Practical SAT Solving
Lecture 5
Carsten Sinz, Tomáš Balyo | May 23, 2016
Lecture Outline

- Details on implementing DPLL
  - Decision Heuristics
  - Restarts
  - Implementing Unit Propagation
DPLL Algorithm: Outline

- **DPLL**: Davis-Putnam-Logemann-Loveland
- **Basic idea**: case splitting (Depth first search on the partial assignments) 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).
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 \cup \{\{L\}\}) return true; // first branch
    else if (DPLL(S \cup \{\{\neg L\}\}) return true; // second branch
    else return false;
}
```
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?
"Modern" DPLL Algorithm with "Trail"

```java
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
```
Properties of a good decision heuristic
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 unsatisfied 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 unsatisfied 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
Restarts

- What is a restart?
Restarts

- What is a restart?
  - Clear the partial assignment
  - Unassign all the variables
  - Backtrack to level 0

Why would anybody want to do restarts in DPLL?
- To recover from bad branching decisions
- You solve more instances
- Might decrease performance on easy instances

Carsten Sinz, Tomáš Balyo – SAT Solving
May 23, 2016
13/28
Restarts

- What is a restart?
  - Clear the partial assignment
  - Unassign all the variables
  - Backtrack to level 0
- Why would anybody want to do restarts in DPLL?
What is a restart?
- Clear the partial assignment
- Unassign all the variables
- Backtrack to level 0

Why would anybody want to do restarts in DPLL?
- To recover from bad branching decisions
- You solve more instances
- Might decrease performance on easy instances
When to Restart?

- After a given number of decisions
- The number of decision between restarts should grow
  - To guarantee completeness
- How much increase?
  - Linear increase – too slow
  - Exponential increase – ok with small exponent
  - MiniSat: $k$-th restart happens after $100 \times 1.1^k$
Inner/Outer Restart Scheduling
Inner/Outer Restart Scheduling

Inner/Outer Restart Algorithm

```c
int inner = 100
int outer = 100

forever do
    do DPLL for inner conflicts ... 
    restarts++
    if inner >= outer then
        outer *= 1.1
        inner = 100
    else
        inner *= 1.1
```

Carsten Sinz, Tomáš Balyo – SAT Solving

May 23, 2016

16/28
Luby Sequence Restart Scheduling
Luby Sequence Algorithm

```c
unsigned luby (unsigned i)
    for (unsigned k = 1; k < 32; k++)
        if (i == (1 << k) - 1) then return 1 << (k - 1)
    for (k = 1;; k++)
        if ((1 << (k - 1)) <= i && i < (1 << k) - 1) then
            return luby(i - (1 << (k-1)) + 1);

limit = 512 * luby (++restarts);
... // run SAT core loop for limit conflicts
```

- Complicated, not trivial to compute
Reluctant Doubling

- A more efficient implementation of the Ruby sequence
- Use the $v_n$ of the following pair

\[(u_1, v_1) = (1, 1)\]  \hspace{1cm} (1)

\[(u_{n+1}, v_{n+1}) = u_n & - u_n = v_n ? (u_n + 1, 1) : (u_n, 2v_n)\]  \hspace{1cm} (2)

- Example: (1,1), (2,1), (2,2), (3,1), (4,1), (4,2), (4,4), (5,1), ...
- Invented by Donald Knuth
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 list 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
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 less 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
zChaff

Literals

<table>
<thead>
<tr>
<th>1</th>
<th>start top end</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>start top end</td>
</tr>
<tr>
<td>2</td>
<td>start top end</td>
</tr>
<tr>
<td>-3</td>
<td>start top end</td>
</tr>
</tbody>
</table>

Stack

Clauses

-8

1  -2  7  -8

-2  3  -5

1

3
Limmat

-2

start
top
end

Watcher of A

-8
1 -2 7 -8
A

Watcher of B

-2 3 -5
B

3

Good for parallel SAT solvers with shared clause database
invariant: first two literals are watched
invariant: first two literals are watched
- Often the other watched literal satisfies the clause.
- For binary clauses no need to store the clause.