Practical SAT Solving

Lecture 5

Carsten Sinz, Tomáš Balyo | May 22, 2018
Lecture Outline: Today

- Repetition
- More Details on implementing DPLL
  - Literal Selection Heuristics
  - Efficient Unit Propagation
“Modern” DPLL Algorithm with “Trail”

boolean mDPLL(ClauseSet S, PartialAssignment α) {
    while ((S, α) contains a unit clause {L}) {
        add {L = 1} to α
    }
    if (a literal is assigned both 0 and 1 in α) return false;
    if (all literals assigned) return true;
    choose a literal L not assigned in α occurring in S;
    if (mDPLL(S, α ∪ {L = 1}) return true;
    else if (mDPLL(S, α ∪ {L = 0}) return true;
    else return false;
}

(S, α): clause set S as “seen” under partial assignment α
How can we implement unit propagation efficiently?

(How can we implement pure literal elimination efficiently?)

Which literal \( L \) to use for case splitting?

How can we efficiently implement the case splitting step?
Properties of a good decision heuristic

- Fast to compute
- Yields efficient sub-problems
- More short clauses?
- Less variables?
- Partitioned problem?
Properties of a good decision heuristic

- Fast to compute
- Yields efficient sub-problems
  - More short clauses?
  - Less variables?
  - Partitioned problem?
Bohm’s Heuristic

- Best heuristic in 1992 for random SAT (in the SAT competition)
- Select the variable $x$ with the maximal vector $(H_1(x), H_2(x), \ldots)$

$$H_i(x) = \alpha \max(h_i(x), h_i(\overline{x})) + \beta \min(h_i(x), h_i(\overline{x}))$$

- where $h_i(x)$ is the number of not yet satisfied clauses with $i$ literals that contain the literal $x$.
- $\alpha$ and $\beta$ are chosen heuristically ($\alpha = 1$ and $\beta = 2$).
- Goal: satisfy or reduce size of many preferably short clauses
MOMS Heuristic

- Maximum Occurrences in clauses of Minimum Size
- Popular in the mid 90s
- Choose the variable $x$ with a maximum $S(x)$.

$$S(x) = (f^*(x) + f^*(\overline{x})) \times 2^k + (f^*(x) \times f^*(\overline{x}))$$

- where $f^*(x)$ is the number of occurrences of $x$ in the smallest not yet satisfied clauses, $k$ is a parameter
- Goal: assign variables with high occurrence in short clauses
Jeroslow-Wang Heuristic

- Considers all the clauses, shorter clauses are more important
- Choose the literal $x$ with a maximum $J(x)$.

$$J(x) = \sum_{x \in c, c \in F} 2^{-|c|}$$

- Two-sided variant: choose variable $x$ with maximum $J(x) + J(\overline{x})$
- Goal: assign variables with high occurrence in short clauses
- Much better experimental results than Bohm and MOMS
- One-sided version works better
(R)DLCS and (R)DLIS Heuristics

- (Randomized) Dynamic Largest (Combined | Individual) Sum
- Dynamic = Takes the current partial assignment in account
- Let $C_P$ ($C_N$) be the number of positive (negative) occurrences
- DLCS selects the variable with maximal $C_P + C_N$
- DLIS selects the variable with maximal $\max(C_P, C_N)$
- RDLCS and RDLIS does a random selection among the best
  - Decrease greediness by randomization
- Used in the famous SAT solver GRASP in 2000
LEFV Heuristic

- Last Encountered Free Variable
- During unit propagation save the last unassigned variable you see, if the variable is still unassigned at decision time use it otherwise choose a random
- Very fast computation: constant memory and time overhead
  - Requires 1 int variable (to store the last seen unassigned variable)
- Maintains search locality
- Works well for pigeon hole and similar formulas
How to Implement Unit Propagation

The Task

Given a partial truth assignment $\phi$ and a set of clauses $F$ identify all the unit clauses, extend the partial truth assignment, repeat until fix-point.

Simple Solution

- Check all the clauses
- Too slow
- Unit propagation cannot be efficiently parallelized (is P-complete)
How to Implement Unit Propagation

The Task

Given a partial truth assignment $\phi$ and a set of clauses $F$ identify all the unit clauses, extend the partial truth assignment, repeat until fix-point.

Simple Solution

- Check all the clauses
- Too slow
- Unit propagation cannot be efficiently parallelized (is P-complete)

In the context of DPLL the task is actually a bit different

- The partial truth assignment is created incrementally by adding (decision) and removing (backtracking) variable value pairs
- Using this information we will avoid looking at all the clauses
How to Implement Unit Propagation

The Real Task

We need a data structure for storing the clauses and a partial assignment $\phi$ that can efficiently support the following operations:

- detect new unit clauses when $\phi$ is extended by $x_i = v$
- update itself by adding $x_i = v$ to $\phi$
- update itself by removing $x_i = v$ from $\phi$
- support restarts, i.e., un-assign all variables at once

Observation

- We only need to check clauses containing $x_i$
Occurrences List and Literals Counting

The Data Structure
- For each clause remember the number unassigned literals in it
- For each literal remember all the clauses that contain it

Operations
- If $x_i = T$ is the new assignment look at all the clauses in the occurrence of $\overline{x_i}$. We found a unit if the clause is not SAT and counter=2
- When $x_i = v$ is added or removed from $\phi$ update the counters
"Traditional" Approach

Crawford, Auton (1993)

**Clause**
- int actPos
- int actNeg
- list<Literal> literals
- Variable* subsumedBy

number of positive / negative literals in clause (to detect units)
literals of the clause
pointer to variable by which this clause first was subsumed (or NIL if cl. is not subsumed); needed for backtracking

**Variable**
- enum {0, 1, OPEN} state
- int nrPosOcc
- int nrNegOcc
- list<Clause*> posOccList
- list<Clause*> negOccList

assignment state
number of positive / negative occurrences of variable
pointers to clauses, in which variable occurs positively / negatively

Crawford, Auton (1993)
Traditional Approach: Example

\[ F = \{\{x, \neg y, z\}, \{\neg z\}\} \]
Traditional Approach: Example

\[ F = \{\{x, \neg y, z\}, \{\neg z\}\} \]

unit propagation: set \( z = 0 \)
Traditional Approach: Example

\[ F = \{ \{x, \neg y, z\}, \{\neg z\}\} \quad \text{unit propagation: set } z = 0 \]
Traditional Approach: Example

\[ F = \{ \{ x, \neg y, z \}, \{ \neg z \} \} \]

unit propagation: set \( z = 0 \)
Head/Tail Lists

Zhang, Stickel (1996)
Head/Tail Lists: Example

\[ F = \{ \{ x, \neg y, z \}, \{ \neg z \} \} \]
Head/Tail Lists: Example

\( F = \{ \{ x, \neg y, z \}, \{ \neg z \} \} \)  
detected unit clause: \( \{ \neg z \} \)
Head/Tail Lists: Example

\[ F = \{ \{ x, \neg y, z \}, \{ \neg z \} \} \]

unit propagation: set \( z = 0 \)
F = \{ \{ x, \neg y, z \}, \{ \neg z \} \}

unit propagation: set z = 0
Head/Tail Lists: Example

\[ F = \{ \{x, \neg y, z\}, \{\neg z\}\} \]

unit propagation: set \( z = 0 \)
2 watched literals

The Data Structure

- In each non-satisfied clause "watch" two non-false literals
- For each literal remember all the clauses where it is watched

Maintain the invariant: two watched non-false literals in non-sat clauses

- If a literal becomes false find another one to watch
- If that is not possible the clause is unit

Advantages
2 watched literals

The Data Structure

- In each non-satisfied clause "watch" two non-false literals
- For each literal remember all the clauses where it is watched

Maintain the invariant: two watched non-false literals in non-sat clauses

- If a literal becomes false find another one to watch
- If that is not possible the clause is unit

Advantages

- visit fewer clauses: when \( x_i = T \) is added only visit clauses where \( \overline{x_i} \) is watched
- no need to do anything at backtracking and restarts
  - watched literals cannot become false
2 Watched Literals: Data Structures

Clause
- `int watched_1_index;`
- `int watched_2_index;`
- `vector<Literal> literals`

watched literals
(indices in literal vector)
literals of the clause

Variable
- `enum { 0, 1, OPEN } state`
- `list<Clause*> posWatched`
- `list<Clause*> negWatched`

assignment state
pointers to clauses, in which variable is watched (positively / negatively)
2 Watched Literals: Example

- \neg x_1, x_4, \neg x_5, x_7, x_9
- x_1 = 1
- x_4, \neg x_5, x_7, x_9
- x_5 = 1, x_9 = 0
- \neg x_1, x_4, \neg x_5, x_7, x_9
- x_4 = 0
- \neg x_1, x_4, \neg x_5, x_7, x_9
- x_4 \cdot x_5 \cdot x_9
- \neg x_1, x_4, \neg x_5, x_7, x_9

- \neg x_1 is false
- add. \neg x_5, x_9 are false
- add. \neg x_4 is false, new unit x_7
- no change on watched literals
Good for parallel SAT solvers with shared clause database
MiniSat

 invariant: first two literals are watched
PicoSat

-2  head

-8  -2  7  1

1  -2

-2  3  -5

Invariant: first two literals are watched
Lingeling

- often the other watched literal satisfies the clause
- for binary clauses no need to store the clause
CRef Solver::propagate()
{
    CRef confl = CRef_Undef;
    int num_props = 0;

    while (qhead < trail.size()){
        Lit p = trail[qhead++]; // propagate 'p'.
        vec<Watcher>& ws = watches.lookup(p);
        Watcher *i, *j, *end;
        num_props ++;
        for (i = j = (Watcher*)ws , end = i + ws.size();
            i != end ;){
            // Try to avoid inspecting the clause:
            Lit blocker = i->blocker;
            if (value(blocker) == l_True){
                *j++ = *i ++; continue ;
            }
            // Make sure the false literal is data[1]:
            CRef cr = i->cref;
            Clause& c = ca[cr];
            Lit false_lit = ~p;
            if (c[0] == false_lit)
            c[0] = c[1]; c[1] = false_lit;
            assert(c[1] == false_lit);
            i ++;

            // If 0th watch is true, clause is satisfied.
            Lit first = c[0];
            Watcher w = Watcher(cr, first);
            if (first != blocker && value(first) == l_True){
                *j++ = w; continue ;
            }

            // Look for new watch:
            for (int k = 2; k < c.size(); k++)
            if (value(c[k]) != l_False){
                c[1] = c[k]; c[k] = false_lit;
                watches[~c[1]].push(w);
                goto NextClause ;
            }
        }
        ws.shrink(i - j);
    }
    propagations += num_props;
    simpDB_props -= num_props;
    return confl ;
}

// Look for new watch:
for (int k = 2; k < c.size(); k++)
if (value(c[k]) != l_False){
    c[1] = c[k]; c[k] = false_lit;
    watches[~c[1]].push(w);
    goto NextClause ;
}

// Did not find watch -- clause is unit
*j++ = w;
if (value(first) == l_False){
    confl = cr;
    qhead = trail.size();
    // Copy the remaining watches:
    while (i < end)
    *j++ = *i++;
} else
    uncheckedEnqueue(first, cr);

NextClause :;
}
}

ws.shrink(i - j);
}

propagations += num_props;
simpDB_props -= num_props;
return confl ;
}