Serial SAT Solver

Abraham White

April 6, 2020

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 $\{\top, \bot\}$ 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.

A more recent aproach is the conflict-driven clause learning algorithm (CDCL), inspired by the original DPLL algorithm.

Implementation

Recursive DPLL

The basic DPLL algorithm can be defined recursively as in Algorithm 1. In the algorithm, $F[l \to \top]$ denotes the formula obtained by replacing the literal l with $\top$ and $\neg l$ with $\bot$ 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 formula is consistent if for every literal in the formula there doesn't also exist the opposite. Unfortunately, this recursive algorithm becomes unusable with larger formulae.

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 $\bot$. This step occurs on line 6 of the recursive algorithm .

2. **Unit Propogation**: 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.

Algorithm 1 The recursive DPLL algorithm

```
function DPLL(F : Formula)
    if F is empty then
        return SAT
    else if F contains an empty clause then
        return UNSAT
    else if F contains a pure literal l then
        return DPLL(F[l → ⊤])
    else if F contains a unit clause [l] then
        return DPLL(F[l → ⊤])
    else
        let l be a literal in F
        if DPLL(F[l → ⊤]) = SAT then
            return SAT
        else
            return DPLL(F[l → ⊥])
        end if
    end if
end function
```

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 integer. Clauses are represented by a list of these integer literals, and a formula is represented by a list of clauses. We exlude 0 from the possible integers. 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 an 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. We also do some work to recognize pure literals, which will be eliminated in a later step.

We define an adjacency list to associate literals with clauses that reference them using an unordered map. The `LitData` struct is used in the undordered map to keep track of the clauses where a literal occurs positively (x), or negatively ($\neg x$). The `ClauseData` struct keeps track of the number of its literals assigned true or false which tells us whether the clause is satisfied, unsatisfied, or unit.

```
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<std::reference_wrapper<ClauseData>> pos_clauses;
  std::vector<std::reference_wrapper<ClauseData>> 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 l, const ClauseData& c) {
  auto s = sign(l);
  auto pos = l*s;
  auto it = this->literals.find(pos);
  LitData data;
  if (it != this->literals.end()) data = it->second;
  if (s == 1) {
    data.pos_clauses.push_back(c);
  } else {
    data.neg_clauses.push_back(c);
  }
  this->literals.insert_or_assign(pos, data);
}
```

Now we need a constructor function to handle initializing these data structures from the list of list of integers formula representation.

```
Formula(std::vector<std::vector<int>> formula) {
  this->remaining = formula.size();
  for (auto c : formula) {
    ClauseData cd;
    cd.literals = c;
    cd.orig_len = c.size();
    this->clauses.push_back(cd);
    for (auto l : c) this->add_literal(l, cd);
  }
}
```

We need to actually implement the recursive DPLL algorithm. The algorithm itself is simple, but the helper functions will be more complicated.

```
bool dpll() {
  for (auto l : this->literals) {
    if (l->second->pure()) {
      this->pure_literal_assign(l->first, l->second);
    }
  }
  for (auto c : this->clauses) {
    if (c.sat()) continue;
    if (c.literals.size() == 0) return false;
    if (c.unit()) this->unit_propogate(c);
  }
  if (this->remaining == 0) return true;
  auto l = this->get_branching_variable();
  this->set_var(l);
  if (this->dpll() == true) return true;
  this->unset_var(l);
  this->set_var(-l);
  return this->dpll();
}
```

The implementation goes as follows: First we handle the pure literal step, which removes whole clauses from consideration by assigning truth values. We use the `pure_literal_assign` function to set a value for the literals. First 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, and update the associated clauses. We remove satisfied clauses from the adjacency lists of other literals, since any other truth assignments don't matter and the clause doesn't have to be considered any more.

```
void remove_satisfied(ClauseData& d) {
  for (auto l : d.literals) {
    auto s = sign(l);
    auto it = this->literals.find(l*s);
    if (it != this->literals.end()) {
      if (s == 1) {
        auto& p = it->second.pos_clauses;
        p.erase(std::remove(p.begin(), p.end(), d));
      } else {
        auto& n = it->second.neg_clauses;
        n.erase(std::remove(n.begin(), n.end(), d));
      }
    }
  }
  d.literals.clear();
}
void pure_literal_assign(int l, LitData& data) {
  auto pos_size = data.pos_clauses.size();
  auto& lclauses = (pos_size == 0) ? data.neg_clauses : data.pos_clauses;
  data.assn = (pos_size != 0) ? 1 : 0;
  for (auto& clause : lclauses) {
    clause.n_t++;
    this->remaining--;
    remove_satisfied(clause);
  }
}
```

The next loop helps with unit propogation. We skip over clauses that have already been satisfied, terminate when we have a clause that is empty, i.e. there was a conflicting literal asignment, and then propogate when the clause is unit. We use the `unit_propogate` function to make a truth asignment for the unit literal. Internally, all `unit_propogate` does is change the formula by setting a truth assignment for the unit literal.

```
void unit_propogate(ClauseData& clause) {
  auto l = clause.literals[0];
  this->set_var(l);
}
void set_var(int l) {
  auto s = sign(l);
  auto it = this->literals.find(l*s);
  if (it == this->literals.end())
    throw std::runtime_error("Couldn't find literal for assignment");
  if (it->second.assn != -1)
    throw std::runtime_error("Literal already assigned");
  LitData& lit = it->second;
  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& clause : sat_c) {
    clause.n_t++;
    this->remaining--;
    remove_satisfied(clause);
  }
  for (auto& clause : unsat_c) {
    clause.n_f++;
    clause.literals.erase(std::remove(clause.literals.begin(),
                                      clause.literals.end(),
                                      (s == 1) ? -l : l));
```

```
  }
}
```

We then check if there are no remaining unsatisfied 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` and `unset_var` functions to handle changing the formula and backtracking.

CDCL

Appendix

Helper Code

sign

Apparently the standard `copysign` is slow

```
int sign(int x) {
  return ( (x > 0) ? 1
         : (x < 0) ? -1
         : 0);
}
```

Read Input

```
auto read_input() {
  std::vector<std::vector<int>> f;
  for (std::string l; std::getline(std::cin, l);) {
    std::stringstream ss(l);
    std::string word;
    std::vector<int> clause;
    while (ss >> word) {
      int v = std::stoi(word);
      if (v == 0) throw std::invalid_argument("0 cannot be a literal");
      clause.push_back(v);
    }
    f.push_back(clause);
  }

  return f;
}
```

References

[1] Filip Marić. Formalization and implementation of modern sat solvers. *Journal of Automated Reasoning*, 43(1):81–119, 2009.