

PA1 Report

Instructor: Fei He

陈嘉杰 (2017011484)

TA: Jianhui Chen, Fengmin Zhu

1 第一部分

第一部分的实验要求是实现朴素的 DPLL 算法。算法实现基本按照课件要求，只不过把递归形式改为了手动维护栈的形式，这样方便后续 backjump 的实现。代码中维护了如下的数据结构：

```
struct LiteralInfo {
    // immutable
    std::vector<uint32_t> clauses;
    std::vector<uint32_t> clause_index;
    // mutable
    uint32_t cur_clauses;
    bool is_assigned;
#ifdef CDCL
    uint32_t unit_clause;
    uint32_t assign_depth;
#endif
};

struct ClauseInfo {
    // immutable
    std::vector<uint32_t> literals;
    // mutable
    uint32_t num_unassigned;
    bool is_satisfied;
};

enum ChangeType { TYPE_DECIDE, TYPE IMPLIED };

struct Change {
    uint32_t assigned_literal;
    uint32_t removed_clauses_begin;
    ChangeType type;
};
```

LiteralInfo 记录了这个 literal 出现在的 clause 和对应的位置、当前出现在未满足 clause 的次数、是否已经赋值和用于 CDCL 的 implication graph 的边的记录。ClauseInfo 记录了这个 clause 中的各个 literal、当前还未赋值的 literal 和是否已经满足。Change 记录了搜索树的一个结点，记录了此时

是选择了一个 literal 还是因为 unit propagation 选择了一个 literal、目前赋值的 literal 并记录了因为赋值而被满足的 clause。

通过这些结构体，可以实现 DPLL 算法中需要用到的各个操作。考虑到这只是个小作业，并且公开的测试样例中数据量不是很大，没有做很深入的性能优化，比如通过位运算缩短在 clause 中寻找 unit clause 的时间（见 [1]）。

程序通过了公开的所有测例，并且我也额外从 SATLIB - Benchmark Problems 找到了一些 DIMACS 格式的测例，并加到了 dpll/tests 目录下，并额外手动构造了几个测例用于测试，一共 43 个测试样例，基于 Zhang Xinwei 和 Wang Yuanbiao 编写的脚本运行进行测试，在 Release 编译条件下都通过：

```
0 unsat for 1 vars pass, time: 0.008457 ms
1 sat for 5 vars pass, time: 0.01464 ms
2 sat for 4 vars pass, time: 0.016514 ms
3 sat for 5 vars pass, time: 0.014561 ms
4 unsat for 20 vars pass, time: 0.083649 ms
5 sat for 12 vars pass, time: 0.04352 ms
6 sat for 20 vars pass, time: 0.207192 ms
7 unsat for 21 vars pass, time: 0.144755 ms
8 sat for 70 vars pass, time: 2.06928 ms
9 sat for 57 vars pass, time: 29.5635 ms
10 unsat for 90 vars pass, time: 1.63332 ms
11 sat for 42 vars pass, time: 0.447705 ms
12 unsat for 35 vars pass, time: 32.835 ms
13 sat for 45 vars pass, time: 2.92227 ms
14 unsat for 80 vars pass, time: 1.78398 ms
15 unsat for 50 vars pass, time: 0.315052 ms
16 unsat for 275 vars pass, time: 2.895 ms
17 sat for 163 vars pass, time: 1.63666 ms
18 unsat for 157 vars pass, time: 1.81069 ms
19 sat for 2 vars pass, time: 0.008153 ms
20 sat for 20 vars pass, time: 0.11078 ms
21 sat for 20 vars pass, time: 0.161564 ms
22 sat for 20 vars pass, time: 0.136904 ms
23 sat for 20 vars pass, time: 0.117698 ms
24 sat for 20 vars pass, time: 0.10649 ms
25 sat for 20 vars pass, time: 0.133034 ms
26 sat for 20 vars pass, time: 0.116897 ms
27 sat for 20 vars pass, time: 0.099551 ms
28 sat for 20 vars pass, time: 0.101954 ms
29 sat for 20 vars pass, time: 0.147096 ms
30 sat for 20 vars pass, time: 0.095605 ms
31 unsat for 50 vars pass, time: 1.46067 ms
32 unsat for 50 vars pass, time: 1.59957 ms
33 unsat for 50 vars pass, time: 1.35715 ms
34 unsat for 50 vars pass, time: 1.44918 ms
35 unsat for 50 vars pass, time: 1.92965 ms
36 sat for 100 vars pass, time: 216.186 ms
37 sat for 100 vars pass, time: 244.769 ms
```

```

38 sat for 100 vars pass, time: 13.4931 ms
39 sat for 100 vars pass, time: 1888.17 ms
40 sat for 100 vars pass, time: 143.096 ms
41 unsat for 9 vars pass, time: 0.082119 ms
42 sat for 7 vars pass, time: 0.033555 ms

```

total score: 43 / 43

2 第二部分

第二部分在第一部分的基础上实现了 CDCL，在代码中通过 CDCL 宏来进行控制，方便两个版本的对比。在遇到冲突的时候，CDCL 算法会按照 implication graph 遍历寻找一个 cut，我实现的是找到所有入度为 0 的结点，也就是所有 Decide 而不是 Propagate 的结点。这样做的好处是可能能够 backjump 到较早的 decision level，坏处是这样的点可能很多，导致插入的 clause 不是很有效。在 [2] 书中讲到一种根据图的最小割求出 conflict clause 的方法，由于生成的 literal 数量会比较少，所以应该会比我的算法得到更好的结果。

在找到这些结点后，把这些结点对应的 literal 的 negation 拼接起来成为一个新的 clause。由于此时这个 clause 所有 literal 都是 assigned 但没有一个是 true，所以它一定是 unsat 的，直接找到这些 literal 中 decision level 最大的一个，回退到 decide 它之前，此时就可以恢复正常的算法执行了，并且由于刚刚插入的这个 clause，会立即发生 unit propagate。

这里实际上也有两种实现的可能，一个可能是选择回到 decision level 最大的那一个的前一步，另一个可能是回到 decision level 最小的一个然后继续。第一种实现的优势是会减少走重复路径的可能，第二种实现的优势是 backjump 更远，并且可能会提前出现新的 unit propagate 路径，但在 conflict clause 很大的时候效果也不好。我没有做特别深入的比较研究，选择了前一种。

为了测试 CDCL，我也构造了几个样例，然后比较了程序输出的统计数据，对比了决策的次数（通过 DEBUG 宏打开）：

样例	无 CDCL 决策次数	CDCL 决策次数
3	10	10
4	441	441
5	12	12
6	20	20
7	350	350
8	70	70
9	189649	1332
41	166	57
42	21	21

References

- [1] Tanbir Ahmed. “An Implementation of the DPLL Algorithm”. In: (), p. 138.

- [2] Daniel Kroening and Ofer Strichman. *Decision procedures: an algorithmic point of view*. Texts in theoretical computer science. OCLC: 244022161. Berlin: Springer, 2008. 304 pp. ISBN: 978-3-540-74104-6 978-3-540-74105-3.