Sudoku as a SAT Problem

1 Introduction

Sudoku is a well-known puzzle that has gained international popularity. The objective is to fill a n^2 x n^2 grid with n^2 digits from 1 to n^2 such that each column and each row contains all numbers from 1 to n^2 and each of the n^2 blocks contains all numbers from 1 to n^2. For each number 1 to n^2, it may only appear at most once in each column and row. In this report, we view the Sudoku problem as a Satisfiability (SAT) problem where the puzzle is encoded into conjunctive normal form (CNF) and is solved using propositional satisfiability inference techniques. Our focus is on comparing the time complexity and space complexity of the inference techniques Unit Propagation, Failed Literal Rule, and Pure Literals where different size test cases will be considered for our analyses. The nature in which the CNF sentence is solved is done through a combination of backtracking and inference techniques.

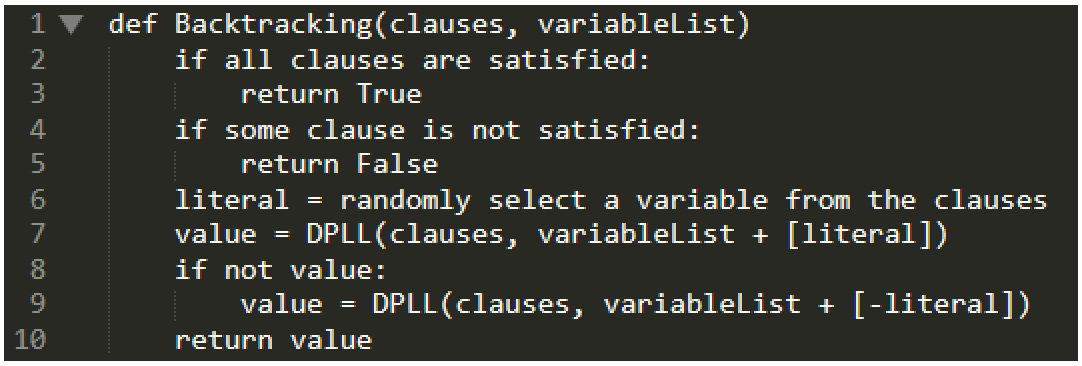
A SAT problem is represented using n propositional variables s_1, s_2, …, s_n which are assigned a truth value 0 (false) or 1 (true). We represent a literal *l* as a positively assigned variable s_i or as a negatively assigned variable $-s_i$. A CNF formula is composed of a conjunction of clauses where each clause is a disjunction of variables. Our objective is to satisfy the entire set of clauses, better known as a sentence. On a smaller scale, this is done by assigning a single variable a value such that the variable resolves to a true literal, thus, satisfying the clause. We perform this across all clauses. When all clauses in our CNF formula are satisfied, this means that the sudoku board has been solved. If the clauses cannot be satisfied using any combination of variable assignments, then this means that the sudoku board is not solvable.

There is a specific set of rules to follow in order to convert a sudoku puzzle into a CNF sentence. Since there are a total of 9x9 = 81 spaces on the board, and 9 possible numbers to put into each space, we reach a maximum of 9x9x9 = 729 variables. What each variable represents is its row number, column number, and value. We can represent each variable as s_{xyz} where x represents the row number, y represents the column number, and z represents a number from 1 to 9 which occupies the space. Additionally, we add a negation to indicate that variable z cannot be in cell xy. For example, s_{385} means that in row 3 and column 8, there is a 5. This will be one variable within the clauses. Negative variables can be encoded similarly. Given the previous example, we would not be able to put the number 5 in row 3, column 5, and its respective 3x3 sub-grid. Many negative variables can come out of this, one of them being $-s_{375}$.

2 Implementation

2.1 - Backtracking

The core of the algorithm is its backtracking nature. It works by assigning a random variable true initially, and eliminating any clause that can be satisfied with this variable assignment, it will then move on to the next variable and assign it to true as well. If it runs into a case where there is no solution given the current variable assignment, then it will backtrack and start assigning variables to false. The only way for the algorithm to run into this is by running into an empty clause, []. This indicates that the given variable assignment is not satisfiable, since our inference techniques will work by removing any variables which have no possibility of resolving their clauses to true, i.e. false literals. By itself, it has the potential to run through every combination of variable assignments, however, this would take unrealistically long for large sudoku instances, so we speed up the search significantly by using inference techniques. Even though many algorithms simply select the first variable from the list of variables, we decided to randomly select variables as it eliminates any deterministic behaviour and the chance of running into an unfavourable arrangement of variables.



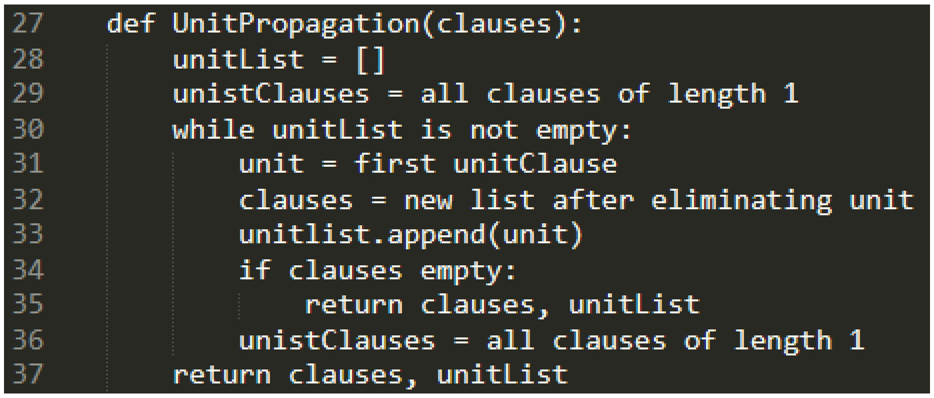
```
def Backtracking(clauses, variableList)
    if all clauses are satisfied:
        return True
    if some clause is not satisfied:
        return False
    literal = randomly select a variable from the clauses
    value = DPLL(clauses, variableList + [literal])
    if not value:
        value = DPLL(clauses, variableList + [-literal])
    return value
```

Figure 1: Backtracking Pseudocode

2.2 - Unit Propagation

The first inference technique we use is Unit Propagation, which is a technique used to simplify a set of clauses. The technique is based on the unit clause, which is defined to be a clause with a single variable. Each unit clause must be assigned such that it is a true literal, or the puzzle is not satisfiable. We iterate through all of the clauses and construct a list of unit clauses. With this, we simplify our sentence by following two rules: every clause containing the unit variable with the same polarity (negation or no negation) as the variable in the unit clause itself can be resolved to true, and every clause containing the unit variable with the opposite polarity must have the variable removed, since there is no possibility of satisfying it with the given assignment.

As an example, we can look at the following sentence: $(a \lor b) \land (-a \lor c) \land (-c \lor d) \land (a)$. The variable a will be detected as a unit clause and the CNF formula can be simplified as a result. The clause $(a \lor b)$ will be removed as a appears positively just as it does in the unit clause. The clause $(-a \lor c)$ will have -a removed as it is false and will not contribute to satisfying its clause. The clause $(-c \lor d)$ would not change as it does not contain the unit variable, and the unit clause itself, (a), will be eliminated.



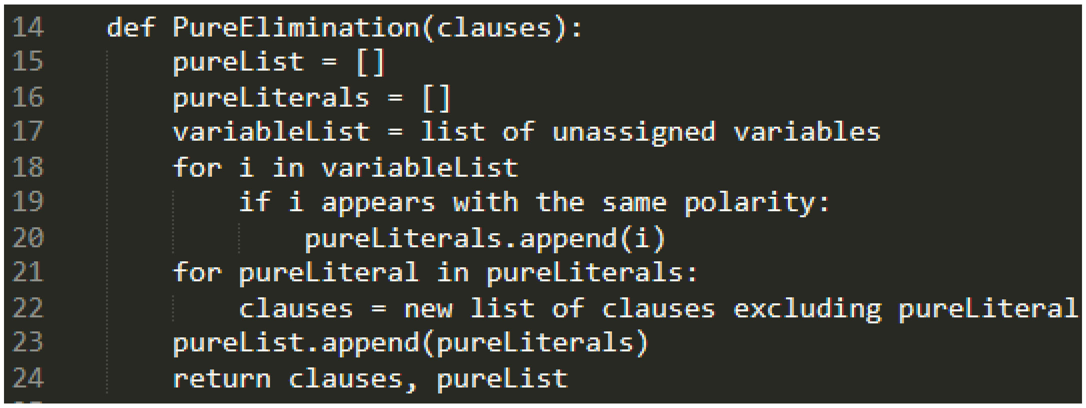
```
def UnitPropagation(clauses):
    unitList = []
    unistClauses = all clauses of length 1
    while unitList is not empty:
        unit = first unitClause
        clauses = new list after eliminating unit
        unitlist.append(unit)
        if clauses empty:
            return clauses, unitList
        unistClauses = all clauses of length 1
    return clauses, unitList
```

Figure 2: Unit Propagation Pseudocode

2.3 - Pure Literal Removal

The next inference technique we use is Pure Literal Removal. A pure literal is defined to be a variable which either always occurs with a negation in front of it, or never occurs with a negation in front of it, or, in other words, a variable which always appears with the same polarity. When a variable is identified as a pure literal, we can simply assign it such that it resolves to a true literal. The algorithm works by iterating through every clause, and selecting variables which have the same polarity, and then adding them to a list of pure literals. After this, we can simply eliminate each clause which contains a variable from the list of pure literals.

As an example, we can look at the following sentence: (-a⋁c⋁-d) ⋀ (a⋁c⋁b) ⋀ (b⋁-c) ⋀ (-d). The algorithm will iterate through each variable, identify any variable which occurs with the same polarity, and add them to the list of pure literals. In this case, the pure literals would be b and d. Any clauses containing these variables will be deleted. The algorithm will delete the clauses (a⋁c⋁b) and (b⋁-c) in the case of b being a pure literal. It will also delete the clauses containing (-a⋁c⋁-d) and (-d) in the case of d being a pure literal.



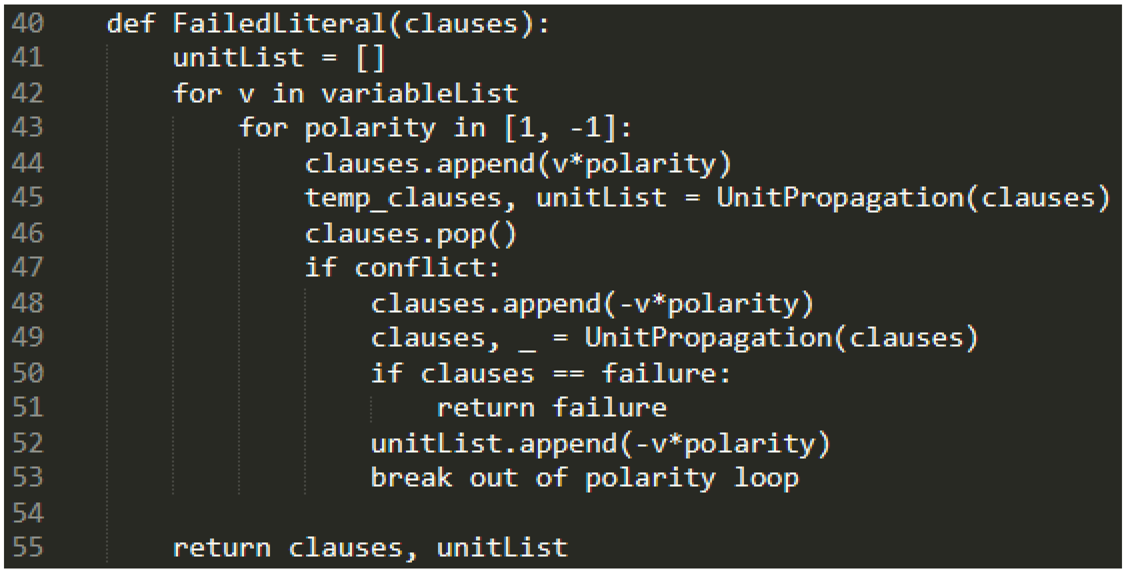
```
def PureElimination(clauses):
    pureList = []
    pureLiterals = []
    variableList = list of unassigned variables
    for i in variableList
        if i appears with the same polarity:
            pureLiterals.append(i)
    for pureLiteral in pureLiterals:
        clauses = new list of clauses excluding pureLiteral
    pureList.append(pureLiterals)
    return clauses, pureList
```

Figure 3: Pure Literal Removal Pseudocode

2.4 - Failed Literal Rule

The next inference technique we use is the Failed Literal Rule. The way this works is that we force an assignment on a variable and run unit propagation on that variable. If the sentence is not satisfiable given this assignment, then we assign the variable the opposite boolean value, and we can conclude that this assignment is necessary as the previous one resulted in unsatisfiability.

As an example, we can look at the following sentence: (-a⋁c⋁-d) ⋀ (a⋁c⋁b) ⋀ (b⋁-c) ⋀ (-d).   The algorithm will initially select -a and turn it into a unit clause, then perform unit propagation. This will yield the new sentence (c⋁b) ⋀ (b⋁-c) ⋀ (-d). We can see that (-a⋁c⋁-d) has been removed as it is satisfiable given the unit clause -a and a has been removed from (a⋁c⋁b) as it cannot be used to satisfy the clause. The same will occur for the next variables in order, which is b, c. d.



```
def FailedLiteral(clauses):
    unitList = []
    for v in variableList
        for polarity in [1, -1]:
            clauses.append(v*polarity)
            temp_clauses, unitList = UnitPropagation(clauses)
            clauses.pop()
            if conflict:
                clauses.append(-v*polarity)
                clauses, _ = UnitPropagation(clauses)
                if clauses == failure:
                    return failure
                unitList.append(-v*polarity)
                break out of polarity loop

    return clauses, unitList
```

Figure 4: Failed Literal Rule Pseudocode

3 Methodology

The main question we will be answering is how effective the algorithm is at solving real sudoku puzzles. To evaluate our algorithm we will look at the running times of the algorithm and the nodes generated by the algorithm with different inference technique combinations. We will use a range of easy puzzles with many spaces filled in, to hard puzzles with few spaces filled in, and observe if the algorithm is able to tackle these puzzles.

The different combinations we will test are: unit propagation, unit propagation + failed literal, pure literal, unit propagation + pure literal, unit propagation + pure literal + failed literal. Unit propagation is needed for failed literal, so they must be used in combination. Pure literal does not rely on any other inference technique, so we can use it independently.

The manner in which we will be comparing easy to hard sudoku puzzles is by removing 5 variables at a time. Each removal of 5 variables will occur in the same positions to allow for equality amongst inference technique combinations. The same arrangement of variables will be used throughout the testing. Once the algorithm takes well over 10 minutes to complete, we terminate the search. We then go back to the previous stage which it passed, and increment the missing variables by 1 per search, in order to get some more meaningful results, until it once again takes well over 10 minutes to complete or passes all of the single increment stages. The single increment sub-stages only have variables removed in the same positions as the next stage.

4 Experimental Setup

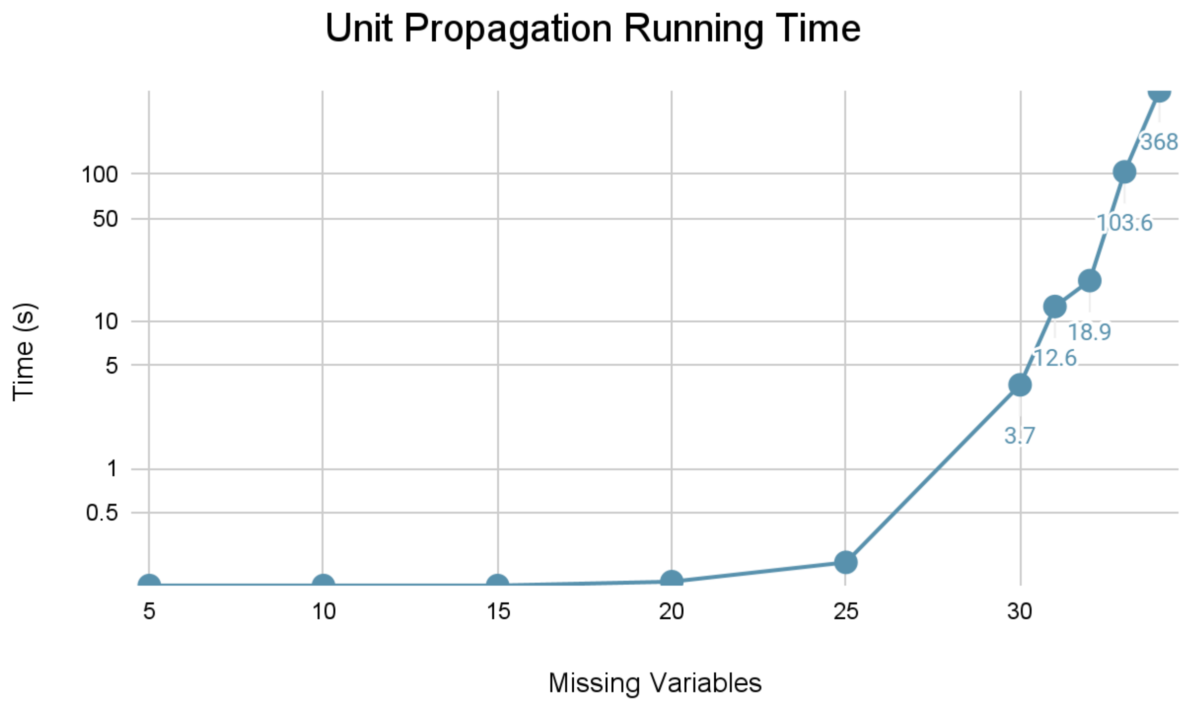
The machine which the program is being run on is the CSIL workstations using the Ubuntu 20.04.2 LTS operating system. It has an i9-9900 16 core @ 3.1GHz processor with 32GB of memory. We're using Python version 3.8 as the only language for this project.

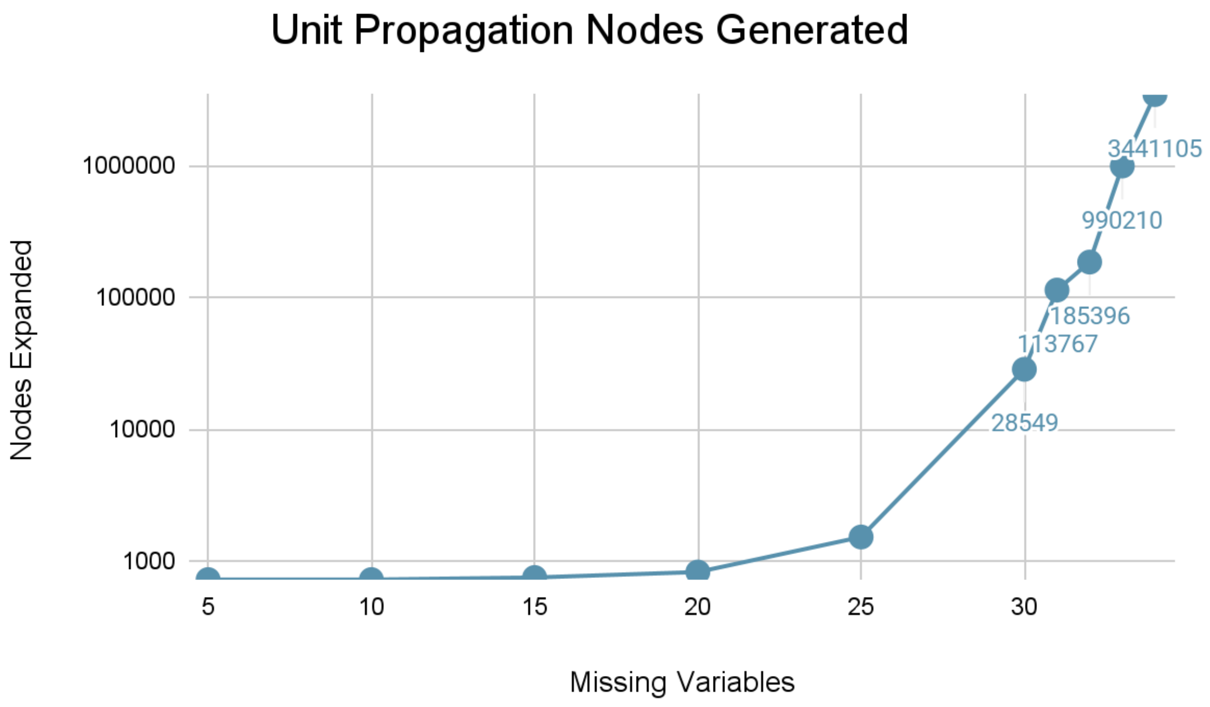
5 Experimental Results

All graphs displayed are logarithmic so smaller data points do not get dwarfed.

5.1 - Unit Propagation

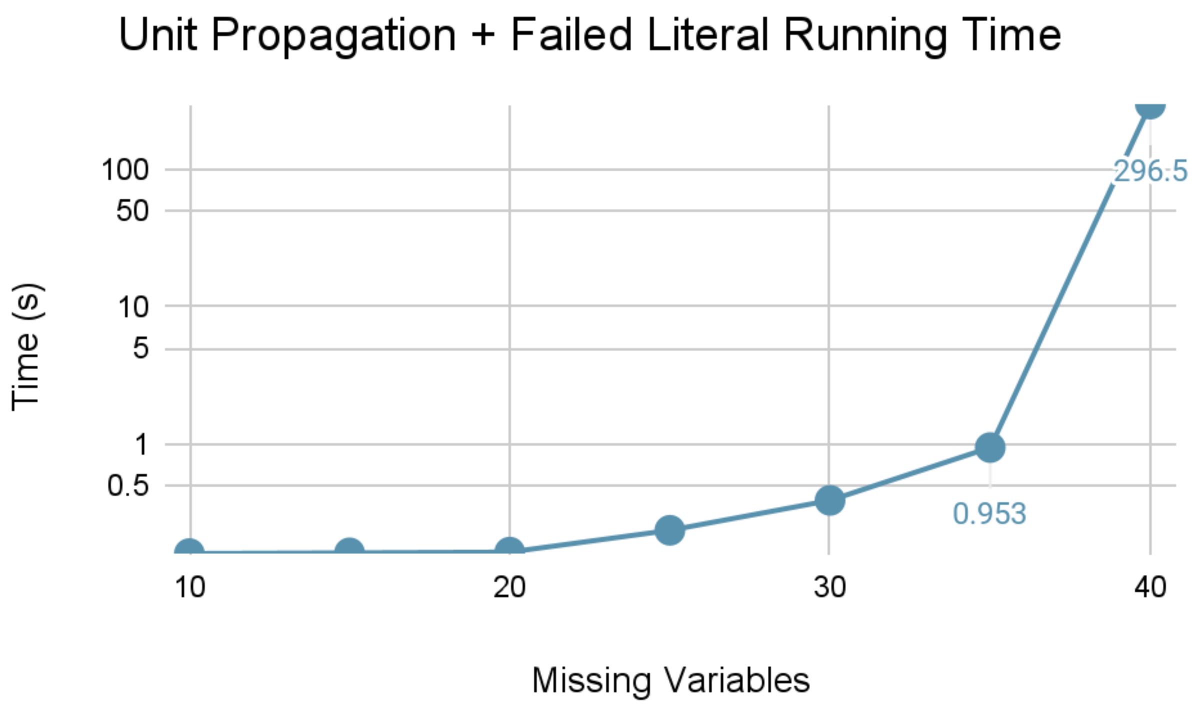
Unit propagation's search took well over 10 minutes to complete when runAny clauses containing these variables will be deleted. with 35 missing variables.

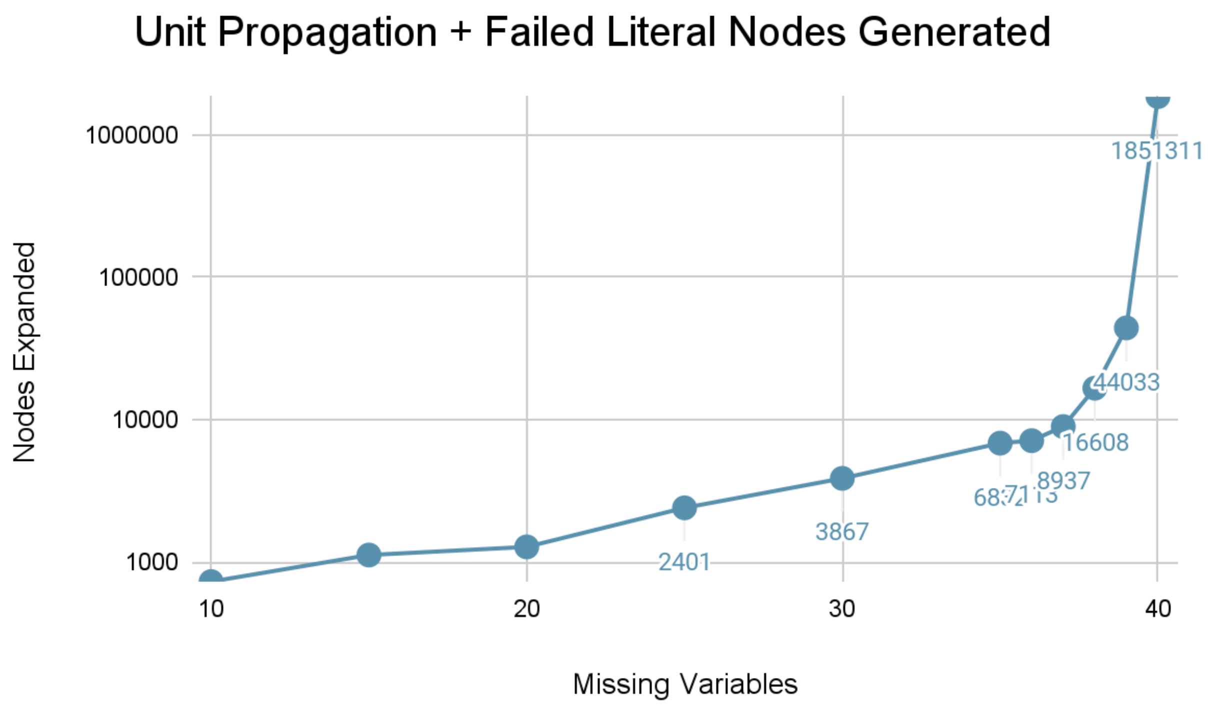




5.2 - Unit Propagation + Failed Literal

Unit propagation + failed literal's search took well over 10 minutes to complete when run with 41 missing variables.

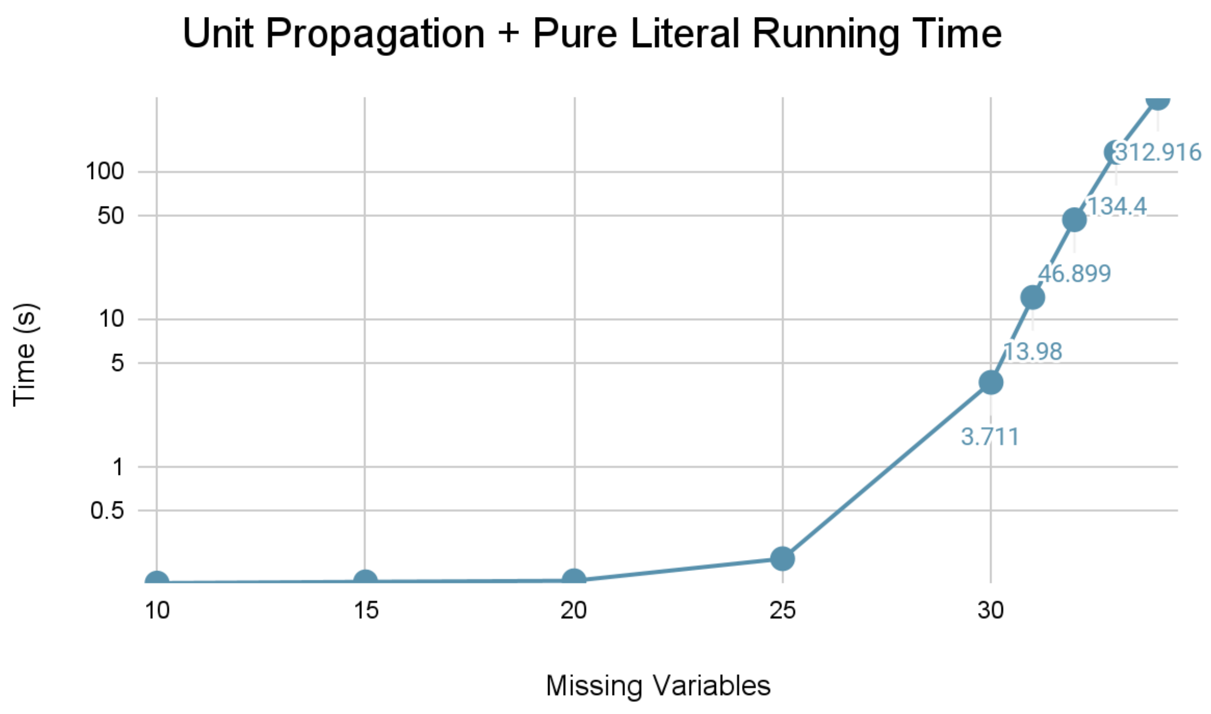




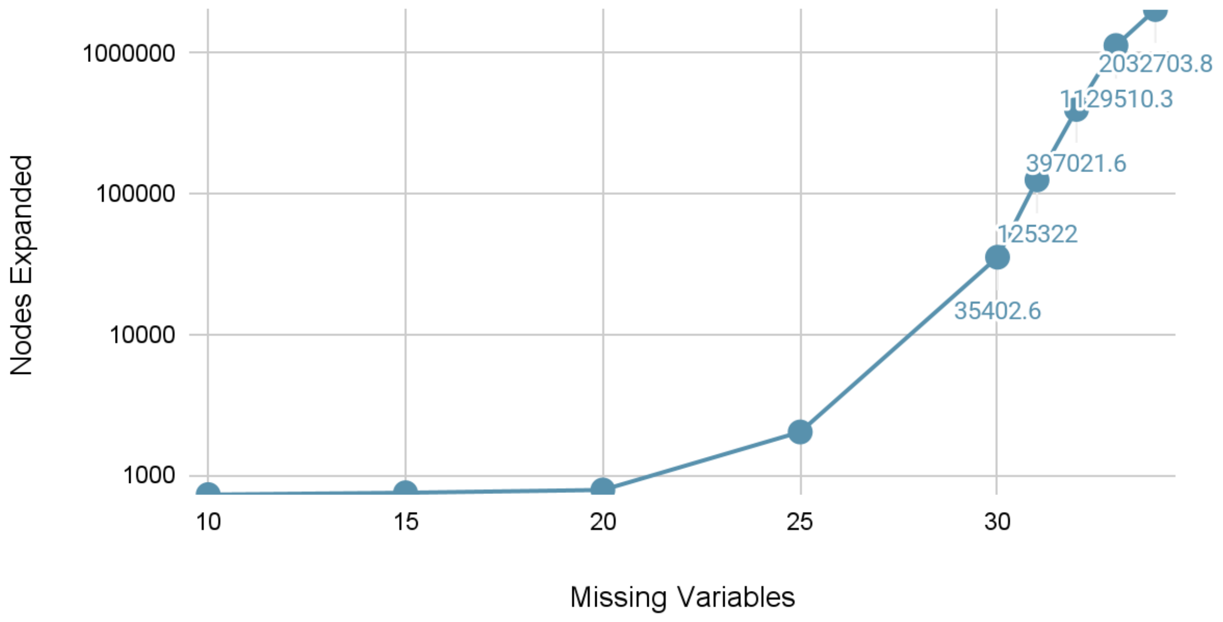
5.3 - Pure Literal

Pure Literal by itself was not able to solve even an instance with 1 variable removed.

5.4 - Pure Literal + Unit Propagation



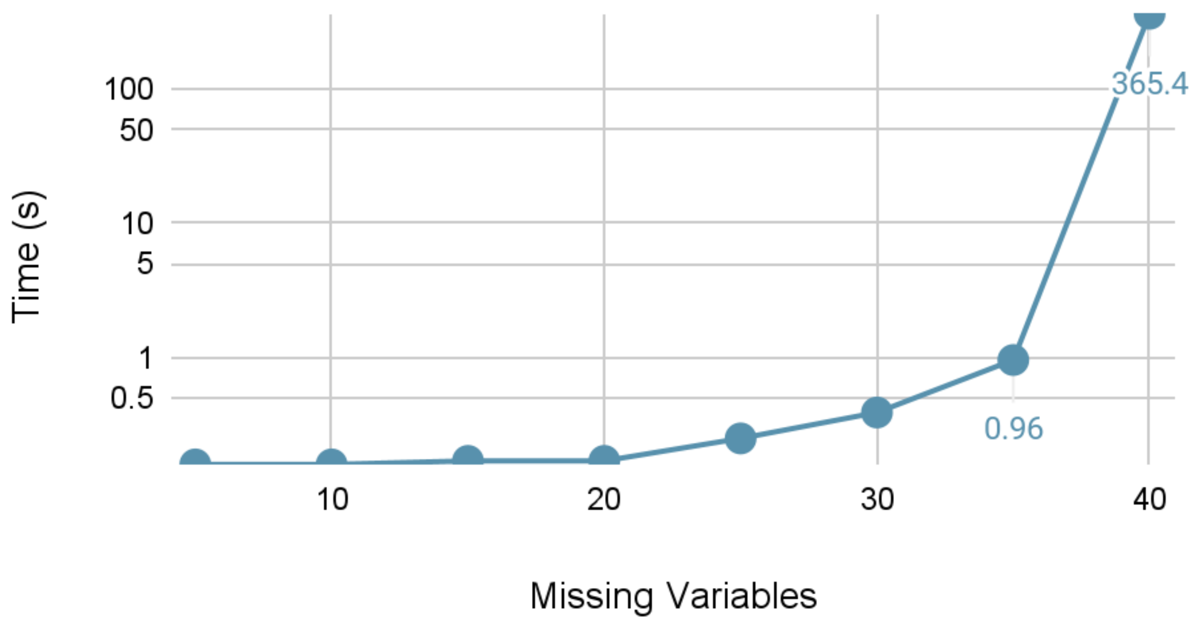
Unit Propagation + Pure Literal Nodes Generated

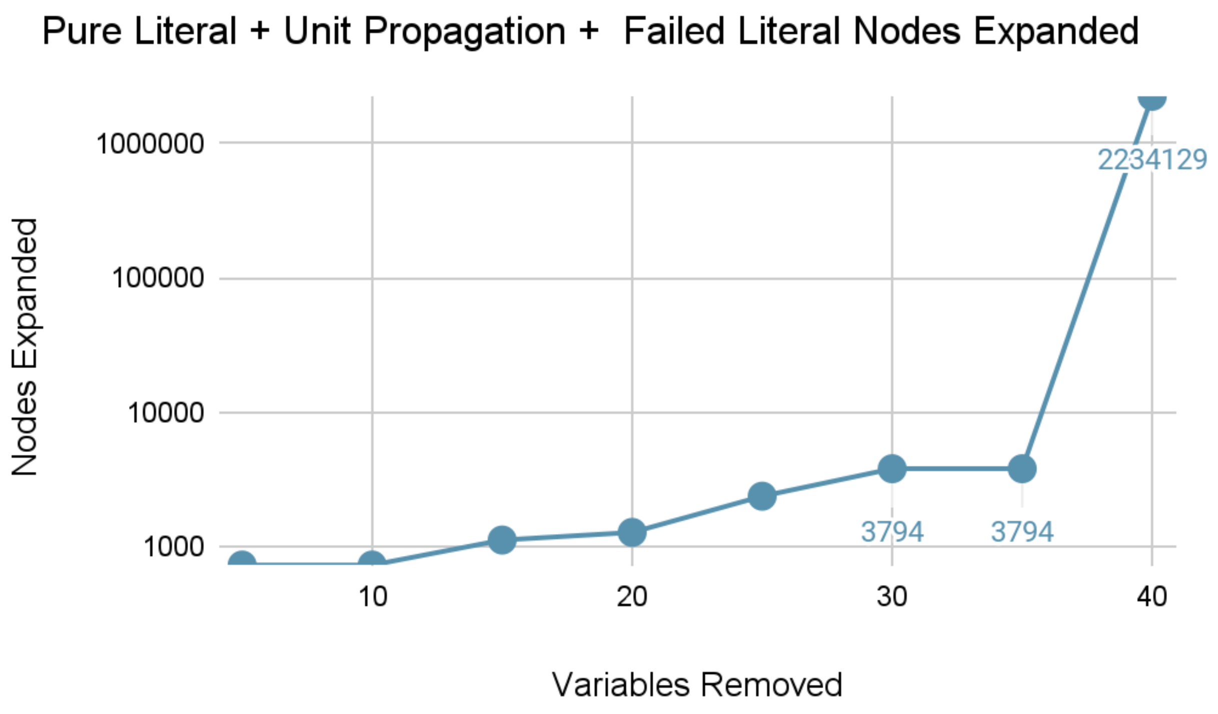


5.5 - Pure Literal + Unit Propagation + Failed Literal

Pure literal + unit propagation + failed literal's search took well over 10 minutes to complete when run with 41 missing variables.

Pure Literal + Unit Propagation + Failed Literal Running Time





6 Conclusion/Results

In summary, different inference technique combinations lead to very different performance. The backtracking algorithm without any inference techniques was not even able to complete a puzzle with a single variable removed due to its brute force nature. When a single variable gets removed, many clauses are still generated within the program, causing backtracking to go down large numbers of arbitrarily deep paths. Similarly, pure literal removal also was not able to. After examining the CNF output for the easy sudoku board, we came to the conclusion that there are little to no pure literals within the file, meaning that running Pure Literal by itself is virtually the same as backtracking alone.

Excluding the above, below are tables displaying inference technique combination's running times and nodes expanded for cases of 30, 35 and 40 variables removed, ordered by their running times.

Running times of inference technique combinations	30	35	40
Unit Propagation + Failed Literal	0.394s	0.953s	296.5s
Unit Propagation + Failed Literal + Pure Literal	0.934s	0.96s	365.4s
Unit Propagation + Pure Literal	3.7s	>> 10min	>> 10min
Unit Propagation	3.7s	>> 10min	>> 10min

Nodes expanded of inference technique combinations	30	35	40
Unit Propagation + Failed Literal	3,867	6,832	1,851,311
Unit Propagation + Failed Literal + Pure Literal	3,794	3,794	2,234,129
Unit Propagation + Pure Literal	35,403	N/A	N/A
Unit Propagation	28,549	N/A	N/A

Looking at these results, we can see that unit propagation + failed literal is the strongest combination. This was initially interesting because it does not use pure literals, however after discovering pure literal's poor performance on sudoku boards, it made sense. By examining the unit propagation vs. unit propagation + pure literal case, we can see that the running time is the same for 30 variables removed, but the latter expands far more nodes.

What is interesting is that not even the strongest combination here is capable of solving a realistic sudoku puzzle. 40 missing variables is only roughly half the board, though typical sudoku puzzles have 50-60 variables removed. This would likely take many, many weeks for unit propagation + pure literal to resolve, when humans can do it in a matter of 10 minutes. The CNF conversion function produces a great number of clauses, many of which we suspect are unnecessary, and likely severely slow down the algorithm. This is an entirely different problem in and of itself[7]. All in all, sudoku solving is very tricky and only with very refined inference techniques, effective combinations, and optimal CNF encoding, can we expect to solve sudoku in a timely fashion.

7 References

1. Paper the report is based on: Sudoku as a SAT Problem
2. Further info on pure literals & backtracking: DPLL algorithm - Wikipedia
3. Further info on unit propagation: Unit propagation - Wikipedia
4. Failed literal: Practical SAT Solving
5. Binary failed literal: SAT and Constraint Satisfaction
6. Sudoku to CNF converter: Solving problems with CNF SAT solvers: The Sudoku example
7. Optimal CNF encoding: Efficient CNF Encoding for Selecting 1 from N Objects

In addition to your individual reports describing your individual contributions to your group, your group will be asked to submit a collective final report for this project. At the very least, your group's final report should contain the following:

1. An introduction section containing a precise mathematical description of the problem being studied.
2. An implementation section containing detailed descriptions of the algorithms your group is using, including pseudocode for each algorithm. This section should document any significant implementation decisions which were made when implementing the algorithms. Did you make any special modifications to optimize the algorithm for the problem instances you are studying?
3. A methodology section where you present the questions which you will be investigating concerning the performance of the algorithms on your problem instances, and describe the experiments you will conduct to answer those questions. What classes of instances will you be using? Make sure to use some challenging instances. If applicable, you may consider using the MAPF Benchmark instances mentioned in the individual project. Make sure to cite your sources if your instances are not hand-crafted. Some examples of questions to investigate are as follows:
    - Does there exist a class of instances on which one algorithm always outperforms another algorithm? Is this true for all instances, or is the other algorithm still sometimes a better choice?
    - How does the performance of your algorithm(s) vary as a function of the number of agents, the percentage of obstacles in instances, the size of the instances, etc? Are there some algorithms whose performance varies

less than others when certain parameters change? Is there one algorithm that is uniformly a horrible choice for this problem?
    - If one of your algorithms sometimes returns suboptimal solutions, how close were those solutions to the optimal solutions? Did there exist a class of instances on which that algorithm was always nearly optimal?
4. An experimental setup section where you describe the environment you are running your experiments on. What language(s) are you using for implementation? What version of the language(s)? What operating system are you using? What sort of processor are you using? How much memory is available?
5. An experimental results section where-in you run your described experiments and display the results to us. You should display your results graphically. For instance, plot the number of solutions expanded by different algorithms, the cost of the solutions found by different algorithms, etc, as a function of the particular instances.
6. A conclusions section where-in you present answers to your questions on the basis of your experimental results. Additionally, what broader takeaways do you have from these experiments? Were there certain features of problem instances that always resulted in poorer performance? Were there certain features of problem instances that always resulted in excellent performance? What might you try next in light of your findings were you to continue to work on this problem?
7. A short bibliography providing citations for the algorithms you are using, and any instances you are using which you did not generate yourself.