Overcoming Resolution- Based Lower Bounds for SAT Solvers

DoRon B. Motter and Igor L. Markov
University of Michigan, Ann Arbor

IWLS 2002
Motivation

Boolean Satisfiability (SAT) has widespread applications
- EDA: Equivalence checking, BMC, Routing, AI: Planning, etc.
- New applications are constantly emerging

Fast SAT solvers abound (GRASP, Chaff, BerkMin)
- Highly tuned implementations improved over years

Many small instances are still difficult to solve

Our Approach
- Algorithms which lead to different classes of tractable instances
- Seek improvements to these algorithms
Motivation

Complete SAT solvers are typically based on DLL
- Resolution-based lower bounds apply to these solvers
- Empirically Chaff, Grasp take exponential time on pigeonholes, etc.

Previous Work:
- We introduced the Compressed Breadth-First Search (CBFS)
- Empirical measurements: our implementation, Cassatt, spends $\Theta(n^4)$ time on pigeonhole-$n$ instances
  - Pigeonhole instances are of size $\Theta(n^3)$
- Analytically: CBFS refutes pigeonhole instances in poly time
  - Resolution-based lower bounds do not apply to CBFS

This Work:
- We augment CBFS with pruning based on the unit clause rule (BCP)
Empirical Performance

Runtime for instances of the pigeon-hole problem

- Cassatt
- Chaff
- ZRes
- GRASP

Runtime for instances of the pigeon-hole problem

- Cassatt
  
- $x^{**4}/800000$
Outline

- Boolean Satisfiability
- Overview of Compressed BFS
- Background
  - Partial Truth Assignments + Open Clauses
  - Zero Suppressed Binary Decision Diagrams
  - Boolean Constraint Propagation
- Compressed BFS
  - Overview
  - Example
- BCP + Compressed BFS
  - Example
  - Extensions
- Results
- Conclusion
Boolean Satisfiability

Boolean Satisfiability (SAT)

- Instance: formula \( \varphi \) in Conjunctive Normal Form (CNF)
  - \( V \): set of variables \{a, b, \ldots, n\}
  - \( C \): set of clauses
    - Each clause is a set of literals over \( V \)
- Question: Is there an assignment to \{a, b, \ldots, n\} which makes this formula true?

Known to be NP-complete

- Unlikely any algorithm will efficiently solve all instances

Many practical applications in EDA

- Bounded model checking, equivalence checking, circuit layout
Compressed-BFS: Overview

In Breadth First Search
- Store “promising” partial solutions of a given depth
- Iteratively increase depth until all variables are processed
  - Main data structure is a set/queue of partial truth assignments

In Compressed BFS
- Store a set of clauses instead of a “promising” partial truth assignment
  - This is enough information to determine satisfiability
- Manipulate all such sets in a compressed form
  - Main data structure is a collection of sets
Background: Partial Assignments

\[ \varphi = (a + c + d)(\overline{g} + \overline{h})(b + e + f)(d + \overline{e}) \]

Partial truth assignment

- Assignment to some \( \mathcal{V} \subseteq V \)
- Consider any assignment to \( \{a, b, c, d\} \):
  - If it is valid, \( a + c + d \) must be satisfied
  - \( \overline{g} + \overline{h} \) is not yet affected by this assignment
  - Hence, The assignment only affects cut clauses

Cut Clauses:
straddle a conceptual line separating assigned variables from unassigned ones

Diagram:

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```markdown
Consider any assignment to \( \{a, b, c, d\} \):
- If it is valid, \( a + c + d \) must be satisfied
- \( \overline{g} + \overline{h} \) is not yet affected by this assignment
- The assignment only affects cut clauses
```
Background: Terminology

Given **partial truth assignment**

Classify all clauses into:
- **Satisfied**: At least one literal assigned true
- **Violated**: All literals assigned, and not satisfied
- **Open**: 1 or more literals assigned, and no literals assigned true
  - Open clauses are activated but not satisfied
- **Activated**: Have at least one literal assigned some value
- **Unit**: Have all but one literal assigned, and are open

A **valid** partial truth assignment ⇔ no violated clauses
Open Clauses

- **Straightforward Breadth-First Search**
  - Maintain all valid partial truth assignments of a given depth; increase depth in steps

- **Valid partial truth assignments → sets of open clauses**
  - No literals assigned
    - Clause is **not activated**
  - All literals assigned
    - Clause must be **satisfied**
      - Because: assignment is valid ⇒ no clauses are violated

- **“Cut” clause = some, but not all literals assigned**
  - Must be either **satisfied** or **open**
  - This is determined by the partial assignment

- **Compressed Breadth-First Search**
  - Store sets of open clauses instead of promising assignments
Zero Suppressed Binary Decision Diagrams

- ZDD: A directed acyclic graph (DAG)
  - Unique source
  - Two sinks: the 0 and 1 nodes
- Each node has
  - Level index $i$
  - Two children at lower levels
    - T-Child and E-Child
- Characterized by reduction rules
  - If two nodes have the same level index, children
    - Merge these nodes
  - Zero-suppression rule
    - Eliminate nodes whose T-Child is 0
    - No node with a given index $i$ ⇒ assume a node whose T-child is 0
- ZDDs can store collections of sets
  - 0 is the empty collection $\emptyset$
  - 1 is the one-collection of the empty set $\{\emptyset\}$
  - At any node $f$, $f = f_T \cup \{i\} \otimes f_E$

$$f = f_E \cup \{i\} \otimes f_T$$
ZDD: Example

Collection of subsets:
- \{1, 3\}
- \{2, 3\}
- \{3\}
Boolean Constraint Propagation

Repeated application of the unit clause rule

Recall: unit clauses (with respect to some partial truth assignment)
- Have one remaining unassigned literal
- Not yet satisfied

In order for this assignment to lead to satisfiability
- This clause must be satisfied
- The remaining literal must be set true

Boolean Constraint Propagation
- Repeatedly apply unit clause rule to deduce new assignments
Compressed BFS: Overview

- Maintain collection of subsets of open clauses
  - Analogous to maintaining all “promising” partial solutions of increasing depth
  - Enough information for BFS on the solution tree
- This collection of sets is called the **front**
  - Stored and manipulated in compressed form (ZDD)
  - Assumes a clause ordering (global indices)
    - Clause indices correspond to node levels in the ZDD
- Algorithm: expand one variable at a time
  - After all variables two cases possible
    - The front is $\emptyset$ $\Rightarrow$ Unsatisfiable
    - The front is $\{\emptyset\}$ $\Rightarrow$ Satisfiable
Compressed BFS: An Example

Process variables in the order \{a, b, c, d\}

Initially the front is set to 1

- The collection should contain one “branch”
- This branch should contain no open clauses \(\Rightarrow \{\emptyset\}\)
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

1 2 3 4 5 6

- **Processing variable a**
  - Activate clauses \{3, 4, 5, 6\}
  - Cut clauses: \{3, 4, 5, 6\}
  - \(a = 0\)
    - Clauses \{3, 4\} become open
  - \(a = 1\)
    - Clauses \{5, 6\} become open

- ZDD contains \{ \{3, 4\}, \{5, 6\} \}
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

1. **Processing variable b**
   - Activate clauses \{1, 2\}
     - Cut clauses: \{1, 2, 3, 4, 5, 6\}
   - \(b = 0\)
     - No clauses can become violated
       - \(b\) is not the end literal for any clause
     - Clause 2 is satisfied
       - Don’t need to add it
     - Clause 1 first becomes activated
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

1 2 3 4 5 6

- **Processing variable** \(b\)
  - Activate clauses \{1, 2\}
    - Cut clauses: \{1, 2, 3, 4, 5, 6\}
  - \(b = 1\)
    - No clauses can become violated
      - \(b\) is not the end literal for any clause
    - Existing clauses 4, 6 are satisfied
    - Clause 1 is satisfied
      - Don’t need to add it
    - Clause 2 first becomes activated
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

- **Processing variable** \(b\)
  - Activate clauses \(\{1, 2\}\)
    - Cut clauses: \(\{1, 2, 3, 4, 5, 6\}\)
  - \(b = 1\)
    - No clauses can become violated
      - \(b\) is not the end literal for any clause
    - Existing clauses 4, 6 are satisfied
    - Clause 1 is satisfied
      - Don’t need to add it
    - Clause 2 first becomes activated
Compressed BFS: An Example

(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)
Compressed BFS: An Example

(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)

Processing variable c

- Finish clause 4
  - Cut clauses: {1, 2, 3, 5, 6}
- c = 0
  - No clauses become violated
    - c ends 4, but c=0 satisfies it
  - Clauses 4,5 become satisfied
  - No clauses become activated
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

- Processing variable \(c\)
  - Finish clause 4
    - Cut clauses: \(\{1, 2, 3, 5, 6\}\)
  - \(c = 1\)
    - Clause 4 may be violated
      - If \(c\) appears in the ZDD, then it is still open
    - Clauses 1, 2, 3 are satisfied
    - No clauses become activated
Compressed BFS: An Example

\[(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)\]

1 2 3 4 5 6

**Processing variable d**

- Finish clauses \{1, 2, 3, 5, 6\}
  - Cut clauses: \{1, 2, 3, 5, 6\}
- \(d = 0, d=1\)
  - All clauses are already satisfied
  - Assignment doesn’t affect this
  - Instance is satisfiable

1 2 3 4 5 6

1 1
Compressed BFS: Pseudocode

CompressedBFS(Vars, Clauses)
front ← 1
for i = 1 to |Vars| do
    front' ← front
    //Modify front to reflect xi = 1
    Form sets Uxi,1, Sxi,1, Axi,1
    front ← front ⊗ 2Cut - Uxi,1
    front ← ExistAbstract(front, Sxi,1)
    front ← front ⊗ Axi,1
    //Modify front' to reflect xi = 0
    Form sets Uxi,0, Sxi,0, Axi,0
    front' ← front' ⊗ 2Cut - Uxi,0
    front' ← ExistAbstract(front', Sxi,0)
    front' ← front' ⊗ Axi,0
    //Combine the two branches via Union
    //and remove Subsumptions
    front ← front ⊃ front'
if front = 0 then
    return Unsatisfiable
if front = 1 then
    return Satisfiable
Boolean Constraint Propagation with CBFS

φ = (a + d)(a + c)(a + c)(a + b)(a + b)

1 2 3 4 5

Consider having processed variable $a$ only
Recall: The front consists of sets of open clauses
\textbf{Conflicting set of clauses}
- A set of open clauses by which it is possible to derive a contradiction by the \textit{unit clause rule}
- Ex. if clauses \{2, 3\} are both open $\Rightarrow c$ and $\overline{c}$ are both implied
- After variable $a$ $\Rightarrow$ \{2, 3\} is a conflicting set of clauses
\textbf{Conflicting sets cannot appear in the same set of open clauses}
- CBFS will eventually determine this
- Repeated application of the unit clause rule may find this more efficiently

In this example: conflicting sets of clauses
- Clauses \{2, 3\} cannot appear together
- Clauses \{4, 5\} cannot appear together
Boolean Constraint Propagation with CBFS

**Basic idea:** recursive search to find all sets of conflicting clauses

- For each unit clause U
  - Find all clauses violated when U is satisfied
  - Find all clauses violated when U is violated (includes U)
  - Form Cartesian Product of these sets
- Can form the ZDD of all conflicting sets of clauses

**Conflicting sets cannot appear in the same set of open clauses**

- If a set in the front contains a conflicting set
  - Can prune with ZDD Subsumed Difference operator
Boolean Constraint Propagation with CBFS

```plaintext
GetConflictZDD(Formula F', Integer Var)
    foreach clause C ∈ F'
        if C has no literals (after the cut) //Then C is a violated clause
            ViolCls ← ViolCls ∪ C
            //Find the set of variables implied by some unit clause
            IVars ← ImpliedVars( Units(F') )
        //Find the lowest index implied variable such that v > Var
        v_low ← UpperBound(IVars, Var)
        if no such v_low exists
            return ViolCls
        ConflZdd ← ViolCls
        ConflZdd ← GetConflictZDD((Assign(F', v=1), v))
        Z0 ← GetConflictZDD((Assign(F', v=0), v))
        Z ← Z0 ⊙ Z1
        ConflZdd ← ConflZdd ∪ Z
    return ConflZdd
```

Extending BCP/CBFS

Bounded Depth BCP
- Want conflicting sets to subsume many sets in the front
  - Should be as small as possible
  - As depth of search increases ⇒ number of clauses in any conflicting sets found increases
- Search for Conflicting ZDD may be time consuming

BCP pruning at step $k$ is similar to step $k+1$
- To help combat this, apply BCP every $2d$ steps
  - $d$ ⇒ depth of BCP search
Empirical Results

<table>
<thead>
<tr>
<th>FPGA</th>
<th>S/U</th>
<th>Cassatt</th>
<th>BCP 2</th>
<th>BCP 3</th>
<th>BCP 4</th>
<th>zChaff</th>
</tr>
</thead>
<tbody>
<tr>
<td>10_11</td>
<td>UNS</td>
<td>0.04</td>
<td>0.12</td>
<td>0.45</td>
<td>1.18</td>
<td>&gt;250</td>
</tr>
<tr>
<td>10_12</td>
<td>UNS</td>
<td>0.05</td>
<td>0.14</td>
<td>0.35</td>
<td>0.96</td>
<td>&gt;250</td>
</tr>
<tr>
<td>10_13</td>
<td>UNS</td>
<td>0.03</td>
<td>0.15</td>
<td>0.59</td>
<td>2.01</td>
<td>&gt;250</td>
</tr>
<tr>
<td>10_15</td>
<td>UNS</td>
<td>0.09</td>
<td>0.34</td>
<td>1.31</td>
<td>6.39</td>
<td>&gt;250</td>
</tr>
<tr>
<td>10_20</td>
<td>UNS</td>
<td>0.24</td>
<td>0.7</td>
<td>2.82</td>
<td>15.1</td>
<td>&gt;250</td>
</tr>
<tr>
<td>11_12</td>
<td>UNS</td>
<td>0.06</td>
<td>0.16</td>
<td>0.59</td>
<td>1.1</td>
<td>&gt;250</td>
</tr>
<tr>
<td>11_13</td>
<td>UNS</td>
<td>0.04</td>
<td>0.15</td>
<td>0.74</td>
<td>2.97</td>
<td>&gt;250</td>
</tr>
<tr>
<td>11_14</td>
<td>UNS</td>
<td>0.04</td>
<td>0.21</td>
<td>0.98</td>
<td>4.09</td>
<td>&gt;250</td>
</tr>
<tr>
<td>11_15</td>
<td>UNS</td>
<td>0.06</td>
<td>0.24</td>
<td>1.06</td>
<td>5.43</td>
<td>&gt;250</td>
</tr>
<tr>
<td>11_20</td>
<td>UNS</td>
<td>0.1</td>
<td>0.51</td>
<td>3.3</td>
<td>20.68</td>
<td>&gt;250</td>
</tr>
<tr>
<td>10_8</td>
<td>SAT</td>
<td>0.03</td>
<td>0.07</td>
<td>0.28</td>
<td>2.6</td>
<td>2.13</td>
</tr>
<tr>
<td>10_9</td>
<td>SAT</td>
<td>0.06</td>
<td>0.13</td>
<td>0.36</td>
<td>1.24</td>
<td>2.01</td>
</tr>
<tr>
<td>12_8</td>
<td>SAT</td>
<td>0.06</td>
<td>0.12</td>
<td>0.37</td>
<td>2.03</td>
<td>&gt;250</td>
</tr>
<tr>
<td>12_9</td>
<td>SAT</td>
<td>0.12</td>
<td>0.19</td>
<td>0.53</td>
<td>2.36</td>
<td>104.7</td>
</tr>
<tr>
<td>12_10</td>
<td>SAT</td>
<td>0.15</td>
<td>0.26</td>
<td>0.87</td>
<td>3.97</td>
<td>&gt;250</td>
</tr>
<tr>
<td>12_11</td>
<td>SAT</td>
<td>0.07</td>
<td>0.2</td>
<td>0.83</td>
<td>4.97</td>
<td>&gt;250</td>
</tr>
<tr>
<td>12_12</td>
<td>SAT</td>
<td>0.52</td>
<td>0.67</td>
<td>1.55</td>
<td>5.68</td>
<td>132.91</td>
</tr>
<tr>
<td>13_9</td>
<td>SAT</td>
<td>0.35</td>
<td>0.44</td>
<td>0.8</td>
<td>2.79</td>
<td>191.63</td>
</tr>
<tr>
<td>13_10</td>
<td>SAT</td>
<td>0.71</td>
<td>0.84</td>
<td>1.43</td>
<td>5.47</td>
<td>66.3</td>
</tr>
<tr>
<td>13_11</td>
<td>SAT</td>
<td>1.61</td>
<td>1.8</td>
<td>2.4</td>
<td>4.47</td>
<td>&gt;250</td>
</tr>
<tr>
<td>13_12</td>
<td>SAT</td>
<td>2.66</td>
<td>2.88</td>
<td>3.62</td>
<td>7.99</td>
<td>&gt;250</td>
</tr>
</tbody>
</table>
## Empirical Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#</th>
<th>S/U</th>
<th>Cassatt</th>
<th>+ BCP Depth 2</th>
<th>+ BCP Depth 3</th>
<th>+ BCP Depth 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Family</td>
<td></td>
<td></td>
<td>%Sol</td>
<td>Avg</td>
<td>%Sol Avg</td>
<td>%Sol Avg</td>
</tr>
<tr>
<td>aim-100*</td>
<td>24</td>
<td>-</td>
<td>70.83</td>
<td>84.04</td>
<td>75 79.5</td>
<td>75 77.74</td>
</tr>
<tr>
<td>aim-50*</td>
<td>24</td>
<td>-</td>
<td>100</td>
<td>0.18</td>
<td>100 0.17</td>
<td>100 0.355</td>
</tr>
<tr>
<td>dubois*</td>
<td>13</td>
<td>UNS</td>
<td>100</td>
<td>0.01</td>
<td>100 0.02</td>
<td>100 0.02</td>
</tr>
<tr>
<td>pret*</td>
<td>8</td>
<td>UNS</td>
<td>100</td>
<td>0.016</td>
<td>100 0.018</td>
<td>100 0.02</td>
</tr>
<tr>
<td>par16*</td>
<td>5</td>
<td>SAT</td>
<td>80</td>
<td>85.52</td>
<td>60 129.19</td>
<td>60 131.338</td>
</tr>
<tr>
<td>par16-c*</td>
<td>5</td>
<td>SAT</td>
<td>60</td>
<td>152.42</td>
<td>60 154.67</td>
<td>60 155.26</td>
</tr>
<tr>
<td>par8*</td>
<td>5</td>
<td>SAT</td>
<td>100</td>
<td>0.71</td>
<td>100 0.488</td>
<td>100 0.89</td>
</tr>
<tr>
<td>par8-c*</td>
<td>5</td>
<td>SAT</td>
<td>100</td>
<td>0.026</td>
<td>100 0.058</td>
<td>100 0.128</td>
</tr>
</tbody>
</table>
Conclusions and Ongoing Work

CBFS runtimes on several families show great improvements over DLL-based solvers
- Potential for a more general purpose combined solver

We introduced a BCP-based pruning into CBFS
- On classes CBFS solves quickly $\Rightarrow$ no further improvement
- On less structured instances $\Rightarrow$ CBFS's runtime is improved by the addition of a restricted BCP

We hope to further improve performance of CBFS/BCP
- BCP reductions need not be complete:
  - Heuristic and randomized approaches can applied to find some, but not all conflicting sets
  - Can tune the application of BCP to improve performance