**SudoCode: an Analysis of 2   
Sudoku Solving Algorithms**

Jordan Stebner-Hoang  
<https://github.com/jstebner/sudocode>

**Introduction**

This project compares methods for solving sudoku puzzles of arbitrary order N, such that the grid that makes up the puzzle is a N\*N grid of N\*N sub grids, resulting in a N2\*N2 grid of numbers (e.g., an "order 3 puzzle" would be a classic 9x9 sudoku puzzle). Given an incomplete, solvable, order N sudoku puzzle, the algorithm should return a completed puzzle such that:

- Every symbol on the grid is one of N2 unique symbols

- Every row of the grid contains N2 unique symbols

- Every column of the grid contains N2 unique symbols

- Every N\*N sub grid of the grid contains N2 unique symbols

The algorithms will be measured by comparing the time taken to solve each puzzle in a set of incomplete order N sudoku puzzles (though, N will likely not exceed 6 due to computing constraints) with M missing symbols (blank spaces in the grid). The 2 algorithms I will compare are:

1. Backtracking in row-wise order. This approach will iterate over every cell in the grid filling in empty cells with the first possible symbol it can be (not violating any rules stated above) until the board is filled and valid. If a cell has no possible symbol, then the algorithm will "backtrack" and update previous "guesses" in LIFO order until the board is completed. I could perform this algorithm with either an iterative or recursive approach, but I have not yet decided which I would like to do (although I'm slightly more attracted to a recursive approach due to simplicity of implementation).
2. Constraint propagation with wave function collapse. This approach will first iterate over each unfilled cell in the grid and record the set of possible states it can exhibit. After doing so, it will iterate over each cell existing in a "superposition" of states and filter down its possible states by propagating the puzzle's constraints (listed above in (1)), reducing the "entropy" (number of possible states) each cell exhibits, collapsing the cell to a definite value if it can only exist in a single possible state. If, when filtering, the algorithm makes a full pass over the grid without reducing the entropy of any cell, the algorithm will collapse the state of the cell with lowest entropy using the first possible state and resume searching for the final grid state like backtracking but applying constraint propagation at each guess made.

**Design**

*\*Note: for the algorithms:*

* *Purple indicates functions*
  + *All external function calls are defined below*
* *Red indicates variables*
* *Blue indicates references to variables*
* *Bold and capitalized indicates operations*
* *Green indicates comments*

*Supporting algorithms:*

Function possible, accepting parameter grid, row, col, value: // O(n^2)

**IF** value is in grid at (row, :) **THEN**

**RETURN** False

**IF** value is in grid at (:, col) **THEN**

**RETURN** False

**IF** value is in the group of grid at (row, col) **THEN**

**RETURN** False

**RETURN** True

Function filled, accepting parameter grid: // O(n^4)

**FOR** each row index in the grid **DO**

. **FOR** each col index in the grid **DO**

. . **IF** grid at (row, col) is 0 **THEN**

.. **RETURN** False

. . **IF** grid at (row, col) is a set **THEN**

. . **RETURN** False

. **END FOR**

**END FOR**

**RETURN** True

1. **Backtracking**

Function solve\_BT, accepting parameter grid:

**FOR** each row index in the grid **DO**

. **FOR** each col index in the grid **DO**

. . **IF** the value of the grid at (row, col) isn’t 0 **THEN**

. . **SKIP** this col and continue to the next col

. . **FOR** each symbol in the set of possible symbols **DO**

. . . **IF** it is possible to set grid at (row, col) to symbol **THEN**

. . . . **SET** grid at (row, col) to symbol

. . . . **CALL** solve\_BT(grid) // recursive call

. . . . **IF** grid is not filled with valid symbols **THEN**

. . . . **SET** grid at (row, col) to 0

. . . **END IF**

. . **END FOR**

. . **RETURN**

. **END FOR**

**END FOR**

1. **Constraint Propagation + Backtracking**

Function solve\_CP, accepting parameter grid:

**IF** there are any 0s in grid **THEN** // create states

. **FOR** each row in the grid **DO**

. . **FOR** each col in the grid **DO**

. . . **IF** the value of the grid at (row, col) isn’t 0 **THEN**

. . . **SKIP** this col and continue to the next col

. . . **CREATE** empty set named states

. . . **FOR** each symbol in the set of possible symbols **DO**

. . . . **IF** it is possible to set grid at (row, col) to symbol **THEN**

.... **ADD** symbol to states

. . . **END FOR**

. . . **SET** grid at (row, col) to states

. . **END FOR**

. **END FOR**

**END IF**

**CREATE** a number named updates starting at 1

**WHILE** updates is greater than 0 **DO** // prune search space

. **SET** updates to 0

. **FOR** each row in the grid **DO**

. . **FOR** each col in the grid **DO**

. . . **IF** the value of the grid at (row, col) isn’t a set **THEN**

. . . **SKIP** this col and continue to the next col

. . . **IF** the set in the grid at (row, col) is empty **THEN**

... **RETURN**

... **IF** the set in the grid at (row, col) has 1 element **THEN** // singleton collapse

. . . . **FOR** each cell in the same row, col, and group as the cell at (row, col) **DO**

. . . . . **IF** the value in the cell is a set **THEN**

. . . . .. **REMOVE** the element from the set stored in cell

. . . . . **END IF**

. . . . **END FOR**

. . . . **INCREMENT** updates

. . . . **SET** grid at (row, col) to the element in the set

. . . . **SKIP** this col and continue to the next col

. . . **END IF**

... **ELSE** // polyzygotic propagation

. . . . **FOR** each subset of [row, col, and group] that the cell at (row, col) is part of **DO**

. . . . . **CREATE** a count of cells in the subset matching the cell at (row, col)

. . . . . **IF** count is less than the length of the set in the cell at (row, col) **THEN**

. . . . . . **SKIP** this col and continue to the next col

. . . . . **END IF**

. . . . . **ELSE IF** count is greater than the length of the set in the cell at (row, col) **THEN**

. . . . . . **RETURN**

. . . . . **END IF**

. . . . . **FOR** each cell in the same subset as the cell at (row, col) **DO**

. . . . . . **IF** the cell has a set that isn’t equal to the set in the cell at (row, col) **THEN**

. . . . . . . **REMOVE** elements from the cell at (row, col) from cell

. . . . . . . **INCREMENT** updates

. . . . . . **END IF**

. . . . . **END FOR**

. . . . **END FOR**

... **END ELSE**

. . . // more propagation checks can be added in series here

. . . **FOR** each symbol in the set in the grid at (row, col) **DO** // elimination collapse

. . . . **FOR** each subset of [row, col, and group] that the cell at (row, col) is part of **DO**

. . . . . **IF** the symbol isn’t in any set in the current subset **THEN**

. . . . . . **FOR** each cell in the same row, col, and group as the cell at (row, col) **DO**

. . . . . . . **IF** the value in the cell is a set **THEN**

. . . . ... . **REMOVE** the element from the set stored in cell

. . . . . . . **END IF**

. . . . . . **END FOR**

. . . . .. **SET** grid at (row, col) to the element in the set

. . . . . . **INCREMENT** updates

. . . . . **END IF**

. . . . **END FOR**

. . . **END FOR**

. . **END FOR**

. **END FOR**

**END WHILE**

// backtrack

**CREATE** r\_min, c\_min both starting with Null

**FOR** each row index in the grid **DO**

. **FOR** each col index in the grid **DO**

. . **IF** the value of the grid at (row, col) isn’t a set **THEN**

. . **SKIP** this col and continue to the next col

. . **IF** r\_min and c\_min are Null **THEN**

. . **SET** r\_min to row and c\_min to col

. . **ELSE IF** the size of the set in grid at (row, col) is smaller than the set in grid at (r\_min, c\_min) **THEN**

. . **SET** r\_min to row and c\_min to col

. **END FOR**

**END FOR**

**CREATE** copy of grid named backup

**IF** r\_min, c\_min are both not Null **THEN**

. **FOR** each symbol in the set in grid at (r\_min, c\_min) **DO**

. . **SET** grid at (r\_min, c\_min) to symbol

. . **CALL** solve\_CP(grid) // recursive call

. . **IF** grid is not filled with valid symbols **THEN**

. . . **SET** grid to a copy of backup

. . **END IF**

. . **ELSE**

. . . **RETURN**

. . **END ELSE**

. **END FOR**

**END IF**

**Analysis**

*N: “order” of puzzle (classic 9x9 would be N=3)*

*k: number of elements removed*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Time complexity | Space Complexity | Basic operations | Input Size Consideration |
| Algo1: Backtrack | O( N2k ) | O( k ) | Hash map search | Input must be a N2-by- N2 grid, with k removed elements |
| Algo2: Const Prop | O( N12k ) | O( k2 N2 ) | Hash map search, set difference | Input must be a N2-by- N2 grid, with k removed elements |

**Hypothesis**

I believe Algorithm 2 will run faster mostly since it aims to heavily reduce the number of recursive calls by pruning the search space repeatedly (and partially because I’m biased towards it since I designed it). While on paper it appears as though it will perform worse in time and space, I am most interested in seeing how many fewer recursive calls it will make than Algorithm 1 (i.e., fewer guesses). I believe that with enough constraint checks, a sudoku solving algorithm can solve a board with 0 guesses, and Algorithm 2 is a step in that direction.

**Test plan**

There are 3 files (n=2,3,4) that contain sudoku puzzles with k removed elements, where the largest k depends on n:

* n=2: max(k)=12
* n=3: max(k)=64
* n=4: max(k)=200

Each (n, k) has 40 randomly generated puzzles whose times, space, recursive calls, and validity will be averaged when comparing results. The expected results will be a valid puzzle created by each solving algorithm (instead of using an expected result puzzle, as some solvable puzzles have multiple valid solutions).

(*I wanted to have up to n=7, but my computer was unable to generate a single n=5 puzzle so I’m mostly going to be comparing changing values of k across 3 classes of n)*

All data can be found in sudocode/data/\*.csv

**Results**

*\* BT refers to Algorithm 1 (backtracking), while CP refers to Algorithm 2 (constraint propagation)*

Chart, line chart

Description automatically generated**Time**

The graph to the left shows the average time taken to run each algorithm across a range of k removed values for order 2 puzzles.

Since order 2 puzzles have few elements, the range of possible removed values is small and doesn’t show a clear trend between number of removed values and time taken.

We can see, however, that the time taken on order 2 puzzles is generally higher for CP than BT.

Chart, histogram

Description automatically generatedThe graph to the left shows the average time taken to run each algorithm across a range of k removed values for order 3 puzzles.

The vertical axis is on a logarithmic scale to show the exponential growth of time taken as k increases. At lower k values (up to ~23), the time taken varies around 0 nano seconds, however at higher k values we can see an upward trend for both BT and CP.

We can see that as k grows for order 3 puzzles, the time taken by BT is significantly larger than CP.

Chart, histogram

Description automatically generated

The graph to the left shows the average time taken to run each algorithm across a range of k removed values for order 4 puzzles.

The vertical axis is once again on a logarithmic scale, as the difference between BT and CP is drastic on a linear scale.

We can see that at lower values of k (<90), BT and CP perform similarly on order 4 puzzles, taking roughly 3 ms with a slight upward trend. However, for higher values of k, the difference between BT and CP becomes large.

**Space**

The graph to the left shows the average peak memory when running each algorithm across a range of k removed values for order 2 puzzles.Chart, line chart

Description automatically generated

Again, since order 2 puzzles are small, we can see odd trends in the graph for BT and CP.

A possible trend we can see from this graph is the slight incline of CP, while BT stays very close to 0.

Chart, line chart

Description automatically generated

The graph to the left shows the average peak memory when running each algorithm across a range of k removed values for order 3 puzzles.

This graph offers much more information on the trend of space usage by each algorithm, showing that peak memory is exponentially related to k in CP, while BT once again stays very close to 0.

While this trend is drastic, we should keep in perspective that the highest peak memory used by CP is ~5\*105 Bytes, or 500 KB of memory.

Chart, line chart

Description automatically generatedThe graph to the left shows the average peak memory when running each algorithm across a range of k removed values for order 2 puzzles.

This graph shows that the peak memory usage as k increased is nonlinear for CP and linear for BT in order 4 puzzles which roughly matches what was predicted in the analysis section.

I like this graph a lot.

**Recursive Calls***(TOTAL recursive calls, not depth of recursion)\**

**Chart, line chart

Description automatically generated**The graph to the left shows the average number of recursive calls when running each algorithm across a range of k removed values for order 2 puzzles.

We can see that the number of recursive calls grows linearly as k increase for BT and stays close to 0 for CP.

While this is for order 2 puzzles and therefore lacks many data points, I am happy to see that the number of recursive calls for CP stays very low, which was the intention when designing the algorithm.

Chart, line chart

Description automatically generated  
The graph to the left shows the average number of recursive calls when running each algorithm across a range of k removed values for order 3 puzzles.

The vertical axis is once again on a logarithmic scale, as the difference between BT and CP is drastic on a linear scale.

We can see that the number of recursive calls for BT begins to grow quickly at higher values of k, while CP gets to at most ~10.

**Chart, line chart, histogram

Description automatically generated**

The graph to the left shows the average number of recursive calls when running each algorithm across a range of k removed values for order 4 puzzles

The vertical axis is once again on a logarithmic scale, as the difference between BT and CP is drastic on a linear scale.

This graph looks very similar to the order 3 graph on average recursive calls, and shows in higher detail the difference between BT and CP.

**Conclusion**

Overall, I would say that my hypothesis was true that the constraint propagation algorithm took less process time and recursive calls than the backtracking algorithm. For both metrics on higher values of k, CP performed much better than BT, even though the time complexity for each algorithm suggested otherwise (in the Analysis section). I somewhat expected this, however, since according to the worst case the CP algorithm would perform many more steps than BT within the same possibility space, even though the extra steps performed are to prune that space and reach a goal state more quickly.

The space complexity estimates roughly matched the actual test results, however the memory used by each algorithm was still relatively small (staying under a MB). As my hypothesis was focused on the speed of the solving algorithms, I will turn a blind eye to the fact that backtracking outperformed my constraint propagation algorithm.

I initially wanted to go up to N=7, but I couldn’t generate an order 5 puzzle and instead stayed with order 2, 3, and 4. I also created puzzles of order 4 up to k=200, but I was only able to run tests up to k=132 as each test took on average ~20 minutes to run and I was running out of time.

An additional metric was collected for each test, the validity of a solution to the puzzle returned by the algorithm (can be seen in sudocode/out/results.csv). This was to ensure that the algorithms didn’t return and invalid puzzle; however, all solutions were valid.

**References**

<https://www.youtube.com/watch?v=G_UYXzGuqvM&t=1s&ab_channel=Computerphile>

<https://en.wikipedia.org/wiki/Mathematics_of_Sudoku>

<https://arxiv.org/pdf/cs/0602027.pdf>

<https://www.sudokuwiki.org/Sudoku_Creation_and_Grading.pdf>

<https://www.inf.tu-dresden.de/content/institutes/ki/cl/study/winter06/fcp/fcp/sudoku.pdf>