Structure in Boolean Satisfiability

Igor L. Markov
University of Michigan at Ann Arbor
Outline

- Why handling structure is important
- Examples of detecting and using structure
  - Sparsity
  - Clusters and hierarchies
  - Directionality and circuit structure
  - Linear constraints (0-1 ILP)
  - Symmetry
Why Handling Structure is Important

- Presence of structure = the difference between typical cases and worst cases (between useful and hopeless)
  - View this as a meta-definition of structure
  - In practice, must distinguish types of structure
  - Some types are more useful than others
- Exploiting realistic structure is a way to beat NP-completeness (unless P=NP)
Implicit Reliance on Structure in Existing SAT Algorithms

- Randomized Local Search
  - Works well when there are many solutions
  - Works well on homogenous instances with well-pronounced statistical properties
- Constraint propagation speeds up backtracking on constrained instances
- Recursive learning
  - Works well on sparse and clustered instances
- Data structures in all modern SAT solvers exploit sparsity
Explicit Use of Structure

- Known algorithms often implicitly exploit several types of structure
  - E.g., sparsity and presence of many solutions
  - Not necessarily by design
- If you are working with only one type of structure, consider handling it explicitly
  - Figure out how to represent it
  - Develop detection / extraction algorithms
  - Learn how to use extracted information
Example: Circuit-derived CNF Instances

- “Long-range” constraints, directionality

- Randomized local search does not maintain such compatibility

- Constraint propagation is essential
  - E.g., UnitWalk = RandomLocalSearch + UnitClauseRule

- Learning / conflict recording also essential
Another Example: Clusters

- Circuit-derived CNFs inherit circuit connectivity
- VLSI circuits have fractal structure
  - Statistical description in terms of Rent’s rule
- Circuits are designed as hierarchy of clusters
  - Gates in circuit blocks (adder, shifter, comparator)
  - Circuit blocks in components (pipeline stage)
  - Components in modules (micro-processor)
- Recursive min-cut partitioning produces non-trivial results (very small cuts)
Finding Clusters in Circuits

- Circuits are modeled as hyper-graphs
  - Gates are vertices
  - Signal nets (wire connections) are hyper-edges
- Min-cut bi-partitioning
  - Assign every vertex to Part0 or Part1 (approximately 50%:50%)
  - Minimize #hyper-edges that have vertices both in Part0 and Part1
- A cluster will end up in either in Par0 or Part1
- Recursive bisection captures clusters
Finding Clusters in CNF-SAT

- Model CNFs as hyper-graphs
  - Variables are vertices, clauses are hyper-edges (ignore literal polarity)
  - OR clauses are vertices, variables are hyper-edges (again, ignore literal polarity)
- Apply hyper-graph partitioning techniques
  - Circuit partitioning software works well on CNFs that are unrelated to circuit
- MINCE: variable ordering for backtrack search in CNF-SAT
Structure of Pigeon-Hole (n=7) Hypergraph Representation
Structure of Pigeon-Hole (n=7) Cut-Width Profile
Structure of FPGA Routing
Cut-Width Profile
Circuit-solvers vs CNF-solvers

- When the original circuit is known, commercial software typically operates directly on the circuit.
  - Circuits have directionality, but CNF instances do not.
Circuit-solvers vs CNF-solvers

- When the original circuit is known, commercial software typically operates directly on the circuit
  - Circuits have directionality, but CNF instances do not

- **Question 1**: can the original circuit structure be restored? **Yes**

- **Question 2**: can a CNF-SAT solver be sped up if the original circuit is known? **Yes**
Generic Circuit Detection

- Convert the CNF instance to an undirected graph
- Convert the CNF-signature of the gate to match to an undirected graph
- Use subgraph isomorphism to match instances of the gate

Conversions of the clauses

\[(b+d+c)(c+a+b')(a+c')(d+a')\]
Generic Circuit-Detection

- To piece together the circuit, create a maximal independent set (MIS) instance
  - one node per detected gate
  - an edge between nodes if the gates are incompatible
    (signatures overlap, etc.)

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

Encodes (1) \(a=\text{AND}(b,c)\),
(2) \(a=\text{AND}(b,d)\), and (3) \(b=\text{AND}(a,c)\)

Only (2) and (3) are compatible.
AND-OR-NOT Circuit Conversion

- Generic alg requires solving NP-hard problems
- Is there a more efficient way, possibly for a slightly more restricted problem?
- Yes: The mapping from AND-OR-NOT circuits to CNF allows no incompatible gate matches
  - Proof examines each clause of the CNF and shows it must have come from a specific gate
  - Proof suggests efficient linear time algorithm
    - based on pattern-matching of clauses
Easily Detectable Gate Types

<table>
<thead>
<tr>
<th>Gate type</th>
<th>Difficulty of restoring circuit structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>OR and AND</td>
<td>Straightforward pattern-matching</td>
</tr>
<tr>
<td>NOR and NAND</td>
<td>Pattern-matching with back-tracking</td>
</tr>
<tr>
<td>NOT, XOR and XNOR</td>
<td>Can be detected by straightforward pattern-matching, but w/o orientation, which can only be determined in the context of other gate types</td>
</tr>
<tr>
<td>MAJ3</td>
<td>More advanced pattern matching with back-tracking</td>
</tr>
</tbody>
</table>

Table 1: The relative difficulty of detecting particular types of logic gates in CNF-SAT formulas. Note that this is not an exhaustive listing of detectable gates.
### Structure in DIMACS Benchmarks

<table>
<thead>
<tr>
<th>Benchmark series</th>
<th>% variables in simple gates</th>
<th>% clauses in simple gates</th>
<th>% variables in XOR/XNORs</th>
<th>% clauses in XOR/XNORs</th>
<th>Detection runtime (s)</th>
<th># of benchmarks</th>
<th># of variables</th>
<th># of clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bf</td>
<td>54.29%</td>
<td>22.12%</td>
<td>1.18%</td>
<td>0.54%</td>
<td>0.43</td>
<td>4</td>
<td>5793</td>
<td>16566</td>
</tr>
<tr>
<td>Dubois</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0.09</td>
<td>13</td>
<td>1275</td>
<td>3400</td>
</tr>
<tr>
<td>Hanoi</td>
<td>43.22%</td>
<td>10.19%</td>
<td>0%</td>
<td>0%</td>
<td>0.37</td>
<td>2</td>
<td>2041</td>
<td>12272</td>
</tr>
<tr>
<td>Parity</td>
<td>33.17%</td>
<td>13.58%</td>
<td>88.35%</td>
<td>68.42%</td>
<td>2.68</td>
<td>30</td>
<td>24267</td>
<td>83330</td>
</tr>
<tr>
<td>Pret</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0.09</td>
<td>8</td>
<td>840</td>
<td>2240</td>
</tr>
<tr>
<td>Ssa</td>
<td>47.25%</td>
<td>18.57%</td>
<td>1.45%</td>
<td>0.69%</td>
<td>1.09</td>
<td>8</td>
<td>7228</td>
<td>17669</td>
</tr>
<tr>
<td>XOR-Chain</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>99.55%</td>
<td>0.38</td>
<td>27</td>
<td>4554</td>
<td>12126</td>
</tr>
</tbody>
</table>

### Before SAT Preprocessor Hypre

<table>
<thead>
<tr>
<th>Benchmark series</th>
<th>% variables in simple gates</th>
<th>% clauses in simple gates</th>
<th>% variables in XOR/XNORs</th>
<th>% clauses in XOR/XNORs</th>
<th>Detection runtime (s)</th>
<th>Hype runtime (s)</th>
<th>% variables remaining</th>
<th>% clauses remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bf</td>
<td>75.31%</td>
<td>46.41%</td>
<td>0.33%</td>
<td>0.12%</td>
<td>0.23</td>
<td>0.49</td>
<td>22.31%</td>
<td>31.17%</td>
</tr>
<tr>
<td>Dubois</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0.09</td>
<td>0.05</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Hanoi</td>
<td>52.24%</td>
<td>12.56%</td>
<td>0%</td>
<td>0%</td>
<td>0.24</td>
<td>0.23</td>
<td>64.97%</td>
<td>66.35%</td>
</tr>
<tr>
<td>Parity</td>
<td>30.70%</td>
<td>17.09%</td>
<td>100%</td>
<td>83.24%</td>
<td>2.13</td>
<td>1.47</td>
<td>54.99%</td>
<td>67.01%</td>
</tr>
<tr>
<td>Pret</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0.09</td>
<td>0.03</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Ssa</td>
<td>58.21%</td>
<td>29.86%</td>
<td>9.25%</td>
<td>4.74%</td>
<td>0.17</td>
<td>0.34</td>
<td>8.91%</td>
<td>8.33%</td>
</tr>
<tr>
<td>XOR-Chain</td>
<td>0%</td>
<td>0%</td>
<td>100%</td>
<td>100%</td>
<td>0.38</td>
<td>0.17</td>
<td>99.41%</td>
<td>99.55%</td>
</tr>
</tbody>
</table>
## Comparison with Solving Runtime

Circuit Based Technique Runtime = Simulation + (Implicit or Explicit Learning)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>ZChaff</th>
<th>Implicit</th>
<th>Explicit</th>
<th>Simulation</th>
<th>Extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>9Vliw001</td>
<td>1057</td>
<td>567</td>
<td>793</td>
<td>93</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw004</td>
<td>953</td>
<td>804</td>
<td>1011</td>
<td>92</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw005</td>
<td>3126</td>
<td>740</td>
<td>1314</td>
<td>88</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw007</td>
<td>140</td>
<td>286</td>
<td>855</td>
<td>110</td>
<td>11</td>
</tr>
<tr>
<td>9Vliw008</td>
<td>1450</td>
<td>239</td>
<td>1914</td>
<td>114</td>
<td>12</td>
</tr>
<tr>
<td>9Vliw009</td>
<td>1006</td>
<td>784</td>
<td>829</td>
<td>117</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw010</td>
<td>867</td>
<td>329</td>
<td>1897</td>
<td>96</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw015</td>
<td>2209</td>
<td>985</td>
<td>1270</td>
<td>97</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw017</td>
<td>1007</td>
<td>175</td>
<td>913</td>
<td>109</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw019</td>
<td>2936</td>
<td>849</td>
<td>1448</td>
<td>129</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw021</td>
<td>1666</td>
<td>1069</td>
<td>1345</td>
<td>99</td>
<td>10</td>
</tr>
<tr>
<td>9Vliw024</td>
<td>1375</td>
<td>965</td>
<td>1282</td>
<td>107</td>
<td>9</td>
</tr>
</tbody>
</table>
Using Circuit Structure
In CNF-SAT

- Motivating app: equivalence checking
  - Comparing two combinational circuits (A and A’), where one is a modified version of the other
  - Often, many wires in circuit A can be matched to equivalent wires in circuit A’

- Technique
  - Simulate both circuits on 32 random inputs
  - Hash each wire by its 32-bit label
  - Identify likely equivalence classes of wires
  - Let the SAT solver check those equivalences first
  - See K.T.Cheng et al. DATE`03 (www.sigda.org)
Linear Structure (1)

- Counting constraints
  - At most/least k out of n variables can be 1
  - Without additional variables, CNF encodings are exponential in (k+1)
  - Several linear-overhead CNF encodings (e.g., use circuits for adding 1 and “less-than” comparison)
- Handling counting constraints natively allows to solve hole-n quickly
- More generally, extend SAT with 0-1 ILP
Linear Structure (2)

- Current SAT-solving techniques can be extended to support 0-1 ILP constraints
  - PBS, Galena, MiniSAT, MiniSAT-PB
- However, extracting 0-1 ILP constraints from CNF does not seem useful
  - Detection too slow
  - Too few constraints in practice
- Counting constraints are symmetric, but their compact CNF encodings are not
Example

\[ c_1x_1 + c_2x_2 + c_3x_3 + \ldots + c_nx_n \leq g \]

\[ (((s^{1\rightarrow1} + s^{2\rightarrow2}) + s^{3\rightarrow3}) + \ldots + s^{n\rightarrow n}) \leq g \]

\[ s^{i\rightarrow i} \equiv c_i x_i \]

\[ s^{i\rightarrow j} \equiv \sum_{i \leq k \leq j} s^{k\rightarrow k} \]
Example (cont.)

Circuit consistency function:

\[ \varphi = (z \leftrightarrow s^{1\rightarrow n} \leq g) \land \]
\[ \land (s^{1\rightarrow i} \leftrightarrow s^{1\rightarrow i-1} + s^{i\rightarrow i}) \land \]
\[ 2 \leq i \leq n \]
\[ \land (s^{i\rightarrow i} \leftrightarrow c_ix_i) \land \]
\[ 1 \leq i \leq n \]

Satisfiability of PB formula is equivalent to: \[ z \land \varphi \]
Symmetry (1)

- Permutations and negations of Boolean variables that preserve the CNF formula
  - Inputs to a parity checker, ECCs
  - Two adders in a micro-processor
- No symmetry in random CNFs or graphs
- Symmetry helps local search
- Complicates back-tracking
- Breaking a symmetry allows one to cut search space by a small constant factor
  - One of the most impactful types of structure
Symmetry (2)

- Interacts with clusters
- Interacts with counting constraints
- Syntactic symmetries – of the formula / circuit
  - Relatively easy to detect
  - Can be hidden by mangling CNF
- Semantic symmetries – of the Boolean function / solutions to SAT
  - Syntactic symmetries are semantic, but not vice versa
  - Recent work on symmetries of Boolean functions
Summary

- Structured instances vs Worst-case complexity
- All existing SAT-solvers exploit structure
  - Implicitly (intuition about applications)
  - Explicitly (understanding applications)
- If structure is detected, can call relevant algos
- Faster SAT-solving with explicit structural info
- Interactions between different types of structure
  - High-level representations expose symmetry
  - Sparsity can be exploited when detecting other structures
Thank you