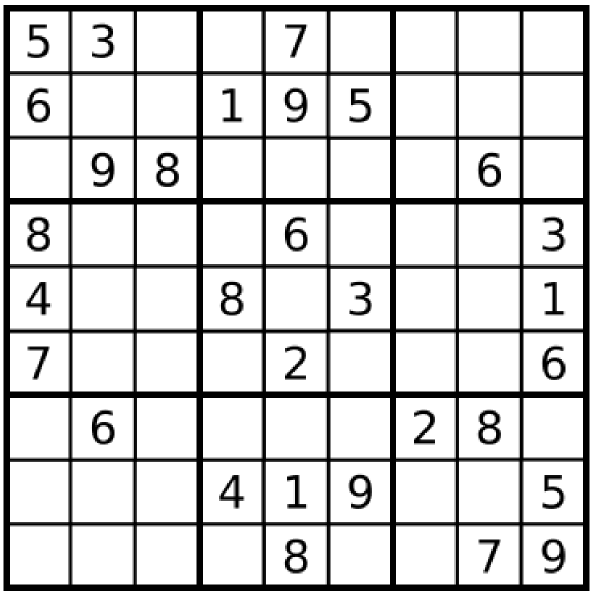
Contents

1. Introduction

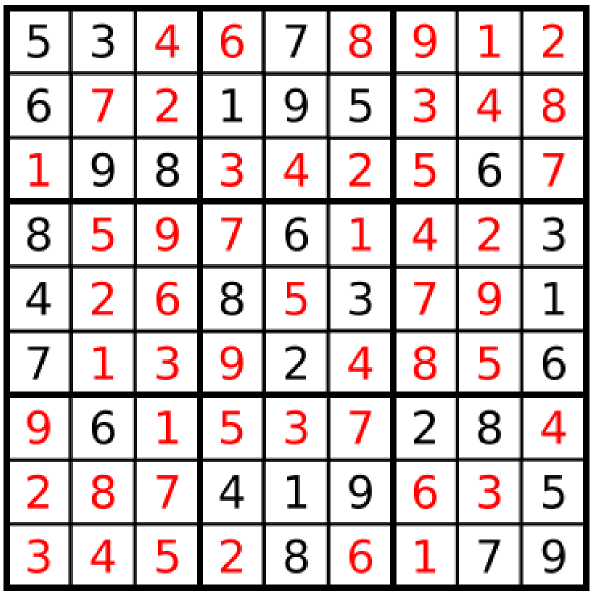
Sudoku is a popular logic-based puzzle game published by the Japanese puzzle company Nikoli in 1986. It involves filling a 9x9 grid of cells with numbers. The grid is divided into nine 3x3 sub-grids, and the objective is to fill each empty cell in the grid with a number from 1 to 9. The rules for Sudoku are simple:

- Each row, each column, and each sub-grid must contain all the numbers from 1 to 9 without repetition.

A sudoku puzzle starts with some of the cells already filled, and the solver has to fill the other empty cells. The puzzle is solved when all the empty cells are filled, and the rules of Sudoku are satisfied, as shown in Fig. 1.



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

Fig. 1 Sudoku puzzle: The initial state on the left, and the goal state on the right.

Based on the number of initial clues and their placements on the grid, a Sudoku puzzle may have a unique solution or multiple valid solutions. The minimum number of clues required for a Sudoku puzzle to have a unique solution is 17. If less than 17 clues are given, multiple valid solutions are expected.

The Sudoku puzzles can range from relatively easy to very challenging, with the difficulty level depending on:

- The number of initially provided clues,
- The logic required to solve the puzzle.

As a rule of thumb, puzzles with fewer initial clues tend to be more challenging to solve, as they require more complex reasoning and deduction to fill in the missing numbers. However, if the number of clues is very low (less than 17), the existence of multiple valid solutions may make it easier to find a valid solution.

Most of the time, it is difficult and time-consuming for humans to solve Sudoku puzzles. If someone wants to pass his time then it is okay to try solving a Sudoku puzzle by himself. Otherwise, if someone wants a fast solution, he should consider writing a program on a computer to solve Sudoku puzzles. A program that solves Sudoku puzzles should:

1. Find a valid solution (basic requirement),
2. Be efficient (desired requirement).

The general approach of solving Sudoku puzzles is using the backtracking technique. It involves systematically exploring possible solutions and backtracking when a conflict or dead-end is encountered. The basic idea is to fill in the empty cells one by one, ensuring that the numbers entered are valid according to the Sudoku rules.

In this work, we have implemented and tested two methods to solve Sudoku puzzles: DFS brute force with backtracking (with and without heuristic), and Knuth's Algorithm X. The goal is to find which method is the most efficient one to solve Sudoku puzzles of different difficulty levels.

2. DFS Sudoku Solver

2.1 Algorithm and implementation

The easiest method to solve a Sudoku puzzle is using brute force search and backtracking. In this case, the used search strategy is Depth-First Search. DFS starts with an arbitrary root node and explores as far as possible along each branch before backtracking and exploring other options. To solve Sudoku puzzles, the algorithm follows these steps:

1. Choose an empty cell in the Sudoku grid (from left to right, from up to down).
2. Try each number from 1 to 9 in that cell.
3. If the chosen number is valid (satisfies Sudoku rules), move to the next empty cell and repeat steps 1-3 recursively.
4. If no valid number is found for a cell, backtrack to the previous cell (decision node) and try a different number.
5. Repeat steps 1-4 until the entire Sudoku is filled.

The implementation is given below:

```
def solve():
  global grid
  # 1) Find an empty cell in the Sudoku grid.
  for row in range(9):
    for col in range(9):
      if grid[row][col] == 0:

        # 2) Try each number from 1 to 9 in that cell.
        for num in range(1,10):

          # 3) If the chosen number is valid, insert it into this empty
          #    cell and work with the other empty cells.
          if allowed(row, col, num):
            grid[row][col] = num
            solve()
            #If the made decision led to a dead-end then reset it.
            grid[row][col] = 0

        # 4) If no valid number is found for a cell, dead-end happened.
        # Backtrack to the previous call(decision node)
        # and try a different number.
        return

  # 5) If there isn't any empty cell then you have a solution.
  print_grid()

  # By backtracking you can find other solutions if they exist.
  # It will try other numbers if allowed.
  user_input = input("'Enter' for other solutions.\n(Type 'Stop' to
exit): ")
  if user_input.lower() == "stop":
      raise StopExecution()
```

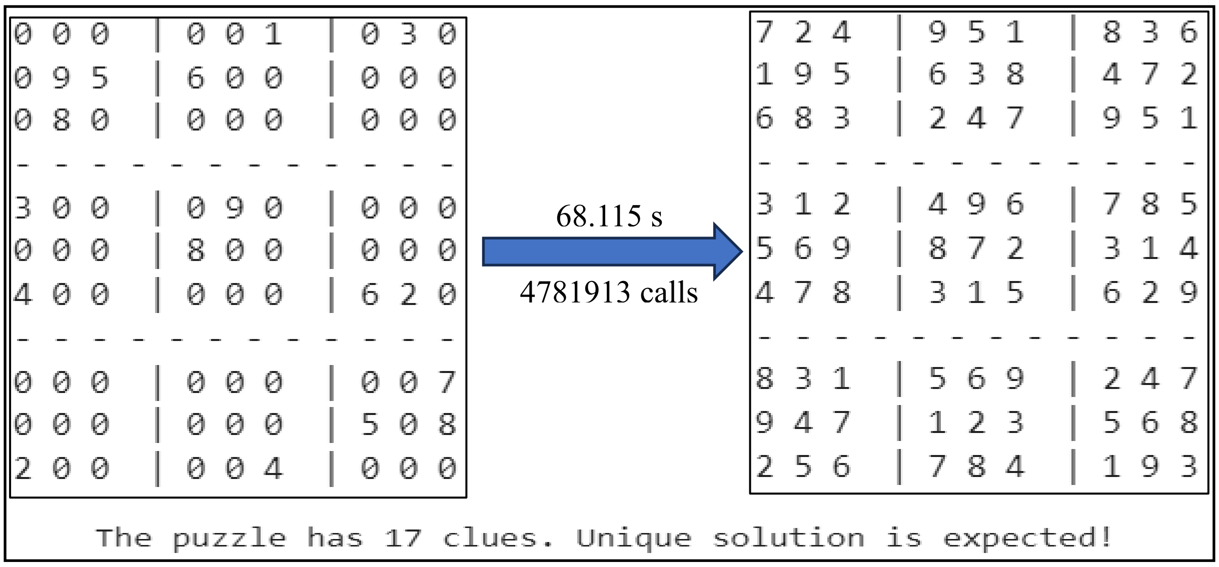
```
# allowed() checks if a number can be inserted into a cell
# without violating the Sudoku rules.

def allowed(p_row,p_col,num):
  global grid
```

```
# 1) Check for row conflict
for col in range(0,9):
  if grid[p_row][col] == num :
    return False

# 2) Check for column conflict
for row in range(0,9):
  if grid[row][p_col] == num :
    return False

# 3) Check for sub-grid conflict
# (row_0,col_0) starting position of the sub-grid
row_0 = (p_row // 3) * 3
col_0 = (p_col // 3) * 3
for row in range(0,3):
  for col in range(0,3):
    if grid[row_0 + row][col_0 + col] == num :
      return False

# If there is not any conflict, return True
return True
```

2.2 Tests and evaluation

The implemented Sudoku solver was tested to solve puzzles of different levels of difficulty: Starting with an easy one (63 clues), then increasing the difficulty (41, 30, 27, 26, 22 clues) up to the most difficult one (17 clues). For these puzzles unique solutions are expected because they have more than 16 clues. Fig. 2 shows results for a puzzle with 17 clues, and Fig. 3 shows results for a puzzle with 22 clues.

Moreover, a test with a puzzle with 16 clues was conducted. The aim was to prove the ability of the implemented Sudoku solver to find multiple valid solutions when they exist. Fig. 4 shows the results.

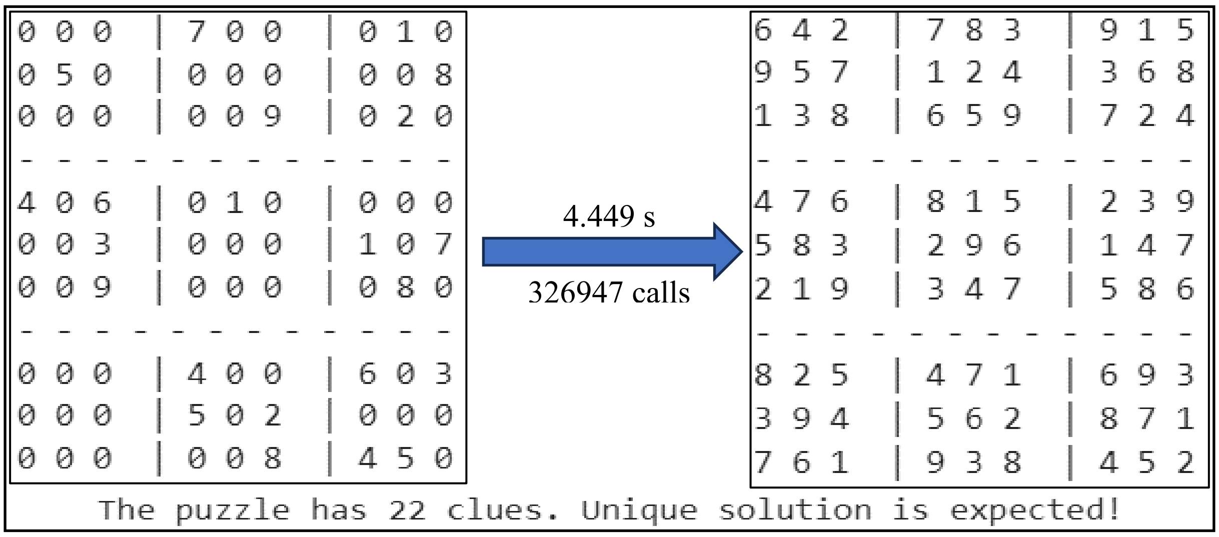


```
0 0 0 | 0 0 1 | 0 3 0
0 9 5 | 6 0 0 | 0 0 0
0 8 0 | 0 0 0 | 0 0 0
- - - - - - - - - - - - -
3 0 0 | 0 9 0 | 0 0 0
0 0 0 | 8 0 0 | 0 0 0
4 0 0 | 0 0 0 | 6 2 0
- - - - - - - - - - - - -
0 0 0 | 0 0 0 | 0 0 7
0 0 0 | 0 0 0 | 5 0 8
2 0 0 | 0 0 4 | 0 0 0
```

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

```
The puzzle has 17 clues. Unique solution is expected!
```

Fig. 2 Results for a puzzle with 17 clues.

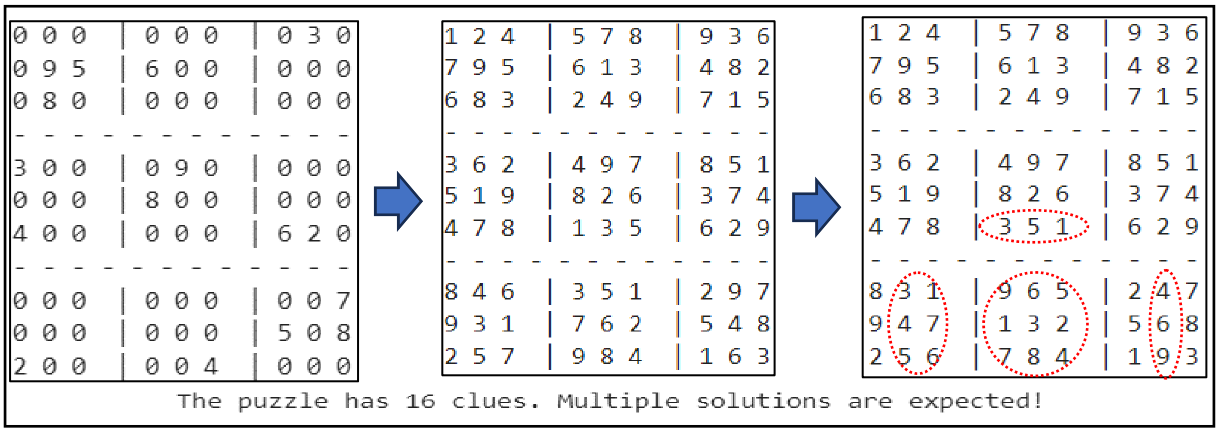


```
0 0 0 | 7 0 0 | 0 1 0
0 5 0 | 0 0 0 | 0 0 8
0 0 0 | 0 0 9 | 0 2 0
- - - - - - - - - - - - -
4 0 6 | 0 1 0 | 0 0 0
0 0 3 | 0 0 0 | 1 0 7
0 0 9 | 0 0 0 | 0 8 0
- - - - - - - - - - - - -
0 0 0 | 4 0 0 | 6 0 3
0 0 0 | 5 0 2 | 0 0 0
0 0 0 | 0 0 8 | 4 5 0
```

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

```
The puzzle has 22 clues. Unique solution is expected!
```

Fig. 3 Results for a puzzle with 22 clues.



```
0 0 0 | 0 0 0 | 0 3 0
0 9 5 | 6 0 0 | 0 0 0
0 8 0 | 0 0 0 | 0 0 0
- - - - - - - - - - - - -
3 0 0 | 0 9 0 | 0 0 0
0 0 0 | 8 0 0 | 0 0 0
4 0 0 | 0 0 0 | 6 2 0
- - - - - - - - - - - - -
0 0 0 | 0 0 0 | 0 0 7
0 0 0 | 0 0 0 | 5 0 8
2 0 0 | 0 0 4 | 0 0 0
```

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

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

```
The puzzle has 16 clues. Multiple solutions are expected!
```

Fig. 4 Results for a puzzle with 16 clues.

From Fig. 4 we can easily observe the effect of backtracking. The idea is that to find another solution the implemented algorithm backtracks to only a few previous calls / previous decision nodes (maybe one or two) and takes another decision there resulting in another solution.

The relationship between the numbers of clues and the time required to solve each puzzle is plotted in Fig. 5.

Number of Clues	Solving Time (s)
17	68.1147
22	4.4486
26	0.0531
27	0.9692
30	2.3366
41	0.1066
63	0.0014

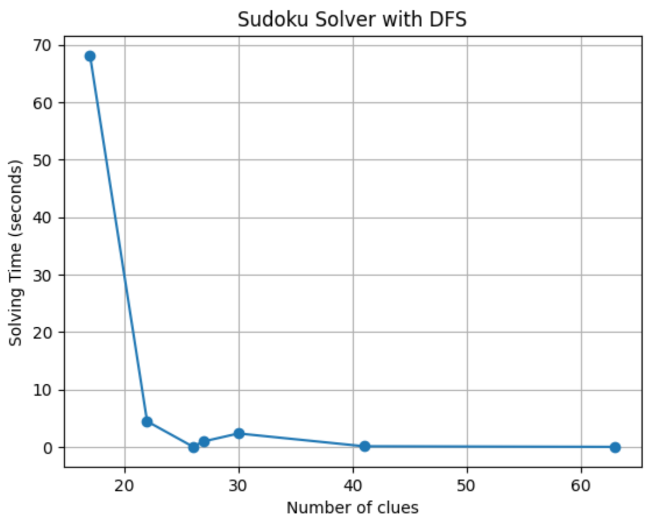


Fig. 5 Solving time for each puzzle (DFS).

From Fig. 5 we can see that the implemented solver performs very well for relatively easy puzzles (more than 40 clues) with a solving time near 0 s, but when it comes to difficult puzzles (with fewer clues) the solving time is high indicating that the implemented solver is not efficient.

The evaluation of the Sudoku solver with DFS:

- **Completeness**: **Yes**, it can find a solution.
- **Optimality**: **No**. Solution found first may not be the shortest possible.
- **Time complexity**: $O(b^d)$, where b is maximum branching factor, and d is the depth of solution.

2.3 Smart DFS Sudoku Solver

When evaluating the DFS Sudoku Solver we found out that it is not efficient when solving difficult puzzles, therefore optimizations are required. The problem with DFS Sudoku Solver is the way it chooses empty cells (from left to right and from up to bottom). This is not a clever way to choose empty cells. Here, to improve the DFS Sudoku Solver we present the heuristic:

- Choose the most constrained empty cell. It is the cell with the fewest allowed numbers on it.

So instead of starting with a cell that has 4 options, start with a cell that has 2 or 1 option. In principle, this heuristic should greatly prune the search tree, resulting in a more efficient solver.

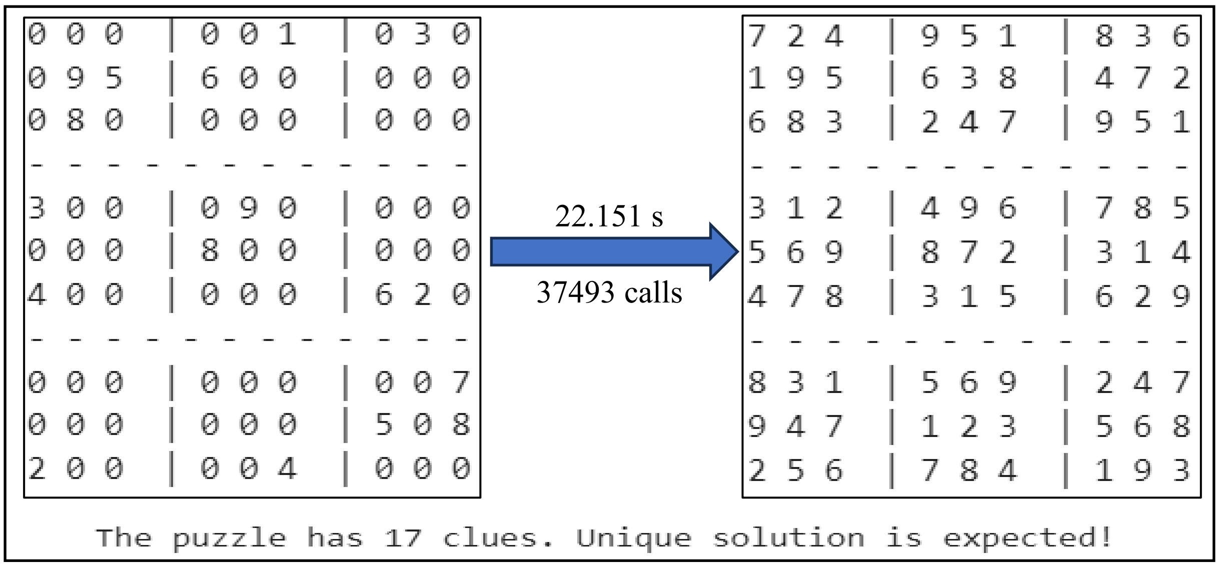
2.3.1 Algorithm and implementation

The algorithm and implementation are almost the same as the DSF Sudoku Solver. The only difference here is the way of choosing the empty cells. The heuristic of the most constrained empty cell is used here. It is implemented using the ***find_candidate()*** function, as shown below:

```python
# This function finds the most constrained empty cell.
# Process: 1) For each empty cell create a list containing
#             the cell coordinates and the allowed numbers in it.
#          2) Return the shortest list (best candidate)
def find_candidate():
  global grid

  list_of_candidates = []
  for row in range(9):
    for col in range(9):
      candidate = []

      # find a candidate
      if grid[row][col] == 0:
        # add its coordinates
        candidate +=[row,col]
        # add its allowed numbers
        for num in range(1,10):
          if allowed(row,col,num):
            candidate.append(num)
        # add candidate to a comparison list
        list_of_candidates.append(candidate)

  # Handle the situation when the grid is filled
  if not list_of_candidates:
    return 0

  # Find the best candidate
  min = list_of_candidates[0]
  for i in range(1, len(list_of_candidates)):
    if len(list_of_candidates[i]) < len(min):
      min = list_of_candidates[i]
  return min
```

2.3.2 Tests and evaluation

In this section, the same tests as in DFS Sudoku Solver are performed. The results are shown below:



```
0 0 0 | 0 0 1 | 0 3 0
0 9 5 | 6 0 0 | 0 0 0
0 8 0 | 0 0 0 | 0 0 0
- - - - - - - - - - - -
3 0 0 | 0 9 0 | 0 0 0
0 0 0 | 8 0 0 | 0 0 0
4 0 0 | 0 0 0 | 6 2 0
- - - - - - - - - - - -
0 0 0 | 0 0 0 | 0 0 7
0 0 0 | 0 0 0 | 5 0 8
2 0 0 | 0 0 4 | 0 0 0
```

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

```
The puzzle has 17 clues. Unique solution is expected!
```

Fig. 6 Results for a puzzle with 17 clues (Smart DFS).

Number of Clues	DFS Time (s)	Smart DFS Time (s)
17	68.1147	22.1509
22	4.4486	0.1705
26	0.0531	0.0244
27	0.9692	0.3107
30	2.3366	1.1206
41	0.1066	0.0469
63	0.0014	0.0023

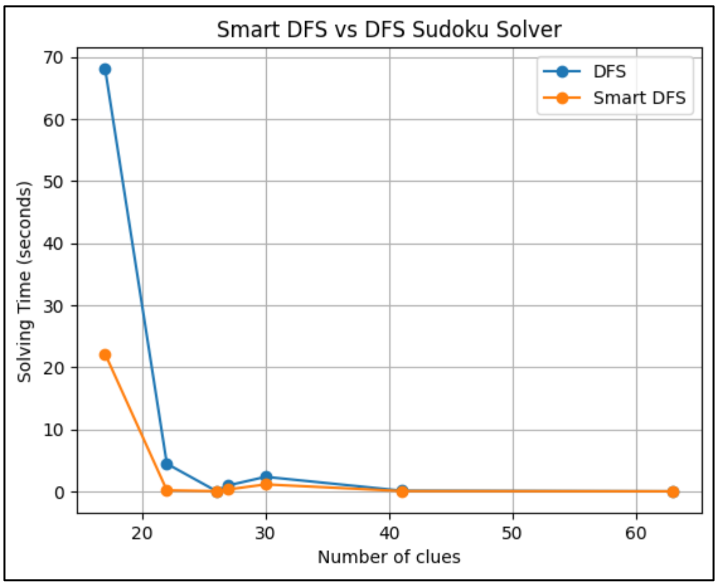


Fig. 7 Solving time for each puzzle (DFS vs Smart DFS)

From Fig. 7, we can see that the Smart DFS Sudoku Solver is much more efficient than the previous solver. For relatively easy puzzles (more than 40 clues), there is no difference in performance between two solvers, but when it comes to difficult puzzles (with fewer clues) the difference is large. For example, to solve the most difficult puzzle (with 17 clues), the solving time is reduced 3 times (from 68 to 22s), and the number of recursive calls (made decisions) is reduced 127 times (from 4.7 million to 37000 calls), which are significant improvements. The proposed heuristic works.

The evaluation of the Smart DFS Sudoku Solver:

- **Completeness**: **Yes**, it can find a solution.
- **Optimality**: **No**. Solution found first may not be the shortest possible.
- **Time complexity**: $O(b^d)$, exponential in the depth of the solution *d*.

3. Sudoku Solver using Algorithm X

Apart from the brute force approach, there is another approach to build a Sudoku solver. In this approach, Sudoku problem is represented as an exact cover problem, and the Algorithm X invented by Donald Knuth is used to solve it.

3.1 Algorithm X and its implementation

Algorithm X is a straightforward recursive, nondeterministic, depth-first, backtracking algorithm. In Algorithm X, the exact cover problem is represented by a matrix A consisting of 0s and 1s. The goal is to select a subset of the rows such that the digit 1 appears in each column exactly once. Algorithm X follows these steps:

```
0. If A is empty, the problem is solved; terminate successfully.
1. Otherwise choose a column, $c$ (deterministically).
2. Choose a row, $r$, such that $A[r,c] = 1$ (nondeterministically).
3. Include $r$ in the partial solution.
4. For each $j$ such that $A[r,j] = 1$,
       delete column $j$ from matrix $A$;
       for each $i$ such that $A[i,j] = 1$,
           delete row $i$ from matrix $A$.
5. Repeat this algorithm recursively on the reduced matrix $A$.
```

If column c is entirely zero, the algorithm has reached a dead-end; terminate unsuccessfully and backtrack to the previous call. The suggested heuristic in choosing column *c* is to choose the column with the smallest number of 1s in it.

In general, to implement Algorithm X, the Dancing Links technique is used. It represents the matrix of the exact cover problem using doubly-linked circular lists. However, here the matrix of the problem is represented using Python dictionaries: one dictionary for the given set X and one dictionary for the collection Y of subsets of X.

The implementation of Algorithm X is shown below:

```
def solve(X, Y, solution):
```

```
# 0) Matrix has no columns, partial solution is valid; terminate
successfully.
    if not X:
        yield list(solution)

    # 1) Otherwise choose a column c.
    else:
        # use heuristic (column with the fewest 1s)
        c = min(X, key=lambda c: len(X[c]))

        # 2) Choose a row r such that A[r][c] = 1.
        for r in list(X[c]):

            # 3) Include row r in the partial solution.
            solution.append(r)

            # 4) Reduce matrix A.
            cols = select(X, Y, r)

        # 5) Repeat this algorithm recursively on the reduced matrix A.
            for s in solve(X, Y, solution):
                yield s

            # Reverse changes made by "select(X, Y, r)",
            deselect(X, Y, r, cols)
            solution.pop()
```

3.2 Sudoku Solver algorithm and implementation

To build a Sudoku solver using Algorithm X we should follow these steps:

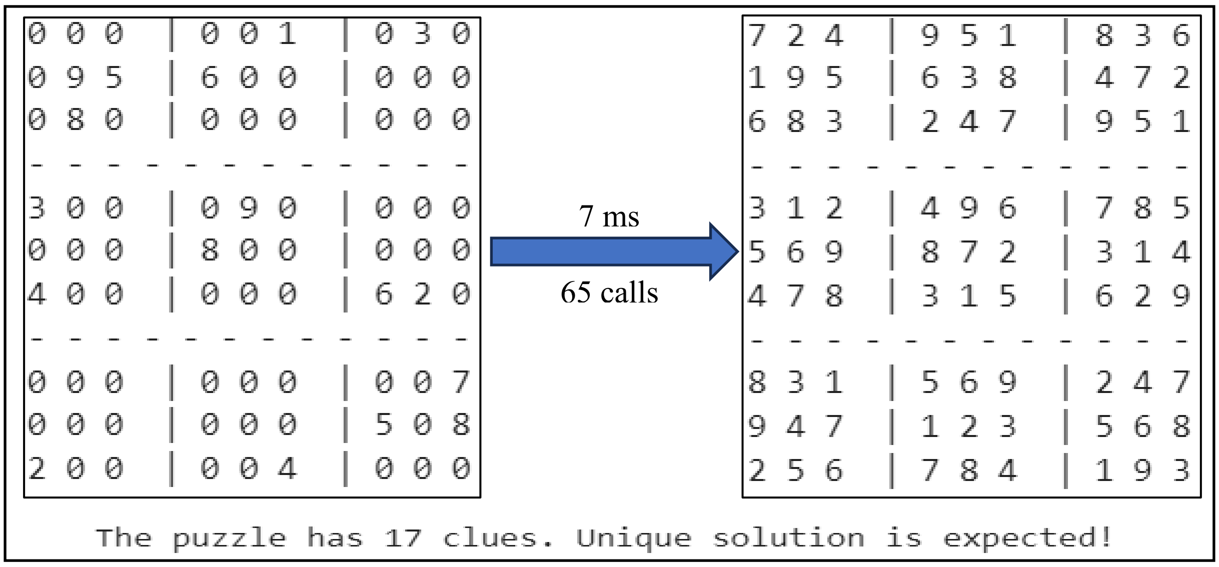
1. Represent Sudoku problem as an exact cover problem, by constructing 'Constraint Dictionary' and 'Mapping Dictionary' (used to represent the matrix of the problem).
2. Remove constraints for given clues.
3. Use Algorithm X to satisfy the constraints of each empty cell, and then remove constraints of each cell from 'Constraint Dictionary'.
4. When 'Constraint Dictionary' is empty, a solution is found.

The implementation of the Sudoku solver is shown below:

```python
def solve_sudoku(size, grid):

    R, C = size
    N = R * C
#----------------------------------------------------------------------
  # 1. Constract 'Constraint Dictionary' and 'Mapping Dictionary'

  # Create a list representing all constraints in the Sudoku problem.
  # Each constraint is a tuple of two elements:
  #   1. The type of constraint (string)
  #   2. The corresponding indices or values involved in the constraint
    X = ([("rc", rc) for rc in product(range(N), range(N))] +
         [("rn", rn) for rn in product(range(N), range(1, N + 1))] +
         [("cn", cn) for cn in product(range(N), range(1, N + 1))] +
         [("bn", bn) for bn in product(range(N), range(1, N + 1))])

  # Y (Mapping Dictionary)---> Mapping cells to constraints:
  #              For each combination row, column(cell) and number,
  #              Y contains its corresponding list of constraints.
  #              Key: The combination (r,c,n) as tuple
  #              Value: List of constraints
    Y = dict()
    for r, c, n in product(range(N), range(N), range(1, N + 1)):
        b = (r // R) * R + (c // C) # Box number
        Y[(r, c, n)] = [
            ("rc", (r, c)),
            ("rn", (r, n)),
            ("cn", (c, n)),
            ("bn", (b, n))]

  # Represent the matrix of the excat cover problem
  # X---> Constraint Dictionary
  # Y---> Mapping Dictionary
    X, Y = exact_cover(X, Y)
#----------------------------------------------------------------------
  # 2. Removing constraints for given clues.
    for i, row in enumerate(grid):
        for j, n in enumerate(row):
            if n:
                select(X, Y, (i, j, n))
#-----------------------------------------------------------------------
  # 3,4. Perform Algorithm X until you have a solution
    for solution in solve(X,Y,[]) :
        for (r, c, n) in solution:
            grid[r][c] = n
        yield grid
```

3.3 Tests and evaluation

In this section, the same tests as in the previous cases are performed. The results are shown below:



```
0 0 0  | 0 0 1  | 0 3 0
0 9 5  | 6 0 0  | 0 0 0
0 8 0  | 0 0 0  | 0 0 0
- - - - - - - - - - - -
3 0 0  | 0 9 0  | 0 0 0
0 0 0  | 8 0 0  | 0 0 0
4 0 0  | 0 0 0  | 6 2 0
- - - - - - - - - - - -
0 0 0  | 0 0 0  | 0 0 7
0 0 0  | 0 0 0  | 5 0 8
2 0 0  | 0 0 4  | 0 0 0
```

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

```
The puzzle has 17 clues. Unique solution is expected!
```

Fig. 8 Results for a puzzle with 17 clues (Using Algorithm X).

Table 1 Solving time of each puzzle (DFS vs Smart DFS vs Algorithm X)

Number of Clues	DFS Time (s)	Smart DFS Time (s)	Algorithm X Time (s)
17	68.1147	22.1509	0.007
22	4.4486	0.1705	0.0077
26	0.0531	0.0244	0.0044
27	0.9692	0.3107	0.0157
30	2.3366	1.1206	0.006
41	0.1066	0.0469	0.0085
63	0.0014	0.0023	0.0033

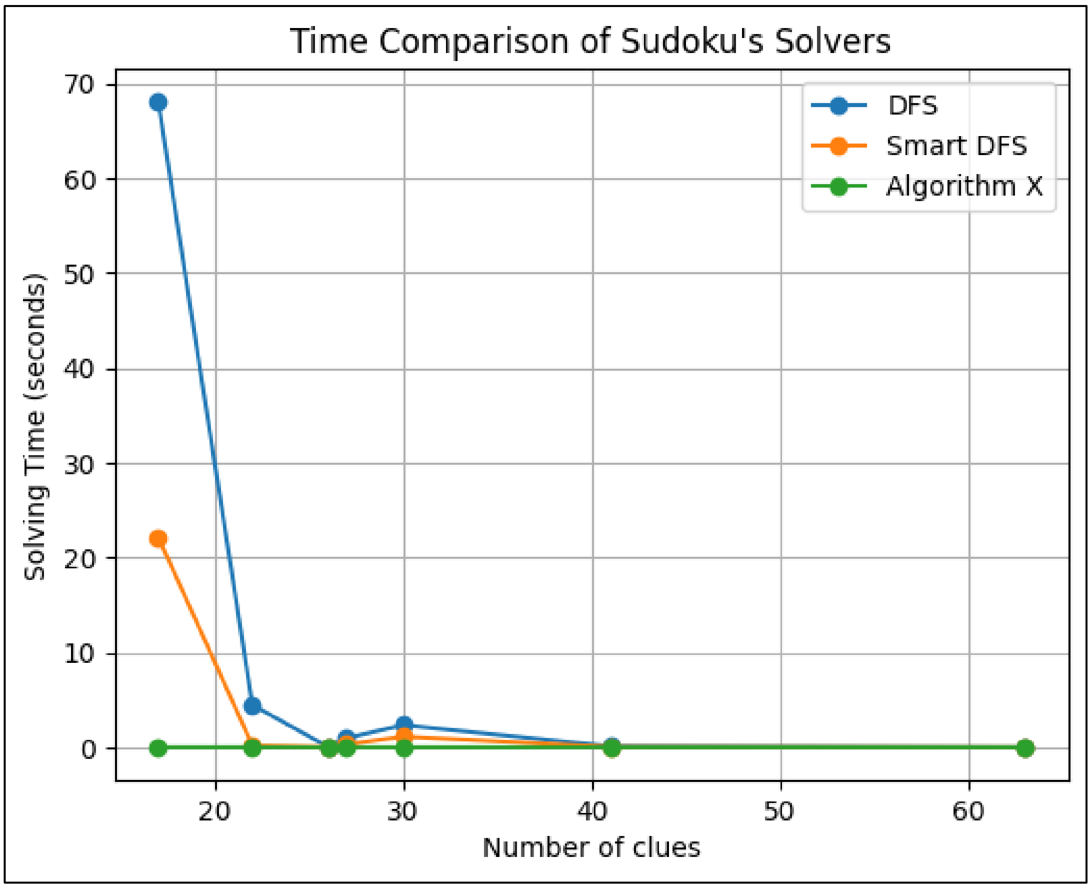


Fig. 9 Time comparison of different Sudoku's solvers.

From Fig. 9, we can easily notice the superiority of the Sudoku solver using Algorithm X. We can see that this solver can solve puzzles of different difficulty levels in milliseconds. This shows that the Sudoku solver using Algorithm X is independent from the puzzle's difficulty, therefore it is the most efficient Sudoku solver.

The evaluation of the Sudoku Solver using Algorithm X:

- **Completeness**: **Yes**, it can find a solution.
- **Optimality**: **No**. Solution found first may not be the shortest possible.
- **Time complexity**: Theoretically $O(4^{d/5})$, but practically near O(1).

4. Discussion and Conclusion

To better understand and compare different Sudoku solvers, a summary of the properties of each Sudoku solver is given in Table 2.

Table 2 Summary of the properties of each Sudoku solver

Sudoku Solver	Completeness	Optimality	Time complexity	Efficiency	Implementation difficulty
DFS	Yes	No	$O(b^d)$	Low	Low
Smart DFS	Yes	No	$O(b^d)$	Medium	Low
Algorithm X	Yes	No	$O(4^{d/5})$	High	High

From the conducted tests and evaluations, we can conclude that:

- The approach of brute force search is very easy to understand and implement, but it is not efficient in solving difficult puzzles.
- By presenting heuristics we can improve the efficiency of the brute force approach, while keeping its implementation difficulty low.
- The approach of presenting the Sudoku problem as an exact cover problem and solving it using Knuth's Algorithm X is the most efficient approach. However, the implementation difficulty of it is high.
- There exists a trade-off between the efficiency and implementation difficulty of each Sudoku solver. If you care about efficiency, use Algorithm X approach, and if you care about implementation difficulty, use brute force approach with heuristics.

5. References

[1] Knuth, D. E. (2000). Dancing Links. Millennial Perspectives in Computer Science, 187-214.

[2] https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X

[3] https://en.wikipedia.org/wiki/Exact_cover

[4] https://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html

[5] https://11011110.github.io/blog/2008/01/10/analyzing-algorithm-x.html

[6] https://en.wikipedia.org/wiki/Sudoku

[7] https://uchida.cis.k.hosei.ac.jp/