Dependency Schemes and Search-Based QBF Solving: Theory and Practice

Florian Lonsing

Friday, 27th April, 2012
Overview

Quantified Boolean Formulae (QBF):
- Extension of propositional logic.
- PSPACE-completeness (propositional logic: NP-completeness).
- Applications in verification and MC: compact encodings.

This Work:
- QBF solving: variable dependencies.
- Dependency schemes to improve QBF solvers.
- DepQBF: search-based QBF solver, integrates dependency schemes.

<table>
<thead>
<tr>
<th>QBFEVAL’10 (568 formulae) – without preprocessing</th>
<th>Solved</th>
<th>Avg. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepQBF</td>
<td>372</td>
<td>334.60</td>
</tr>
<tr>
<td>QuBE7.2-nopp</td>
<td>319</td>
<td>431.47</td>
</tr>
<tr>
<td>Nenofex</td>
<td>211</td>
<td>573.65</td>
</tr>
<tr>
<td>Quantor 3.0</td>
<td>203</td>
<td>590.15</td>
</tr>
<tr>
<td>squolem 2.02</td>
<td>124</td>
<td>708.80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QBFEVAL’10 score-based ranking</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DepQBF</td>
<td>2896.68</td>
</tr>
<tr>
<td>DepQBF-pre</td>
<td>2508.96</td>
</tr>
<tr>
<td>aqme-10</td>
<td>2467.96</td>
</tr>
<tr>
<td>qmaiga</td>
<td>2117.55</td>
</tr>
<tr>
<td>AIGSolve</td>
<td>2037.22</td>
</tr>
<tr>
<td>quantor-3.1</td>
<td>1235.14</td>
</tr>
<tr>
<td>struqs-10</td>
<td>947.83</td>
</tr>
<tr>
<td>nenofex-qbfeval10</td>
<td>829.11</td>
</tr>
</tbody>
</table>
Propositional Logic (SAT):
- Boolean variables $V := \{x_1, \ldots, x_n\}$, literals $l := \nu$ and $l := \bar{\nu}$ for $\nu \in V$.
- Clauses $C_i := (l_1 \lor \ldots \lor l_{k_i})$, CNF $\phi := \bigwedge_{i=1}^{m} C_i$.

Quantified Boolean Formulae (QBF):
- Prenex CNF: quantifier-free CNF over quantified Boolean variables.
- PCNF $F := Q_1 x_1 \ldots Q_n x_n. \phi$, where $Q_i \in \{\exists, \forall\}$, no free variables.
- $Q_i x_i \leq Q_{i+1} x_{i+1}$: variables are linearly ordered.
- Applications: compact encodings, e.g. bounded model checking (BMC).

QBF Semantics: recursively based on formula structure.
- $\forall x \phi$ is satisfiable iff both $\phi[x/0]$ and $\phi[x/1]$ are satisfiable.
- $\exists x \phi$ is satisfiable iff $\phi[x/0]$ or $\phi[x/1]$ is satisfiable.
- Related to search-based QDPLL algorithm (see later).

Problem: prefix ordering might limit the freedom in QBF solving.
Semantical Evaluation:
- $Q_1x_1 \ldots Q_nx_n. \phi$: must assign variables in prefix ordering in general.
- $\exists a \forall x, y \exists b. \phi$: assigning $b$ is possible as soon as $x$ and $y$ are assigned.

Example (Depending Variables)
- $\forall x \exists y. (x = y)$ is satisfiable: value of $y$ depends on value of $x$.
- $\exists y \forall x. (x = y)$ is unsatisfiable: value of $y$ is fixed for all values of $x$.

Breaking the prefix ordering might yield unsound results!

Example (Independent Variables)
- $\forall x \exists y. (x \lor \neg y) \land (\neg x \lor \neg y)$ is satisfiable: assign $x$, then $y$.
- $\exists y \forall x. (x \lor \neg y) \land (\neg x \lor \neg y)$ is satisfiable: assign $y$, then $x$.

Breaking the prefix ordering might be sound and increase freedom!
Goal: identify independent variables in a given PCNF.
- $x$ and $y$ are independent if they can be assigned in arbitrary order.
- Can we go from linear prefix ordering to partial ordering on variables?

Dependency Schemes: relation $D \subseteq (V \times V)$.  
- General framework for expressing (in)dependence in a given PCNF.
- $(x, y) \notin D$: $y$ independent from $x$.
- $(x, y) \in D$: conservatively regard $y$ as depending on $x$.
- Interpret $D$ as a partial ordering on the variables in general.
- Interesting cases: $(x, y) \notin D$ and $(y, x) \notin D$.

Assignment Trees:
- Theoretical foundation of dependency schemes.
- Tree-like models of PCNFs.
- Represent choice of values for $\exists$-variables.
- Explain variable independence.

∀x∃y. (x ∨ ¬y) ∧ (¬x ∨ y).
**Syntactic Approaches:** tradeoff quality vs. efficiency of computation.

- Trivial dependency scheme $D^{\text{triv}}$: given quantifier prefix.
- Quantifier trees $D^{\text{tree}}$: non-deterministic mini-scoping.
- Standard dependency scheme $D^{\text{std}}$: connections between variables.
- $D^{\text{std}} \subseteq D^{\text{tree}} \subseteq D^{\text{triv}}$: apply $D^{\text{std}}$ in practice.

**Example ($D^{\text{tree}}$ vs. $D^{\text{std}}$)**

$\exists a, b \forall x, y \exists c, d. (a \lor x \lor c) \land (a \lor b) \land (b \lor d) \land (y \lor d)$.

Either $(a, y) \in D^{\text{tree}}$ or $(b, x) \in D^{\text{tree}}$ but $(a, y) \notin D^{\text{std}}$ and $(b, x) \notin D^{\text{std}}$. 

---

Florian Lonsing  
**Dependency Schemes and Search-Based QBF Solving**
Dependency Graphs

**Dependency Scheme** $D$ as Directed-Acyclic Graph (DAG):
- Explicit edges $x \to y$ iff $(x, y) \in D$.

**Compressed Dependency Graphs:** equivalence relations, aux. edges.
- “Outgoing” edges: $x \approx \downarrow y$ iff $D(x) = D(y)$.
- “Incoming” edges: $x \approx \uparrow y$ iff $D^{-1}(x) = D^{-1}(y)$.
- Efficient algorithm to compute graph for $D_{\text{std}}$ (see later).

**Example**

```
x_1 \quad x_2 \quad x_3
  |    |    |
  v    v    v
y_1 \quad y_2 \quad y_3
  |    |    |
  v    v    v
z_1 \quad z_2
```

- $[x_1, x_2]_{\uparrow} = [x_1, x_2]_{\downarrow}$
- $[x_3]_{\uparrow} = [x_3]_{\downarrow}$
- $[y_1, y_2]_{\uparrow} = [y_1, y_2]_{\downarrow}$
- $[y_3]_{\uparrow} = [y_3]_{\downarrow}$
- $[z_1]_{\uparrow} = [z_1]_{\downarrow}$
- $[z_2]_{\uparrow} = [z_2]_{\downarrow}$
State qdpll ()
while (true)
    State s = qbcp ();
    if (s == UNDET)
        // Make decision.
        v = select_dec_var ();
        assign_dec_var (v);
    else
        // Conflict or solution.
        // s == UNSAT or s == SAT.
        btlevel = analyze_leaf (s);
        if (btlevel == INVALID)
            return s;
        else
            backtrack (btlevel);

DecLevel analyze_leaf (State s)
    R = get_initial_constraint (s);
    // s == UNSAT: 'R' is empty clause.
    // s == SAT: 'R' is sat. cube...
    // ..or new cube from assignment.
    while (!stop_res (R))
        p = get_pivot (R);
        R’ = get_antecedent (p);
        R = constraint_res (R, p, R’);
        add_to_formula (R);
    return get_asserting_level (R);

Figure: QDPLL with conflict-directed clause and solution-directed cube learning.

Backtracking Search with Constraint Learning:
- Classical QDPLL based on quantifier prefix, i.e. \(D^{\text{triv}}\).
- \texttt{qbcp}: propagate implied (i.e. necessary) assignments.
- \texttt{select\_dec\_var}: decision making.
- \texttt{analyze\_leaf}: add learned constraint produced by Q-resolution.
State qdpll ()
while (true)
    State s = qbcp ();
    if (s == UNDET)
        // Make decision.
        v = select_dec_var ();
        assign_dec_var (v);
    else
        // Conflict or solution.
        // s == UNSAT or s == SAT.
        btlevel = analyze_leaf (s);
        if (btlevel == INVALID)
            return s;
        else
            backtrack (btlevel);
    DecLevel analyze_leaf (State s)
    R = get_initial_constraint (s);
    // s == UNSAT: 'R' is empty clause.
    // s == SAT: 'R' is sat. cube...
    // ..or new cube from assignment.
    while (!stop_res (R))
        p = get_pivot (R);
        R' = get_antecedent (p);
        R = constraint_res (R, p, R');
        add_to_formula (R);
    return get_asserting_level (R);

Figure: QDPLL with conflict-directed clause and solution-directed cube learning.

Replacing $D^{triv}$ with Arbitrary Dependency Scheme $D \subseteq D^{triv}$:

- Same basic framework: considering $D$ as a parameter of QDPLL.
- Only change: $D$ used for dependency checking and decision making.
- Expecting more implications, shorter learned constraints.
- Expecting more freedom for selecting decision variables.
Constraint Reduction (CR):

**Definition**

Let $D$ be a dependency scheme. Given a clause $C$, *constraint reduction* on $C$ by $D$ produces the clause

$$CR_D(C) := C \setminus \{l \in L_\forall(C) \mid \forall l' \in L_\exists(C) : (v(l), v(l')) \notin D\}.$$

- Part of QBCP and Q-resolution for constraint learning.
- Deleting “largest” universal literals: shortens clauses.
- If $D \subset D'$, then $CR$ by $D$ might produce shorter clauses than $CR$ by $D'$.
- Potentially more unit/empty clauses.

**Example**

$$\exists x \forall a \exists y. \phi' \land (x \lor a \lor y).$$

Given $D^{\text{triv}}$ from prefix: $a$ is irreducible in $(x \lor a \lor y)$ since $(a, y) \in D^{\text{triv}}$.

Given $D \subseteq D^{\text{triv}}$ where $(a, y) \notin D$: $a$ is reducible in $(x \lor a \lor y)$, yielding $(x \lor y)$. 

Florian Lonsing  
Dependency Schemes and Search-Based QBF Solving
Dependency Graph for $D_{std}$: efficient incremental construction.

- Statistics for QBFEVAL’08 set (3328 formulae).
- Max. time 8.11s, avg. time 0.09s.
- Compare: explicit computation times out (900s) on 135 formulae.
- For $x \in V_\forall$, $x \in V_\exists$, avg. $|D_{std}(x)| = 19807$ and $|D_{std}(x)| = 4$.
- Graph compactly represents sets of depending variables.
- Dep. classes/dep. variables: 0.01 and 0.02 for $x \in V_\forall$, $x \in V_\exists$.
- Graph is tightly integrated in QDPLL.

State qdpll ()
  while (true)
    State s = qbcp ();
    if (s == UNDET)
      // Make decision.
      v = select_dec_var ();
      assign_dec_var (v);
    else
      // Conflict or solution.
      // s == UNSAT or s == SAT.
      btlevel = analyze_leaf (s);
      if (btlevel == INVALID)
        return s;
      else
        backtrack (btlevel);

DecLevel analyze_leaf (State s)
  R = get_initial_constraint (s);
  // s == UNSAT: ‘R’ is empty clause.
  // s == SAT: ‘R’ is sat. cube...
  // ..or new cube from assignment.
  while (!stop_res (R))
    p = get_pivot (R);
    R’ = get_antecedent (p);
    R = constraint_res (R, p, R’);
  add_to_formula (R);
  return get_asserting_level (R);

Figure: QDPLL with conflict-directed clause and solution-directed cube learning.
Dependency Schemes in QDPLL: implemented in our solver DepQBF.
- Pays off despite overhead.
- Expect performance increase from more powerful dependency schemes.

Table: Comparing different dependency schemes in QDPLL.

<table>
<thead>
<tr>
<th></th>
<th>$D^\text{triv}$</th>
<th>$D^\text{tree}$</th>
<th>$D^\text{std}$</th>
<th>QuBE6.6-nopp</th>
<th>QuBE6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solved</strong></td>
<td>1223</td>
<td>1221</td>
<td><strong>1252</strong></td>
<td>1106</td>
<td><strong>2277</strong></td>
</tr>
<tr>
<td><strong>Avg. Time</strong></td>
<td>579.94</td>
<td>580.64</td>
<td><strong>572.31</strong></td>
<td>608.97</td>
<td><strong>302.49</strong></td>
</tr>
</tbody>
</table>

Table: Dynamic effects of different dependency schemes in QDPLL.

<table>
<thead>
<tr>
<th></th>
<th>$D^\text{triv} \cap D^\text{tree}$</th>
<th>$D^\text{triv} \cap D^\text{std}$</th>
<th>$D^\text{tree} \cap D^\text{std}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>solved</strong></td>
<td><strong>1172</strong></td>
<td>1196</td>
<td>1206</td>
</tr>
<tr>
<td><strong>time</strong></td>
<td><strong>23.15</strong></td>
<td>26.68</td>
<td>23.73</td>
</tr>
<tr>
<td><strong>implied/assigned</strong></td>
<td>90.4%</td>
<td><strong>90.7%</strong></td>
<td>88.6%</td>
</tr>
<tr>
<td><strong>backtracks</strong></td>
<td>32431</td>
<td><strong>27938</strong></td>
<td>34323</td>
</tr>
<tr>
<td><strong>learnt constr. size</strong></td>
<td>157</td>
<td><strong>99</strong></td>
<td>150</td>
</tr>
</tbody>
</table>
Table: DepQBF and other solvers with and without preprocessing.

<table>
<thead>
<tr>
<th>QBF EVAL'10 (568 formulae) – with preprocessing</th>
<th>Solved</th>
<th>Avg. Time</th>
<th>SAT</th>
<th>UNSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloqpr + QxBF + DepQBF</td>
<td>468</td>
<td>197.31 (16.47)</td>
<td>224</td>
<td>244</td>
</tr>
<tr>
<td>Bloqpr + DepQBF</td>
<td>466</td>
<td>198.50 (7.69)</td>
<td>223</td>
<td>243</td>
</tr>
<tr>
<td>QuBE7.2</td>
<td>435</td>
<td>264.70 (–)</td>
<td>202</td>
<td>233</td>
</tr>
<tr>
<td>QxBF+ DepQBF</td>
<td>378</td>
<td>323.19 (7.21)</td>
<td>167</td>
<td>211</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QBF EVAL'10 (568 formulae) – without preprocessing</th>
<th>Solved</th>
<th>Avg. Time</th>
<th>SAT</th>
<th>UNSAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DepQBF</td>
<td>372</td>
<td>334.60</td>
<td>166</td>
<td>206</td>
</tr>
<tr>
<td>QuBE7.2-nopp</td>
<td>319</td>
<td>431.47</td>
<td>144</td>
<td>175</td>
</tr>
<tr>
<td>Nenofex</td>
<td>211</td>
<td>573.65</td>
<td>103</td>
<td>108</td>
</tr>
<tr>
<td>Quantor 3.0</td>
<td>203</td>
<td>590.15</td>
<td>99</td>
<td>104</td>
</tr>
<tr>
<td>squolem 2.02</td>
<td>124</td>
<td>708.80</td>
<td>53</td>
<td>71</td>
</tr>
</tbody>
</table>
Figure: Sorted run times of selected solvers from Table 3.
Conclusions

**Drawbacks of Prenex CNF:**
- Quantifier prefix limits freedom of QBF decision procedures.
- Linear ordering of variables might often be relaxed.

**Dependency Schemes:**
- Variable independence: quality vs. efficiency of computation.
- Related to QBF semantics: inherent property.
- From linear to partial orders on variables: increased freedom.
- Relevant for *arbitrary* QBF solvers.

**DepQBF:** search-based, competitive, open-source.
- Combining QDPLL with $D^{std}$.
- Improved overall performance despite overhead.
- Fewer backtracks, shorter learnt constraints, more implications.

**Open Problems and Future Work:**
- Theoretical results related to QDPLL with $D \subseteq D^{triv}$.
- Applying more powerful dependency schemes than $D^{std}$.
- Constraint learning in QDPLL.
References
U. Bubeck and H. Kleine Büning.
Bounded Universal Expansion for Preprocessing QBF.

M. Benedetti.
Quantifier Trees for QBFs.

A. Biere.
Resolve and Expand.

H. Kleine Büning, M. Karpinski, and A. Flögel.
Resolution for Quantified Boolean Formulas.

M. Cadoli, A. Giovanardi, and M. Schaerf.
An Algorithm to Evaluate Quantified Boolean Formulae.

E. Giunchiglia, M. Narizzano, and A. Tacchella.
Learning for Quantified Boolean Logic Satisfiability.

