On Proof Systems Behind Efficient SAT Solvers

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

- Best complete SAT solvers are based on DLL
  - Runtime (on unSAT instances) is lower-bounded by the length of resolution proofs
  - Exponential lower bounds for pigeonholes
- Previous work: we introduced the Compressed Breadth-First Search algorithm (CBFS/Cassatt)
  - Empirical measurements: our implementation of Cassatt spends $\Theta(n^4)$ time on $\text{PHP}_n^{n+1}$
- This work: we show analytically that CBFS refutes pigeonhole instances $\text{PHP}_n^{n+1}$ in poly time
  - Hope to find a proof system behind Cassatt
Empirical Performance
Related Work

- We are pursuing novel algorithms for SAT facilitated by data structures with compression
  - Zero-suppressed Binary Decision Diagrams (ZDDs)
- Existing algorithms can be implemented w ZDDs
  - The DP procedure: Simon and Chatalic, [ICTAI 2000]
  - DLL: Aloul, Mneimneh and Sakallah, [DATE 2002]
- We use the union-with-subsumption operation
- Details of the Cassatt algorithm are in
  - Motter and Markov, [ALENEX 2002]
Outline

- Background
- Compressed BFS
  - Overview
  - Example
  - Algorithm
- Pigeonhole Instances
- Outline of Proof
  - Some bounds
- Conclusions and Ongoing Work
Background

\[(a+c+d)(-g + -h)(-b + e + f)(d + -e)\]

\[
\begin{array}{c|c}
(a + c+d) & (d+\overline{e}) \\
(b + e+f) & (\overline{g}+\overline{h}) \\
\hline
\end{array}
\]

\[a \ b \ c \ d \ e \ f \ g \ h\]
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+ literal 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
Binary Decision Diagrams

- BDD: A directed acyclic graph (DAG)
  - Unique source
  - Two sinks: the 0 and 1 nodes
- Each node has
  - Unique label
  - Level index
  - Two children at lower levels
    - T-Child and E-Child
- BDDs can represent Boolean functions
  - Evaluation is performed by a single DAG traversal
- BDDs are characterized by reduction rules
  - If two nodes have the same level index and children
    - Merge these nodes
Zero-Suppressed BDDs (ZDDs)

- Zero-suppression rule
  - Eliminate nodes whose T-Child is $0$
  - No node with a given index $\Rightarrow$ assume a node whose T-child is $0$

- ZDDs can store a collection of subsets
  - Encoded by the collection’s characteristic function
  - $0$ is the empty collection $\emptyset$
  - $1$ is the one-collection of the empty set $\{\emptyset\}$

- Zero-suppression rule enables compact representations of sparse or regular collections
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
  - When all variables are processed two cases possible
    - The front is $\emptyset$ $\Rightarrow$ Unsatisfiable
    - The front is $\{\emptyset\} \Rightarrow$ Satisfiable
Compressed BFS

\[ \text{Front} \leftarrow 1 \quad \# \text{ assign } \{\emptyset\} \text{ to front} \]

\textbf{foreach } v \in \text{Vars} \n
\enspace \text{Front2} \leftarrow \text{Front} \n
\enspace \text{Update(Front, v } \leftarrow 1) \n
\enspace \text{Update(Front2, v } \leftarrow 0) \n
\enspace \text{Front} \leftarrow \text{Front} \cup \text{Front2} \n
\textbf{if } \text{Front} == 0 \textbf{ return } \text{Unsatisfiable} \n
\textbf{if } \text{Front} == 1 \textbf{ return } \text{Satisfiable}
Compressed BFS: An Example

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

\[1\ 2\ 3\ 4\ 5\ 6\]

- 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)

- 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

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

- 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

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

- **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)

1 2 3 4 5 6

- 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. **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
Compressed BFS: Pseudocode

CompressedBfs(Vars, Clauses)

front ← 1

for i = 1 to |Vars| do

    front’ ← front

    //Modify front to reflect \( x_i = 1 \)
    Form sets \( U_{x_i,1}, S_{x_i,1}, A_{x_i,1} \)
    front ← front ∩ 2^{Cut - U_{x_i,1}}
    front ← ExistAbstract(front, \( S_{x_i,1} \))
    front ← front ⊗ \( A_{x_i,1} \)

    //Modify front’ to reflect \( x_i = 0 \)
    Form sets \( U_{x_i,0}, S_{x_i,0}, A_{x_i,0} \)
    front’ ← front’ ∩ 2^{Cut - U_{x_i,0}}
    front’ ← ExistAbstract(front’, \( S_{x_i,0} \))
    front’ ← front’ ⊗ \( A_{x_i,0} \)

    //Combine the two branches via Union with Subsumption
    front ← front ∪_{s} front’

if front = 0 then
    return Unsatisfiable

if front = 1 then
    return Satisfiable
The Instances $\text{PHP}_n^{n+1}$

- Negation of the pigeonhole principle
  - “If $n+1$ pigeons are placed in $n$ holes then some hole must contain more than one pigeon”

- Encoded as a CNF
  - $n(n+1)$ Boolean variables
    - $v_{ij}$ represents that pigeon $i$ is in hole $j$
  - $n+1$ “Pigeon” clauses: $(v_{i1} + v_{i2} + \ldots + v_{in})$
    - Pigeon $i$ must be in some hole
  - $n(n+1)$ “Pairwise Exclusion” clauses (per hole): $(\overline{v_{i1j}} + \overline{v_{i2j}})$
    - No two pigeons can be in the same hole

- Unsatisfiable CNF instance

- Use the “hole-major” variable ordering
  - $\{x_1, x_2, \ldots x_{n(n+1)}\} \leftrightarrow \{v_{11}, v_{21}, \ldots, v_{(n+1)1}, v_{12}, v_{22}, \ldots\}$
The Instances $\overline{\text{PHP}}_{n}^{n+1}$
Outline of Proof

- Bound the size of the ZDD-based representation throughout execution
  - With most ZDD operations:
    - \( h = \text{zdd\_op}(\text{ZDD } f, \text{ZDD } g) \)
    - \( h \) is built during a traversal of ZDDs \( f, g \)
    - The execution time is bounded by \( \text{poly}(|f|, |g|) \)
- Do not consider all effects of reduction rules
  - These obscure underlying structure of the ZDD
  - Reduction rules can only eliminate nodes
    - This will still allow an upper bound on ZDD size
Outline of Proof

- Main idea: Bound the size of the partially reduced ZDD
  - First compute a simple bound between “holes”
  - Prove that the size does not grow too greatly inside “holes”
- Show the ZDD at given step has a specific structure
Bounds Between $H_k$

- **Lemma.** Let $k \in \{1, 2, \ldots, n\}$. After assigning values to variables $x_1, x_2, \ldots, x_{k(n+1)}$, we may satisfy at most $k$ of the $n+1$ pigeon clauses.
  - Valid partial truth assignment to the first $k(n+1)$ variables
    - Must set only one variable in $H_i$ true, for each $i < k$.

- For CBFS
  - Remove subsumed sets
    - front contains all sets of $(n+1-k)$ pigeon clauses
  - How many nodes does this take?
ZDD of all k-Element Subsets

- To reach 1 ⇒ function must select the T-Child on exactly k indices
  - Less than k ⇒ Traverse to 0
  - More than k ⇒ Zero-Suppression Rule
- Contains \((n+1-k)k\) nodes
- ZDDs are a canonical representation
  - When this is encountered in CBFS, we are assured of this structure
  ⇒ CBFS uses \((n+1-k)(k+1)\) nodes after variable \(x_k(n+1)\)
The *front* within $H_k$

- After variable $x_{k(n+1)+i}$ the ZDD contains $(i+1)$ “branches”
- Main branch corresponds to all $x_{k(n+1)} + 1, \ldots, x_{k(n+1)} + i$ false
- $i+1$ other branches correspond to one of $x_{k(n+1)} + 1, \ldots, x_{k(n+1)} + i$ true
- Squares correspond to ZDDs of all subsets of a given size
- Can show this structure is correct by induction
- Bound comes from counting nodes in this structure
Analytical vs. Empirical
Conclusions and Ongoing Work

- Understanding why CBFS can quickly solve pigeonhole instances depends on recognizing structural invariants within the ZDD
- We hope to understand exactly what proof system is behind CBFS
- We hope to improve the performance of CBFS
  - DLL solvers have been augmented with many ideas (BCP, clause subsumption, etc)
  - These ideas may have an analogue with CBFS giving a performance increase
Thank you!!!
The Utility of Subsumption

- Cassatt empirically solves pigeonhole instances in $O(n^4)$ without removing subsumptions.
- Without subsumption removal:
  - Instead of ZDD’s for all $k$-element subsets
  - ZDDs for all $(k \text{ or greater})$-element subsets
    - Still $O(n^2)$
- To find a bound, need to factor in the additional nodes due to keeping all $(k \text{ or greater})$ element subsets.
Opportunistic Subsumption Finding

- ‘Subsume’-able sets can occur as the result of Existential Abstraction or Union
  - In pigeonhole instances, this only occurs when we satisfy 1 pigeon clause
    ⇒ Smaller sets will have only one less element than larger sets they subsume
- Can detect some subsumptions by recursively searching for nodes of the form
  - Captures subsumptions which occur in CBFS’s solution of pigeonhole instances
Thanks again!!!
Processing a Single Variable

- **Given:**
  - Assignment of 0 or 1 to a single variable $x$
- It violates some clauses: $V_{x \leftarrow \{0,1\}}$
  - $V_{x \leftarrow \{0,1\}}$: Clauses which are unit, and this assignment makes the remaining literal false
    - If any clause in $V_{x \leftarrow \{0,1\}}$ is open then the partial truth assignment for that set of open clauses cannot yield satisfiability
  - Remove all such sets of open clauses
    - Can use ZDD Intersection
Processing a Single Variable

- **Given:**
  - Assignment of 0 or 1 to a single variable $x$
- **It satisfies some clauses:** $S_{x \leftarrow \{0,1\}}$
  - $S_{x \leftarrow \{0,1\}}$: Clauses in which $x$ appears, and the assignment makes the corresponding literal true
    - If any clause in $S_{x \leftarrow \{0,1\}}$ is open, it should no longer be
  - Remove all such clauses $S_{x \leftarrow \{0,1\}}$ from any set
    $\Rightarrow$ ZDD $\exists$ Abstraction
Processing a Single Variable

- Given:
  - Assignment of 0 or 1 to a single variable \( x \)
  - It activates some clauses, \( A_x \leftarrow \{0, 1\} \)
    - \( A_x \leftarrow \{0, 1\} \): Clauses in which \( x \) is the first literal encountered, and \( x \) does not satisfy
      - These clauses are open in any branch of the search now
  - Add these clauses \( A_x \leftarrow \{0, 1\} \) to each set
    \( \Rightarrow \) ZDD Cartesian Product