminates all possibilities in the same direction.

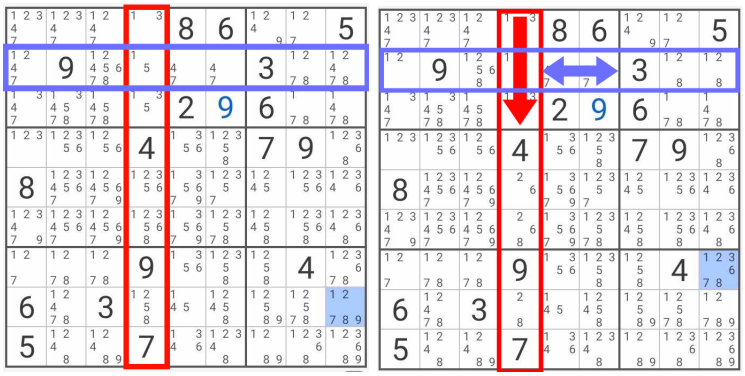


Figure 5: Possibilities in a row and column after two lines are found

The following chapters will explain how I have implemented these strategies into my Sudoku solving algorithm, in order to decrease solution time, that can solve Sudoku's of various sizes.

2 Data structures and Initial values

```
#define DEMO 0
int N;
int sqrt_N;
```

- `DEMO` is a boolean that determines if the board should be printed for demonstration. Zero means *off*, for performance.
- `N` is a global int that determines the board size, set by user at start. This has to be global because the board will be on the stack, making all function calls look like `int board[N][N]`, thus forcing `N` to be outside the function scope.
- `sqrt_N` is the square root of N, set globally as it is used by several functions.

```
typedef struct unAssignInd{
    int x;
    int y;
} unAssigned_t;
```

`unAssigned_t` contains the x / y coordinate of a cell that has not yet been assigned.

```
typedef struct pair{
    int x1;
    int x2;
    int y1;
    int y2;
    int val;
} pair_t;
```

`pair_t` contains the x / y coordinates of two cells that CAN assign the same value.

```
typedef struct line{
    int val;
    int x_coord;
    int y_coord;
    int dir;
} line_t;
```

The `line_t` struct contains a value which forms a line of possibilities in a box. The x / y coordinates are the starting point of the line and `dir` is the direction in which the line is going, whether it is vertical or horizontal.

3 Implementation and Optimization

All performance tests have been made in WSL(Ubuntu) on a AMD Ryzen 3 7320U with Radeon Graphics CPU. Compiled with `-O3 -march=native`

- `-O3` This flag enables aggressive optimizations. It includes optimizations like loop unrolling, function inlining, and aggressive floating-point optimizations.
- `-march=native` This flag instructs the compiler to generate code specific to the architecture of the machine on which it is being run. It utilizes the instruction set extensions and features available on the processor, potentially improving performance by tailoring the generated code to the specific hardware.

The following functions are written in the order they were implemented and tested.

3.1 Naive Solution

The strategies, explained in chapter one, are only logical optimizations to increase solution efficiency. The initial implementation is a backtracking brute force algorithm.

```
bool Solve(int board[N][N], unAssigned_t **unAssignInd, int N_unAssign){
    if(N_unAssign == 0){
        return true;
    }

    int x = unAssignInd[N_unAssign-1]->x;
    int y = unAssignInd[N_unAssign-1]->y;

    for(int val = 1; val < (N+1); val++){
        board[y][x] = val; // Set guess
        if(ValidateBoard(board, y, x)){
            bool solution;
            solution = Solve(board, unAssignInd, N_unAssign - 1);
            if(solution){
                return true;
    }   }   }
    board[y][x] = 0; // No solution, reset value in backtracking
    return false;
}
```

This is the backtracking brute force algorithm, which is a recursive function. This function will start from the bottom right corner of the Sudoku board and work it's way up to the top left corner. For each cell will all numbers ($1 \rightarrow N$) be tried. If a number is valid, will `Solve` jump to the next cell. If all numbers in a cell failed to find a solution, backtrack (return to previous cell).

While it's difficult to provide an exact worst case time complexity with backtracking due to the variable nature of the algorithm's behavior depending on the puzzle. This function will (theoretically) have a worst case of $O(N^M)$, where N = Number of Values and M = Number of Unassigned cells. With backtracking will the worst case time complexity be significantly lower, but harder to predict. I found out during benchmark testing that the solution time could vary

from 0.8 seconds all the way to 160 seconds, on a standard Sudoku with 17 filled cells. As mentioned before, it all comes down to luck.

This of course leads me to my next point, performance. As this function has a time variance of almost three minutes to find a solution, I felt that the best way to see if I was improving performance was to measure the average solution time of several puzzles. Therefore all improvements have been tested to solve 18 standard Sudoku's with 17 filled cells, generated by an external part. [1] This gave an average solution time of 15.0893 seconds.

3.2 Possibilities

Since the naive implementation tries all numbers for each cell, will a lot of unnecessary checks be done. By introducing an array of possibilities for each unassigned cell, can the number of checks be significantly reduced.

The possibilities are set by testing all numbers, one cell at a time, and validating the board. Example of possibilities for cell zero: `possibilities[y][x] = [0, 2, 3, 0, 0, 0, 7, 0, 9]`, zero means the value cannot be assigned. This is done before the brute force algorithm is started, and a check is inserted inside `Solve`.

```
for(int i = 0; i < N; i++){
    if(possibilities[y][x][i] == '0') continue;
    board[y][x] = i+1; // Set guess
    ...
}
```

The algorithm will now check if the value was possible on the initial board before assigning it, and then does a new validation check on the current board. This saves almost three seconds, giving an average solution time of 12.1995 seconds.

3.3 Parallelized Brute Force

Checking if a number is valid is an independent parallel task, but two tasks can not work on the same board at the same time. When the solution is found, no more tasks should be issued and previous tasks cancelled. New tasks has to be small enough so that threads can handle them and increase performance, but big enough to avoid memory-allocation and task-creation overhead. (Project Description)

The brute force algorithm is parallelized with OpenMP in order to make it work with these constraints.

```
int main(){
    ...
    omp_set_num_threads(N_threads);
    #pragma omp parallel
    {
        #pragma omp master
        {
            Solve(board, unAssignInd, N_unAssign, &solution, 0, possibilities);
            if(!solution){
                printf("NO SOLUTION\n");
            }
        }
    }
    ...
}
```

In `main`: `Solve` is now called by a thread master, meaning only one of the threads can execute this block.

```
bool Solve(char board[N][N], unAssigned_t **unAssignInd, int N_unAssign, bool *
    solutionFound, int depth, char poss[N][N][N]){
    if(N_unAssign == 0){
        return true;
    }
    int tmp = N_unAssign - 1;
    int x = unAssignInd[tmp]->x;
    int y = unAssignInd[tmp]->y;

    #pragma omp taskgroup
    {
        for(int i = 0; i < N; i++){
            if(*solutionFound) break;
            if(poss[y][x][i] == '0') continue;

            board[y][x] = (i+1) + '0'; // Set guess
            if(ValidateBoard(board, y, x)){
                bool local_solution = false;
                #pragma omp task firstprivate(board, N_unAssign) final(depth > 1)
                {
                    char board_copy[N][N];
                    copy_board(board, board_copy);
                    board_copy[y][x] = (i+1) + '0';
                    local_solution = Solve(board_copy, unAssignInd, tmp,
    solutionFound, depth+1, poss);
                    if(local_solution){
                        if(check_entire_board(board_copy)){
                            printf("SOLUTION\n");
                        }else{
                            printf("NO SOLUTION\n");
                        }
                        *solutionFound = true; // Workaround to stop all threads
                    }
                }
            }
        }
        #pragma omp taskwait
    }
    board[y][x] = '0'; // No solution, reset value in backtracking
    return false;
}
}
```

The parallelized version executes in the following order:

1. After checking base-case and de-referencing variables, a taskgroup is created. The taskgroup ensures that the one task cannot continue before all child-tasks are completed.

2. Check if a solution is found. That is if the shared boolean has been set to *TRUE*, if so stop the thread. Closely resembling a spinning lock as it will be checked very frequently.

3. After validating the insertion of a new number upon the board, a new task is created. The new task will receive a private copy of the boards current state and a private copy of the number of unassigned indices. The task will then be picked up by a thread and tried for a solution.

Two important notes here. Firstly notice the `final(depth > 1)` variable. This determines the depth of which new tasks should be allowed to be created. I.e. It will prevent child-tasks at a certain depth to create new threads. This value needs to be adjusted depending on how

many cores/threads that is being used and the size of the Sudoku puzzle. In general will `final(depth > 1)` give best performance.

Secondly note that the final return will always be *FALSE*. Because if one thread finds the solution, it cannot return, and if all threads were to return, then you would get a lot of trash data I.e. unsolved boards. Therefore if one thread finds the solution then it has to be printed immediately.

3.4 Possibility Elimination Functions

This part will explain how the strategies explained in chapter one are implemented. The following functions are all made with the same purpose, eliminate possibilities. All functions (except `the_lucky_one`) have a similar structure but vary in execution. In pseudo code:

```
int count = 0;
int boxSize = sqrt_N;
set_other_initial_variables();

for(int box = 0; box < N; box++){
    int start_x = (box % boxSize) * boxSize;
    int start_y = (box / boxSize) * boxSize;

    do_function_specific_operations(box, start_x, start_y);
    count += count_updates();
}
return count;
```

All functions:

- Sets initial variables
- Iterates through one box at a time and performs function specific operations. `start_x`, `start_y` are the coordinates to the top-left cell of each box.
- Returns the count. This can be the number of values that was inserted to the board, or the number of pairs/lines that was found.

3.4.1 The Lucky One

This function is named `the_lucky_one` as you should consider yourself lucky if it actually finds a value to set on the board. As mentioned before this function differs from the other elimination function, but it's implementation is just as simple as the strategy, described in chapter one. `the_lucky_one` will not regard the boxes, as the others do, it will only check all unassigned cells if they have a single possibility, if so insert it to the board.

This function does not give much improvement by it self as it only gives a slight improvement of 0.2 seconds, and puts the average solution time at 5.43527 seconds. But it still has its value which can bee seen in chapter 3.4.5 Early Solution.

3.4.2 The Easy One

This function is so named because it's easy for a human to see if a number can only be assigned to one cell in the box. It gets a bit more tricky for a computer since it got no eyes.

```
bool changed = true;
while(changed){
    changed = false;
    ...
```

```
    int local_changed = 0;
    for(int val = 1; val < (N+1); val++){
        local_changed = the_easy_one_helper(board, poss, boxSize, start_y, start_x,
     val);
        if(local_changed){
            count++;
            changed = true;
    }   }
   update_possibilities(board, poss);
}
return count;
```

the_easy_one will keep checking the entire board until it can no longer insert a value. The function has to manually try all numbers in all boxes and check if only one of the cells can set the given value (done by the helper function). Count all numbers that could be inserted and update the array of possibilities according to the new state of the board.

A lot of "easy" values will get filled by this function which would take the brute-force ages to find, and thus cutting the execution time in half. Putting the average execution time at 2.38004 seconds.

3.4.3 Pairs

```
for(int val = 1; val < (N+1); val++){
    pairs_helper(poss, boxSize, start_y, start_x, val, pairs[val-1]);
}

for(int i = 0; i < N; i++){
    if(pairs[i]->val == 0) continue;
    for(int j = i+1; j < N; j++){ // j = i+1 to avoid duplicates
        if(pairs[j]->x1 == pairs[i]->x1
        && pairs[j]->x2 == pairs[i]->x2
        && pairs[j]->y1 == pairs[i]->y1
        && pairs[j]->y2 == pairs[i]->y2){
            for(int k = 0; k < N; k++){
                if(k != pairs[i]->val-1 && k != pairs[j]->val-1){
                    poss[pairs[i]->y1][pairs[i]->x1][k] = '0';
                    poss[pairs[i]->y2][pairs[i]->x2][k] = '0';
                    count++;
    }   }   }   }
    pairs[i]->val = 0; // Reset pair
}
```

This function will first get all values that can only be placed in two cells, and the cells coordinates, using the helper function. It will then do a check between all values that can only be placed in two cells if they have the same coordinates. If two values have the same possibility coordinates, a pair is found and all other possibilities with the same coordinates as the pair is removed (replaced with 0).

Even doe pairs are quite common in Sudoku puzzles, it will still not eliminate that many possibilities as one might hope. Less eliminations means less time is cut, putting the average execution time at 1.93348 seconds.

3.4.4 Lines

```
for(int val = 1; val < (N+1); val++){
    line->val = 0; // Reset line
    get_line(poss, boxSize, start_y, start_x, val, line);
```

```
    if(line->val != 0){
        eliminate_line(poss, line, start_x, start_y);
        count++;
    }
}
```

This function is very easy to grasp, very complex to implement (see source code). For each value, it checks if a line is formed. It also has to determine if the line is horizontal or vertical. If a line exist eliminate the possibility from all cells outside the box that are in the same row or column (dependent on the direction of the line).

Lines are a very good tool for eliminating possibilities, since a line will affect an entire row or column on the board. `Lines` will further reduce the average execution time down to 0.288 seconds.

3.4.5 Early Solution

In `main`:

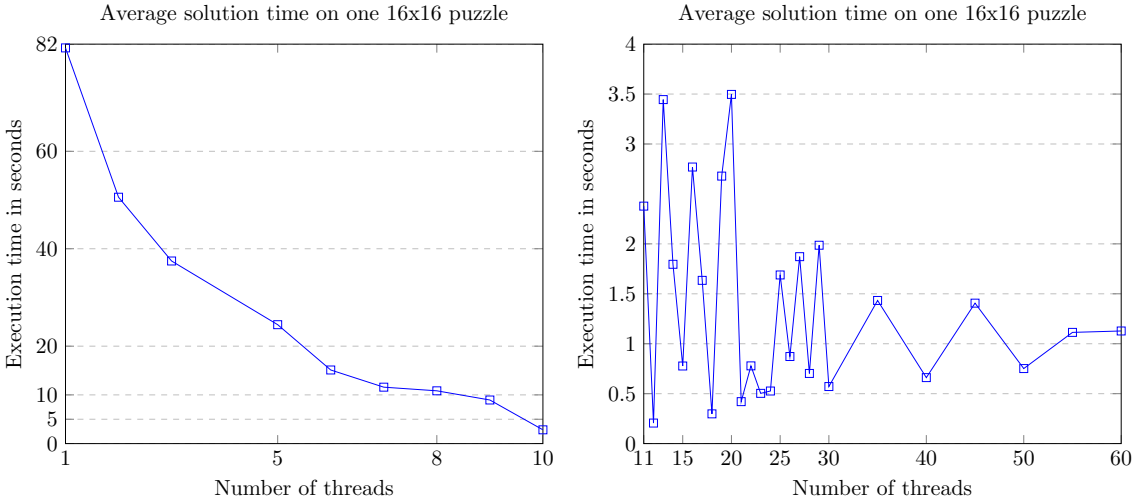
```
int lucky = 1, easy = 1, pair = 1, previous_pair = 0, line = 1, previous_line = 0;
while(lucky != 0 || easy != 0 || line != previous_line || pair != previous_pair){
    // Loop until no changes
    lucky = the_lucky_one(board, possibilities);
    easy = the_easy_one(board, possibilities);
    previous_pair = pair;
    pair = pairs(possibilities);
    previous_line = line;
    line = lines(possibilities);
}
```

Since the possibility elimination functions affect one another are they put into a while loop that keeps iterating until no changes are made. We know if a change was made by checking the `count` returned by each function. `the_lucky_one` and `the_easy_one` will return how many numbers they set on the board (zero if none where set). `pairs` and `lines` will return the number of pairs and lines that exists on the board, therefore its value has to be compared with the previous iteration to see if it stayed the same.

When these functions were combined into a while loop, a phenomenon which I call "Early Solution" occurred. This phenomenon happens when a puzzle is solved solely by the possibility elimination functions, and the brute-force is never used. This significantly reduces the average execution time, which is now 0.0278 seconds. This makes the Sudoku solver more then 500 times faster then the naive solution.

4 Testing

One 16x16 Sudoku, hand picked by me, have been used to demonstrate how the number of threads affects the execution time of the brute force algorithm. Ran on Intel(R) Xeon(R) CPU E5520 @2.27GHz Model 26, 8 cores.



Interestingly the "golden rule of threading" (one thread per core) does not apply here. As we can see the execution time is significantly reduced when the number of threads is increased from 1 to 10, but best performance is not achieved at 8 threads. Instead we get very "violent" swings in execution time when the number of threads > 11, where 12 or 18 threads (depending on server traffic) gives best performance. Both having an solving time of 0.2-0.3 seconds on average. Time starts panning out when the number of threads > 30, slowly increasing in execution time as thread creation overhead increases.

A significant amount of other tests were performed as well in order to validate the solver. But I still came to the same conclusion no matter what other strategies, input, or hardware I tried.

Everything depends on the puzzle.

Should an *EARLY SOLUTION* be found then the solution time is close to constant. An early solution on a 9x9 puzzle can be found in 0.0015 seconds and on a 25x25 in 0.006 seconds, where the first have 64 unassigned cells and the second have between 280-290 unassigned cells. Note that this time is achieved without threads, as only the brute force is threaded.

However if the brute force is used, then the order of which the unassigned cells are written heavily affects solution time. The brute force will start in the bottom right corner and work its way up, one row at a time. In a worst-case scenario would all unassigned cells be in consecutive, this of course never happens, but just by having a few unassigned cells in the bottom right corner can drastically change the execution time.

5 Discussion

Following up on the statement given in chapter four that "everything depends on the puzzle". I want to comment on my last and most serious attempt to bust this "myth". In this final attempt I tried to make the logical solution part of the brute force, as a sort of hybrid solver. The main premise was that the brute-force would set a number and then try to solve the puzzle with logic,

with slight variation between attempts. One variation would actually fill the board, but the likelihood for it to be correct were the same as to win a casino jackpot. Another variation had so much overhead that one might as well use the naive implementation. The main issues with all variations were backtracking and correctly updating possibilities.

One could also look into optimization strategies such as loop unrolling and reducing heavy instructions. I have not done so, since the number of calculations are close to none, and the intention of making functions as "light" as possible have been prioritized through out the development of this program.

Another idea I didn't try, because I can't really see how it would work, is to have the brute force solver jump from the bottom-right corner to the top-left. I.e. every other number would be set starting from the opposite corner. This can potentially make the solver discover wrong numbers earlier then the current solver.

I had to come to the conclusion in the end that my Sudoku solver can, most likely, never have a stable execution time that scales with the size of the board, rather then the layout of the input. Or at least, I cannot achieve it.

References

[1] Generate and solve sudoku. https://kjell.haxx.se/sudoku/.

[2] How to win sudoku. https://towardsdatascience.com/how-to-win-sudoku-3a82d05a57d. Original publisherhttps://blogs.unimelb.edu.au/sciencecommunication/files/2016/10/sudoku-p14bi6.png.

[3] Sudoku. https://sv.wikipedia.org/wiki/Sudoku.