DPLL Algorithm Implementation Report

Yujun Kim

May 20, 2024

Abstract

This report details the implementation of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for solving SAT problems.

1 Introduction

The DPLL algorithm is a backtracking-based search algorithm used to determine the satisfiability of propositional logic formulas in Conjunctive Normal Form (CNF). It extends the Davis-Putnam algorithm by incorporating key techniques such as unit propagation, pure literal elimination, and clause learning. This report outlines the implementation of the DPLL algorithm, the design decisions involved, and the rationale behind these decisions.

2 Implementation Details

The implementation is structured into several key functions, each responsible for different aspects of the DPLL algorithm. The main function orchestrates the process by calling these helper functions.

2.1 Data Structure

In implementation of the algorithm, selecting proper data structure is crucial in both completion and the performance of the algorithm.

- **Variables**. Each variable is parsed as distinct natural number, as imposed from the input structure.
- **Literal, clause, and formula.** Literal is parsed as an integer. Clause is parsed as a list of integer. Formula is parsed as a list of clause. i.e. list of list of integers.
- `v_dict`: Saves the assignment of each variables and keep track whether the assignment is implied or decided. Its type is a `Dict[Int, (Bool, Int)]`. Given a variable `x`, `v_dict(x)` gives tuple of boolean and integer, where boolean indicates whether the assignment is true or false. The integer part indicates whether the assignment of the variable is decision assignment or implied assignment. If the assignment is implied by `i`-th clause, the corresponding integer is `i`. If the assignment is decided, then the corresponding value is `None`
- `v_order`: Keeps track of order of assignment. Its type is `List(int)`. It is necessary in backtracking and clause learning.

2.2 Reading Input and Parsing Output

- `read_input` function reads a CNF formula from a file and parses it into a list of clauses. Moreover, it also returns number of variables and number of clauses in the formula.
- `parse_assignment` function takes `s_bool, v_dict` as input and parse it to the required output format. `s_bool` is simply a boolean indicating satisfiability of the formula, and `v_dict` follows above description.

2.3 Main DPLL Function

The `DPLL` function initializes the assignment and iteratively performs unit propagation, clause learning, and simplification until it finds a solution or determines that the formula is unsatisfiable.

```
def DPLL(nvar, nclause, formula):
    s_bool = False
    v_dict = dict()
    v_order = []
    while True:
        copied_formula_for_unit_propagate = [clause[:] for clause in formula]
        copied_formula_for_simplify = [clause[:] for clause in formula]
        v_dict, v_order = unit_propagate(copied_formula_for_unit_propagate, v_dict, v_or
        simplified_formula = simplify(copied_formula_for_simplify, v_dict)
        if len(simplified_formula) == 0:
            return True, v_dict
        elif [] in simplified_formula:
            copied_formula_for_clause_learning = [clause[:] for clause in formula]
            learned_clause = clause_learning(copied_formula_for_clause_learning, v_dict,
            formula.append(learned_clause)
            if len(learned_clause) == 0:
                return False, v_dict
            else:
                v_dict, v_order = backtrack(v_dict, v_order, learned_clause)
        else:
            v_dict, v_order = dumb_decision_strategy(nvar, v_dict, v_order)
    return s_bool, v_dict
```

2.4 Supporting Functions

Several supporting functions are used within the DPLL function to handle unit propagation, clause learning, simplification, and decision strategies. These include:

- `unit_propagate`: Simplifies the formula by assigning values to variables that appear as single literals in any clause.
- `clause_learning`: Identifies a conflict and learns a new clause to prevent the same conflict from occurring again. Uses `v_order` to backtrace.
- `simplify`: Removes literals and clauses based on the current variable assignments.
- `dumb_decision_strategy`: Makes a decision by assigning a value to the first unassigned variable.

3 Design Decisions and Rationale

The primary goal of the implementation was to create a clear and modular DPLL algorithm that can be easily extended with additional optimizations. Several design decisions were made to achieve this:

3.1 Unit Propagation

Unit propagation is a critical step in the DPLL algorithm as it simplifies the formula significantly before making any decisions. By propagating units early, the search space is reduced, making the algorithm more efficient.

```
def unit_propagate(formula, v_dict, v_order):
    formula = simplify_without_deleting_clause(formula, v_dict)
    length_list = [len(clause) for clause in formula]
    while 1 in length_list:
        idx = length_list.index(1)
        L = formula[idx][0]
```

```
        v_dict[abs(L)] = (L > 0, idx)
        v_order.append(abs(L))
        formula = simplify_without_deleting_clause(formula, v_dict)
        length_list = [len(clause) for clause in formula]
    return v_dict, v_order
```

3.2 Clause Learning

Clause learning is implemented to prevent the algorithm from revisiting the same conflicts, thus pruning the search space. This optimization significantly improves performance on difficult SAT instances.

```
def clause_learning(formula, v_dict, v_order):
    D = formula[get_conflict_idx(formula, v_dict)]
    for p in v_order[::-1]:
        value = v_dict[p]
        assign, implied = value
        if implied is None or not variable_is_in_clause(p, D):
            continue
        else:
            D = resolve_p(formula[implied], D, p)
    return D
```

3.3 Simplification

Simplification removes clauses that are already satisfied and literals that are false, keeping the formula as small as possible. This helps in speeding up the subsequent steps of the algorithm.

```
def simplify(formula, v_dict):
    for i in range(len(formula))[::-1]:
        clause = formula[i]
        for j in range(len(clause))[::-1]:
            literal = clause[j]
            literal_eval = eval_literal(literal, v_dict)
            if literal_eval == 1:
                formula.pop(i)
                break
            elif literal_eval == 0:
                formula[i].pop(j)
    return formula
```

3.4 Decision Strategy

Implementation of decision strategy follows code below. As the name of the function suggests, it follows simple heuristic: Assign true to first undecided variable. This part clearly has a possibility to be optimized further by using better heuristics.

```
def dumb_decision_strategy(nvar, v_dict, v_order):
    for i in range(1, nvar + 1):
        try:
            v_dict[i]
        except KeyError:
            v_dict[i] = (True, None)
            v_order.append(i)
            return v_dict, v_order
    assert False, "All variables already assigned"
```

4 Possible Optimizations and Their Analysis

4.1 Challenging Points

Copying lists. During the algorithm, the formula implemented as list of list is copied several times. This is to prevent original formula to be changed when not intended. However, this may cause a drawback of performance, especially in the aspect of the memory.

4.2 Analysis

Comparision to Brute-Force Search. To compare the efficiency of DPLL compared to brute-force search, number of iteration of loop has been counted. Brute-force searching requires at most 2^k number of iterations where k is the number of variables. A provided satisfiable CNF formula with 544 variables required 737 iterations and another satisfiable CNF formula with 60 variables required 637 iterations. Both of cases are extremly efficient compared to the brute-force search.

5 Conclusion

The DPLL algorithm implementation described in this report incorporates key optimizations such as unit propagation, clause learning, and simplification. These choices were driven by the need to balance complexity and performance. The experimental results confirm that these optimizations significantly enhance the algorithm's efficiency, making it well-suited for solving SAT problems.