SAT Solver: Concepts and Code Walkthrough

September 30, 2024

Abstract

This report provides a comprehensive explanation of the SAT (Boolean Satisfiability) problem, SAT solvers, and a detailed walkthrough of a DPLL-based SAT solver implemented in C++. The report explains the SAT problem, the DPLL algorithm, and relevant code sections from the provided implementation. Performance considerations and potential optimizations are discussed as well.

1 Introduction

The Boolean Satisfiability Problem (SAT) is a decision problem where, given a Boolean formula, we need to determine if there exists an assignment of truth values to variables that makes the formula true. SAT was the first problem proven to be NP-complete, and it serves as a cornerstone in complexity theory. SAT solvers are widely used in areas such as formal verification, automated reasoning, and planning.

2 Boolean Satisfiability Problem

In the SAT problem, a formula is typically expressed in Conjunctive Normal Form (CNF), where a formula is a conjunction (AND) of clauses, and each clause is a disjunction (OR) of literals. For example, the formula:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \tag{1}$$

has two clauses. The goal is to assign truth values to the variables x_1, x_2, x_3 such that the formula evaluates to true.

3 Applications of SAT Solvers

SAT solvers are used in many real-world applications, including:

- **Formal verification:** Verifying the correctness of hardware and software systems.
- **Cryptography:** Attacking cryptographic primitives.
- **Planning and scheduling:** Solving constraint satisfaction problems in AI.

4 The DPLL Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm is a backtracking-based search algorithm for solving SAT problems. The core steps include:

- **Unit propagation:** If a clause is a unit clause (containing only one literal), that literal must be true.
- **Pure literal elimination:** If a literal appears with only one polarity (always positive or negative), it must be assigned to satisfy the formula.
- **Backtracking:** If the current assignment cannot satisfy the formula, the algorithm backtracks and tries a different assignment.

5 Code Walkthrough: DPLL Solver in C++

The provided implementation of the SAT solver follows the DPLL algorithm. Below, we break down the key components of the code.

5.1 Main Function

The main function of the code initializes the input and calls the DPLL solver.

```
int main() {
    vector<vector<int>> clauses = {
        {1, -3, 4},
        {-1, 2},
        {-2, -4}
    };
    DPLLSolver solver;
    if (solver.solve(clauses, 4)) {
        cout << "SATISFIABLE" << endl;
    } else {
        cout << "UNSATISFIABLE" << endl;
    }
    return 0;
}
```

This example creates a set of clauses and invokes the DPLL solver. If the solver finds a satisfying assignment, it prints "SATISFIABLE", otherwise, it prints "UNSATISFIABLE".

5.2 DPLL Function

The core of the SAT solver is the DPLL function, which performs recursive backtracking to solve the SAT instance.

```
bool DPLL(vector<vector<int>> clauses, unordered_map<int, bool>& assignment
    if (clauses.empty()) {
        return true;
    }
```

```
    for (const auto& clause : clauses) {
        if (clause.empty()) {
            return false;
        }
    }

    for (const auto& clause : clauses) {
        if (clause.size() == 1) {
            int unit = clause[0];
            bool value = (unit > 0);
            assignment[abs(unit)] = value;

            return DPLL(simplify(clauses, unit), assignment, numVars);
        }
    }

    int variable = chooseVariable(clauses, numVars);
    assignment[variable] = true;
    if (DPLL(simplify(clauses, variable), assignment, numVars)) {
        return true;
    }
    assignment[variable] = false;
    if (DPLL(simplify(clauses, -variable), assignment, numVars)) {
        return true;
    }

    return false;
}
```

The DPLL function first checks for base cases: if all clauses are satisfied, it returns true, and if any clause is unsatisfied, it returns false. The function then applies unit propagation and attempts to assign truth values to variables, backtracking when necessary.

5.3 Simplification of Clauses

The ‘simplify‘ function removes satisfied clauses and updates others based on the current assignment of literals.

```
vector<vector<int>> simplify(const vector<vector<int>>& clauses, int litera
    vector<vector<int>> simplified;

    for (const auto& clause : clauses) {
        if (find(clause.begin(), clause.end(), literal) != clause.end()) {
            continue;
        }
        vector<int> new_clause;
        for (int lit : clause) {
            if (lit != -literal) {
```

```
                new_clause.push_back(lit);
            }
        }
        simplified.push_back(new_clause);
    }
    return simplified;
}
```

6 Optimizations and Heuristics

Although the basic DPLL algorithm works, various heuristics can improve its efficiency. For example, choosing the variable with the highest frequency in unsatisfied clauses can reduce the number of backtracking steps.

7 Conclusion

This report has provided a detailed overview of the SAT problem, the DPLL algorithm, and a code-based implementation of a SAT solver. SAT solvers are crucial in many domains, and their continued optimization is a subject of ongoing research.