tags: `Report`

# YaSat milestone 1

309551130 謝柏威

## file structure

```
.
├── makefile
├── benchmarks
│   ├── *.cnf
│   ├── SAT
│   │   ├── sanity
│   │   └── tiny
│   └── UNSAT
│       ├── sanity
│       └── tiny
├── doc
│   ├── project_description.pdf
│   └── report1.md
├── include
│   ├── dpll.hpp
│   ├── parser.h
│   └── state.hpp
├── lib
│   ├── MiniSat_v1.14_linux
│   └── sat_comparer
├── scripts
│   ├── check.cpp
│   └── test.sh
└── src
    ├── parser.cpp
    └── sat.cpp
```

# Usage

- `makefile` is provided.
- All the source code can be compiled and run on `linux{id}.cs.nctu.edu.tw` by simply typing `make`. The resulting binary will be generated named `yasat`.

```
$ make
g++ -Iinclude/ -std=c++17 -Ofast -Wall -Wextra -Wshadow -Wold-style-cast -c -o obj/parser.o sr
g++ -Iinclude/ -std=c++17 -Ofast -Wall -Wextra -Wshadow -Wold-style-cast -c -o obj/sat.o src/s
g++ -Iinclude/ -std=c++17 -Ofast -Wall -Wextra -Wshadow -Wold-style-cast -o yasat obj/parser.c
```

```
$ la yasat
-rwxr-xr-x 1 pwhsieh gcs 48K Apr 19 20:24 yasat
```

# Feature

In this project We are going to implement a SAT solver, and these are the features I've implemented in this milestone.

1. iterative DPLL
2. two literal watching

## 1. iterative DPLL

Since there's going to have all kinds of branching heuristics added to this project in the future, I thought it would be a good idea to maintain the recursive stack on my own.

### DPLL structure

- This is the DPLL structure declaration. It takes `number of variables` and `clauses` as the constructor argument.

```
// #Declaration
struct DPLL {
    DPLL(int _num_vars, vector<vector<int>> &_clauses)
        : num_vars(_num_vars), num_clauses(_clauses.size()), clauses(_clauses) {}
    void init();
    void watch_not_false(int&, int&, int, int);
    bool watch_is_true(int, int);
    bool unit_propagate();
    bool backtrack();
    std::optional<std::vector<int>> solve();

    int num_vars;
    int num_clauses;
    std::queue<int> prop;
    std::stack<int> branch;
    std::stack<State, std::vector<State>> call_stack;
    std::vector<std::vector<int>> &clauses;
};
```

### DPLL procedure

This is the main loop of the iterative DPLL, including `unit_propagate`, `backtrack`, and decision making.

```
std::optional<std::vector<int>> DPLL::solve() {
    // 0. init
    init();

    while (!call_stack.empty()) {
        // 1. unit propagate
        if (!unit_propagate()) {
            if (!backtrack()) return std::nullopt; //UNSAT
            continue;
        }

        // 2. check if curent state is satisfied already
        if (call_stack.top().done) break;

        // 3. if not, apply new decision
        call_stack.push(call_stack.top());
        int g = call_stack.top().pick_variable();
        call_stack.top().set_variable(g, prop);
        branch.push(-g);
    }

    // 4. SAT: return ans;
    auto res = std::vector<int>(num_vars);
    auto &state = call_stack.top();
    for (int i = 1; i <= num_vars; i++) {
        if (state.var(i) and *state.var(i)) {
            res[i-1] = i;
        } else {
            res[i-1] = -i;
        }
    }
    return res;
}
```

# 2. two literal watching

- During **unit propagation**, we only look at the first two literal in each clause. This so-called `two literal watching` method enable us to do quick unit propagation without iterating through all the clauses.
- This method drastically decrease the complexity of the unit propagation, since we at most look at all the clauses and literal **once**.
- `prop` record all the false literal. (ex: decision on `1 == true`, imply that `1 == false` is the false literal)

```
bool DPLL::unit_propagate() {
    auto &state = call_stack.top();
    while (!prop.empty()) {
        int false_literal = prop.front(); prop.pop();
```

```
        state.done = true;
        for (int i = 0; i < num_clauses; i++) {
            if (clauses[i].size() == 1) break;

            // this clause is satisfied
            auto &[la, lb] = state.watch[i];
            if (la == -1 and lb == -1) continue;

            // one of the watched literal is already true
            if (watch_is_true(la, i) or watch_is_true(lb, i)) {
                la = lb = -1;
                continue;
            }

            // update watching literals that are not false
            watch_not_false(la, lb, i, false_literal);
            watch_not_false(lb, la, i, false_literal);

            // check if this became unit clause (implication)
            if (la == -1) std::swap(la, lb);
            if (la != -1 and lb == -1) {
                int tmp = clauses[i][la];
                if (state.var(tmp) and *state.var(tmp) != (tmp > 0)) {
                    prop = {};
                    return false; //conflict
                }
                state.set_variable(tmp, prop);
                la = -1;
            }
            if (la != -1 or lb != -1) state.done = false;
        }
    }
    return true;
}
```

# Results

- These result are verified by my own written `checker.cpp`, which is placed under `scripts/`.
- It checks the correctness of the SAT solver by taking the `.cnf` file and `.sat` file as input, and check if there's any confict between them.

## 75 variable, 325 clauses

- These are some randomly generated 3-SAT dataset.
- 5 SAT and 5 UNSAT, 10 dataset in total.
- 10 cnf test cases take about ***a second***.

> make test will run the test.sh script to run the benchmark

```
$ time make test
./scripts/test.sh
[SAT]
[ACCEPTED]
[SAT]
[ACCEPTED]
[SAT]
[ACCEPTED]
[SAT]
[ACCEPTED]
[SAT]
[ACCEPTED]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
[UNSAT]
make test  1.03s user 0.04s system 91% cpu 1.176 total
```

## Sudoku

- Out of curiousity, I use the encoded sudoku data from the previous project. And the result is actually quite good.
- `9 x 9` sudoku can be solved in `0.132` seconds, and `16 x 16` sudoku can be solved in `1.436` seconds.

### 9 x 9

```
$ time ./yasat benchmarks/9.cnf
[SAT]
./yasat benchmarks/9.cnf  0.03s user 0.00s system 23% cpu 0.132 total
```

### 16 x 16

```
$ time ./yasat benchmarks/16.cnf
[SAT]
./yasat benchmarks/16.cnf  1.42s user 0.01s system 99% cpu 1.436 total
```

# Discussion

- The material provided in class has taught us how to implement a recursive version of `DPLL`, however, for better future proof, I decided to implement it in an *iterative way*.
- This is quite hard in the beginning because I can't quite figure out the right way to iterate through all the possible outcomes of the `cnf` while writing efficient code.
- Another obstacle I've encountered during this assignment is the implementation of `two-literal watching`. The whole concept is actually bug-prone and hard to debug.
- After all these hard work, fortunately, the result is great. Much better than I would have thought. It is capable of **solving `16 x 16` sudoku in about a second**. Nice!