Lecture Outline

- VSIDS heuristic
- Clause forgetting
- Parallel SAT Solving
**CDCL Algorithm**

```plaintext
input : Formula F in CNF
output: SAT / UNSAT

1. \( dl \leftarrow 0 \)  // initialize decision level
2. \( V \leftarrow \emptyset \)  // initialize trail (variable assignment)
3. while not all variables assigned do
   4. if unit_propagation\((F, V)\) == CONFLICT then
      5. \((c, bl)\) ← analyze conflict
      6. if \( bl < 0 \) then
         7. return UNSAT
      8. else
         9. add_clause\((c)\)
         10. backtrack to \( bl \)
         11. \( dl \leftarrow bl \)
   12. else
      13. \((x, b)\) ← pick branching literal
      14. \( dl \leftarrow dl + 1 \)
      15. \( V \leftarrow V \cup \{(x, b)\} \)
5. return SAT
```

**CDCL Round-Up**

Carsten Sinz, Tomáš Balyo – SAT Solving

June 11, 2019
Variable Selection in CDCL

- Previous heuristics (MOMS, Bohm’s, etc.): global, “static”
  - E.g. MOMS: \( S(x) = (f^*(x) + f^*(\overline{x})) \times 2^k + (f^*(x) \times f^*(\overline{x})) \)
  - \( f^*(x) \) is the number of occurrences of \( x \) in the smallest not yet satisfied clauses, \( k \) is a parameter
  - static: \( S(x) \) often computed only at root node of search
  - global: based on whole CNF
Variable Selection in CDCL

- Previous heuristics (MOMS, Bohm’s, etc.): global, “static”
  - E.g. MOMS: $S(x) = (f^*(x) + f^*(\overline{x})) \times 2^k + (f^*(x) \times f^*(\overline{x}))$
  - $f^*(x)$ is the number of occurrences of $x$ in the smallest not yet satisfied clauses, $k$ is a parameter
  - static: $S(x)$ often computed only at root node of search
  - global: based on whole CNF

- Idea for CDCL: Make heuristics more “focused”
  - try to find small unsatisfiable subsets
  - prefer variables that occurred in a recent conflict
VSIDS Heuristic

- **VSIDS: Variable State Independent Decaying Sum**
  - **General approach**: Compute score for each variable, select variable with highest score
  - Initial variable score is number of literal occurrences
  - New conflict clause $c$: Score is incremented for all variables in $c$
  - Periodically, divide all scores by a constant

First presented in SAT solver Chaff, 2001 [1]
VSIDS (or a variant of it) implemented in most current CDCL solvers
VSIDS Heuristic

- VSIDS: Variable State Independent Decaying Sum
  - General approach: Compute score for each variable, select variable with highest score
  - Initial variable score is number of literal occurrences
  - New conflict clause $c$: Score is incremented for all variables in $c$
  - Periodically, divide all scores by a constant

- First presented in SAT solver Chaff, 2001 [1]
VSIDS Heuristic

- **VSIDS**: Variable State Independent Decaying Sum
  - **General approach**: Compute score for each variable, select variable with highest score
  - Initial variable score is number of literal occurrences
  - New conflict clause $c$: Score is incremented for all variables in $c$
  - Periodically, divide all scores by a constant

- First presented in SAT solver Chaff, 2001 [1]

- VSIDS (or a variant of it) implemented in most current CDCL solvers
VSIDS Example

Initial $F$:
\[
\{x_1, x_4\} \\
\{x_1, \overline{x_3}, \overline{x_8}\} \\
\{x_1, x_8, x_{12}\} \\
\{x_2, x_{11}\} \\
\{\overline{x_7}, \overline{x_3}, x_9\} \\
\{\overline{x_7}, x_8, x_9\} \\
\{x_7, x_8, \overline{x_{10}}\}
\]

Scores:
4 : $x_8$
3 : $x_1, x_7$
2 : $x_3$
1 : $x_2, x_4, x_9, x_{10}, x_{11}, x_{12}$
VSIDS Example

Initial $F$:  
\[
\begin{align*}
\{x_1, x_4\} \\
\{x_1, \overline{x_3}, \overline{x_8}\} \\
\{x_1, x_8, x_{12}\} \\
\{x_2, x_{11}\} \\
\{x_7, \overline{x_3}, x_9\} \\
\{x_7, x_8, x_9\} \\
\{x_7, x_8, \overline{x_{10}}\}
\end{align*}
\]

$F$ with new learned clause added:  
\[
\begin{align*}
\{x_1, x_4\} \\
\{x_1, \overline{x_3}, \overline{x_8}\} \\
\{x_1, x_8, x_{12}\} \\
\{x_2, x_{11}\} \\
\{x_7, \overline{x_3}, x_9\} \\
\{x_7, x_8, x_9\} \\
\{x_7, x_8, \overline{x_{10}}\} \\
\{x_7, x_{10}, \overline{x_{12}}\} \quad \text{(new learned clause)}
\end{align*}
\]

Scores:  
4 : $x_8$  
3 : $x_1, x_7$  
2 : $x_3$  
1 : $x_2, x_4, x_9, x_{10}, x_{11}, x_{12}$

Scores:  
4 : $x_8, x_7$  
3 : $x_1$  
2 : $x_3, x_{10}, x_{12}$  
1 : $x_2, x_4, x_9, x_{11}$
Possible: Keep list of variables sorted by score
Implementation of VSIDS

- **Possible**: Keep list of variables sorted by score
- **Many implementations**: Use priority queues
  - **Operations**:
    - `insert_with_priority`, `pull_highest_priority_element`
Implementation of VSIDS

- **Possible:** Keep list of variables sorted by score
- **Many implementations:** Use priority queues
  - Operations:
    - `insert_with_priority`, `pull_highest_priority_element`
- **Often implemented as binary heaps**
  - Insert: $O(\log n)$
  - Delete: $O(\log n)$
  - Peek: $O(1)$
Variants of VSIDS

**Question:** Why periodically divide scores?

**Answer:** Give priority to recently learned clauses

Chaff: half scores every 256 conflicts (“decay”); sort priority queue after each decay only

**Variants of VSIDS:**

- Berkmin’s strategy (Berkmin, 2002) – bump all literals in implication graph, divide scores by 4
- VMTF: variable move to front (Siege, 2004)
- CMTF: clause move to front (HaifaSAT, 2008)
- eVSIDS – exponential VSIDS
Comparison of Heuristics

SAT Competition 2013 Application Track Benchmarks Solved by Lingeling

static
inc
sum
vmtf
vsids256
evsids
avg
sc13
Learned Clause Removal

- **Problem:** Too many learned clauses!
  - ...and not all of them are helpful (e.g. subsumed clauses)
  - BCP gets slower, memory consumption

- **Solution:** Forget clauses after some time
  - also called **Clause Database Reduction**
  - **size heuristics:** discard long clauses
  - **least recently used (LRU) heuristics:** discard clauses not involved in recent conflict clause generation
  - “**Glucose level**”: number of distinct decision levels in learned clauses
    (called LBD in original paper [2])
1994 First parallel implementation of DPLL
completely distributed (no master and slave roles)
A list of partial assignment is generated
Each processors receives the entire formula and a few partial assignments
Each Processors consists of
  - Worker (solve or split the formula, use the partial assignments)
  - Balancer (estimate workload, communicate, stopping)
If a worker has nothing to do (all its partial assignments lead to UNSAT) a balancing process is launched.
- Centralized master-slave architecture
- Communication only between master and slaves
- Master assigns partial assignments based on the **Guiding Path**
  - Each node in the search tree is open or closed (closed means one branch is explored)
  - Master splits the open nodes and assigns job to slaves
- All processors can get stuck on unpromising branches
Guiding Path Example

guiding path
{(X1,0,0),(X3,1,0),(X4,1,1),(X5,0,0)}

*** : explored branch
+ : current node
? : remaining subtree
The solver \textit{Satz} improves PSATO the by adding \textit{work stealing} for workload balancing.

- An idle slave request work from the master.
- The master splits the work of the most loaded slave.
- The idle slave and most loaded slave get the parts.
2001 – Clause learning invented

I FREAKING LOVE LEARNING!!!!
Clause Sharing Parallel Solvers

- 2001, Blochinger et al.: PaSAT – the first parallel DPLL with "intelligent backtracking" and clause sharing
  - Similar to PSATO and SATZ: master slave, guiding path, randomized work stealing
- 2004, Feldman et al. – the first shared memory parallel solver
  - Multi-core processors started to be popular
  - Uses same techniques as the previous solvers (guiding path etc.)
  - Bad performance explained by high number of cache misses (DPLL/CDCL is otherwise highly optimized for cache)
- ... and many many more similar solvers
Cube and Conquer

**Basic Idea**

Generate a large amount of partial assignments (millions) and then assign each to one of the slaves.

- it is unlikely that any of the slaves will run out tasks
- The partial assignments are usually generated using a look-ahead solver (breadth-first search up to a limited depth)
- Examples of such solvers
  - march (Heule) + iLingeling (Biere) introduced the idea in 2011
  - Treengeling (Biere) – still state of the art for combinatorial problems
  - This kind of solver was used in the 200TB proof
Pure Portfolios

Basic Idea
Each processor works on the entire problem (no partial assignment restrictions). Each processors uses a (slightly) different solver (different heuristics, random seeds, etc.) All processors stop when one solver solves the problem.

- PPfolio – winner of Parallel Track in the 2011 SAT Competition
  - It is just a bash script that combines the best solvers from the 2010 Competition
  - The author: “it’s probably the laziest and most stupid solver ever written, which does not even parse the CNF and knows nothing about the clauses”
  - This kind of solvers is not allowed since then in SAT Competitions
Portfolios with Clause Learning

- Same as pure portfolio but clauses are shared
- Usually the same solver with different parameters is used for each processor
- 2009, Hamadi et al.¹: ManySAT – the first solver using this idea (based on MiniSat)

¹Microsoft® Research
Portfolios with Clause Learning

- Same as pure portfolio but clauses are shared
- Usually the same solver with different parameters is used for each processor
- 2009, Hamadi et al.\(^1\): ManySAT – the first solver using this idea (based on MiniSat)

This is most successful approach since then

\(^1\)Microsoft\® Research
What Makes a Good Portfolio Solver

Two Pillars of Portfolios:

**Diversification**
- The search space of the solvers should not overlap too much
- Use different configuration values of heuristic parameters
- Partial assignment recommendations (no restrictions!)

**Clause Sharing**
- Which clauses to share?
- How many?
- How often?
- How to implement efficiently?
Experiments – Random Satisf. 3-SAT

Satisfiable Instances

No Diversification, No Sharing
Only Sharing
Only Diversification
Diversification and Sharing

Time in seconds

Problems

CDCL Round-Up
Carsten Sinz, Tomáš Balyo – SAT Solving
June 11, 2019
21/32
Advice for Satisfiable problems

DIVERSIFY

YOUR PORTFOLIO
Experiments – Random Unsat. 3-SAT

Unsatisfiable Instances

- No Diversification, No Sharing
- Only Sharing
- Only Diversification
- Diversification and Sharing

Clause sharing is important for UNSAT

CDCL Round-Up
Carsten Sinz, Tomáš Balyo – SAT Solving
June 11, 2019
Clause sharing is important for UNSAT
A recent portfolio implementation

- HordeSAT – a Massively Parallel SAT Solver
- A scalable SAT solver for up to 2048 processors
HordeSAT Design Principles

- **Modular Design**
  - blackbox approach to SAT solvers
  - any solver implementing a simple interface can be used

- **Decentralization**
  - all nodes are equivalent, no central/master nodes

- **Overlapping Search and Communication**
  - search procedure (SAT solver) never waits for clause exchange
  - at the expense of losing some shared clauses

- **Hierarchical Parallelization**
  - running on clusters of multi-cpu nodes
  - shared memory inter-node clause sharing
  - message passing between nodes
Modular Design

Portfolio Solver Interface

```cpp
void addClause(vector<int> clause);
SatResult solve(); // SAT, UNSAT, UNKNOWN
void setSolverInterrupt();
void unsetSolverInterrupt();
void setPhase(int var, bool phase);
void diversify(int rank, int size);
void addLearnedClause(vector<int> clause);
void setLearnedClauseCallback(LCCallback* clb);
void increaseClauseProduction();
```

- Lingeling implementation with just glue code
- MiniSat implementation, small modification for learned clause stuff
## Diversification

### Setting Phases – "void setPhase(int var, bool phase)"

- **Random** – each variable random phase on each node
- **Sparse** – each variable random phase on exactly one node
- **Sparse Random** – each variable random phase with prob. $\frac{1}{\#\text{solvers}}$

### Native Diversification – "void diversify(int rank, int size)"

- Each solver implements in its own way
- Example: random seed, restart/decision heuristic
- For lingeling we used plingeling diversification

- Best is to use Sparse Random together with Native Diversification.
## Clause Sharing

Regular (every 1 second) collective all-to-all clause exchange

### Exporting Clauses
- Duplicate clauses filtered using Bloom filters
- Clause stored in a fixed buffer, when full clauses are discarded, when underfilled solvers are asked to produce more clauses
- Shorter clauses are preferred
- Concurrent Access – clauses are discarded

### Importing Clauses
- Filtering duplicate clauses (Bloom filter)
  - Bloom filters are regularly cleared – the same clauses can be imported after some time
  - Useful since solvers seem to "forget" important clauses
Overall Algorithm

The Same Code for Each Process

SolveFormula(F, rank, size)

    for i = 1 to numThreads do
        s[i] = new PortfolioSolver(Lingeling);
        s[i].addClauses(F);
        diversify(s[i], rank, size);
        new Thread(s[i].solve());

forever do
    sleep(1) // 1 second
    if (anySolverFinished) break;
    exchangeLearnedClauses(s, rank, size);
Experiments – SAT 2011+2014

Time in seconds
Problems
Lingeling
1x4x4
2x4x4
4x4x4
8x4x4
16x4x4
32x4x4
64x4x4
128x4x4

Time in seconds
Problems
Lingeling
1x4x4
2x4x4
4x4x4
8x4x4
16x4x4
32x4x4
64x4x4
128x4x4

CDCL Round-Up
Carsten Sinz, Tomáš Balyo – SAT Solving
June 11, 2019
30/32
Experiments – Speedups

Big Instance = solved after $10 \cdot (#\text{threads})$ seconds by Lingeling

<table>
<thead>
<tr>
<th>Core Solvers</th>
<th>Parallel Solved</th>
<th>Both Solved</th>
<th>Speedup All</th>
<th>Speedup Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x4x4</td>
<td>385</td>
<td>363</td>
<td>303</td>
<td>25.01</td>
</tr>
<tr>
<td>2x4x4</td>
<td>421</td>
<td>392</td>
<td>310</td>
<td>30.38</td>
</tr>
<tr>
<td>4x4x4</td>
<td>447</td>
<td>405</td>
<td>323</td>
<td>41.30</td>
</tr>
<tr>
<td>8x4x4</td>
<td>466</td>
<td>420</td>
<td>317</td>
<td>50.48</td>
</tr>
<tr>
<td>16x4x4</td>
<td>480</td>
<td>425</td>
<td>330</td>
<td>65.27</td>
</tr>
<tr>
<td>32x4x4</td>
<td>481</td>
<td>427</td>
<td>399</td>
<td>83.68</td>
</tr>
<tr>
<td>64x4x4</td>
<td>476</td>
<td>421</td>
<td>377</td>
<td>104.01</td>
</tr>
<tr>
<td>128x4x4</td>
<td>476</td>
<td>421</td>
<td>407</td>
<td>109.34</td>
</tr>
</tbody>
</table>
Experiments – Speedups on Big Inst.

Big Instance = solved after $10 \cdot (\#\text{threads})$ seconds by Lingeling