Practical SAT Solving

Lecture 5

Carsten Sinz, Tomáš Balyo | May 21, 2019
Stochastic Local Search (SLS)

SAT as an optimization problem: minimize the number of unsatisfied clauses

Start with a complete random assignment $\alpha$:

```
0 0 1 0 1 1 0 0 1 1 0 1 0 1 0
```

Repeatedly flip (randomly/heuristically chosen) variables to decrease the number of unsatisfied clauses:

```
0 0 1 0 1 0 0 0 1 1 0 1 0 1 0
```
SLS Algorithms

- Local search algorithms are **incomplete**: they cannot show unsatisfiability!
- Many variants of local search algorithms
- Main question: Which variable should be flipped next?
  - select variable from an unsatisfied clause
  - select variable that increases the number of satisfied clauses most
- How to avoid local minima?
Maybe[Assignment]  GSAT(ClauseSet S)
{
    for i = 1 to MAX_TRIES do {
        α = random-assignment to variables in S
        for j = 1 to MAX_FLIPS do {
            if (α satisfies all clauses in S) return α
            x = variable that produces least number of
                unsatisfied clauses when flipped
            flip x
        }
    }
    return Nothing // no solution found
}
SLS: Illustration

[Source: Alan Mackworth, UBC, Canada]
Walksat [2]

- Variant of GSAT
- Try to avoid local minima by introducing “random noise”
  - Select unsatisfied clause $C$ at random
  - If by flipping a variable $x \in C$ no new unsatisfied clauses emerge, flip $x$
  - Otherwise:
    - With probability $p$ select a variable $x \in C$ at random
    - With probability $1 - p$ select a variable that changes as few as possible clauses from satisfied to unsatisfied when flipped
Consider a flip taking $\alpha$ to $\alpha'$

- **breakcount**: number of clauses satisfied in $\alpha$, but not in $\alpha'$
- **makecount**: number of clauses unsatisfied in $\alpha$, but satisfied in $\alpha'$
- **diffscore**: number of unsatisfied clauses in $\alpha$ minus number of clauses unsatisfied in $\alpha'$

Typically, **breakcount**, **makecount** and **diffscore** are updated after each flip.

Question: How can we do this efficiently?
GSAT and Walksat Flip Heuristics

- **GSAT**: select variable with highest **diffscore**
- **Walksat**:
  - First randomly select unsatisfied clause $C$
  - If there is a variable with **breakcount** 0 in $C$, select it
  - Otherwise with probability $p$ select a random variable from $C$, and with probability $1 - p$ a variable with minimal **breakcount** from $C$
Runtime Comparison Walksat vs. GSAT

Table 4: Comparing an efficient complete method (DP) with local search strategies on circuit synthesis problems. (Timings in seconds.)

<table>
<thead>
<tr>
<th>formula</th>
<th>id</th>
<th>vars</th>
<th>clauses</th>
<th>DP time</th>
<th>GSAT+w time</th>
<th>WSAT time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2bitadd.12</td>
<td>708</td>
<td>1702</td>
<td>*</td>
<td>0.081</td>
<td>0.013</td>
<td></td>
</tr>
<tr>
<td>2bitadd.11</td>
<td>649</td>
<td>1562</td>
<td>*</td>
<td>0.058</td>
<td>0.014</td>
<td></td>
</tr>
<tr>
<td>3bitadd.32</td>
<td>8704</td>
<td>32316</td>
<td>*</td>
<td>94.1</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>3bitadd.31</td>
<td>8432</td>
<td>31310</td>
<td>*</td>
<td>456.6</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>2bitcomp.12</td>
<td>300</td>
<td>730</td>
<td>23096</td>
<td>0.009</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>2bitcomp.5</td>
<td>125</td>
<td>310</td>
<td>1.4</td>
<td>0.009</td>
<td>0.001</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Comparing DP with local search strategies on circuit diagnosis problems by Larrabee (1989). (Timings in seconds.)

<table>
<thead>
<tr>
<th>formula</th>
<th>id</th>
<th>vars</th>
<th>clauses</th>
<th>DP time</th>
<th>GSAT+w time</th>
<th>WSAT time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa7552-038</td>
<td>1501</td>
<td>3575</td>
<td>*</td>
<td>7</td>
<td>129</td>
<td>2.3</td>
</tr>
<tr>
<td>ssa7552-158</td>
<td>1363</td>
<td>3034</td>
<td>*</td>
<td>90</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ssa7552-159</td>
<td>1363</td>
<td>3082</td>
<td>*</td>
<td>14</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>ssa7552-160</td>
<td>1391</td>
<td>3126</td>
<td>*</td>
<td>18</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>

[Source: Selman, Kautz, Cohen Local Search Strategies for Satisfiability Testing, 1993]
Repetition: Resolution/Saturation

Saturation Algorithm

- **INPUT:** CNF formula $F$
- **OUTPUT:** \{SAT, UNSAT\}

```
while (true) do
    $R = \text{resolveAll}(F)$
    if ($R \cap F \neq R$) then $F = F \cup R$
    else break

if ($\bot \in F$) then return UNSAT else return SAT
```

Properties of the saturation algorithm:

- it is sound and complete – always terminates and answers correctly
- has exponential time and space complexity
Can we do better?

- Question: Can we do better than saturation-based resolution?
  - Avoid exponential space complexity
  - Improve average-case complexity (for important problem classes)
Davis-Putnam Algorithm [3]

- Presented in 1960 as a procedure for first-order (predicate) logic
- Procedure to check satisfiability of a formula $F$ in CNF
- Three (deduction) rules:
  1. **Unit propagation**: if there is a unit clause $C = \{l\}$ in $F$, simplify all other clauses containing $l$
  2. **Pure literal elimination**: If a literal $l$ never occurs negated in $F$, add the clause $\{l\}$ to $F$
  3. **Case splitting**: Assume that $F$ is put in the form $(A \lor l) \land (B \lor \bar{l}) \land R$, where $A$, $B$, and $R$ are free of $l$. Replace $F$ by the clausification of $(A \lor B) \land R$
- Apply deduction rules (giving priority to rules 1 and 2) until no further rule is applicable
The superiority of the present procedure over those previously available is indicated in part by the fact that a formula on which Gilmore’s routine for the IBM 704 causes the machine to compute for 21 minutes without obtaining a result was worked successfully by hand computation using the present method in 30 minutes.
DPLL Algorithm: Outline

- Algorithmic improvements over DP algorithm
- Basic idea: case splitting and simplification
- Simplification: unit propagation and pure literal deletion
- Unit propagation: 1-clauses (unit clauses) fix variable values: if \( \{x\} \in S \), in order to satisfy \( S \), variable \( x \) must be set to 1.
- Pure literal deletion: If variable \( x \) occurs only positively (or only negatively) in \( S \), it may be fixed, i.e. set to 1 (or 0).
Let $F_0 = \{\{x, y\}, \{\neg x, y, \neg z\}, \{\neg x, z, u\}, \{x, \neg u\}\}$.

All clauses containing $y$ may be deleted, as $y$ occurs only positively in $F$. This yields:

$$F_1 = \{\{\neg x, z, u\}, \{x, \neg u\}\}$$

Each solution $\alpha_1$ of $F_1$ can be extended to a solution $\alpha_0$ of $F_0$ by setting $\alpha_0(y) = 1$.

Moreover, if $F_1$ does not possess a solution, then so does $F_0$.

Repeating yields $F_2 = \{\{x, \neg u\}\}$ and $F_3 = \emptyset$, thus $F_0$ is satisfiable.
DPLL Algorithm

```java
boolean DPLL(ClauseSet S) {
    while (S contains a unit clause \{L\}) {
        delete from S clauses containing L;  // unit-subsumption
        delete \neg L from all clauses in S;  // unit-resolution
    }
    if (\bot \in S) return false;          // empty clause?
    while (S contains a pure literal L) {
        delete from S all clauses containing L;
        if (S = \emptyset) return true;    // no clauses?
        choose a literal L occurring in S;  // case-splitting
        if (DPLL(S U \{\{L\}\}) return true;  // first branch
        else if (DPLL(S U \{\{-L\}\}) return true;  // second branch
        else return false;
    }
}
```
DPLL: Implementation Issues

- How can we implement unit propagation efficiently?
- Which literal $L$ to use for case splitting?
- How can we efficiently implement the case splitting step?
“Modern” DPLL Algorithm with “Trail”

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

($S, \alpha$): clause set $S$ as “seen” under partial assignment $\alpha$
DPLL: Implementation Issues

- 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

**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

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

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

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

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

- **Clause**
  - int headIndex
  - int tailIndex
  - list<Literal> literals
  - Index of first/last unassigned literal (to detect units)
  - Literals of the clause

- **Variable**
  - enum { 0, 1, OPEN } state
  - list<Clause*> posHeadList
  - list<Clause*> negHeadList
  - list<Clause*> posTailList
  - list<Clause*> negTailList
  - Assignment state
  - Pointers to clauses, in which first/last unassigned variable occurs positively/negatively

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

\[ F = \{\{x, \neg y, z\}, \{\neg z\}\} \quad \text{unit propagation: set } z = 0 \]
Head/Tail Lists: Example

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

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

- $\neg x_1$
- $x_4$
- $\neg x_5$
- $x_7$
- $x_9$

$x_1 = 1$ → $\neg x_1$
- $x_4$
- $\neg x_5$
- $x_7$
- $x_9$

DPLL rec. call 1

$x_5 = 1$ → $\neg x_5$
- $x_4$
- $\neg x_5$
- $x_7$
- $x_9$

DPLL rec. call 2

$x_4 = 0$ → $\neg x_4$
- $x_4$
- $\neg x_5$
- $x_7$
- $x_9$

DPLL rec. call 3

Conflict, backtrack last two decisions

$\neg x_1$ 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
invariant: first two literals are watched
PicoSat

 invariant: first two literals are watched
- Often the other watched literal satisfies the clause.
- For binary clauses no need to store the clause.
**MiniSAT** 

**propagate()** - Function

```c
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 ; }

            // 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);
                }
        }
        ws.shrink(i - j);
    }
    propagations += num_props;
    simpDB_props -= num_props;
    return confl ;
}
```

---

**Davis-Putnam DPLL Algorithm Heuristics Unit Propagation**

Carsten Sinz, Tomáš Balyo – SAT Solving  
May 21, 2019  
48/48
URL http://dl.acm.org/citation.cfm?id=1867135.1867203


URL http://doi.acm.org/10.1145/321033.321034