41052 Assessment Task 3: Computational Problem

3-SAT

Oliver Salman (13881750)
Zach Clare (13907908)

November 11, 2022

Contents

1 Overview and Problem Definition

Boolean Satisfiability is a type of decision problem that that plays a large role in decision algorithms in computer science. 3-SAT is a subset of the boolean satisfiability problem and is of the form:

$$C_1 \wedge C_2 \wedge ... \wedge C_n \tag{1}$$

Where each C_i is an $\vee$ of three or less variables.

In general, the boolean satisfiability problem is the problem of determining whether there exist an interpretation of variables such that the entire boolean formula is true. That is, whether every variable in C_i can be replaced with either true or false in such a way that the entire formula evaluates to true. If there is an interpretation of these variables, then the boolean formula is satisfiable, and unsatisfiable otherwise.

The 3-SAT problem is proved by the Cook-Levin Theorem to be NP-Complete, meaning there is no deterministic algorithm that can solve the 3-SAT problem in polynomial time. This report will explain how we solved the 3-SAT problem using some well known SAT solving algorithms and frameworks.

2 Definitions

Below are some handy definitions that we will be referring to throughout the report.

- **Conjunctive Normal Form (CNF)**: CNF is a way of representing a boolean expression and is of the form $C_1 \wedge C_2 \wedge ... \wedge C_n$ where each C_i contains literals separated by $\vee$. CNF is often considered the canonical representation for SAT formulas whereby all SAT problems can be reduced to 3-SAT when represented in CNF form.

- **Pure Literal**: A pure literal is a literal that only exists as either only positive or only negative throughout the entire boolean expression. These variables can then be assigned in such a way that makes all clauses containing it true. This has no bearing on the other literal assignments, so pure literals can be removed form the boolean formula, hence reducing the search space.

- **Unit Clause**: A unit clause is a CNF clause containing a single unassigned literal. For the SAT to be satisfisiable, this clause will also need to be satisfiable, thus we can assume that this value must be the valid assignment of the literal.

3 Brute force

As mentioned previously with NP-complete problems, a non-deterministic machine is a formalisation of the idea of a brute force search algorithm. With this in mind, we have decided to implement a naive brute force search algorithm that determines the satisfiability of a boolean expression by checking every possible assignment the clauses can take and enumerating these until a solution is found. It is clear that this approach is inefficient as the search space is exponential to the number of variables in the boolean expression. Though, implementing a brute force search algorithm to decide satisfiability is always a good step in understanding the problem at hand, and finding other methods (i.e heuristics, approximations) to transform or reduce the problem into smaller problems.

We first started by generating all possible assignments that the variables can take, which is a powerset of the number of variables in the expression. For example, a trivial boolean expression with 3 variables will have 8 possible assignments ([0,0,0], [1,0,0], [0,1,0], [0,0,1], [1,1,0], [0,1,1], [1,0,1], [1,1,1]). We then iterate over all literals in all clauses in the boolean expression and try every assignment until we find a solution. The program terminates when either a solution exists, or no solution exists.

Below is pseudocode for our brute force implementation.

```
def brute_force(cnf, n_vars, assignments=[]):
    for i in range(2**n_vars):
        for j in range(n_vars):
            create_all_possible_literal_assignments(i,j)
    for ass in assignments:
        for clause in cnf:
            for lit in clause:
                if clause satisfies assignment:
                    return SAT
                return UNSAT
```

4 DPLL

The second algorithm we implemented is the Davis–Putnam–Logemann–Loveland (DPLL) Algorithm, which is a search algorithm based on backtracking that decides the satisfiability of a boolean expression. Below is pseudocode of the algorithm.

```
function DPLL(cnf)
    while there is a unit clause {l} in cnf do
        cnf = unit-propagate(l, cnf);
    while there is a literal l that occurs pure in cnf do
        cnf = pure-literal-assign(l, cnf);
    if cnf is empty then
        return true;
    if cnf contains an empty clause then
        return false;
    l = choose-literal(cnf);
    return DPLL(cnf AND {l}) or DPLL(cnf AND {not(l)});
```

The algorithm starts off by assigning each literal a truth value, and then simplifies the expression via unit propagation and pure literal elimination. **Pure literal elimination** is the process of removing all literals that only exist with one polarity (either all true or not true). **Unit propagation** is the process of finding literals that exist by themselves in a clause, and removing all other clauses that contain these literals. These two techniques are able to drastically reduce the search space.

It then chooses a branching literal (either naively or with heuristics) which is then used to recurse using a reduced boolean expression. Naively choosing a branching literal may include branching on a random literal in the expression. Other heuristic methods (or branching heuristics) of choosing a branching literal may include conflict driven learning, whereby favouring literals that have fewer conflicts, or choosing literals that occur more frequently. Methods of choosing this branching literal can reduce the number of recursions needed to converge to a decision. There exist instances for which the running time is constant or exponential depending on the choice of branching literals. Though in general, the branching literal is assigned to true (reducing the search space of the boolean expression), and recursing on this reduced expression. If a conflict arises, it chronologically backtracks to the most recent conflict and reassigns it to false and continues its procedure.

This process continues until the algorithm reaches one of two terminating cases. Either the boolean expression is empty, meaning it is satisfied by any assignment so all of its clauses are by default true. The second case is when the expression contains an empty clause, which is false by default as no variables exists to make the clause true.

5 Complexity

Boolean Satisfiability is an NP-Complete decision problem, meaning:

- Its solution can be verified in polynomial time (by a deterministic turing machine) and a brute force algorithm can find the solution to the problem by enumerating all possible solutions.
- Can be used to simulate all other problems that are at most as hard as Boolean Satisifiability.

Below, we will state both the theoretical and practical complexity of 3-SAT.

5.1 Theoretical

For the complexity of our brute force algorithm, we first need to figure out what the complexity of the CNF we're trying to solve. Our CNF has a number of clauses c, over n boolean variables, which can either be true or false - we can determine that the total number of possibilities to be tested is 2^n, therefore our brute force algorithm has a theoretical complexity of O(2^n). To store these possibilities, we also require a space complexity of also $O(2^n)$.

As for DPLL, it has a theoretical best case time complexity of $\Omega(1)$, and a worse case of $\tilde{O}(2^n)$, where n is the number of variables included in the SAT problem. It's worst case is the same as the brute force complexity as it is an NP-complete problem. As for it's space complexity, it is $O(n)$.

5.2 Experimental

The time complexity of our brute force algorithm starts off by creating all possible truth assignments for the powerset of all variables, which is $O(2^n)$. It also iterates through every literal in every clause checking if each assignment will satisfy, taking an additional $O(n^2)$ time. This leaves us with an overall time complexity of $O(2^n)$ where n is the number of variables in the problem.

For the space complexity of brute force, as it tests every possible option, for each variable n that has 2 possible values true or false, we have a space complexity of $O(2^n)$.

To model the time complexity of our DPLL algorithm implementation correctly, we can utilise the Master Theorem for decreasing functions. First, we mark the size of our problem n as n, as worst case for n is the amount of variables we have. We can then set our a to 2, as the

count of the subproblems for each step is 2 for each literal, being true or false. As for the size of our subproblems, they will be $n - l$, where l is a literal. Thus, we can model our problem fully as $T(n) = 2T(n - l) + n$. We can then use the master theorem to reduce this, giving us the final complexity of $\theta(n^{\frac{n}{l}} * n)$.

As for space complexity, we start with the initial size of $O(n)$. We than also generate a sat map to help with heuristics, which has a worst case space complexity of $O(n * m)$, as it generates a map of what clauses each literal is in which may include duplicates, leading to its increased size over the problem size. Finally, we generate a dictionary of literal assignments, which has a space complexity of $O(n)$. Thus, our total space complexity for our implementation is $O(n * m)$.

6 Tests

To test our implementations, we found it a bit difficult as there aren't really any unit tests that we can test against, as the result is either true or false. To tackle this, we utilised a pre-existing SAT Solver library, and used recursive randomised testing to ensure that our algorithm had perfect accuracy across all cases. This involved generating a number of test cases - we used 1000 SAT cases and 1000 UNSAT cases, totalling 2000 total cases. We then iterated through each of these random cases, and compared the results of our algorithm with the result from the SAT library. If any of these failed, we'd be able to run through the algorithm using debugging print statements and breakpoints to diagnose and solve the issue.

We've since done extensive testing to ensure that no matter the amount of clauses or variables, we can determine its satisfiability, albeit dependent on time for the larger cases.