Overcoming Resolution- Based Lower Bounds for SAT Solvers

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

Michigan Engineering

IWLS 2002

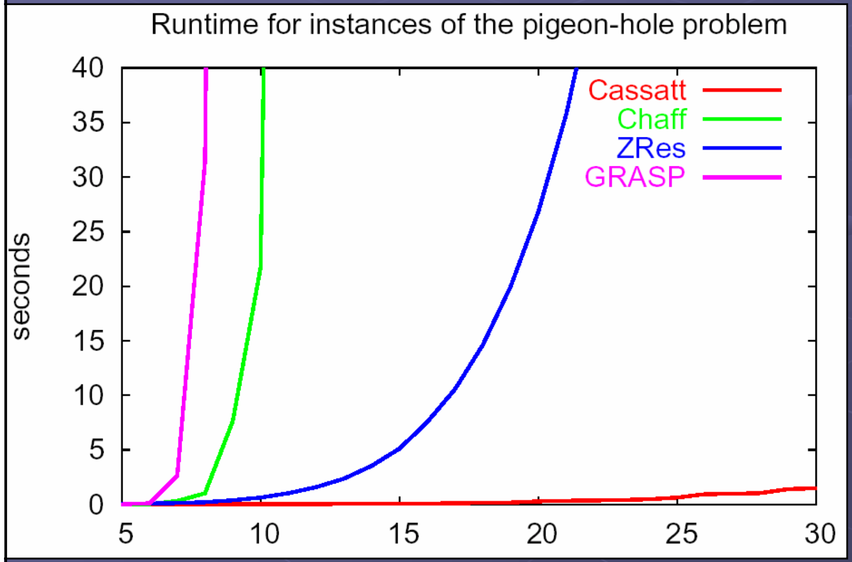
Motivation

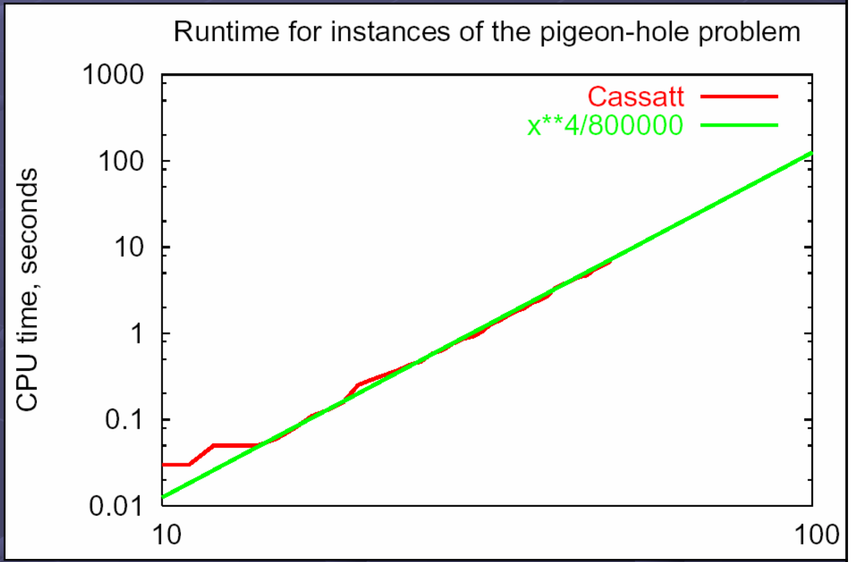
- Boolean Satisfiability (SAT) has widespread applications
  - EDA: Equivalence checking, BMC, Routing, AI: Planning, etc.
  - New applications are constantly emerging
- Fast SAT solvers abound (GRASP, Chaff, BerkMin)
  - Highly tuned implementations improved over years
- Many small instances are still 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





MichiganEngineering

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 φ in Conjunctive Normal Form (CNF)
    - V: set of variables {a, b, …, n}
    - C: set of clauses
    - Each clause is a set of literals over V
  - Question: Is there an assignment to {a, b, …, 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

MichiganEngineering

Background: Partial Assignments

$$\varphi = (a + c + d)(\bar{g} + \bar{h})(\bar{b} + e + f)(d + \bar{e})$$

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

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

$(\bar{d} + e)$

$(\bar{b} + e + f)$

$(a + c + d)$ $(\bar{g} + \bar{h})$

a b c d | e f g h

Background: Terminology

- Given <u>partial truth assignment</u>
- 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 <u>valid</u> partial truth assignment ⇔ <u>no violated clauses</u>

disjoint

$(\bar{d} + e)$

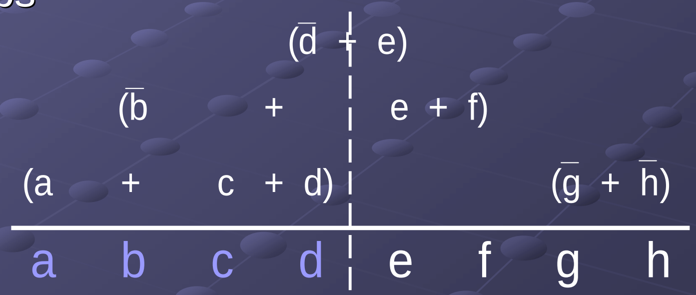
$(\bar{b} + e + f)$

$(a + b + c + d)$ $(\bar{g} + \bar{h})$

a b c d | e f g h

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

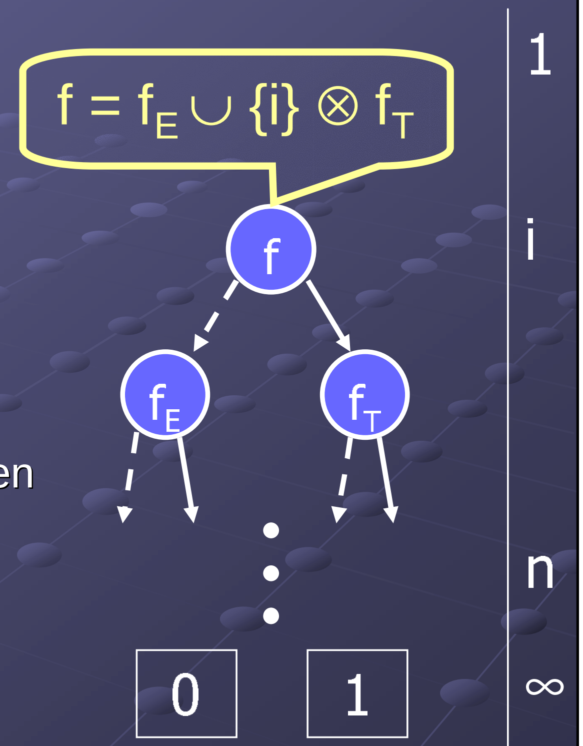


- "**Cut**" clause = **some**, but not all **literals assigned**
  - Must be either **satisfied** or **open**
  - This is determined by the partial assignment
- Compressed Breadth-First Search
  - Store sets of open clauses instead of promising assignments

MichiganEngineering

Zero Suppressed Binary Decision Diagrams

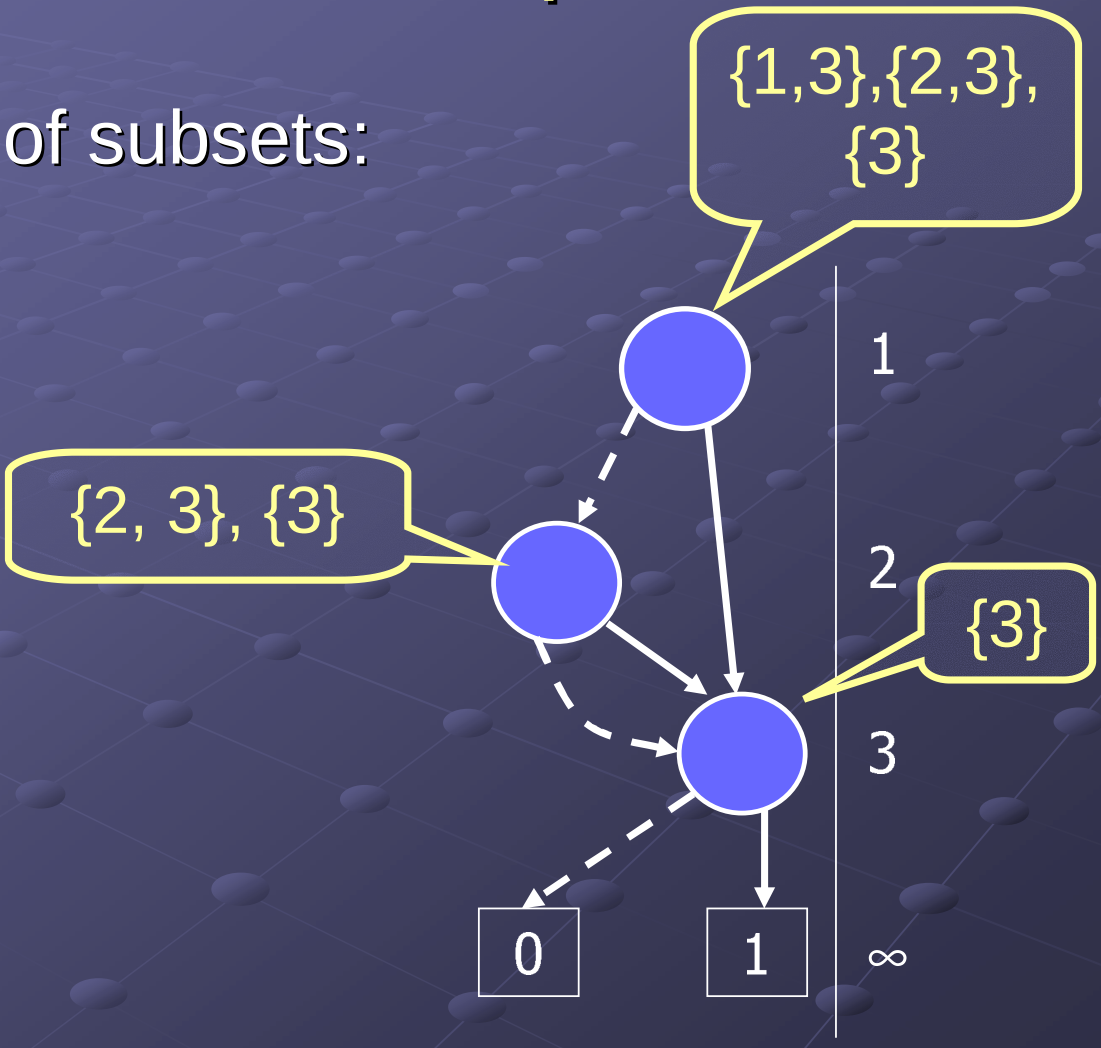
- ZDD: A directed acyclic graph (DAG)
  - Unique source
  - Two sinks: the **0** and **1** nodes
- Each node has
  - Level index i
  - Two children at lower levels
    - T-Child and E-Child
- Characterized by reduction rules
  - If two nodes have the same level index, children
    - Merge these nodes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



MichiganEngineering

ZDD: Example

- Collection of subsets:
  - {1, 3}
  - {2, 3}
  - {3}



MichiganEngineering

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

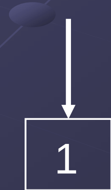
Compressed BFS: Overview

- 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 $\varnothing$ $\Rightarrow$ Unsatisfiable
    - The front is $\{\varnothing\}$ $\Rightarrow$ Satisfiable

Compressed BFS: An Example

$$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$$

- 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 $\Rightarrow \{\varnothing\}$

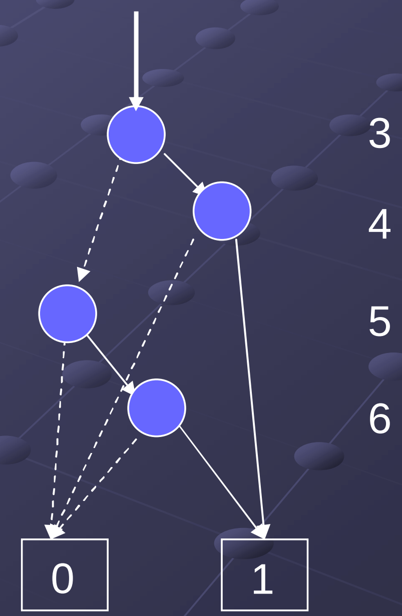


MichiganEngineering

Compressed BFS: An Example

$$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$$

- 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} }

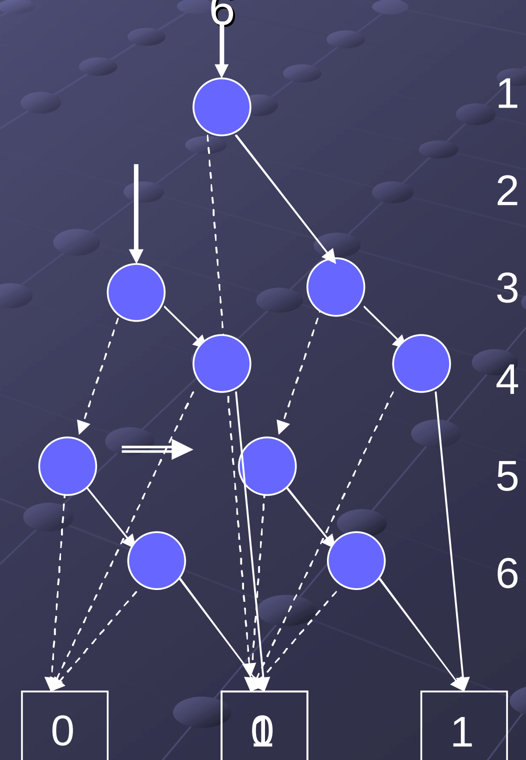


Compressed BFS: An Example

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

1 2 3 4 5 6

- 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

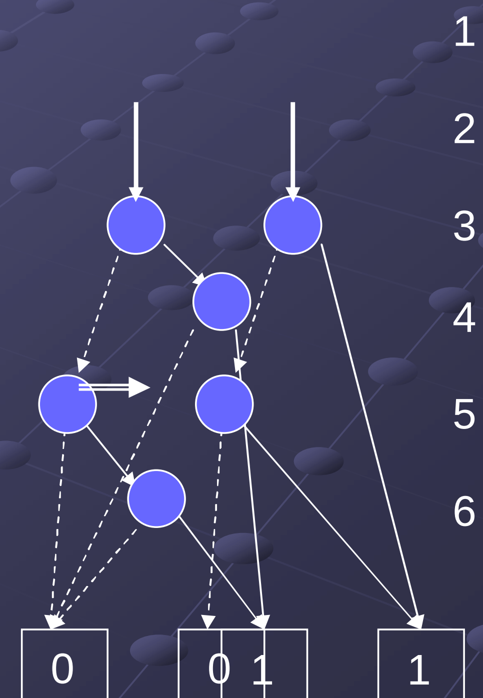


Compressed BFS: An Example

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

1 2 3 4 5 6

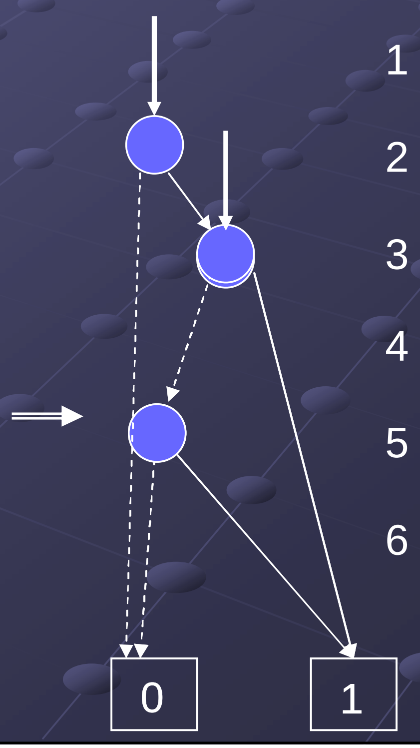
- Processing variable b
  - Activate clauses {1, 2}
    - Cut clauses: {1, 2, 3, 4, 5, 6}
  - b = 1
    - No clauses can become violated
      - b is not the end literal for any clause
    - Existing clauses 4, 6 are satisfied
    - Clause 1 is satisfied
      - Don’t need to add it
    - Clause 2 first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



Compressed BFS: An Example

$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$

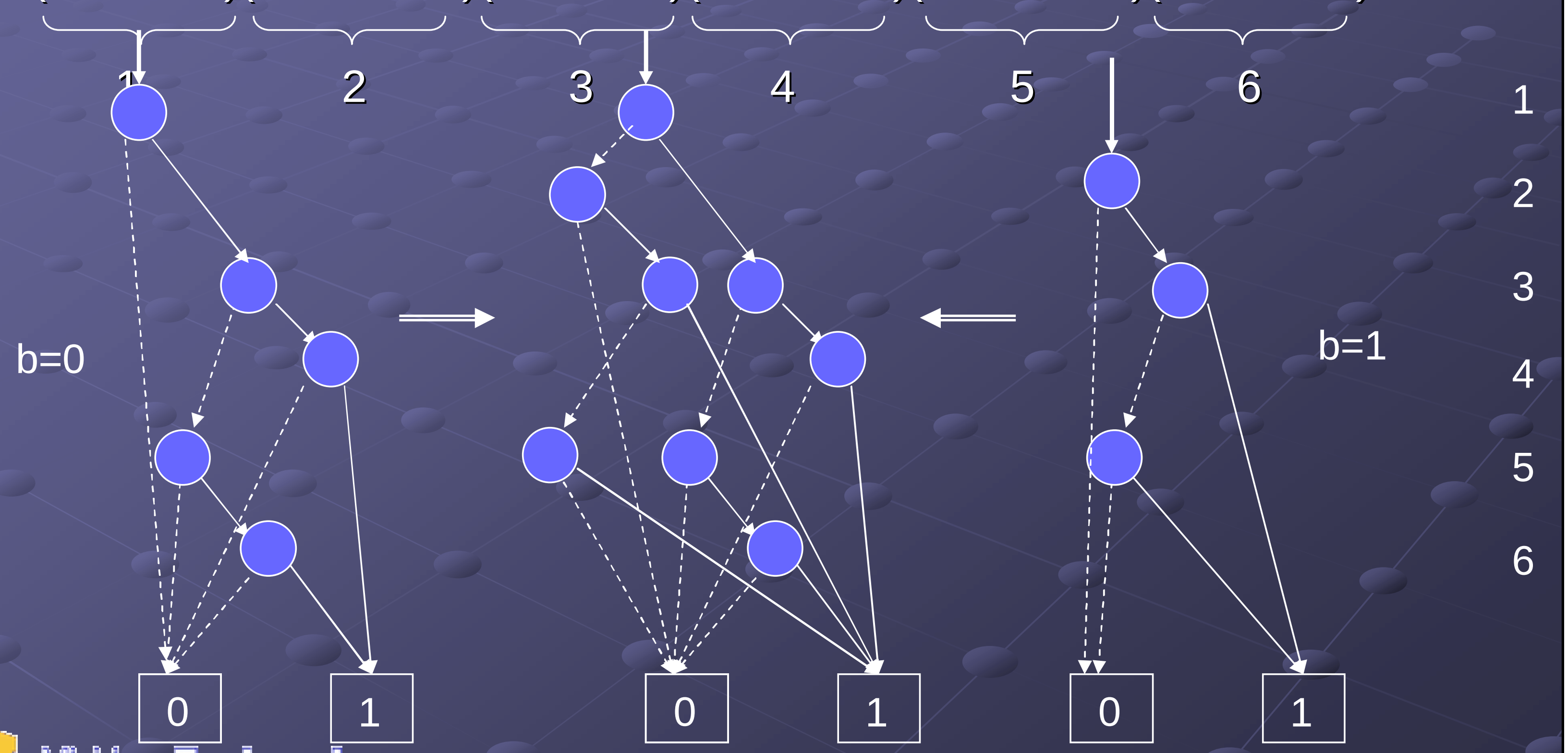
- Processing variable b
  - Activate clauses {1, 2}
    - Cut clauses: {1, 2, 3, 4, 5, 6}
  - b = 1
    - No clauses can become violated
      - b is not the end literal for any clause
    - Existing clauses 4, 6 are satisfied
    - Clause 1 is satisfied
      - Don't need to add it
    - Clause 2 first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



MichiganEngineering

Compressed BFS: An Example

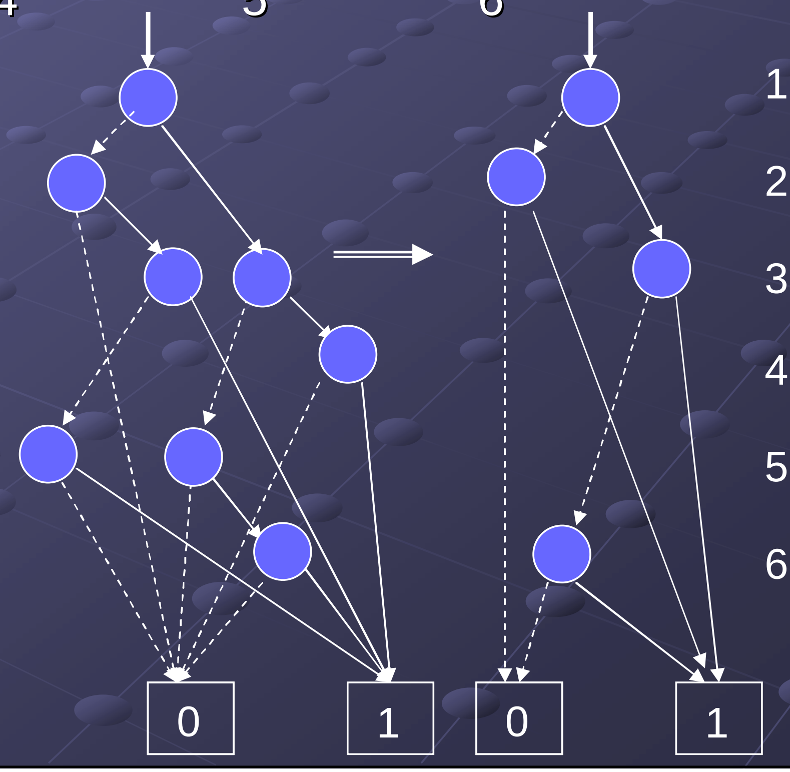
(b + c + d)(-b + c + -d)(a + c + d)(a + b + -c)(-a + -c + d)(-a + b + d)



Compressed BFS: An Example

$$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$$

- 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

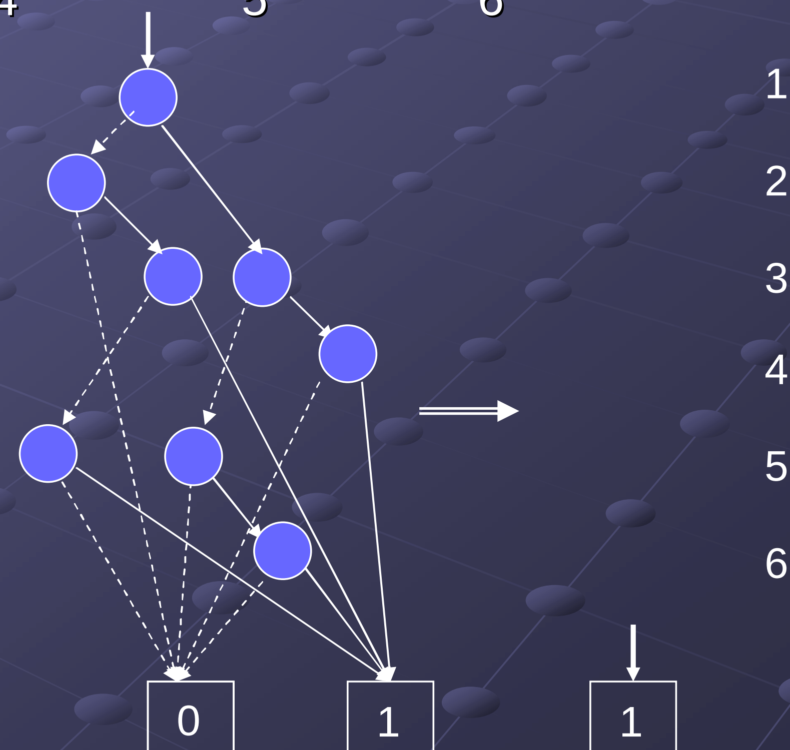


MichiganEngineering

Compressed BFS: An Example

$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$

- 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



MichiganEngineering

Compressed BFS: An Example

$\underbrace{(b + c + d)}_{1}\underbrace{(-b + c + -d)}_{2}\underbrace{(a + c + d)}_{3}\underbrace{(a + b + -c)}_{4}\underbrace{(-a + -c + d)}_{5}\underbrace{(-a + b + d)}_{6}$

- 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

$\Rightarrow$

$\boxed{\mathbf{1}}$ $\boxed{1}$

Compressed BFS: Pseudocode

CompressedBFS(Vars, Clauses)
  front $\leftarrow$ **1**
  for i = 1 to |Vars| **do**
    front' $\leftarrow$ front
    //Modify front to reflect $x_i = 1$
    Form sets $U_{xi,1}$, $S_{xi,1}$, $A_{xi,1}$
    front $\leftarrow$ front $\cap$ $\mathbf{2}^{Cut - Uxi,1}$
    front $\leftarrow$ ExistAbstract(front, $S_{xi,1}$)
    front $\leftarrow$ front $\otimes$ $A_{xi,1}$
    //Modify front' to reflect $x_i = 0$
    Form sets $U_{xi,0}$, $S_{xi,0}$, $A_{xi,0}$
    front' $\leftarrow$ front' $\cap$ $\mathbf{2}^{Cut - Uxi,0}$
    front' $\leftarrow$ ExistAbstract(front', $S_{xi,0}$)
    front' $\leftarrow$ front' $\otimes$ $A_{xi,0}$
    //Combine the two branches via Union
    //and remove Subsumptions
    front $\leftarrow$ front $\cup_s$ front'
  if front = **0 then**
    return Unsatisfiable
  if front = **1 then**
    return Satisfiable

MichiganEngineering

Boolean Constraint Propagation with CBFS

$$\varphi = \underbrace{(\bar{a} + \bar{d})}_{1}\underbrace{(\bar{a} + \bar{c})}_{2}\underbrace{(\bar{a} + c)}_{3}\underbrace{(a + \bar{b})}_{4}\underbrace{(a + b)}_{5}$$

- Consider having processed variable a only
- Recall: The front consists of sets of open clauses
- Conflicting set of clauses
  - A set of open clauses by which it is possible to derive a contradiction by the unit clause rule
  - Ex. if clauses {2, 3} are both open $\Rightarrow$ c and $\bar{c}$ are both implied
  - After variable a $\Rightarrow$ {2, 3} is a conflicting set of clauses
- 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

MichiganEngineering

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

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

    //Iterate over all implied variables ≥v
    forall $v \in$IVars such that $v \geq v_{low}$
        Z1← **GetConflictZDD**(**Assign**(*F'*, *v*=1), v)
        *Z*0 ← **GetConflictZDD**(**Assign**(*F'*, *v*=0), v)
        $Z \leftarrow Z0 \otimes Z1$
        ConflZDD ← ConflZDD ∪ *Z*
    return ConflZDD

Extending BCP/CBFS

- Bounded Depth BCP
  - Want conflicting sets to subsume many sets in the front
    - $\Rightarrow$ Should be as small as possible
    - As depth of search increases $\Rightarrow$ 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 $\Rightarrow$ depth of BCP search

Empirical Results

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

Empirical Results

Benchmark	#	S/U	Cassatt		+ BCP Depth 2		+ BCP Depth 3		+ BCP Depth 4	
Family			%Sol	Avg	%Sol	Avg	%Sol	Avg	%Sol	Avg
aim-100*	24	-	70.83	84.04	75	79.5	75	**77.74**	75	79.39
aim-50*	24	-	100	0.18	100	**0.17**	100	0.355	100	1.69
dubois*	13	UNS	100	**0.01**	100	0.02	100	0.02	100	**0.01**
pret*	8	UNS	100	**0.016**	100	0.018	100	0.02	100	0.02
par16*	5	SAT	80	**85.52**	60	129.19	60	131.338	60	136.75
par16-c*	5	SAT	60	**152.42**	60	154.67	60	155.26	60	159.00
par8*	5	SAT	100	0.71	100	**0.488**	100	0.89	100	2.04
par8-c*	5	SAT	100	**0.026**	100	0.058	100	0.128	100	0.45

MichiganEngineering

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

MichiganEngineering

Thank You

MichiganEngineering