Solving Difficult SAT Instances
In The Presence of Symmetry

Igor Markov
The University of Michigan (Ann Arbor), EECS
CNF

Graph

Symmetry-breaking

Pre-processed CNF instance

Graph symmetries

Call generic SAT solver

CNF symmetries

nauty or saucy

Shatter

Shatter

Shatter

Shatter
Outline

- Reasons to worry about symmetries
- Representing symmetries efficiently
- Finding symmetries efficiently
- How symmetry-breaking works
- Asymptotic and empirical results
- Conclusions
- On-going work
A Simple Routing Instance

Sample clauses:
(y'+z')(y+z)
(z'+t'+x') ...

Try to route two sink-source pairs at the same time

Reducing such instances to SAT
- Allows finding provably-best routing solutions
- Can handle techn.-specific routing constraints
Why Worry About Symmetries (1)

- Designer’s intent
  - Copying & re-use
- A “mix” of many symmetries
- Proofs by symmetry
Why Worry About Symmetries (2)

- Known facts from theorem-proving
  - Reasoning by symmetry gives shorter proofs, e.g., poly-size versus exp-size
  - Proof length *bounds* runtime of thm provers

- Many symmetries in practice
  - Geometry-related symmetries (routing)
  - Logic-related symmetries (circuits)
  - Architectural symmetries (micro-processors)
Symmetries and Permutations

- **Idea:** represent symmetries of an object by permutations that preserve the object.
- Composition of symmetries is modeled by composition of permutations.
  - Composition is associative.
  - The do-nothing symmetry is the identity.
  - Every symmetry has an inverse.
- This enables applications of group theory.
Compact Representations

- Represent the group of all symmetries
  - Do not list individual symmetries
  - List generating permutations (generators)

Elementary group theory proves:
- If redundant generators are avoided,
- A group with $N$ elements can be represented by at most $\log_2(N)$ generators

Guaranteed exponential compression
Compact Representations (2)

- Sometimes can do better than $\log_2(N)$
- E.g., consider the group $S_k$ of all $k!$ permutations of $1..k$
  - Can be generated by (12) and (123..k)
  - Or by (12), (23), (34), ..., (k-1 k)
- To use this guaranteed compression, we rely on algorithms in terms of permutation generators
Finding Symmetries of Graphs

- **Symmetry (automorphism) of a graph**
  - Permutation of vertices that maps edges to edges

- **Additional constraints**
  - Vertex colors (labels): integers
  - Every vertex must map into a vertex of same color

- **Computational Graph Automorphism**
  - Find generators of a graph’s group of symmetries
  - GraphAuto ∈ NP, and is believed to ∉ P and ∉ NPC
  - Algorithms implemented in GAP(GRAPE(NAUTY))
Symmetries of CNF Formulae

- Permutations of variables that map clauses to clauses
  - E.g., symmetries of \((a+b+c)(d+e+f)\)
    include \((ab), (abc)\) as well as \((ad)(be)(cf)\)
  - Considering single swaps only is not enough

- Ditto for variable negations \((a \rightarrow a')\) and compositions with permutations
  - E.g., symmetries of \((a+b+c)(d+e'+f')\)
    include \((de')\) as well as \((ad)(be')(cf')\)
Reduction to Graph Automorphism

- **CNF formula** $\rightarrow$ **colored graph**
  - Linear time and space
- **Find graph's [colored] symmetries**
  - Worst-case exponential time
- **Interpret graph symmetries found as symmetries of the CNF formula**
  - Permutational symmetries
  - Variable-negation symmetries
Reduction to Graph Automorphism

- Vertices of two colors: clauses and vars
  - One vertex per clause, two per variable
- Edges of three types: (i) incidence, (ii) consistency, and (iii) 2-literal clauses

Clauses: A \((x' + y + z)\), B \((x + y' + z')\), C \((y' + z)\)

Symmetry: 
\((x \: x')(y \: z')(y' \: z)\)
Reduction to Graph Automorphism

• **Consistency edges must map to consistency edges**
  - we do not explicitly enforce that
  - previous reductions do → create larger graphs
  - we reduce the input size for GraphAuto by a constant; recall that $O(2^n) \neq O(2^{cn})$

• **Must ensure correctness (!)**
  - a graph symmetry that maps consistency edges somewhere else is termed *spurious*
  - *spurious symmetries can and do happen*
Lemma: Let $M = (V, R)$ be a perfect matching on a finite vertex set $V$ and let its edges $R$ be colored red. Let $F = (V, G)$ be any graph on $V$ and let its edges $G$ be colored green, where $R \cap G = \emptyset$. Let $\Gamma = (V, R \cup G)$ be the graph on $V$ formed by taking the disjoint union of edge sets $R$ and $G$.

If $\Gamma$ has no cycles with edges of alternating colors, then every automorphism of $\Gamma$ must preserve the color of every edge.

Proof: Suppose $\sigma$ is an automorphism of $\Gamma$ under which, w.l.o.g., a red edge $r_0$ maps to a green edge $g_0$. Note that because the red edges form a perfect matching, the green edge $g_0$ shares each of its end points with exactly one red edge. Let them be $r_1$ and $r_1'$ respectively. Now $\sigma$ must map $r_1$ and $r_1'$ to two distinct edges each of which shares exactly one end point with the edge $r_0$. Furthermore, these two images must be green edges since red edges cannot share end points with other red edges. Let them be $g_1$ and $g_1'$ respectively. So, by now we have two paths of alternating colors, where an edge in $P_1$ is the $\sigma$-image of the corresponding edge in $P_0$ (of opposite color):

$$P_0 = (g_1', r_0, g_1) \quad \text{and} \quad P_1 = (r_1', g_0, r_1).$$

But then, $g_1$ and $g_1'$ must share their “outer” end points with two red edges, say $r_2$ and $r_2'$ respectively. In turn, images of $r_2$ and $r_2'$ must be green edges $g_2$ and $g_2'$ extending from the terminals of the path $P_1$. In effect, we have extended paths $P_0$ and $P_1$ to

$$P_0 = (r_2', g_1', r_0, g_1, r_2) \quad \text{and} \quad P_1 = (g_2', r_1', g_0, r_1, g_2).$$

Repeating the foregoing argument for $g_2$ and $g_2'$ and continuing in this manner, we can “grow”
Correct symmetries:

\[
() \\
(\bar{a}\bar{b}\bar{c})(abc) \\
(\bar{a}\bar{c}\bar{b})(acb) \\
(\bar{a}\bar{a})(c\bar{b})(\bar{c}b) \\
(b\bar{b})(a\bar{c})(\bar{a}c) \\
(\bar{c}\bar{c})(b\bar{a})(\bar{b}a)
\]

Spurious symmetries:

\[
(a\bar{b}\bar{c}\bar{c}a) \\
(a\bar{a}\bar{c}\bar{b}\bar{b}) \\
(a\bar{c})(\bar{b}c)(\bar{b}a) \\
(ab)(\bar{a}\bar{c}) \\
(ac)(\bar{b}\bar{c}) \\
(bc)(\bar{b}\bar{a})
\]

Figure 3: An illustration of spurious symmetries: a CNF formula and its graph. Boolean consistency edges are shown by double-lines, but are indistinguishable from other edges by graph automorphism software NAUTY which cannot handle double-edges. Therefore the graph has 12 symmetries: 6 rotations and 6 axial flips. Only 6 of them — 3 rotations and 3 flips, — preserve Boolean consistency edges and correspond to symmetries of the CNF formula. The remaining 6 symmetries are spurious (the first three spurious symmetries shown are rotations, and the remaining three are axial flips).
Pre-processed CNF instance → CNF → Graph

Symmetry-breaking

Graph symmetries

Call generic SAT solver

nauty or saucy

Shatter
Symmetry Breaking With NAUTY

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Sym</th>
<th>Search</th>
<th>Total</th>
<th>% Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hole-n</td>
<td>0.38</td>
<td>0.07</td>
<td>0.45</td>
<td>84.4</td>
</tr>
<tr>
<td>Urq</td>
<td>0.76</td>
<td>1.17</td>
<td>1.93</td>
<td>39.4</td>
</tr>
<tr>
<td>GRoute</td>
<td>38.76</td>
<td>5.08</td>
<td>43.84</td>
<td>88.4</td>
</tr>
<tr>
<td>FPGAroute</td>
<td>3.24</td>
<td>0.21</td>
<td>3.45</td>
<td>93.9</td>
</tr>
<tr>
<td>ChnlRoute</td>
<td>25.86</td>
<td>0.17</td>
<td>26.03</td>
<td>99.4</td>
</tr>
<tr>
<td>XOR</td>
<td>11.43</td>
<td>2.41</td>
<td>13.84</td>
<td>82.6</td>
</tr>
<tr>
<td>2pipe</td>
<td>23.50</td>
<td>8.01</td>
<td>31.51</td>
<td>74.6</td>
</tr>
</tbody>
</table>

- On all but the synthetic Urquhart instances, symmetry detection with naudy dominates run time.
- Further improvements must come from improved symmetry detection.
**saucy**: Exploiting Structure

- *nauty* works very well on small graphs but fails to scale
  - Runs out of memory on formulas with corresponding graphs having >50,000 vertices
- *saucy* improvement #1: sparse representation
- *saucy* improvement #2: can use bipartiteness
  - Clause vertices only connected to literals
  - Never connected to each other
- *saucy* improvements #3 and #4: algorithmic (e.g., asymptotically faster partition refinement)
### saucy: Empirical Performance

<table>
<thead>
<tr>
<th>Inst.</th>
<th>n</th>
<th>zChaff</th>
<th>nauty</th>
<th>% Sym</th>
<th>saucy</th>
<th>% Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>s4-4-3-1</td>
<td>10354</td>
<td>218.53</td>
<td>88.74</td>
<td>28.9</td>
<td>0.11</td>
<td>0.05</td>
</tr>
<tr>
<td>s4-4-3-2</td>
<td>9974</td>
<td>877.59</td>
<td>79.67</td>
<td>8.3</td>
<td>0.10</td>
<td>0.01</td>
</tr>
<tr>
<td>s4-4-3-3</td>
<td>9970</td>
<td>884.78</td>
<td>75.98</td>
<td>7.9</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>s4-4-3-4</td>
<td>10714</td>
<td>464.46</td>
<td>155.31</td>
<td>25.1</td>
<td>0.14</td>
<td>0.03</td>
</tr>
<tr>
<td>s4-4-3-5</td>
<td>11072</td>
<td>134.09</td>
<td>101.63</td>
<td>43.1</td>
<td>0.11</td>
<td>0.08</td>
</tr>
<tr>
<td>s4-4-3-6</td>
<td>9620</td>
<td>13.24</td>
<td>76.48</td>
<td>85.2</td>
<td>0.10</td>
<td>0.75</td>
</tr>
<tr>
<td>s4-4-3-7</td>
<td>10362</td>
<td>18.27</td>
<td>78.96</td>
<td>81.2</td>
<td>0.10</td>
<td>0.54</td>
</tr>
<tr>
<td>s4-4-3-8</td>
<td>6608</td>
<td>0.68</td>
<td>28.42</td>
<td>97.7</td>
<td>0.06</td>
<td>8.11</td>
</tr>
<tr>
<td>2pipe</td>
<td>3575</td>
<td>0.13</td>
<td>2.93</td>
<td>95.8</td>
<td>0.02</td>
<td>13.33</td>
</tr>
<tr>
<td>3pipe</td>
<td>10048</td>
<td>6.44</td>
<td>57.53</td>
<td>89.9</td>
<td>0.13</td>
<td>1.98</td>
</tr>
<tr>
<td>4pipe</td>
<td>21547</td>
<td>153.50</td>
<td>523.64</td>
<td>77.3</td>
<td>0.49</td>
<td>0.32</td>
</tr>
<tr>
<td>5pipe</td>
<td>38746</td>
<td>122.85</td>
<td>3144.85</td>
<td>96.2</td>
<td>1.65</td>
<td>1.33</td>
</tr>
</tbody>
</table>

[http://vlsicad.eecs.umich.edu/BK/SAUCY/](http://vlsicad.eecs.umich.edu/BK/SAUCY/)
How Backtrack SAT Solvers Work

Let’s solve \((x'+y')(x+y)(x'+y+z)(x+y'+z)(z')\)

- Try \(x=0\)
  - \(y=1\)
  - \(z=1\)
  - violated clause, must backtrack

- Try \(x=1\)
  - \(y=0\)
  - \(z=1\)
  - violated clause, must backtrack

We just repeated similar/same steps twice

Branches \(x=0, y=1\) and \(x=1, y=0\) are symmetric
Speeding up SAT Search

Search space may have symmetries
- May have regions that map 1:1
- This makes search redundant

Ideas for speed-ups
- Prev. slide: require that \((x \leq y)\), i.e., \((x' + y)\)
- Consider equivalence classes under symmetry
- Pick a representative for each class
- Search only one representative per class

This restricted search is \(\equiv\) to original
Symmetry-breaking Predicates

- To restrict search
  - Add clauses to the original CNF formula ("symmetry-breaking" clauses)
  - They will pick representatives of classes
- Our main task is to find those clauses
  - Use only permutations induced by generators
  - Permutation $\rightarrow$ group of clauses (a "symmetry-breaking" predicate)
Construction of S.-b. Predicates

- Input in cycle notation
  - E.g., (12)(34)(567)

- Every cycle considered separately

- In practice almost all are 2- or 3-cycles
  - Two types of 2-cycles: (aa’) and (ab)
  - Symm.-breaking predicates: (a) and (a’+b) resp.

- For multiple cycles
  - Procedure to chain symmetry-breaking predicates
Details: Individual Cycles (1)

- Use an ordering of all variables (arbitrary)
- Symmetry-breaking predicate for (ab):
  - \((a \Rightarrow b)\) aka \((a \leq b)\), if a precedes b
  - Think of truth assignments to b and a
    - Must choose one from 01 and 10

\[\begin{align*}
&\rightarrow 00 \\
&\rightarrow 01 \\
&\rightarrow 10 (a'+b) \\
&\rightarrow 11
\end{align*}\]
Details: Individual Cycles (2)

S.-b. predicate for cycle (abc) is $(a' + b)(b' + c)$

- For 3-var partial assignments, can cycle all 0s to front
Details: Multiple Cycles (1)

Solution space reduction

- By $2x$ when (a) is added to break cycle (aa’)
- Still by $2x$ if permutation has cycles (aa’) and (bb’)
- By $4/3x$ when (a’+b) is added to break cycle (ab)
- **What if** a permutation has cycles (ab) and (cd) ?
  - By $2x$ when (a≤b≤c) is added to break (abc)

Suppose you have cycles (aa’) and (uvt)

- Adding both predicates cuts solution space by $4x$

**Rule of thumb**: after breaking a 2-cycle, symmetry-break the square of the permutation
Details: Multiple Cycles (2)

- **Rule of thumb:** after breaking a 3-cycle, symmetry-break the cube of the permutation
- What if we have both (xy) and (uv)?
  - Squaring will kill the second cycle, so don’t square!
  - Look at partial assignments for x,y: 00, 01, 10 and 11
  - For 10 or 01, (x'+y) is all we can do
  - For 00 or 11, can add (u'+v)
  - Adding (x≤y) and (x=y)⇒(u≤v)
    cuts the solution space by $\frac{8}{5}x$ (better than $\frac{4}{3}x$)
- For 3-cycles, add (x=y=z)⇒(u≤v≤w) or the like
- For multiple cycles ((x=y=z)&(a=b))⇒(u≤v), etc
Multiple Permutations

- Conjoin SBPs for multiple permutations
  - Recall that SBPs pick unique representatives of symmetry orbits
  - The conjunction must pick representatives from each respective [larger] orbit

- Ensuring consistency
  - Use lex-leader SBPs wrt a global var order
  - Sort each cycle, lex-sort all cycles
  - Skip all cycles of length >2 (or use tricks)
Symmetry-Breaking Predicates

Classes of symmetric truth assignments

SAT assignments
How to Select Lex-leaders

Idea: select lexicographically smallest assignments from each equivalence class

Crawford et al. construct an SBP for that:
- map a given assignment by all symmetries
- and require that every image be lex-greater

→ Conjunction over all symmetries 😞
Linear-Size Lex-Leader SBPs

- Take a specific permutation $\pi$
  - Consider an arbitrary truth assignment $x^*$
  - The LL-SBP for $\pi$ is $x^* \leq \pi(x^*)$

Key ideas

- Exploit sparsity: only look at support of $\pi$
- Use a $\leq$-comparator circuit

- Crawford et al used an $n^2$-sized circuit
- An obvious linear-size circuit exists
- Additionally, we account for phase-symmetries

http://www.eecs.umich.edu/~faloul/Tools/shatter/
\[ \pi = \left( x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10} \right) \]
\[ = (x_1, x_4)(x_3, x_8, x'_6)(x_5, x'_4) \]

(a) Permutation in tabular and cyclic notation

\[ \text{BP}(\pi, 1) = x_1 \leq x_4 \]
\[ \text{BP}(\pi, 2) = (x_1 = x_4) \rightarrow (x_2 \leq x_2) \]
\[ \text{BP}(\pi, 3) = (x_1 = x_4)(x_2 = x_2) \rightarrow (x_3 \leq x_8) \]
\[ \text{BP}(\pi, 4) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8) \rightarrow (x_4 \leq x_1) \]
\[ \text{BP}(\pi, 5) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1) \rightarrow (x_5 \leq x'_3) \]
\[ \text{BP}(\pi, 6) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x'_3) \rightarrow (x_6 \leq x'_6) \]
\[ \text{BP}(\pi, 7) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x'_3)(x_6 = x'_7) \rightarrow (x_7 \leq x_7) \]
\[ \text{BP}(\pi, 8) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x'_3)(x_6 = x'_7)(x_7 = x_7) \rightarrow (x_8 \leq x'_6) \]
\[ \text{BP}(\pi, 9) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x'_3)(x_6 = x'_7)(x_7 = x_7)(x_8 = x'_6) \rightarrow (x_9 \leq x_9) \]
\[ \text{BP}(\pi, 10) = (x_1 = x_4)(x_2 = x_2)(x_3 = x_8)(x_4 = x_1)(x_5 = x'_3)(x_6 = x'_7)(x_7 = x_7)(x_8 = x'_6)(x_9 = x_9) \rightarrow (x_{10} \leq x_{10}) \]

(b) Various index sets associated with permutation

\[ \text{supp}(\pi) = \{1, 3, 4, 5, 6, 8\} \]
\[ \text{phase-shift}(\pi) = 5 \]
\[ \text{succ}(\text{phase-shift}(\pi), l_{10}) = \{6, 7, 8, 9, 10\} \]
\[ \text{ends}(\pi) = \{4, 8\} \]
\[ \text{supp}(\pi) \setminus \text{ends}(\pi) = \{1, 3, 5, 6\} \]
\[ \text{supp}(\pi) \setminus \text{ends}(\pi) \setminus \text{succ}(\text{phase-shift}(\pi), l_{10}) = \{1, 3, 5\} \]

(c) Bit predicates according to (10). BPs enclosed in boxes with square corners are tautologous because \( \pi \) maps the corresponding bits to themselves. BPs enclosed in boxes with rounded corners are tautologous because they correspond to cycle “ends.” The BPs for bits 6 to 10 are tautologous because \( \pi \) maps bit 5 to its complement.

\[ \text{PP}(\pi) = (p_1)(p_1 \rightarrow l_4 p_3)(p_3 \rightarrow g_1 \rightarrow l_3 p_5)(p_5 \rightarrow g_3 \rightarrow l_5) \]
\[ = (p_1)(p_1 \rightarrow (x_1 \leq x_4)p_3)(p_3 \rightarrow (x_1 \geq x_4) \rightarrow (x_3 \leq x_8)p_5)(p_5 \rightarrow (x_3 \geq x_8) \rightarrow x'_4) \]

(d) Linear formulation of the permutation predicate according to (18), based only on irredundant bits
Shatter

Pre-processed CNF instance

Symmetry-breaking

Graph

Graph symmetries

CNF symmetries

Call generic SAT solver

naughty or saucy
Asymptotic Results

- Proving the pigeon-hole principle w/o induction
  - A series of SAT instances of growing size
  - Conventional SAT solvers take exponential time
- Our approach empirically takes polynomial time

Graphs:
- "gapholes.gpl": $x^{**7/200000000}$
- "chaffholes.gpl": $x^{**3.3/300000}$
Empirical Results with Chaff

<table>
<thead>
<tr>
<th>Instance</th>
<th>S/U</th>
<th>#V</th>
<th>#CL</th>
<th>Plain Chaff</th>
<th>Time - Chaff</th>
<th>Symmetries</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sec</td>
<td>Out</td>
<td>Finding Sec</td>
<td>Number of Sec</td>
</tr>
<tr>
<td>grout3.3-01</td>
<td>S</td>
<td>864</td>
<td>7592</td>
<td>19.01</td>
<td>0%</td>
<td>4.79</td>
<td>8.71E+09</td>
</tr>
<tr>
<td>grout3.3-03</td>
<td>S</td>
<td>960</td>
<td>9156</td>
<td>44.35</td>
<td>0%</td>
<td>8.94</td>
<td>6.97E+10</td>
</tr>
<tr>
<td>grout3.3-04</td>
<td>S</td>
<td>912</td>
<td>8356</td>
<td>19.36</td>
<td>0%</td>
<td>6.81</td>
<td>2.61E+10</td>
</tr>
<tr>
<td>grout3.3-08</td>
<td>S</td>
<td>912</td>
<td>8356</td>
<td>21.30</td>
<td>0%</td>
<td>7.14</td>
<td>3.48E+10</td>
</tr>
<tr>
<td>grout3.3-10</td>
<td>S</td>
<td>1056</td>
<td>10862</td>
<td>28.18</td>
<td>0%</td>
<td>10.65</td>
<td>3.48E+10</td>
</tr>
<tr>
<td>chnl10x11</td>
<td>U</td>
<td>220</td>
<td>1122</td>
<td>22.17</td>
<td>0%</td>
<td>0.45</td>
<td>4.20E+28</td>
</tr>
<tr>
<td>chnl10x12</td>
<td>U</td>
<td>240</td>
<td>1344</td>
<td>81.88</td>
<td>0%</td>
<td>0.61</td>
<td>6.04E+30</td>
</tr>
<tr>
<td>chnl10x15</td>
<td>U</td>
<td>300</td>
<td>2130</td>
<td>657.61</td>
<td>25%</td>
<td>1.28</td>
<td>4.50E+37</td>
</tr>
<tr>
<td>chnl11x12</td>
<td>U</td>
<td>264</td>
<td>1476</td>
<td>207.37</td>
<td>0%</td>
<td>0.75</td>
<td>7.31E+32</td>
</tr>
<tr>
<td>chnl11x13</td>
<td>U</td>
<td>286</td>
<td>1742</td>
<td>788.32</td>
<td>20%</td>
<td>1.08</td>
<td>1.24E+35</td>
</tr>
<tr>
<td>chnl11x20</td>
<td>U</td>
<td>440</td>
<td>4220</td>
<td>1000</td>
<td>100%</td>
<td>4.4</td>
<td>1.89E+52</td>
</tr>
</tbody>
</table>
## Empirical Results with Chaff

<table>
<thead>
<tr>
<th>Instance</th>
<th>S/U</th>
<th>#V</th>
<th>#CL</th>
<th>Plain Time - Chaff out</th>
<th>Symmetries</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Finding sec of Number #generators</td>
<td>Search Time</td>
</tr>
<tr>
<td>fpga10_8</td>
<td>S</td>
<td>120</td>
<td>448</td>
<td>7.56 sec 0%</td>
<td>0.63 sec 6.00E+71 of all 62</td>
<td>0.05</td>
</tr>
<tr>
<td>fpga10_9</td>
<td>S</td>
<td>135</td>
<td>549</td>
<td>3.80 sec 0%</td>
<td>0.88 sec 6.33E+77 of all 68</td>
<td>0.03</td>
</tr>
<tr>
<td>fpga12_11</td>
<td>S</td>
<td>198</td>
<td>968</td>
<td>694.00 sec 50%</td>
<td>3.76 sec 7.18E+77 of all 95</td>
<td>0.06</td>
</tr>
<tr>
<td>fpga12_12</td>
<td>S</td>
<td>216</td>
<td>1128</td>
<td>80.20 sec 0%</td>
<td>5.31 sec 7.44E+77 of all 104</td>
<td>0.13</td>
</tr>
<tr>
<td>fpga12_8</td>
<td>S</td>
<td>144</td>
<td>560</td>
<td>246.70 sec 10%</td>
<td>1.23 sec 8.41E+77 of all 72</td>
<td>0.08</td>
</tr>
<tr>
<td>fpga12_9</td>
<td>S</td>
<td>162</td>
<td>684</td>
<td>885.00 sec 80%</td>
<td>1.7 sec 2.25E+77 of all 79</td>
<td>0.05</td>
</tr>
<tr>
<td>fpga13_9</td>
<td>S</td>
<td>176</td>
<td>759</td>
<td>550.00 sec 85%</td>
<td>2.57 sec 2.56E+77 of all 84</td>
<td>0.06</td>
</tr>
<tr>
<td>fpga13_10</td>
<td>S</td>
<td>195</td>
<td>905</td>
<td>1000 sec 100%</td>
<td>4.04 sec 5.76E+77 of all 93</td>
<td>0.08</td>
</tr>
<tr>
<td>fpga13_12</td>
<td>S</td>
<td>234</td>
<td>1242</td>
<td>1000 sec 100%</td>
<td>6.9 sec 8.85E+77 of all 110</td>
<td>0.08</td>
</tr>
</tbody>
</table>
# Empirical Results with Chaff

<table>
<thead>
<tr>
<th>Instance</th>
<th>S/U</th>
<th>#V</th>
<th>#CL</th>
<th>Plain Chaff sec</th>
<th>Time-Chaff %</th>
<th>Symmetries Finding sec</th>
<th>Symmetries Number of cycle</th>
<th>#generators</th>
<th>Search Time (S/U)</th>
<th>Speedup Tot</th>
<th>Speedup Only</th>
</tr>
</thead>
<tbody>
<tr>
<td>2dlx_ca_mc</td>
<td>U</td>
<td>3250</td>
<td>24640</td>
<td>6.54</td>
<td>0%</td>
<td>38.36</td>
<td>9.36E+77</td>
<td>10</td>
<td>66</td>
<td>6.30</td>
<td>0.15</td>
</tr>
<tr>
<td>2pipe</td>
<td>U</td>
<td>892</td>
<td>6695</td>
<td>2.08</td>
<td>0%</td>
<td>10.74</td>
<td>2.26E+45</td>
<td>10</td>
<td>38</td>
<td>1.56</td>
<td>0.17</td>
</tr>
<tr>
<td>2pipe_1_ooo</td>
<td>U</td>
<td>834</td>
<td>7026</td>
<td>2.55</td>
<td>0%</td>
<td>9.37</td>
<td>8.00E+00</td>
<td>10</td>
<td>3</td>
<td>1.80</td>
<td>0.23</td>
</tr>
<tr>
<td>2pipe_2_ooo</td>
<td>U</td>
<td>925</td>
<td>8213</td>
<td>3.43</td>
<td>0%</td>
<td>11.14</td>
<td>3.20E+01</td>
<td>10</td>
<td>5</td>
<td>2.82</td>
<td>0.25</td>
</tr>
<tr>
<td>3pipe</td>
<td>U</td>
<td>2468</td>
<td>27533</td>
<td>36.44</td>
<td>0%</td>
<td>463.57</td>
<td>7.29E+77</td>
<td>10</td>
<td>85</td>
<td>19.65</td>
<td>0.08</td>
</tr>
<tr>
<td>2dlx_ca_mc</td>
<td>U</td>
<td>3250</td>
<td>24640</td>
<td>6.54</td>
<td>0%</td>
<td>3.17</td>
<td>2.34E+77</td>
<td>10</td>
<td>64</td>
<td>5.42</td>
<td>0.76</td>
</tr>
<tr>
<td>2pipe</td>
<td>U</td>
<td>892</td>
<td>6695</td>
<td>2.08</td>
<td>0%</td>
<td>10.47</td>
<td>2.26E+45</td>
<td>10</td>
<td>38</td>
<td>1.30</td>
<td>0.18</td>
</tr>
<tr>
<td>2pipe_1_ooo</td>
<td>U</td>
<td>834</td>
<td>7026</td>
<td>2.55</td>
<td>0%</td>
<td>9.02</td>
<td>8.00E+00</td>
<td>10</td>
<td>3</td>
<td>1.80</td>
<td>0.24</td>
</tr>
<tr>
<td>2pipe_2_ooo</td>
<td>U</td>
<td>925</td>
<td>8213</td>
<td>3.43</td>
<td>0%</td>
<td>11.09</td>
<td>3.20E+01</td>
<td>10</td>
<td>5</td>
<td>2.80</td>
<td>0.25</td>
</tr>
<tr>
<td>3pipe</td>
<td>U</td>
<td>2468</td>
<td>27533</td>
<td>36.44</td>
<td>0%</td>
<td>3.63</td>
<td>1.42E+77</td>
<td>10</td>
<td>78</td>
<td>36.20</td>
<td>0.91</td>
</tr>
<tr>
<td>4pipe</td>
<td>U</td>
<td>5237</td>
<td>80213</td>
<td>337.61</td>
<td>0%</td>
<td>9.32</td>
<td>1.03E+78</td>
<td>10</td>
<td>142</td>
<td>334.00</td>
<td>0.98</td>
</tr>
<tr>
<td>5pipe</td>
<td>U</td>
<td>9471</td>
<td>195452</td>
<td>325.92</td>
<td>0%</td>
<td>29.42</td>
<td>3.64E+78</td>
<td>10</td>
<td>227</td>
<td>290.50</td>
<td>1.02</td>
</tr>
</tbody>
</table>
Applicability to Other SAT-solvers

- Similar results achieved with BerkMin, JeruSAT and GRASP
  - Generally, the slower the solver, the greater the speed-up
- Symmetry-breaking predicates with WalkSAT and CPLEX
  - Devastating slow-downs
- Known: WalkSAT’s poor performance on structural CNFs, e.g., chains
Conclusions

- Pre-processing speeds up SAT solvers on difficult instances with symmetries
  - In some cases, the speed-up is exponential
- New constructions
  - Symmetry-finding
  - Symmetry-breaking predicates
- Applications at Synopsys, Siemens, etc
Acknowledgements

Joint work with
- Arathi Ramani, Profs Fadi Aloul and Karem Sakallah

Thanks for insightful comments and help
- Prof. Satya Lokam
- DoRon Motter (MSFT) and Ted Stanion (SNPS)

Support provided by:
- Gigascale Silicon Research Center (GSRC)
- National Science Foundation (NSF), IT Research
- Fellowships
  - Motorola, Agere Systems/SRC, Design Automation Conf.
Thank you

For additional details see
http://www.eecs.umich.edu/~imarkov/pubs