Sudoku Solver Presentation

Group 19: Calell Figuerres, Alex Larsen, Son Nguyen, Tenitonimi Olugboyega, Owen Bartel

Brief Overview

- Problem Breakdown
- CNF Generation Algorithms
- Hypothesis
- Clause Comparisons
- Testing Results
- Adding Preprocessing Domain Awareness
- Testing Results
- Conclusion

Problem Breakdown

- Goal: Create CNF formulas based on the constraints of Sudoku of puzzle size n and feed them into an SAT solver which would then use all the boolean statements to compute a result.
- Reminders:
  - CNF stands for Conjunctive Normal Form. This is logical propositions in the form $(X_1 \vee X_2 \vee \ldots.)$ & $(\sim X_4 \vee \sim X_5)$.
  - SAT is a boolean satisfiable algorithm that computes all true variables to make the given boolean expression entirely true.
- How efficient our algorithm was depended on our total clause count and the structure of the clauses we added. Increasing the puzzle size can *dramatically* increase complexity.

Encoding and Identifying Constraints

- Our SAT solvers need to know two things:
  - What are the values we already know? We will consider these as *clues*.
  - What are the rules of Sudoku?
- Constraints for Sudoku:
  - C1: Each cell on the Sudoku board 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 Subgrid or block must have all numbers from 1 - N
- SATs our team has chosen consider a variable like X_1 to be a unique integer
  - Positive numbers represent True variables
  - Negative numbers represent False variables

Pairwise CNF Algorithm

- The most straightforward algorithm, but also the most naive…
  - Creates unique variables to represent different possibilities. Variables are created using a unique formula that takes into account the row, column, and value.
  - We create clauses and slowly append them to a CNF structure (List of lists).

```
# 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

    # This represents clauses for C4
    # Each column must have one of each value
    for c in range(1, n + 1):
        cnf.append([var(r, c, v, n) for r in range(1, n + 1)])

    # This represents clauses for C5
    # Each n x n block must have one of each value
    for br in range(0, N):
        for bc in range(0, N):
            cnf.append([var(br * N + rd, bc * N + cd, v, n) for rd in range(1, N + 1) for cd in range(1, N + 1)])
```

Sequential Counters CNF Algorithm

- Summary:
  - Local propagation
  - Almost like running memory
  - Uses auxiliary variables to help deal scaling as we get into larger puzzles.
    - No longer just X variables, but now we have R, C, and B variables.
  - Instead of comparing every pair of variables for a given row, column, or block, we just track when values appear and propagate information to necessary cells?
  - Example:
    - 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 cells to the right this value is claimed.
      - If it doesn't, we check other cells next to us to make sure we can claim a value.

Three Rules of Sequential Counters

- Focuses on three rules for a row constraint:
  - 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)
- The R variables with c-1 is how we propagate information.
- Other constraints are encoding identically with unique variable forms.

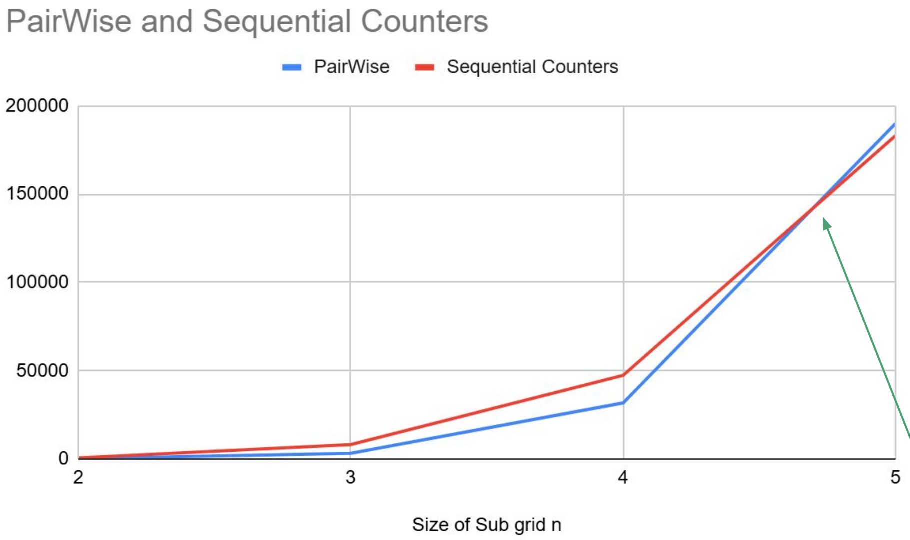
PairWise vs Sequential Counters

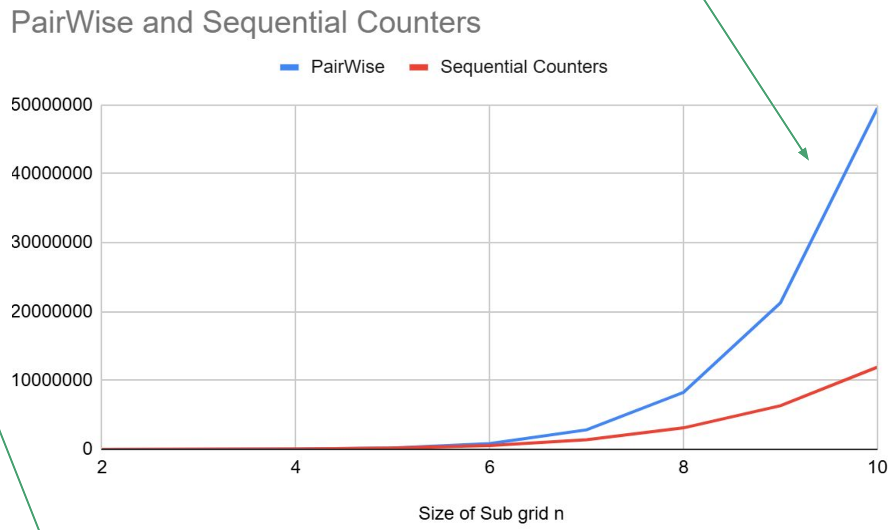
PairWise	Sequential Counters

• Naive solution
• Logically sound
• Minimum space complexity
• Scales poorly with large puzzles
• Clause generation:
  ○ C1 = N^2
  ○ C2 = N^3(N-1)/2
  ○ C3 - C5 = N^2
  ○ **Total: 4(N)^2 + (N^3(N-1)/2**
	• Local Propagation
	• Auxiliary variables
	• High space complexity
	• Scales well against larger puzzles
	• Solver friendly clauses
	• Clause generation:
	  ○ C1 = N^2
	  ○ C2 - C5 = N^2(3N-2)
	  ○ **Total: 12(N)^3 - 7(N)^2**

Clause Count Comparison

Poor Scaling!



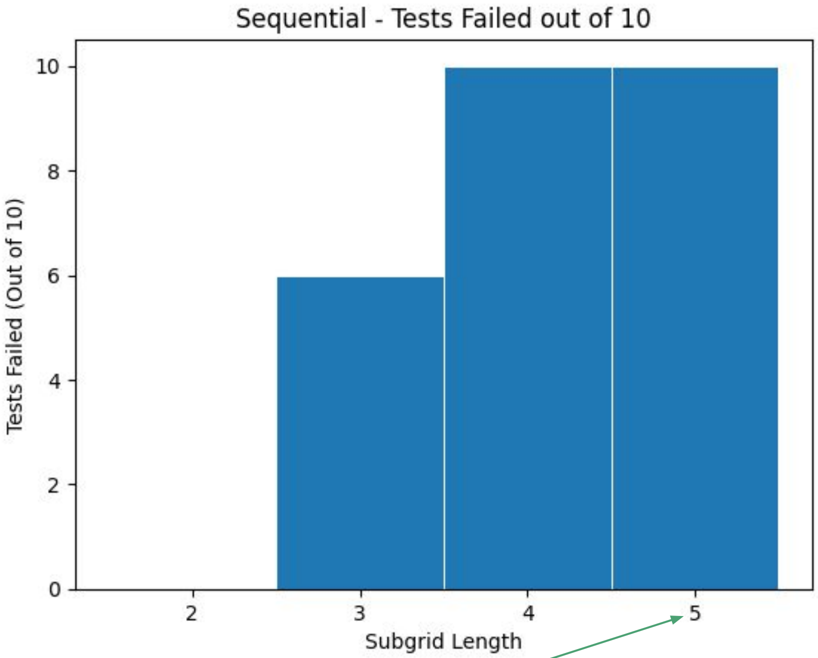


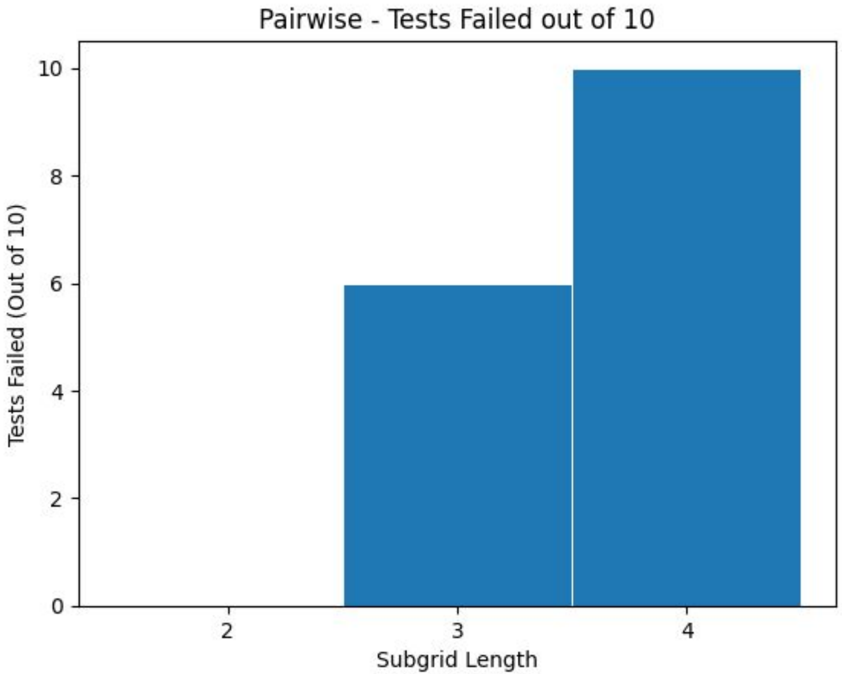
SQ overtakes PW at n >= 5

Experiment and Hypothesis

- Experiment
  - Generate 10 unique puzzles for each size n (2, 3, 4, 5, 6)
  - Push the boundary to see how big of a puzzle we can solve
  - Run both pairwise and sequential counters to compare **accuracy** and **runtime**
- Hypothesis
  - Sequential counters will be able to handle larger puzzles more efficiently
  - Pairwise will be more constrained and lead to less mistakes (better accuracy)

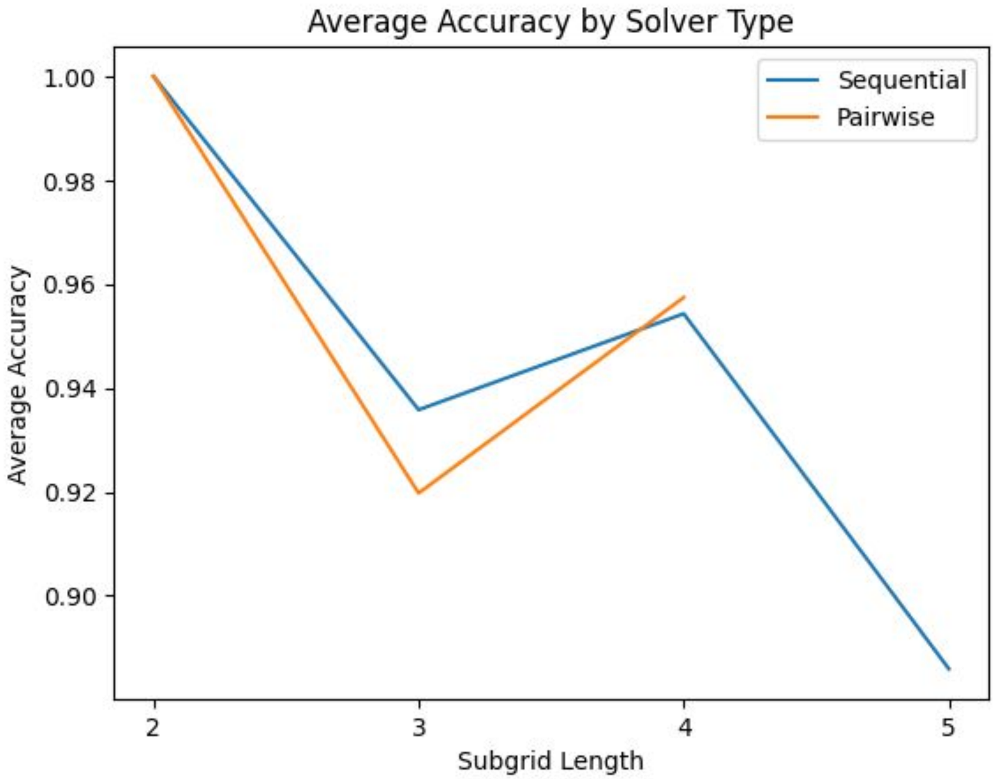
Testing Results: Comparison of Invalid Results





Can do bigger puzzles

Testing Results: Comparison of Accuracy



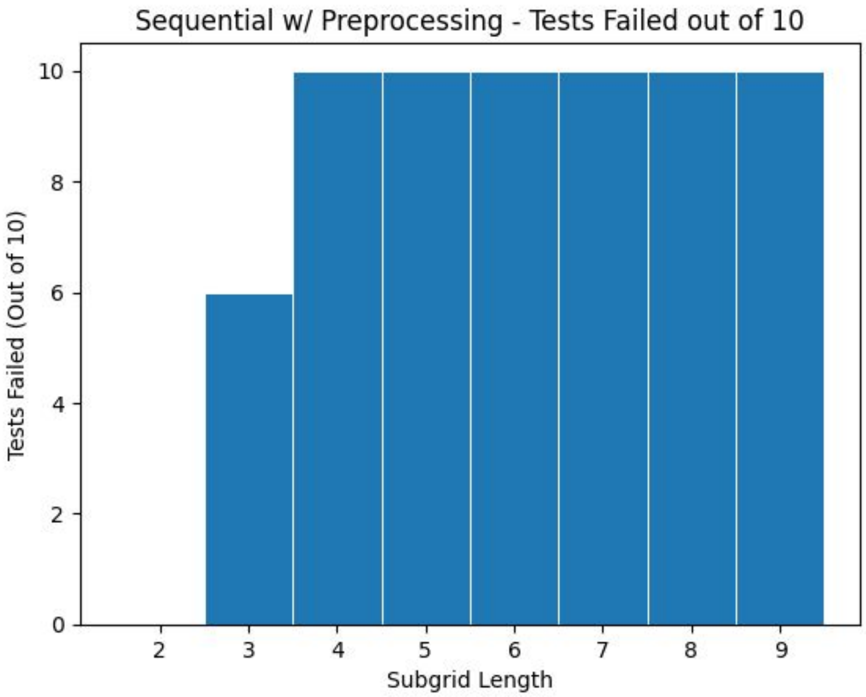
Result Analysis

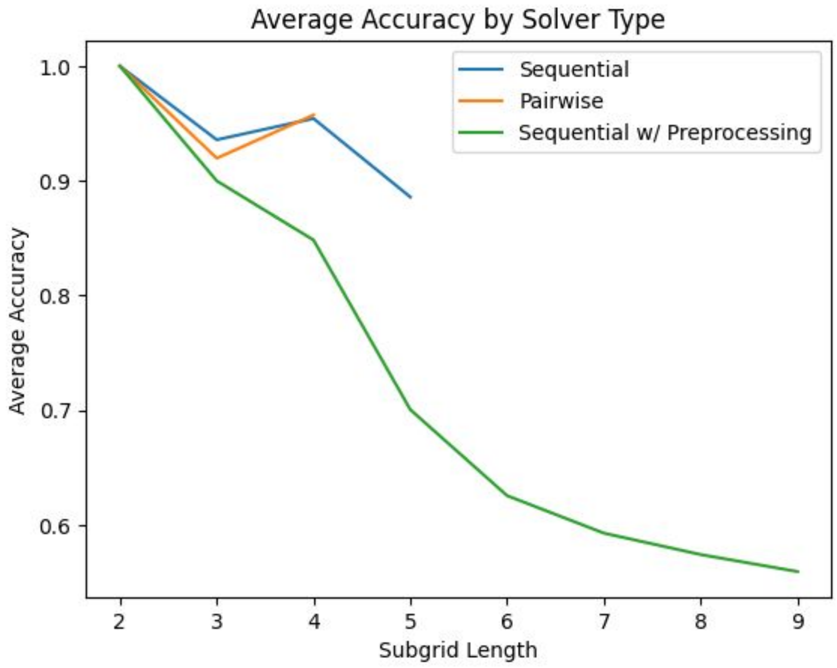
- Both algorithms hit a complete wall at some point to which the SAT couldn't solve in a reasonable amount of time.
  - Pairwise max puzzle size was n = 4, 16 x 16
  - Sequential counters max puzzle size was n = 5, 25 x 25
- Sequential Counters out performed pairwise in both puzzle solving size, valid puzzles for n, and overall accuracy.
  - Although we assumed sequential counters would be able to solve larger puzzles, we did not expect it to out perform pairwise in accuracy.
  - Things that could explain this:
    - Pairwise might being over constraining variables with its redundancy
    - Coincidence that it over performed for these puzzles

What if we added Preprocessing?

- Despite our advantages with our sequential counters, we still hit a wall with at where our algorithms can no longer reach a bigger puzzle size.
- What if we add a preprocessing step that takes into account the given clues of a puzzle and make only necessary clauses?
  - Advantages:
    - Lower clause count
    - Becoming "**domain aware**"
    - Shrinking the problem before CNF generation even starts

Preprocessing Testing Results





Conclusive Statements

- Overall accuracy plummeted and was worse than algorithms without preprocessing.
  - Explanations:
    - Coding is imperfect
    - Sequential Counters must have all the variables to propagate information, else information cannot travel.
    - Should’ve done preprocessing for pairwise instead of sequential counters
    - Overall problem was underconstrained
- Larger puzzles were able to run
  - Much fewer clauses were generated
  - No longer overwhelming the SAT solver

Thank You for Listening

Any Questions?