

Parallel Portfolio SAT Solver

Abraham White

April 14, 2020

Contents

1	Description	1
2	Implementation	2
2.1	Serial	2
2.1.1	Literal Elimination	4
2.1.2	Unit Propagation	4
2.1.3	Branching	4
2.1.4	Branching Rules	5
2.1.5	Assigning literals and removing satisfied clauses	7
2.2	Parallel	8
3	Build and Run	9
4	Performance	10
5	Appendix	10
5.1	Helper Code	10
5.1.1	Read Input	10
5.1.2	Branching Helpers	10
5.1.3	Full Source	11

1 Description

The Boolean Satisfiability Problem (SAT) is an NP-complete decision problem, where the goal is to find whether there is an assignment of values to boolean variables in a propositional formula such that the formula evaluates to true. For example, determining whether

$$a \wedge \neg a$$

can be true for some value of a is an SAT problem.

Most SAT solvers take their input as a propositional logic formula in conjunctive normal form (CNF), involving variables and the operators *negation* (\neg), *disjunction* (\vee), and *conjunction* (\wedge). A propositional logic formula in CNF is the conjunction of a set of clauses. A *clause* is a disjunction of literals, and a *literal* is a boolean variable A which can be either positive (A), or negative ($\neg A$).

An *interpretation* is a mapping from a CNF formula to the set of truth values $\{\top, \perp\}$ through assignment of truth values to the literals in the formul. SAT solvers use *partial interpretations*, where only some of the literals in the formula are assigned truth values. These variables are replaced with their truth values in the formula and the formula is then simplified using the rules of propositional logic.

The Boolean Satisfiability Problem is the question of whether there exists an interpretation for a formula such that the formula evaluates to \top under this interpretation.

There are two types of SAT solvers: complete, and stochastic. Complete solvers attempt to either find a solution, or show that no solutions exist. Stochastic solvers cannot prove that a formula is unsolvable, but can find solutions for specific kinds of problems very quickly. We are attempting to build a complete SAT solver.

Many modern complete SAT solvers are based on a branch and backtracking algorithm called Davis-Putnam-Logemann-Loveland (DPLL), a refinement of the earlier Davis-Putnam algorithm and introduced in 1962 by Martin Davis, George Logemann, and Donald W. Loveland. Many of these solvers add additional heuristics on top of the DPLL algorithm, which can increase efficiency, but adds significant complexity to the implementation.

Parallel SAT solvers come in two variants, *portfolio* and *divide-and-conquer*. A portfolio solver uses many different serial solvers, all racing to complete a single formula. This strategy is competitive because different formulas may benefit from different choices in branching strategy, but determining this from the beginning would be difficult.

A divide-and-conquer solver splits the search space, sending part of the problem to identical serial solvers. These solvers communicate back to the main process, sending learned variables or conflict clauses.

2 Implementation

We will implement a portfolio SAT solver, which uses a basic DPLL based serial solver with varying branching strategies.

2.1 Serial

Algorithm 1 The recursive DPLL algorithm

```

1: function DPLL( $F$  : Formula)
2:   if  $F$  is empty then
3:     return SAT
4:   else if  $F$  contains an empty clause then
5:     return UNSAT
6:   else if  $F$  contains a pure literal  $l$  then
7:     return DPLL( $F[l \rightarrow \top]$ )
8:   else if  $F$  contains a unit clause  $[l]$  then
9:     return DPLL( $F[l \rightarrow \top]$ )
10:  else
11:    let  $l$  be a literal in  $F$ 
12:    if DPLL( $F[l \rightarrow \top]$ ) = SAT then
13:      return SAT
14:    else
15:      return DPLL( $F[l \rightarrow \perp]$ )
16:    end if
17:  end if
18: end function

```

The basic DPLL algorithm can be defined recursively as in Algorithm 1. In the algorithm, $F[l \rightarrow \top]$ denotes the formula obtained by replacing the literal l with \top and $\neg l$ with \perp in F . A literal is pure if it occurs in F but its opposite does not. A clause is unit if it contains only one literal.

The DPLL algorithm consists of two key steps:

1. **Literal Elimination:** If some literal is only seen in pure form, we can immediately determine the truth value for that literal. For instance, if the literal is in the form A , we know that A must be \top , and if the literal is in the form $\neg A$, A must be \perp . This step occurs on line 6 of the recursive algorithm.
2. **Unit Propagation:** If there is a unit clause then we can immediately assign a truth value in the same way we do for literal elimination. This is done on line 8 of the recursive algorithm.

For both the DPLL and CDCL algorithms, we will take our input in conjunctive normal form. For implementation, we represent literals as integers, with a negative integer being the logical negation of the corresponding positive literal. Clauses are represented by a list of these integer literals, and a formula is represented by a list of clauses. We exclude 0 from the possible literals. For instance, we can encode the formula $(A \vee \neg B \vee \neg C) \wedge (\neg D \vee E \vee F)$ with

```
std::vector<std::vector<int>> formula = {{1, -2, -3}, {-4, 5, 6}};
```

Now we begin the implementation of the recursive DPLL algorithm in C++. Since C++ doesn't support tail-recursive calls, we have to transform the recursive algorithm into a mostly iterative one.

First we set up a data structure to keep track of the assignment of truth values, and another to keep track of the clauses a literal appears in and allow identifying pure literals.

First we define a data structure to keep track of the formula. We define an adjacency list to associate literals with clauses that reference them using an unordered map in the `literals` variable. The `LitData` struct is used to keep track of the clauses where a literal occurs positively (x), or negatively ($\neg x$). This also allows us to easily identify pure literals. We keep a list of clauses in the `clauses` variable, an array of the `ClauseData` struct. This structure has an adjacency list associating the clause to its member literals, and tracks the number of literals assigned true or false which we can use to tell whether the clause is satisfied, unsatisfied, or unit. We keep a running tally of the number of clauses that still need to be satisfied with the `remaining` variable.

```
struct Formula {
    struct ClauseData {
        int n_t = 0;
        int n_f = 0;
        std::vector<int> literals;
        int orig_len;
        bool sat() { return n_t >= 1; }
        bool unsat() { return n_f == orig_len; }
        bool unit() { return n_t == 0 && n_f == (orig_len - 1); }
    };
    struct LitData {
        int assn = -1;
        std::vector<int> pos_clauses;
        std::vector<int> neg_clauses;
        bool pure() {
            return assn == -1 && (pos_clauses.size() == 0 || neg_clauses.size() == 0);
        }
    };
    std::vector<ClauseData> clauses;
    std::unordered_map<int, LitData> literals;
    int remaining;
    void add_literal(int, int);
    Formula(std::vector<int>);
};
```

Now we implement the recursive DPLL algorithm. The algorithm itself is simple, but the helper functions will be more complicated. The literal elimination step is on lines 3 and 4, unit propagation on line 9, and the branching step on lines 14-22. We check for termination because of empty formula on line 11, and termination because of empty clause on line 8. The terminate function is called at the beginning of each loop to check if the algorithm should terminate because of an external factor, e.g. another process solving the formula.

```
1  std::tuple<bool, Formula> dpll(Formula& f, BranchRule rule, std::function<bool()> terminate) {
2      if (terminate()) return {false, f};
3
4      for (auto&& [_f, l] : f.literals)
5          if (l.pure()) pure_literal_assign(f, l);
6
7      for (auto& c : f.clauses) {
8          if (c.sat()) continue;
9          if (c.literals.size() == 0) return {false, f};
10         if (c.unit())
11             if (!unit_propagate(f, c)) return {false, f};
12     }
13
14     if (f.remaining == 0) return {true, f};
```

```

15
16     auto l = get_branching_variable(f, rule);
17     Formula oldf(f);
18     set_var(f, l);
19     auto [res, ff] = dp11(f, rule, terminate);
20     if (res) return {res, ff};
21
22     f = oldf;
23     set_var(f, -l);
24     return dp11(f, rule, terminate);
25 }

```

2.1.1 Literal Elimination

First we handle the pure literal step, which removes whole clauses from consideration by assigning truth values. In the `pure_literal_assign` function, we determine the sign of the literal by the clauses it is contained in, since the map removes that information from the key. We then make a truth assignment. Finally, we update the associated clauses, removing satisfied clauses from the adjacency lists of other literals, since once the clause has a single truth assignment the whole clause can be considered true.

```

void pure_literal_assign(Formula& f, Formula::LitData& data) {
    auto pos_size = data.pos_clauses.size();
    auto s = (pos_size == 0) ? -1 : 1;
    auto lclauses = (s == 1) ? data.pos_clauses : data.neg_clauses;
    data.assn = (s == 1) ? 1 : 0;
    for (auto cidx : lclauses) remove_satisfied(f, cidx);
}

```

2.1.2 Unit Propagation

The next loop in the dp11 implementation helps with unit propagation. We skip over clauses that have already been satisfied, terminating when we have a clause that is empty, i.e. there was a conflicting literal assignment. We call the `unit_propagate` function when the clause is unit, which simply creates a truth assignment for the only literal in the clause. We return the result of `set_var` because a clause may become empty as a result of the unit propagation.

```

bool unit_propagate(Formula& f, Formula::ClauseData clause) {
    return set_var(f, clause.literals[0]);
}

```

2.1.3 Branching

Back in the dp11 implementation, we check if there are any remaining undetermined clauses, returning true if we have satisfied all. Finally, we pick a variable using a heuristic and branch, backtracking if the first choice of assignment doesn't work. For this we use the `get_branching_variable` function to determine a branching variable using a heuristic, and the `set_var` function to handle changing the formula.

Since we may have more processors than branching rules, we also define a branching strategy which randomly chooses a literal out of the unassigned literals.

```

enum class BranchRule { dlis, dlcs, jw, jw2, dsj, rand };

std::string branch_rule_name(BranchRule rule) {
    switch (rule) {
        case BranchRule::dlis:
            return "dlis";
        case BranchRule::dlcs:
            return "dlcs";
        case BranchRule::jw:
            return "jw";
    }
}

```

```

    case BranchRule::jw2:
        return "jw2";
    case BranchRule::dsj:
        return "dsj";
    case BranchRule::rand:
        return "rand";
}
throw std::runtime_error("branch_rule_name didn't handle all cases");
}

int get_branching_variable(Formula f, BranchRule rule) {
    int curr = 0;
    switch (rule) {
        case BranchRule::dlis:
            curr = apply_rule(f, &dlis);
            break;
        case BranchRule::dlcs:
            curr = apply_rule(f, &dlcs);
            break;
        case BranchRule::jw:
            curr = apply_rule(f, &jw);
            break;
        case BranchRule::jw2:
            curr = apply_rule(f, &jw2);
            break;
        case BranchRule::dsj:
            curr = apply_rule(f, &dsj);
            break;
        case BranchRule::rand: {
            bool positive = std::rand() % 2;
            int i = 0;
            long unsigned int iter = 0;
            do {
                i = 1 + std::rand() % f.literals.size();
                iter++;
            } while (f.literals[i].assn != -1 && iter < f.literals.size());
            if (i == 0) throw std::runtime_error("random branch failed");
            curr = (positive ? i : -i);
            break;
        }
        default:
            throw std::runtime_error("get_branching_variable didn't handle all cases");
    }
    if (curr == 0)
        throw std::runtime_error("branching heuristic failed: " + branch_rule_name(rule));
    return curr;
}

BranchRule branch_rule_int(int i) {
    return (i > static_cast<int>(BranchRule::rand)) ? BranchRule::rand : static_cast<BranchRule>(i);
}

```

2.1.4 Branching Rules

Branching rules are used for choosing which literal to set to true during the last step of the DPLL algorithm. These are typically based on heuristics, and various strategies have been formalized in papers over the years. Ouyang [1] created a paradigm which associates with each literal u a weight $w(F, u)$, and then chooses a function Φ of two variables:

- Find a variable x that maximizes $\Phi(w(F, x), w(F, \neg x))$; choose x if $w(F, x) \geq w(F, \neg x)$, choosing $\neg x$ otherwise. Ties in the case that more than one variable maximizes Φ are broken by some rule.

Usually $w(F, u)$ is defined in terms of the number of clauses of length k in F that contain the literal u , denoted $d_k(F, u)$. A selection of some branching rules follow:

1. Dynamic Largest Individual Sum (DLIS)

$$w(F, u) = \sum_k d_k(F, u)$$

$$\Phi(x, y) = \max\{x, y\}$$

Notice that $\sum_k d_k(F, u)$ is simply the number of clauses in which u is present, since k can range from 1 to ∞ .

```
auto dlis(Formula f, int l) {
    int wp = nclauses(f, -1, l);
    int wn = nclauses(f, -1, -l);
    return std::make_tuple(wp, wn, std::max(wp, wn));
}
```

2. Dynamic Largest Combined Sum (DLCS)

$$w(F, u) = \sum_k d_k(F, u)$$

$$\Phi(x, y) = x + y$$

```
auto dlcs(Formula f, int l) {
    int wp = nclauses(f, -1, l);
    int wn = nclauses(f, -1, -l);
    return std::make_tuple(wp, wn, wp + wn);
}
```

3. Jeroslow-Wang (JW) rule

$$w(F, u) = \sum_k 2^{-k} d_k(F, u)$$

$$\Phi(x, y) = \max\{x, y\}$$

```
auto jw(Formula f, int l) {
    auto largest_k = get_largest_k(f);
    int wp = 0;
    int wn = 0;
    for (int k = 1; k <= largest_k; ++k) {
        wp += std::pow(2, -k) * nclauses(f, k, l);
        wn += std::pow(2, -k) * nclauses(f, k, -l);
    }
    return std::make_tuple(wp, wn, std::max(wp, wn));
}
```

4. 2-Sided Jeroslow-Wang rule

$$w(F, u) = \sum_k 2^{-k} d_k(F, u)$$

$$\Phi(x, y) = x + y$$

```
auto jw2(Formula f, int l) {
    auto largest_k = get_largest_k(f);
    int wp = 0;
```

```

int wn = 0;
for (int k = 1; k <= largest_k; ++k) {
    wp += std::pow(2, -k) * nclauses(f, k, 1);
    wn += std::pow(2, -k) * nclauses(f, k, -1);
}
return std::make_tuple(wp, wn, wp + wn);
}

```

5. DSJ rule

$$w(F, u) = 4d_2(F, u) + 2d_3(F, u) + \sum_{k \geq 4} d_k(F, u)$$

$$\Phi(x, y) = (x + 1)(y + 1)$$

```

auto dsj(Formula f, int l) {
    auto largest_k = get_largest_k(f);
    int wp = 4*nclauses(f, 2, 1) + 2*nclauses(f, 3, 1);
    int wn = 4*nclauses(f, 2, -1) + 2*nclauses(f, 3, -1);
    for (int k = 4; k <= largest_k; ++k) {
        wp += nclauses(f, k, 1);
        wn += nclauses(f, k, -1);
    }
    return std::make_tuple(wp, wn, (wp+1)*(wn+1));
}

```

2.1.5 Assigning literals and removing satisfied clauses

We can remove satisfied clauses from the graph using the `remove_satisfied` function. This function first increments the number of literals assigned true contained in the clause, and decrements the number of remaining unsatisfied clauses in the formula. Next we iterate over the associated literals for the clause, removing the clause from that literal's adjacency list. Finally, we remove all literals from the clause's adjacency list.

```

void remove_satisfied(Formula& f, int d) {
    auto& clause = f.clauses[d];
    clause.n_t++;
    f.remaining--;
    auto lits = clause.literals;
    for (auto l : lits) {
        auto s = sign(l);
        auto& lit = f.literals[l*s];
        if (s == 1) {
            auto& p = lit.pos_clauses;
            p.erase(std::remove(p.begin(), p.end(), d), p.end());
        } else {
            auto& n = lit.neg_clauses;
            n.erase(std::remove(n.begin(), n.end(), d), n.end());
        }
    }
    clause.literals.clear();
}

```

We set a truth assignment for a literal using the `set_var` function. We first determine an assignment based on whether the literal is positive or negative. Next, we determine out of the clauses that the literal is present in, which are unsatisfied by the change, and which are satisfied. We remove the satisfied clauses using the `remove_satisfied` function. Since a disjunction is not false until all members are false, we can remove the literal from all unsatisfied clauses, also incrementing the number of false literals in that clause. If a clause becomes empty as a result of setting the variable we return early, as this interpretation of the formula is unsat.

```

bool set_var(Formula& f, int l) {
    auto s = sign(l);
    auto pos = l*s;
    auto& lit = f.literals[pos];
    if (lit.assn != -1) throw std::runtime_error("literal already assigned");
    lit.assn = (s == 1) ? 1 : 0;
    auto sat_c = (lit.assn == 1) ? lit.pos_clauses : lit.neg_clauses;
    auto& unsat_c = (lit.assn == 0) ? lit.pos_clauses : lit.neg_clauses;
    for (auto cidx : sat_c) remove_satisfied(f, cidx);
    for (auto cidx : unsat_c) {
        auto& clause = f.clauses[cidx];
        clause.n_f++;
        clause.literals.erase(std::remove(clause.literals.begin(),
                                           clause.literals.end(),
                                           (lit.assn == 0) ? pos : -pos),
                              clause.literals.end());
        if (clause.literals.size() == 0) return false;
    }
    unsat_c.clear();
    return true;
}

```

2.2 Parallel

First, process 0 reads the formula from standard input and distributes it to all other processes.

```

int* form;
int form_c;
std::vector<int> f;
if (rank == 0) {
    f = read_input();
    form = f.data();
    form_c = f.size();
}
MPI_Bcast(&form_c, 1, MPI_INT, 0, MCW);
if (rank != 0) form = (int*)malloc(sizeof(int) * form_c);
MPI_Bcast(form, form_c, MPI_INT, 0, MCW);
if (rank != 0) f = std::vector<int>(form, form + form_c);
Formula formula(f);

```

Next, all the other processes create the formula and begin their computations. The branching strategy is determined by the process rank. On each iteration of the dpll algorithm the process checks if the master has indicated that the formula has been solved. If this process finds the formula to be SAT, it creates an array consisting of the literals assigned true, a zero to separate, and the literals assigned false. This array is sent to the master process. Otherwise, if the formula is UNSAT, a single 0 is sent to the master process to indicate that this process found it to be UNSAT.

```

if (rank != 0) {
    std::srand(rank);
    auto rule = branch_rule_int(rank - 1);
    int early_term = 0;
    auto [sat, finalf] = dpll(formula, rule,
                              [&]() {
                                  MPI_Iprobe(0, 0, MCW, &early_term, MPI_STATUS_IGNORE);
                                  return static_cast<bool>(early_term);
                              });
}

```



```

if (sat) {
    std::vector<int> assn;
    for (auto l : finalf.literals) {
        if (l.second.assn == 1) assn.push_back(l.first);
    }
    assn.push_back(0);
    for (auto l : finalf.literals) {
        if (l.second.assn == 0) assn.push_back(l.first);
    }

    MPI_Send(assn.data(), assn.size(), MPI_INT, 0, 0, MCW);
} else if (!early_term) {
    int data = 0;
    MPI_Send(&data, 1, MPI_INT, 0, 0, MCW);
}
}
}

```

The master process waits until it receives a message from a process. Since our sub-processes use a complete SAT solver, we can finish once a single process has determined the formula to be SAT or UNSAT. The master then tells all of the workers to terminate and reports the results.

```

if (rank == 0) {
    int* buff = (int*)malloc(sizeof(int) * (formula.literals.size() + 1));
    MPI_Status status;
    MPI_Recv(buff, formula.literals.size() + 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &status);
    int solver_rank = status.MPI_SOURCE;
    for (int i = 1; i < size; ++i) {
        if (i == solver_rank) continue;
        int data = 0;
        MPI_Send(&data, 1, MPI_INT, i, 0, MCW);
    }

    if (buff[0] == 0) {
        std::cout << "Formula is: UNSAT" << std::endl;
        std::cout << "Solved By process " << solver_rank << " with branching strategy "
                    << branch_rule_name(branch_rule_int(solver_rank-1)) << std::endl;
    } else {
        std::cout << "Formula is: SAT" << std::endl;
        std::cout << "Solved By process " << solver_rank << " with branching strategy "
                    << branch_rule_name(branch_rule_int(solver_rank-1)) << std::endl;
        std::cout << "Variables assigned TRUE:" << std::endl;
        for (long unsigned int i = 0; i < formula.literals.size() + 1; ++i) {
            if (buff[i] == 0)
                std::cout << "\nVariables assigned FALSE:" << std::endl;
            else
                std::cout << buff[i] << " ";
        }
    }
}
}
}

```

3 Build and Run

Compile with

```
mpic++ -Wall -Wextra -std=c++17 -lm -g -o portfolio parallel_dp11.cpp
```

Run with the following, where \$CNF_FILE is the path to the test file in CNF format.

```
mprun -np 8 --use-hwthread-cpus ./portfolio < $CNF_FILE
```

4 Performance

5 Appendix

5.1 Helper Code

5.1.1 Read Input

Reads input from stdin as the DIMACS cnf format.

```
auto read_input() {
    std::vector<int> f;
    for (std::string l; std::getline(std::cin, l);) {
        if (l.empty()) continue;
        std::stringstream ss(l);
        std::string word;
        ss >> word;
        if (word == "c") continue;
        if (word == "p") {
            ss >> word;
            if (word != "cnf") throw std::invalid_argument("Data must be in cnf format, got " + word);
            continue;
        }
        do {
            if (word == "%") return f;
            int v = std::stoi(word);
            f.push_back(v);
        } while (ss >> word);
    }

    return f;
}
```

5.1.2 Branching Helpers

Included here to save space in the main section.

```
int nclauses(Formula f, int k, int u) {
    auto s = sign(u);
    auto& lit = f.literals[u*s];
    auto cs = (s == 1) ? lit.pos_clauses : lit.neg_clauses;
    int counter = 0;
    if (k == -1) return cs.size();
    for (auto c : cs) {
        if (f.clauses[c].literals.size() == (unsigned int)k) counter++;
    }
    return counter;
}

int get_largest_k(Formula f) {
    return std::max_element(f.clauses.begin(), f.clauses.end(),
        [](auto a, auto b) {
            return a.literals.size() < b.literals.size();
        })->literals.size();
}
```

```

int apply_rule(Formula f, std::function<std::tuple<int,int,int>(Formula, int)> rule) {
    int maximum = 0;
    int curr = 0;
    for (auto l : f.literals) {
        if (l.second.assn != -1) continue;
        auto [wp, wn, phi] = rule(f, l.first);
        if (phi >= maximum) {
            curr = wp >= wn ? l.first : -l.first;
            maximum = phi;
        }
    }
    return curr;
}

```

5.1.3 Full Source

See `parallel_dp11.cpp`.

References

- [1] Ming Ouyang. How good are branching rules in dp11? *Discrete Applied Mathematics*, 89(1):281 – 286, 1998.