Sudoku as a Constraint Problem

Suyash Kushwaha
CPSC 450
Fall 2024

Summary

This project focused on algorithms that can be used to solve Sudokus. Specifically, 9x9 Sudokus were used to test and evaluate the algorithms. The naive algorithm used was backtracking, and the more sophisticated algorithm solves Sudokus by looking at it as a constraint problem (referred to as the constraint algorithm in this report). The algorithms were implemented in Typescript. For performance tests, two different datasets from Kaggle were used. The constraint algorithm performs significantly faster with hard Sudoku problems, however it fails at solving extremely hard Sudokus, which backtracking excels at.

1. ALGORITHM SELECTED

The problem of Sudoku can be viewed as three different problems: solving the rows, columns, and the blocks.

- The constraint algorithm takes advantage of this and encodes the interaction between rows, columns, and blocks to continuously reduce the search space of each cell, until nothing can be reduced.
- Backtracking tries every possible value to solve the Sudoku.

To implement the constraint algorithm, my primary source was [1] and [2] helped with parsing out what the [1] was trying to say. For the backtracking algorithm, I did not reference any external resources, as its implementation was straightforward.

Constraint Algorithm Pseudocode:

```
calculate the search space


do while search space can't be reduced anymore
   reduce the search space using row-block interaction
   reduce the search space using column-block interaction
   reduce the search space using row-column interaction
   reduce the search space using block interaction
   reduce the search space using rows-blocks interaction
   reduce the search space using columns-blocks interaction
```

The search space is reduced using a combination of maximum matching algorithm and Berge's theorem (SCCs needed to be calculated to employ this theorem).

For an NxN Sudoku, if the problem is hard, the search space will make a dense graph, and on easier problems, it will be a sparse graph. The major cost is in reducing the search tree, which has a complexity of $O(n^2)$ for sparse graphs and $O(n^3)$ for dense graphs.

The reduction needs to happen multiple times for every block, row, and column. So for:

1. Hard problems: $O(n^4)$
2. Easy/medium problems: $O(n^3)$

However this is with an assumption that the search space isn't reduced at all, which isn't realistically the case, so runtime complexity is much better than $O(n^4)$ for hard problems.

Note: Hard problems here refers to a large search space, i.e. more missing values in the Sudoku grid.

Backtracking Algorithm Pseudocode:

```
if Sudoku has no missing values
 return true
row = floor(index / 9)
column = index % 9


if grid[row][column] != 0
 for i = 1 to 9
   if i can be placed in grid[row][column]
     grid[row][column] = i


     if bruteForceSolve(index + 1)
       return true
     grid[row][column] = 0
else
 bruteForceSolve(index + 1)
```

The worst case for the backtracking algorithm is an empty Sudoku grid. In an empty NxN Sudoku, we have to try N values in a NxN grid. And for every value we check, we need to check if the value is legal, which has a complexity of O(N). So the complexity ends up being $O((N+N)^{N*N}) = O((2N)^{N*N})$.

However, since we check if a move is legal or not, we can backtrack early most of the time, which reduces the complexity significantly. It is tricky to say what the complexity would be in this case since it depends greatly on the specific problem and when backtracking will happen. Although it's most likely possible to find this complexity, it's outside the scope of this project due to its non-trivial nature.

2. IMPLEMENTATION

The algorithms were implemented in Typescript and no external libraries were used. Since [1] didn't include any pseudocode for the algorithm, the pseudocode in Section 1 is based on my implementation. It was really hard to parse what [1] was trying to say, so the tricky part with the implementation was to figure out what the algorithm would even look like. [2] helped me quite a bit with this.

Implementation of backtracking algorithm

The implementation of the backtracking algorithm is fairly trivial and uses recursion for backtracking.

Implementation of constraint algorithm

To implement the constraint algorithm, first the search space is calculated, which is represented using a JS object, which is basically a map. Then the search space is reduced using multiple ways, as outlined in the pseudocode. The actual reduction happens by creating a new graph that represents the maximum matching problem. After we get back the maximum matches, the SCCs are calculated to find the cycles, and then these cycles are removed from the graph. The remaining edges are then removed from the search space.

The implementation to calculate the strongly connected components was taken from HW6, which uses a set to store the linearization and a map to store the SCCs. To calculate maximum matching, two arrays were used to store the different "colors" of the bipartite graphs and the matching is returned as a map. At first maximum matching was implemented using Ford–Fulkerson algorithm, however Hopcroft-Karp was able to considerably outperform it, so I ended up using that.

Unit tests

Unit tests were written to test multiple components of the constraint algorithm:

- **Helper methods:** methods such as `getRow`, `getColumn`, etc that are used throughout the algorithm. For example, for the grid (0 represents empty elements):

  ```
  0 0 3 0
  3 4 0 1
  4 3 1 2
  2 1 4 3
  ```

  `getRow(0)` should return [0, 0, 3, 0]

- **Graph Algorithms:** The strongly connected components algorithm and the maximum flow algorithm were tested using multiple unit tests. To test SCCs, the tests from HW6 were used. For the maximum flow algorithm, I had to write new tests. For example, consider the following graph:

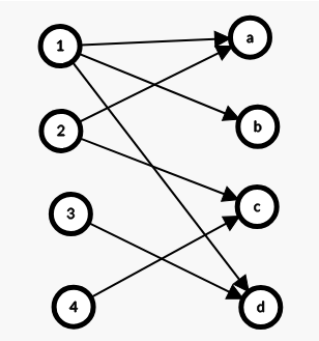

Fig 1. Graph used to test maximum matching

The maximum matching for the graph in Fig 1. would be the graph in Fig 2.

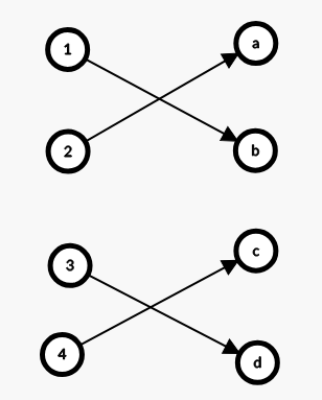

Fig 2. Maximum matching of graph in Fig 1.

- **Core components:** Core components such as setting a value in the Sudoku grid, reducing the search space, and calculating the search space were tested using unit tests. For example, consider the following grid:

  ```
  0 0 0 0
  0 0 0 0
  0 0 0 0
  0 0 0 0
  ```

  If we set index (0,0) to 1, then 1 should be removed from search spaces of (0, 1), (0, 2), (0, 3), (1, 0), (2, 0), (3, 0) and (1, 1).

Other than these tests, hard Sudokus from The New York Times [3] and Euler Project were used [4]. They both tested the combinations of different interactions as outlined in the pseudocode in Section 1.

3. PERFORMANCE TESTS

For performance tests, two datasets from Kaggle were used [5] [6]. [5] had harder Sudoku problems, and [6] had much easier problems. 500,000 Sudokus from each of these datasets were solved. All of the Sudokus were 9x9 in size, and the number of missing values (i.e. difficulty) was used to increase the size of the "input". Fig 3. shows the system used to run the performance tests.

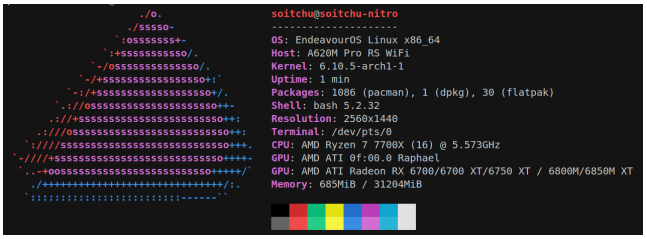

Fig 3. Output of neofetch on the system used to run the performance tests.

To run the performance tests, the CLI of the program can be used. Specifically, for the following results, I ran these commands:

```
# Easy dataset
node --experimental-strip-types index.ts --dataset kaggle-9m --limit 500000 --strategy constraint
node --experimental-strip-types index.ts --dataset kaggle-9m --limit 500000 --strategy backtracking
node --experimental-strip-types index.ts --dataset kaggle-9m --limit 500000 --strategy hybrid

# Hard dataset
node --experimental-strip-types index.ts --dataset kaggle-3m --limit 500000 --strategy constraint
node --experimental-strip-types index.ts --dataset kaggle-3m --limit 500000 --strategy backtracking
node --experimental-strip-types index.ts --dataset kaggle-3m --limit 500000 --strategy hybrid
```

After running the commands above, 6 files will be generated in the `output` directory. These JSON files include an array of a tuple. The first element of the tuple is the time taken, in milliseconds, to solve the Sudoku. If it's negative, then it means it couldn't be solved. The second element of the tuple is how many missing values were there in the sudoku grid.

Table 1. Easy dataset solved using the constraint algorithm

No. of missing values	Avg. Time to Solve per Sudoku in µs	Percentage of Sudokus solved
20 - 24	597.28	100.00%
25 - 29	601.60	100.00%
30 - 34	614.09	100.00%
35 - 39	630.95	99.95%

Table 2. Hard dataset solved using the constraint algorithm

No. of missing values	Avg. Time to Solve per Sudoku in µs	Percentage of Sudokus solved
52 - 53	987.43	33.19%
54 - 55	1061.70	49.57%
56 - 57	1110.01	61.52%
58 - 59	1197.22	69.62%
60 - 61	1285.80	66.13%

Table 3. Easy dataset solved using the backtracking algorithm

No. of missing values	Avg. Time to Solve per Sudoku in µs	Percentage of Sudokus solved
20 - 24	31.27	100.00%
25 - 29	35.60	100.00%
30 - 34	40.44	100.00%
35 - 39	48.74	100.00%

Table 4. Hard dataset solved using the backtracking algorithm

No. of missing values	Avg. Time to Solve per Sudoku in µs	Percentage of Sudokus solved
52 - 53	1467.90	100.00%
54 - 55	3602.11	100.00%
56 - 57	9137.62	100.00%
58 - 59	31976.40	100.00%
60 - 61	114661.27	100.00%

4. EVALUATION RESULTS

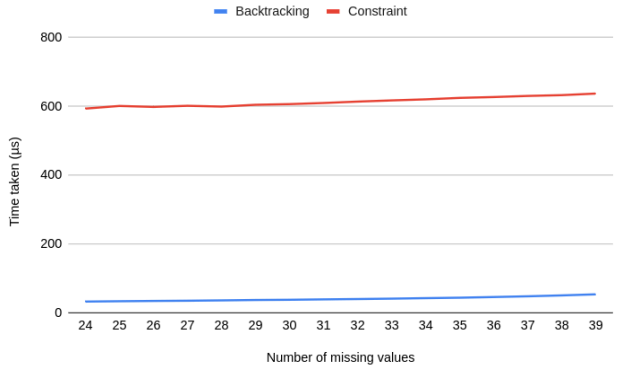

Fig 4. Plot of Table 1 and 3, i.e. how backtracking and constraint algorithms perform with the easy dataset

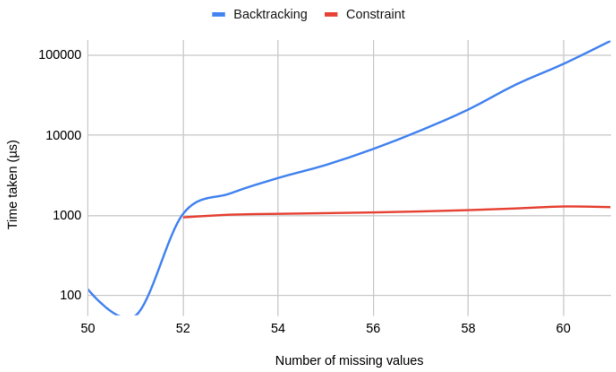

Fig 5. Plot of Table 2 and 4, i.e. how backtracking and constraint algorithms perform with the hard dataset (Note that the Y-axis scales logarithmically)

Fig 4 outlines the cost of setting up both algorithms, showing that initializing the constraint algorithm is significantly more resource-intensive than the backtracking algorithm. This is why backtracking outperforms the constraint algorithm, with the constraint algorithm requiring 600µs compared to approximately 40µs for backtracking.

However when the problem becomes hard, as shown in Fig 5, the constraint algorithm significantly outperforms backtracking. The constraint algorithm takes 1ms on average, whereas the backtracking algorithm takes exponentially more time for each missing value. This lines up with the complexities that were mentioned in Section 1.

Another important thing is that my implementation of the constraint algorithm, although way faster, cannot solve extremely hard Sudoku problems, which can be seen in Table 2. This is most likely due to not encapsulating all the interactions that rows, columns, and blocks can have. As mentioned in section 5.4 of [1], we cannot use simple bipartite matching to encapsulate rows-columns-blocks interactions since it would involve three sets of nodes, and a flow model would be used. I wasn't able to get to implementing this part of the paper due to time constraints. That all said, one thing that my current implementation is good at is solving "hard" problems designed for humans. It was able to solve all 580 hard Sudokus published by the New York Times over the past year, and all the Sudokus in the Project Euler problem.

Keeping the results in Fig 4. and Fig 5. in mind, I came up with a hybrid method to solve Sudokus:

- First, use the constraint algorithm
- If the Sudoku hasn't been solved, use backtracking

This combines the best of both algorithms: the efficiency of the constraint algorithm, and the solvability rate of the backtracking algorithm. It was able to solve all the 500,000 Sudoku problems significantly faster than the backtracking algorithm. The results for this hybrid algorithm can be seen in Fig 6. and Fig 7.

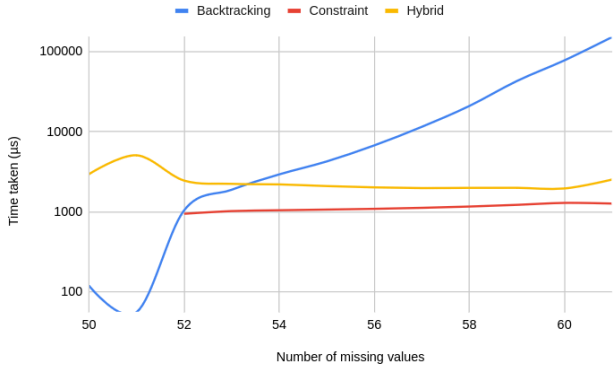

Fig 6. Using backtracking, constraint, and hybrid algorithm to solve the hard dataset (Logarithmic Y-scale)

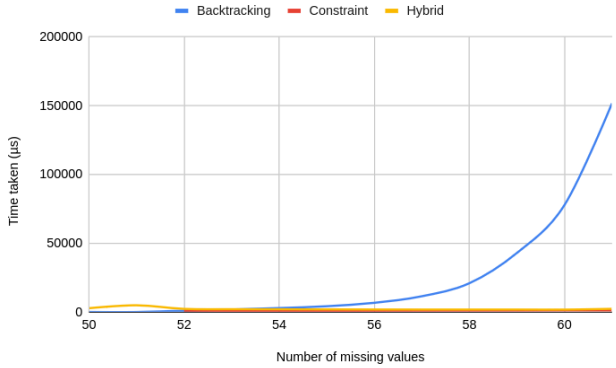

Fig 7. Using backtracking, constraint, and hybrid algorithm to solve the hard dataset.

5. REFLECTION

The entirety of the constraint algorithm was entirely new to me. It felt incredibly satisfying from not being even able to comprehend what [1] was saying, to implementing something that was able to solve most of the Sudokus I was able to throw at it. Another thing I found very interesting was how the core logic of the algorithm used two of the algorithms we went through in class: SCCs and maximum flow. If I had more time, I would want to work on increasing the solvability rate of the constraint algorithm for hard problems, but I am decently satisfied with the results I got with the hybrid algorithm. I think I would also like to test this program with 16x16 Sudoku grids.

6. RESOURCES

[1] was used as the primary source for the algorithm

[2] was used to help parsing out what [1] was saying

[3] and [4] were used for testing whether the overall constraint algorithm was working as expected.

[5] and [6] were used to perform performance tests

7. REFERENCES

[1] Simonis, H. 2005. Sudoku as a constraint problem. *CP Workshop on modeling and reformulating Constraint Satisfaction Problems* Vol. 12, 13-27

[2] Ole Kröger. 2017. Sudoku puzzles, Constraint programming and Graph Theory. Retrieved December 13, 2024 from https://opensourc.es/blog/sudoku/

[3] Abbe98. NYT Sudoku scraper. Retrieved December 9, 2024 from https://github.com/Abbe98/nyt-sudoku-scraper

[4] Project Euler. Su Doku. Retrieved December 13, 2024 from https://projecteuler.net/problem=96

[5] David Radcliffe. 3 million Sudoku puzzles with ratings. Retrieved December 13, 2024 from https://www.kaggle.com/datasets/radcliffe/3-million-sudoku-puzzles-with-ratings

[6] Vopani. 9 Million Sudoku Puzzles and Solutions. Retrieved December 13, 2024 from https://www.kaggle.com/datasets/rohanrao/sudoku