选题

自行设计CNF的SAT求解算法。

需求分析

需要搭建一个CNF的SAT求解器，可以使用现有算法、自行设计的算法，可独立设计程序，或修改现有开源程序的核心算法。
需要构建/查找测试集，并与现有工具（如Z3）进行性能比较。
CNF的定义（和CNF FILE的输入格式）：**DIMACS CNF**

> **Quote**
>
> 1. *clauses* joined by **AND**;
> 2. each clause, in turn, consists of *literals* joined by **OR**;
> 3. each literal is either the name of a variable *(a positive literal*, or the name of a variable preceded by **NOT** *(a negative literal.*

概要分析

现有算法及求解器

DPLL

算法的主要思想：求解CSP问题的基本思路

> **Note**
>
> Algorithm 1: DPLL
>
> X:=unit-resol(X)
> if $\perp \in X$ then
> return(unsatisfiable)
> if $X = \varnothing$ then
> return(satisfiable)
> if $\perp \notin X$ then
> choose variable p in X
> ret1 = DPLL($X \cup \{p\}$)
> ret2 = DPLL($X \cup \{\neg p\}$)
> return ret1$\vee$ret2

一些改进的算法包括：根据某种启发式函数，给变量随机赋值（以任意的顺序），若找不到解，再重新赋值。

CDCL

在DPLL的基础上，用到了backjumping, clause learning, adaptive branching, "two-wathced-literals" unit propagation, random restarts等技术。

> **Note**
>
> Algorithm2: CDCL
>
> $dl \leftarrow 0$; // decision level
> if $UnitPropagation(F, \alpha)$==CONFLICT then return UNSAT;
> while ∃ unassigned variables do
> /* PickBranchVar picks a variable to assign and
> picks the respective value */
> $(x, v) \leftarrow PickBranchVar(F, \alpha)$;
> $dl \leftarrow dl + 1$;
> $\alpha \leftarrow \alpha \cup \{(x, v)\}$;
> if $UnitPropagation(F, \alpha)$==CONFLICT then
> $bl \leftarrow ConflictAnalysis(F, \alpha)$;
> if $bl < 0$ then
> return UNSAT;
> else
> $BackTrack(F, \alpha, bl)$;
> $dl \leftarrow bl$;
> return SAT;

实现了这个算法的求解器包括：MiniSAT, Chaff, GRASP.

SLS

Stochastic Local Search: Begin with a complete assignment and iteratively modify the assignment.

Parallel SAT-solving

分为三类：portfolio, divide-and-conquer 和 parallel local search 算法。
portfolio让多个求解器共同工作，而divide-and-conquer将问题规模划分到多个进程上。

重要的branching启发式函数

1. Maximum Occurences on Minimum sized clauses (MOM)
   优先给那些在短子句中频繁出现的变量进行赋值。
   是贪心算法，试图通过下一次赋值，使最多的子句满足，或实现最多的implication
2. Dynamic Largest Individual Sum (DLIS)
   对每个变量，计算它出现过的未满足的子句，选择数量最多的变量进行赋值。
   是动态的，每次赋值后都要更新，所以开销大。
3. Variable State Independent Decaying Sum (VSIDS)
   计算文字（包括正反）在所有子句中出现的次数，选择数量最多的文字对应的变量进行赋值。
   只当新的子句加入database时才更新计数，开销小，是DLIS的改良版。

选取的求解器

zChaff

应用了Chaff algorithm，主要思想是基于DPLL，应用VSIDS的方法来启发式地选取变量。

MiniSat

基于CDCL，主要实现的优化有：

1. 选取变量的顺序。使用改进版的VSIDS启发式算法选取变量并赋值，并且避免使用out-dated变量；
2. 对二元子句的处理。直接将二元子句中的文字（literal）存到wathcer list里，有利于进行传播；
3. 子句消减（clause deletion）。过多的learned clause会造成空间的开销，所以MiniSat以启发式算法将不需要的子句进行消减；在有限步内没有得到答案的话，restart时扩大允许的子句数量

性能比较

变量数目	子句数目	zChaff的求解时间t1	MiniSat的求解时间t2	差值（t1-t2）(s)
16	18	0.000137	0.004211	-0.004119
42	133	0.010476	0.009996	0.00048
50	80	0.000385	0.002096	-0.001711
60	160	0.002982	0.004462	-0.00148
63	168	0.002683	0.004772	-0.002089
64	254	0.000284	0.005664	-0.00538
66	176	0.003286	0.004739	-0.001453
100	160	0.000722	0.005055	-0.004333
155	1135	0.001496	0.000913	0.000583

变量数目	子句数目	zChaff的求解时间t1	MiniSat的求解时间t2	差值（t1-t2）(s)
1040	3668	0.018241	0.01346	0.004781

总体看来，尽管MiniSat应该是比zChaff做了更多优化，但在benchmark里并没有体现出明显的优势，尤其是当变量和子句较少的时候。
于是考虑优化zChaff，让它更快。

具体实现

对zChaff的详细介绍：Intro to zChaff.
本部分是zChaff原本的代码，未做任何优化。

添加变量和子句

1. 根据CNF file的格式扫描输入文件：

- 开头第一个字母是'c'：注释，跳过
- 开头第一个字母是'p'：定义问题的规模（多少变量，多少子句）
- 其他：具体子句的定义，包含正反变量，以0结尾
  例如：

```
c  simple_v3_c2.cnf
c
p cnf 3 2
1 -3 0
2 3 -1 0
```

2. 特殊情况：
   若一个子句里包含同个变量的正和反（contain var of both polarity），则这个子句自动满足，可以消除。
   如何判断包含这一特殊情况呢？
   解决办法是在扫描一个子句时，将依次扫描到的变量同时加入（insert）两个`std::set<int>`集合里：

- 加入第一个集合时，添加的是变量下标的绝对值`index`（则正反变量只会被添加一次）；
- 加入第二个集合时，添加的是`index << 1 + sign`，`sign`是一个bool变量，当这是反变量时它为true（则正反变量会被添加两次）
  如果扫描完这个子句时，发现两个集合大小不一样，说明包含同个变量的正和反。
  关键代码：

```C++
set<int> clause_vars;
set<int> clause_lits;
if (var_idx != 0) { // 0 is the end of this clause
    if (var_idx < 0) {
        var_idx = -var_idx;
        sign = 1;
    }
    clause_vars.insert(var_idx);
    clause_lits.insert((var_idx << 1) + sign);
}
else {
    // add this clause
    if (clause_vars.size() != 0 && (clause_vars.size() ==
clause_lits.size())) { // yeah, can add this clause
        vector<int> temp;
        for (set<int>::iterator itr = clause_lits.begin();
            itr != clause_lits.end(); ++itr)
            temp.push_back(*itr);
        SAT_AddClause(mng, &temp.begin()[0], temp.size());
    }
    else {
    } // it contain var of both polarity, so is automatically
satisfied, just skip it
```

3. 对counter的初始化：
   如前面所述，VSIDS给每个变量（包括正反文字）一个counter，记录该变量在子句中出现的次数，从而估计它的重要程度。
   当把子句加入database时，需要对counter+1。
   关键代码：

```C++
add_clause(int * lits, int n_lits, int gflag) {
// lits是前面提到的变量集合，每个元素是index << 1 + sign
// n_lits是lits集合的大小
    if(n_lits==2) {
        // >> can omit sign's impact
        // 对二元子句特殊处理
        ++variable(lits[0]>>1).two_lits_count(lits[0]&0x1);
        ++variable(lits[1]>>1).two_lits_count(lits[1]&0x1);
    }
    for (int i=0; i< n_lits; ++i) {
        int var_idx = lits[i]>>1;
```

```
        assert((unsigned)var_idx < variables().size());
        int var_sign = lits[i]&0x1;
        cl.literal(i).set(var_idx, var_sign);
        ++variable(var_idx).lits_count(var_sign); // 对counter++, 用
var_sign区分了正反文字
    }
}
```

求解

初始化

1. 找出定义了但没有在子句中出现过的变量：此变量的正、反counter的出现次数均为0；
2. 找出只出现了正或只出现了反文字的变量：此变量的某一个counter为0。这类变量可以被加入implication队列，可单一赋值；

核心算法概述

核心函数：

```C++
decide_next_branch(); // 选择一个变量并赋值
deduce(); // UnitPropagate：若子句中只有一个文字未赋值，其他文字为假，则把未赋值的文字赋值为1
// 应用了Boolean Constraint Propagation思想；据说占到了运行时的大部分时间
analyze_conflict(); // 找到出现矛盾的原因
```

solver的核心代码：

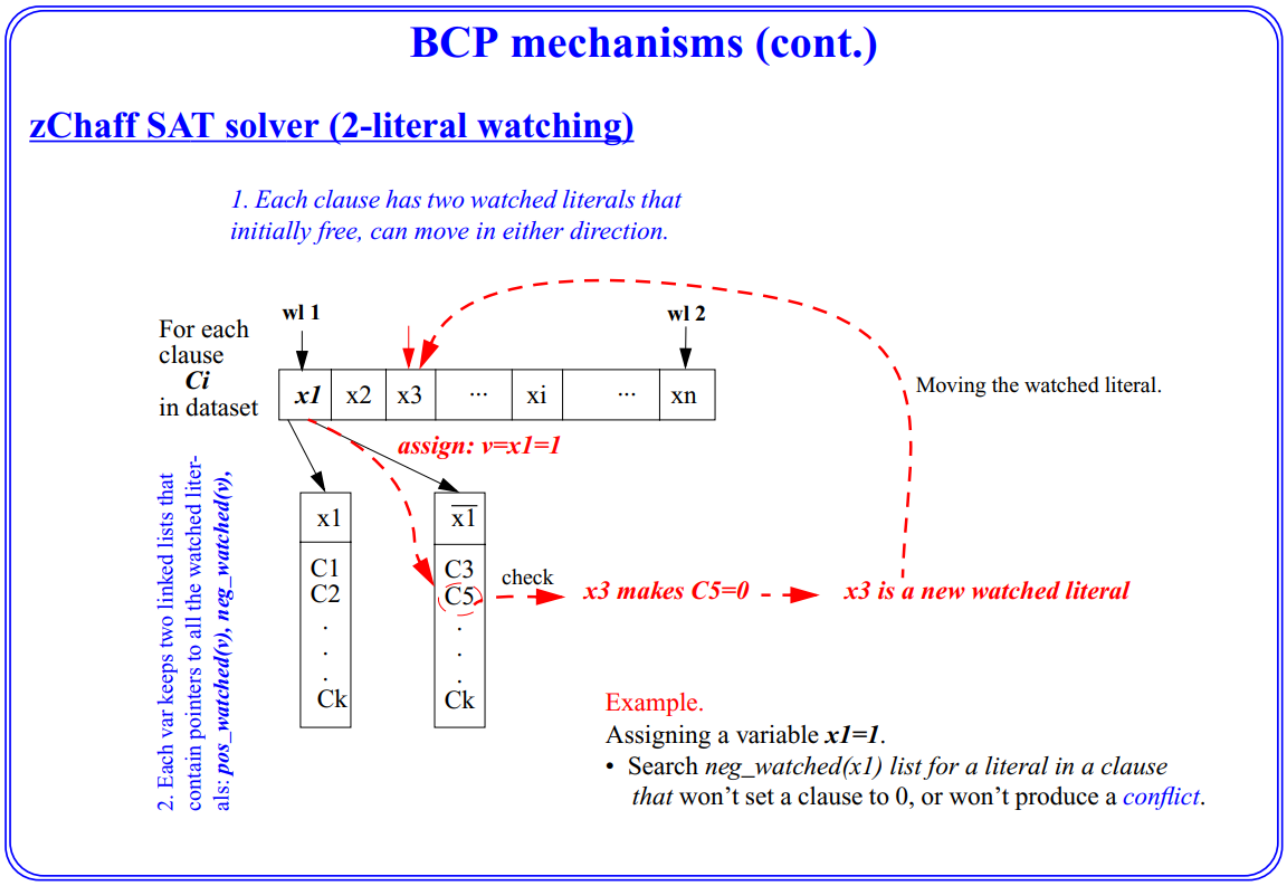
```C++
void CSolver::real_solve(void) {
    while (_stats.outcome == UNDETERMINED) {
        run_periodic_functions();
        // in this function, may:
        // a. restart; b. decay variable score; c. run hook functions
        if (decide_next_branch()) {
        // 如果单元子句不再能传播，启发式地选择一个变量去赋值
            while (deduce() == CONFLICT) {
                int blevel;
                blevel = analyze_conflicts(); // CDCL
                if (blevel < 0) {
                    _stats.outcome = UNSATISFIABLE;
                    return;
```

```
                }
            }
        }
        else {
            if (_sat_hook != NULL && _sat_hook(this))
                continue;
            _stats.outcome = SATISFIABLE;
            return;
        }
        // 处理其他异常情况，如超时、内存不足……
    }
}
```

- 每当状态是UNDETERMINED时，先判断是否还有空闲的变量可赋值
  - 如果有，判断赋值后是否出现CONFLICT矛盾
    - 若有矛盾，进行`analyze_conflicts()`并回溯`backtracking()`
  - 如果没有，继续
- 状态更新为：SAT或UNSAT或OUT_MEM等等

Deduce/UnitPropagate

用2-literal watching判断需要被传播的子句。如图：



每个子句设置两个watched literal：可以被赋值为1的文字。

每个变量设置两个链表：pos_watched(v)和neg_watched(v)，用来存储v被标记为watched literal的那些子句。
当变量v被赋值为1时，由于对$\bar{v}$所在的子句有影响，故遍历neg_watched(v)链表，对其中的每个子句寻找剩余的可以被赋值为1的文字l，有以下情况：

- l是另一个watched literal
  - 若l已被赋值为1，则此子句SAT；
  - 若l是自由文字
    - 除l外找不出一个其他文字是可以被赋值为1的，则这个子句应该被传播
    - 除l外还能找到一个其他文字可以被赋值为1，则把这个文字设为watched literal（替代v）
- 找不到l可以被赋值为1，则出现CONFLICT子句
  任何子句的watched literal尽量避免被设置成0->否则容易CONFLICT

优化思路

本部分是对zChaff的优化，修改了部分核心算法。

原理

用local search的思路优化。
经过调研学习到，一般的local search思路大致如下（伪代码）：

```C++
for (int try_i = 0; try_i < max_tries; try_i++) {
    init_assign()
    // compute scores for each var only once at each try;
    // after flip, only update impacted var
    for (int step_j = 0; step_j < max_steps; step_j++) {
        if (_stats.outcome == SATISFIABLE) return;
        else {
            x = choose var(assign,s) // with max score
            s = s with value of x flipped
            // within limited steps
            // flip a var that satis all clauses
            // otherwise, flip a var that satis most clauses
        }
    }
}
```

即，先给所有变量赋初值，如果遇到CONFLICT，尝试启发式地选取一个变量进行flip（改变其赋值）。如果尝试次数超过一定限制，认为UNKNOWN（不能确定是不是SAT）。
借鉴选取变量赋值的这个思路，我对zChaff的算法进行了修改：
对变量赋值并发现矛盾后，

- 在一定的范围内，尝试：
  - 从导致CONFLICT的变量中随机选取一个，进行flip，并从`_conflicts`子句中消除其所在的子句（`_conflicts`数组存所有产生矛盾的子句，flip后，该变量所在的子句直接SAT，不再有矛盾）
  - 判断`_conflicts`数组是否为空，为空则退出local search循环
- 若local search成功，直接清空`_conflicts`并继续选择新的变量赋值；
- 若local search失败，需要做回溯，消除local search的影响；并且做`analyze_conflicts()`.

核心代码

C++

```
void CSolver::real_solve(void)
{
    int max_steps = 50; // local search尝试的最大次数
    while (_stats.outcome == UNDETERMINED)
    {
        run_periodic_functions();
        if (decide_next_branch()) // 选取一个变量进行赋值
        {
            while (deduce() == CONFLICT) // 赋值后出现CONFLICT
            {
                vector<int>& assignments =
*_assignment_stack[dlevel()]; // dlevel是decision level：目前已赋值（非推导）的变量数目
                vector<int> changed_var_index;
                bool SLS_flag = false;
                for (int step_i = 0; step_i < max_steps; step_i++) {
                    // 在最大次数的范围内进行尝试
                    vector<int> conflict_copy; // 本地对_conflict数组的拷贝，方便回溯
                    conflict_copy.assign(_conflicts.begin(),
_conflicts.end());
                    int flip_index;
                    for (int ass_i = assignments.size() - 1; ass_i
>=0; --ass_i) { // 从目前已赋值的变量中，寻找导致CONFLICT的变量
                        int assigned = assignments[ass_i];
                        if (variable(assigned >> 1).is_marked()) {
                            // 被标记，说明该变量导致CONFLICT
                            variable(assigned >> 1).clear_marked();
                            flip_index = assigned >> 1;
                            break;
                        }
```

```
                    }
                    // now flip
                    changed_var_index.push_back(flip_index); //保存下
标，方便回溯
                    int assigned = assignments[flip_index];
                    int svar = assigned ^ 0x1;
                    variable(flip_index).set_new_cl_phase((assigned ^
0x1) & 0x1); // 重新赋值（取反）
                    for (auto iter = conflict_copy.begin(); iter <
conflict_copy.end(); iter++) {
                    // 从_conflicts数组中消除该变量出现过的子句
                        int clause_i = *iter;
                        int len_i = (int)clause(clause_i).num_lits();
                        for (int j = 0; j < len_i; j++) {
                            auto literal_j =
clause(clause_i).literal(j);
                            if (literal_j.var_index() == flip_index) {
                                conflict_copy.erase(iter);
                                iter--;
                                break;
                            }
                        }
                    }
                    if (conflict_copy.size() == 0) {
                    // 判断local search是否结束
                        SLS_flag = true;
                        break;
                    }
                }

                /* original code is below */
                int blevel = -1;
                if (SLS_flag == false) {// local search失败，需回溯
                    while(!changed_var_index.empty()) {
                        int index = changed_var_index.back();
                        changed_var_index.pop_back();
                        int phase = variable(index).new_cl_phase();
                        variable(index).set_new_cl_phase(phase ^
0x01);

                        variable(index).set_marked();
                    }
                    blevel = analyze_conflicts();
                }
```

```
                else { // local search成功，清空所有矛盾的子句（现在已不矛盾了）
                    blevel = 1;
                    _conflicts.clear();
                }
                if (blevel < 0)
                {
                    _stats.outcome = UNSATISFIABLE;
                    return;
                }
            }
        }
    }
}
```

测试

和原本的zChaff对比，并修改local search尝试的次数：zChaff-LS-X表示使用local search尝试X次。

变量数目	zChaff	zChaff-LS-5	zChaff-LS-50	zChaff-LS-100	zChaff-LS-500
16	0.000162	0.000174	0.000243	0.00032	0.00082
42	0.018902	0.021178	0.026546	0.034056	0.095403
50	0.000728	0.000811	0.001349	0.00137	0.006303
60	0.007918	0.00912	0.012305	0.013608	0.035784
63	0.007851	0.009373	0.012112	0.014982	0.034624
64	0.000755	0.000965	0.001211	0.001481	0.00386
66	0.008167	0.009796	0.012174	0.014073	0.039814
100	0.001847	0.002059	0.003512	0.005047	0.009703
155	0.003235	0.002328	0.005359	0.004299	0.0129
1040	0.053791	0.057644	0.057644	0.122597	0.381011

由此可知，采用local search之后，和原来的zChaff相比，并没有什么提升……（除了变量数目为155的时候）。可能我选的数据集太小了，或者local search回溯的代价太大：我直接全部回溯了，或许也可以回溯一部分，剩下的CONFLICT子句继续做`analyze_conflicts()`。