Sudoku Solver Project Overview

By: Alex Larsen, Calell Figuerres, Owen Bartel, Son Nguyen, Teni Olugboyega

Slides:
https://docs.google.com/presentation/d/1BmiWHKrngC5fUt2pi1guOX2G_Ccyh5TQYKuFWMYBdcw/edit#slide=id.g351169c9f1f_0_560

Code:
https://colab.research.google.com/drive/14Pr_PiLrOEacfJxvTwLBD9E2LPa7iGAx?usp=sharing

Problem Statement

The summary of the entire project is to create conjunctive normal form (CNF) formulas based on the constraints of Sudoku of various puzzle sizes of n and feed them into an boolean satisfiability algorithm (SAT) solver which would then use all the boolean statements to compute the result. The SAT should have a solution that shows which variables are true for the formula to be logically valid and sound.

This problem can be broken down into steps in order to find a solution if possible:

- Identifying all the constraints (rules) of Sudoku
- Encoding all the constraints in a single CNF formula
- Feed the CNF formula into a SAT
- Solve the solution
- Decode the solution variables and see results

Glossary

CNF: Conjunctive Normal Form.
SAT: Satisfiability algorithm for boolean formula problems.
n: The subgrid size of the board. n would = 3 for a normal 9 x 9.
N: The total length of the board. Would be N = 9 for a normal board.
Clues: Cells values already filled in for the puzzle.
Clause: Logical formula used to represent a rule.

Identifying Constraints

When breaking down the game of Sudoku, we all know the obvious rule for rows, columns, and blocks. However, when encoding the game for an algorithm, it's important to also recognize the rules that we don't even notice cuz we assume them. Below are all of the constraints of Sudoku:

- C1: Each cell must have a value from 1 - N
- C2: Each cell must have only one value
- C3: Each row has all numbers from 1 - N
- C4: Each column has all numbers from 1 - N
- C5: Each n x n sub grid has all numbers from 1 - N

Now that we've identified the constraints we can now encode clauses for each cell given its position into one massive formula.

Encoding Constraints into CNF

When encoding constraints for Sudoku there are a few ways to go about it. The three main algorithms used for Sudoku are pairwise, sequential counters, and binary encodings. For this project, our team kept the scope of the project to only pairwise and sequential counters.

When talking about implementation, the SAT algorithms we used from the pysat library consider logic statements as integers with each unique integer representing a different statement. Positive integers represent true statements while negative integers represent false statements. No matter what algorithm we used, we had to encode each cell given its row, column, and possible values. This leads to a space complexity of N^3 of the total possibilities a blank sudoku board will have.

Pairwise Algorithm

Pairwise would be the most simplest and naive solution for tackling the problem of encoding the constraints. Its clause generation rises very quickly for the problem size and each clause can also be quite large as well. That being said, there won't be a statement that isn't logical for the game, only redundant ones once we add clues.

For creating the unique variables to represent solution possibilities, our team used a function that would generate a unique number based off of the cells given row, column, and value we are considering. Below is a snippet:

```python
from pysat.solvers import Solver

# We will call this function to generate a unique variable (integer)
# to represent the case of a cell at row r, column c, with value v
def var(r, c, v, N):

    # Assertion statements to help debug and make sure we never have a row, col., or val that breaks index
    assert(1 <= r and r <= N)
    assert(1 <= c and c <= N)
    assert(1 <= v and v <= N)
    return (r-1)*N*N+(c-1)*N+(v-1)+1

# We do a plus one as a 0 can cause trouble in CNF literals as they negate our var number by making them negative
```

Later in our algorithm, we will use a decoder function that will break apart a unique integer into its row, column, and value by reversing the above formula.

With our var() function we can now easily create logic statements. To encode the different constraints, we just need to create a logical representation by going through row, column, and value numbers. Below is a code snippet on how encoded the row constraint:

```python
    # Iterate over all possibel values
    for v in range(1, n + 1):

        # This represents clauses for C3
        # Each row must have one of each value
        for r in range(1, n + 1):
            cnf.append([var(r, c, v, n) for c in range(1, n + 1)]) # We use c here to iterate through a row
```

Once we've created a CNF object with all clauses conjoined, we can return it so it can later be fed to an SAT solver.

Sequential Counters Algorithm

Beyond a naive solution, sequential counters use auxiliary variables to create local propagation of information amongst cells. By having an auxiliary variable for each constraint, we can create smaller clauses and have a more efficient way of having cells knowing only necessary information. One way of thinking of sequential counters is instead of comparing every pair of variables for a given row, column, or block, we just track when values appear and propagate information to only necessary cells.

An example way of thinking about this algorithm is a cell is checking for what values it can claim for a row. It will look directly to the cell left of it to ask if the variable has appeared yet. If it does we make sure we can alert the cell to the right of this cell that value is claimed. If the cells to our left haven't seen the value, we check others cells next to us to make sure we can claim a value.

When encoding the constraints, we can achieve sequential counters by creating a clause of each of the following rules. Note, this is using the row constraint auxiliary variable:

Rule 1: If our number is true for a given cell, then all of the sequential counters involved with this cell need to know.

- ~X(r, c, v) V R(r, v, c)

Rule 2: Once we realize a sequential counter is true, it must stay true to tell the others.

- ~R(r, v, c-1) V R(r, v, c)

Rule 3: We must prevent other cells within this row, column, or block from claiming the same value v.

- ~X(r, c, v) V ~R(r, v, c-1)

Given that we have auxiliary variables, we will need a way to efficiently create different types of variables. Instead of creating variables via different functions, we used a dictionary to store variables of type X, R, C, B, and A. A is used for constraint C2. When we decode now, we will instead just check the dictionary with the key as a tuple of r, c, and v.

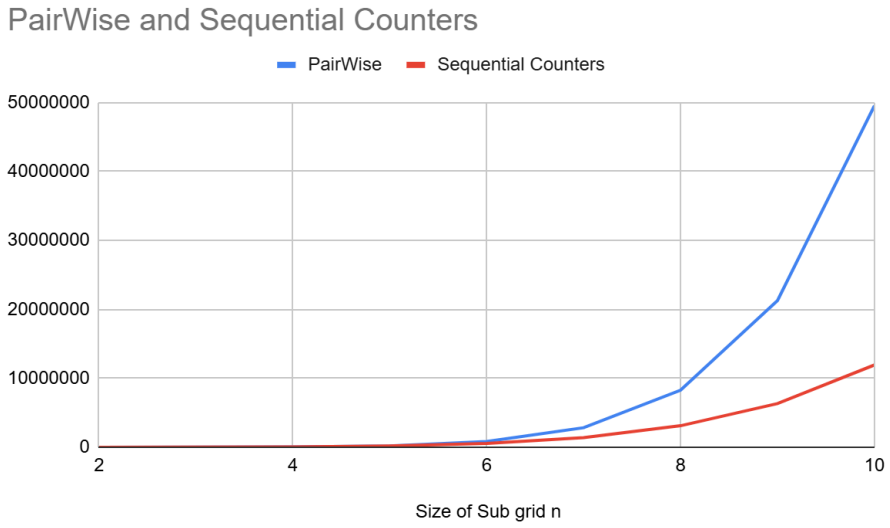
Below is our team's code for how we generate the row constraint clauses for sequential counters:

```
# Generates cnf formulas for row constraint
for r in range(1, n + 1):
    for v in range(1, n + 1):
        for c in range(1, n + 1):
            x = var_X(r, c, v)
            R = var_R(r, v, c)

            # Rule [1]: ~X(r,c,v) ∨ R(r,v,c)
            solver.add_clause([-x, R])
            cnf.append([-x, R])

            if c > 1:
                r_prev = var_R(r, v, c - 1)

                # Rule [2]: ~R(r,v,c-1) ∨ R(r,v,c)
                solver.add_clause([-r_prev, R])
                cnf.append([-r_prev, R])

                # Rule [3]: ~X(r,c,v) ∨ ~R(r,v,c-1)
                solver.add_clause([-x, -r_prev])
                cnf.append([-x, -r_prev])
```

Clause Count Comparison

An important factor to consider when pushing an SAT to its limits is the amount of clauses our CNF will end up containing. Overwhelming the SAT with too many will ultimately lead to the solver having an unreasonable solve time we cannot wait for.

As previously mentioned, sequential counters are more efficient with its clause creation and will scale better as n increases than pairwise. This is the main advantage sequential counters have, and it's a great one. Below is a graph showing the growth in clauses for each algorithm as n increases:



Questions and Predictions

Both algorithms solve a sudoku puzzle for a given length n in different ways, each with its own strengths and weaknesses. Our team decided to do an experiment of testing three things:

1. What is the biggest puzzle we can solve in a reasonable amount of time?
2. How many times do we get a puzzle entirely correct as n increases?
3. How does our accuracy for our solution look as n increases.

The team predicted that pairwise would be more accurate and get more puzzles completely right, but sequential counters will be able to solve bigger puzzles.

Puzzle Creation

The puzzle is created in two steps:

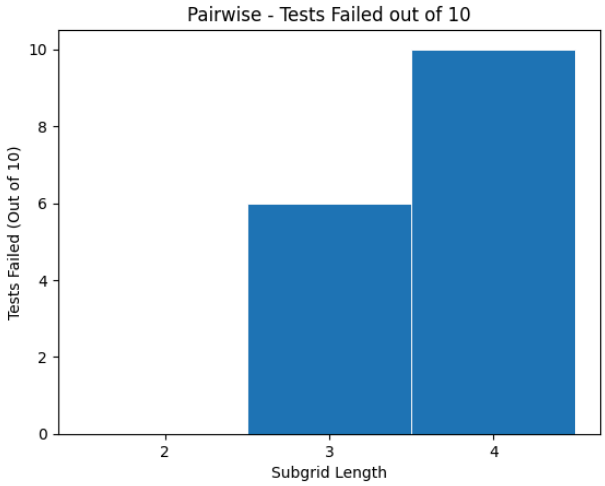
First, a complete, valid Sudoku solution is generated by applying a simple “base” pattern—shifting rows, columns, and blocks in a deterministic way—and then randomly shuffling the order of rows, columns, and the digits themselves to ensure a unique, scrambled full grid.

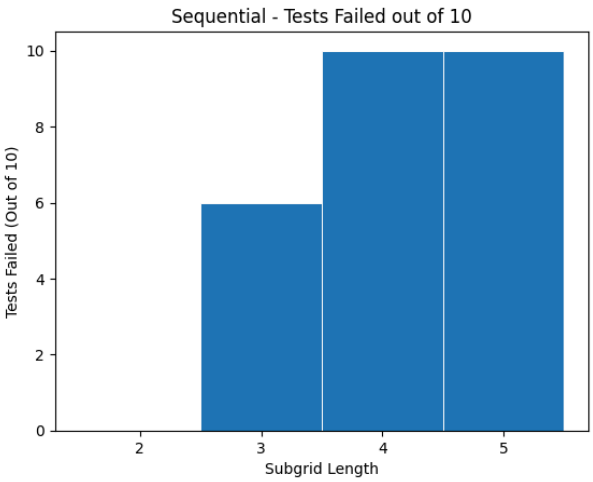
Once the full board is created, cells are “removed” one by one at random based on a specified removal rate, replacing those entries with zeros (empty spots). The result is a proper Sudoku puzzle: a partially filled grid that retains a unique solution but requires the solver to deduce the missing numbers.

Testing Results

To begin our tests, we used the above methods to create puzzles ranging from 4 by 4 (a subgrid size of 2 by 2) to 100 by 100 (a subgrid size of 10 by 10). We created 10 puzzles of each subgrid size then ran those puzzles through each algorithm, stopping when the algorithm would take too long (e.g., more than a few minutes for a puzzle) to solve a given puzzle.

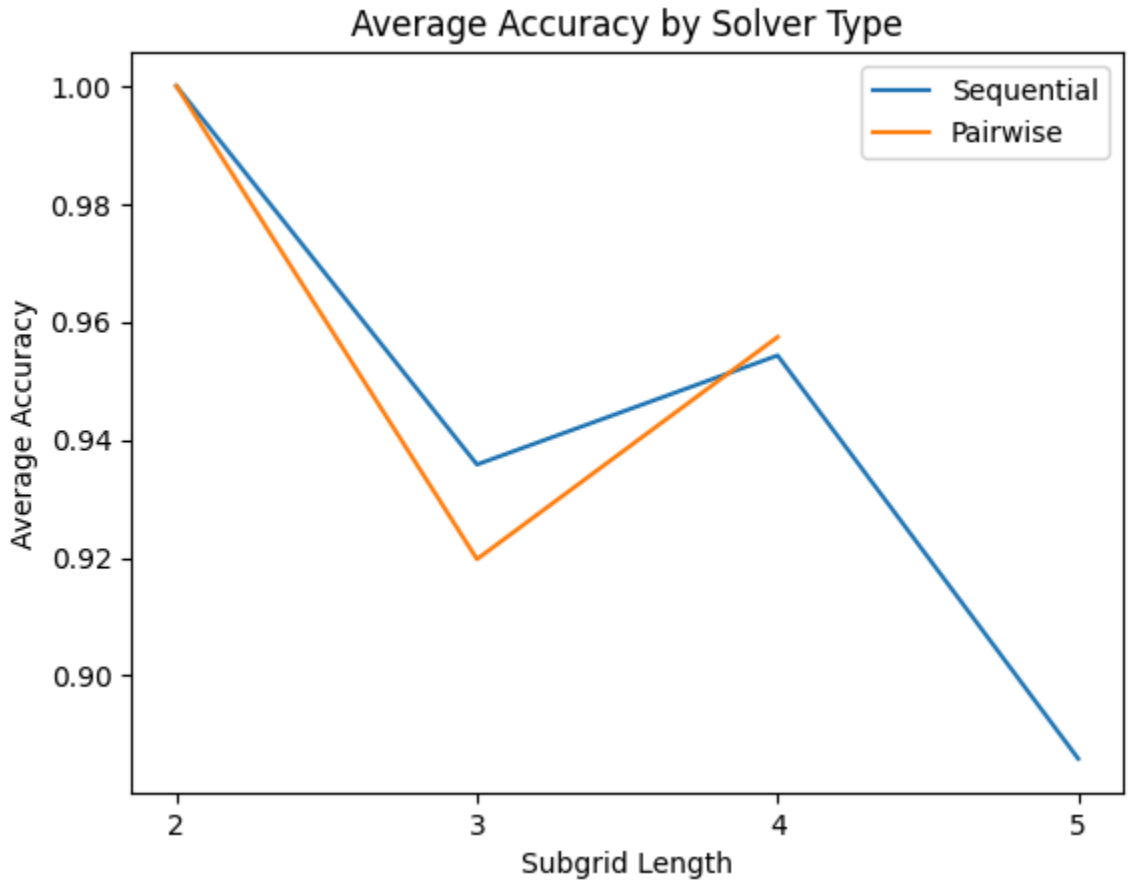
We collected from each test case the number of CNF clauses the given algorithm generated after putting in the clues for the puzzle and the accuracy of the answer the algorithm produced. We then used the accuracy metric to calculate the number of tests that solver failed (i.e., didn’t get completely correct).





As can be seen from these two graphs, both pairwise and sequential algorithms are completely correct with a subgrid size of 2, but quickly start falling off in accuracy for a regular sudoku

puzzle of size 3, and always fail (our admittedly small test size) at sizes four and above. It is worth noting that, even though it still failed, sequential counters did handle slightly larger subgrids than pairwise before taking significantly too long to solve.



As can be seen from this second graph, the average accuracy of each algorithm also quickly plummets beginning with subgrids of size 3 and above. We can also see that the pairwise algorithm never made it past subgrids of size 4, and sequential never got past subgrids of size 5.

Preprocessing

As shown by our testing results, we can see that despite our optimization efforts we end up hitting a wall of computation once we've reached a certain puzzle size. For pairwise we couldn't get past n = 5 while sequential counters came to halt on n = 6. Without knowing the clues of a puzzle when generating CNF formulas, we will consistently create the max amount of clauses for our algorithms even though we don't necessarily have to.

To help get over this time complexity issue, our team decided that we needed to add some preprocessing steps to allow our algorithms to become aware of the clues already done for the puzzle.

Adding Domain Awareness

When a human solves a sudoku puzzle, they typically look at the row, column, or block most filled and narrow down the values a cell has to be. This can be broken down as *domain awareness*. Essentially, by following the rules we know that a cell can only have so many possible values. Thus, instead of encoding CNF constraints for all values 1 - N, we just encode the ones that are in a cell's domain.

For example, a blank sudoku board when initialized with each cell has a domain from 1 - N. But then we add clues to the board of the cells already filled in. The cells given only have a domain of the value they start with. They are a single value that is correct and cannot be changed. If a cell in a row notices there is a three to its left, we now know that our cell cannot have value three. Thus, we take three out of the domain of our cell.

To summarize domain awareness, it represents cells knowing which values they can possibly have not based on the clauses we encode, but strictly off the clues given beforehand.

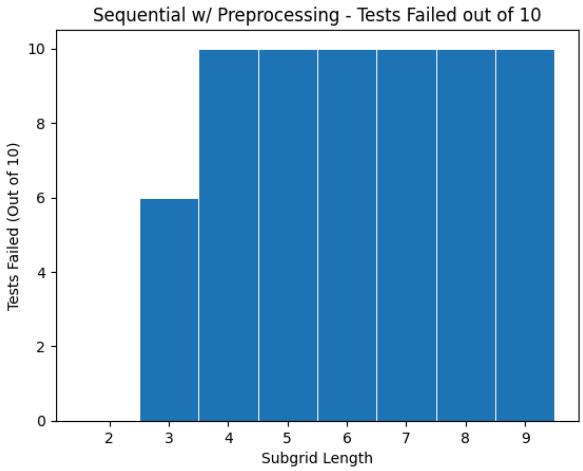
Second Experiment

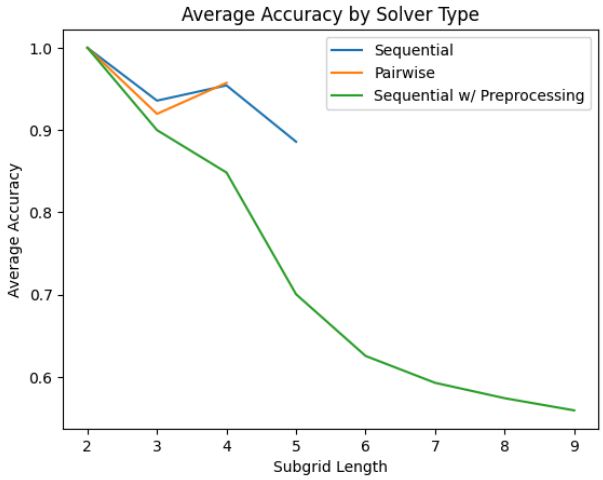
By creating a preprocessing method that stored a 2D grid of domains of all cells, our team was ready to try and reach n = 10. We combined our sequential counters code with our preprocessing code to achieve what we thought would be the most efficient algorithm in our project yet.

Our team assumed this algorithm could perhaps be underconstrained and lead to greater inaccuracies, especially as we see n increase. However, our main goal was to see how big of a puzzle it could actually complete with somewhat decent accuracy.

Testing Results with Domain Awareness

For testing with domain awareness, we used the same puzzles as before, and measured the same results (time taken for each test, accuracy of each result, and number of clauses generated).





As can be seen from these two graphs, adding preprocessing to our sequential algorithm allowed it to complete the tests for all subgrid sizes from 2 to 10–albeit, still continuing to fail at solving the puzzles. However, we can also see that the accuracy of the algorithm plummets even faster and farther than either potential or sequential without preprocessing.

Final Conclusions

With all of our testing results of improved algorithms, we clearly have some final statements. In terms of accuracy, sequential counters were the most accurate while pairwise was a close second. We believe this is because sequential counter clauses were more solver friendly and helped the SAT. For efficiency of problem size, our domain aware sequential counters were the best as it could go to n = 10, but highly inaccurate. I believe this is because sequential counters only work if all the auxiliary variables are created, and domain awareness means we skipped certain cells. Without all variables created, we cannot propagate information. Regular sequential counters could go to n = 5 while pairwise could only go to n = 4.

After doing the entire project, I wish we had more time to discover what made our solvers have any inaccuracy for our first two algorithms. By our checking, we encoded the possibilities correctly and somehow made a couple mistakes even in a 9 x 9 board. Furthermore, we also should have done domain awareness with our pairwise code first as sequential counters only work with all required auxiliary variables. We jumped the gun here as sequential counters was better than pairwise being the shallow reason of why we chose it.