Integrating Dependency Schemes in Search-based QBF Solvers (F Lonsing and A Biere)
presented by Dustin Wehr
Dependency schemes

A dependency scheme $D$ for a PCNF $F$ is first of all a DAG on the variables of $F$, with edges only between differently-quantified variables.

$y$ depends on $x$ according to $D$ if there’s an edge from $x$ to $y$.

$D(x)$ denotes the set of vars that depend on $x$ (according to $D$).

$D$ must have the property: It is sound for QDPLL to assign $x$ just as soon as every predecessor of $x$ is assigned. In that case we say $x$ is enabled.
Quantifier trees give dependency schemes
(the $D_{\text{tree}}$ family of dependency schemes)

Fig. 1. Quantifier trees for the PCNF $\exists a, b\forall x, y\exists c, d. (a \lor b) \land (a \lor x \lor c) \land (b \lor c) \land (b \lor y \lor d)$.

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

Right: $\exists b. (\forall y\exists d. b \lor y \lor d) \land (\exists a. (a \lor b) \land (\forall x\exists c. a \lor x \lor c) \land (b \lor c))$
Dependency schemes

- Dependency scheme $D_1$ is **less restrictive** than dependency scheme $D_2$ iff $D_1 \subseteq D_2$ (i.e. $\text{edges}(D_1) \subseteq \text{edges}(D_2)$)

- Trivial dependency scheme $D_{\text{triv}}$:
  - We saw this in last friday’s lecture.
  - $y$ depends on $x$ iff $x$ and $y$ are differently-quantified and $y$ is bound after $x$ (in the given PCNF).

- Every $D_{\text{tree}}$ scheme is less restrictive than $D_{\text{triv}}$.

- Authors also use a scheme called $D_{\text{std}}$, that is less restrictive than the $D_{\text{tree}}$ schemes.
A nice way of viewing the state of the solver...

- Let $F = Q_1 \ldots Q_r \phi$ be an input PCNF.
- As QDPLL proceeds, learned clauses are added to $\phi$. Let $\phi'$ be the expanded set.
- When $\psi$ is the disjunction of the cubes learned so far while evaluating $F$, the augmented CNF form is $Q_1 \ldots Q_r \phi' \lor \psi$.
- At this stage, QDPLL has proved that the augmented CNF has the same truth value as $F$. 
Survey of modifications to QDPLL

State qdplll ()
    while (true)
        State s = bcp ();
        if (s == UNDEF)
            // 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_reason (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);
            A = get_antecedent (p);
            R = constraint_res (R, p, A);
            add_to_formula (R);
            assign_forced_lit (R);
        return get_asserting_level (R);
Unit propagation with arbitrary dependency schemes

• A clause $C$ is **unit** iff:
  • No literal in $C$ is assigned true.
  • Exactly one $\exists$-literal $l_e$ in $C$ is unassigned.
  • $l_e$ doesn't depend on any of the unassigned $\forall$-literals in $C$.

• A cube $C$ is **unit** iff:
  • No literal in $C$ is assigned false.
  • Exactly one $\forall$-literal $l_u$ in $C$ is unassigned.
  • $l_u$ doesn't depend on any of the unassigned $\exists$-literals in $C$. 
Unit propagation with arbitrary dependency schemes

(for clauses - defn for cubes is similar)

• Generalization of two-literal watching scheme:
  Two unassigned literals \(l_1, l_2\) in each clause are watched such that:
  • \(q(l_1) = q(l_2) = \exists\), OR
  • \(q(l_1) = \forall, q(l_2) = \exists\) and \(l_2\) depends on \(l_1\).
Constraint learning with arbitrary dependency schemes (i.e. clause and cube learning)

- Constraint learning is reminiscent of 1-UIP. We’ll focus on clause learning; cube learning is dual.

- The antecedent clauses on the conflicting path (all of which were unit clauses, as in 1-UIP) that are resolved with the conflicting clause to generate the learned asserting clause are first universally-reduced.

- Universal reduction of C: Remove a ∀-literal from C if there is no ∃-literal in C that depends on it.
Constraint learning with arbitrary dependency schemes

- The **antecedent clauses** on the conflicting path (all of which were unit clauses, as in 1-UlP) that are resolved with the **conflicting clause** to generate the learned **asserting clause** are first universally-reduced.

- **Universal reduction** of $C$: Remove a $∀$-literal from $C$ if there is no $∃$-literal in $C$ that depends on it.

- Starting with $R = the conflicting clause$, $∃$-literals in $R$ (which were forced) of maximum decision level are resolved away until there is only one such literal $l$. After each such resolution, the resolvent is universally reduced.

- That literal $l$ must have this property: any $∀$-literal $l_u$ in $R$ that $l$ depends on was assigned false earlier (at a smaller decision level).
Constraint learning with arbitrary dependency schemes

DecLevel analyze_leaf (State s)
    R = get_initial_reason (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);
        A = get_antecedent (p);
        R = constraint_res (R, p, A);
    add_to_formula (R);
    assign_forced_lit (R);
    return get_asserting_level (R);
Incrementally maintaining the set of enabled vars using a compressed dependency graph

• Recall: var x is enabled iff all the variables that it depends on have been assigned.

• Efficiency issue 1: If the number of variables is large, it’s prohibitive to keep the dependency graph in memory explicitly (it can have size quadratic in # of vars)

• So merge vars into equivalence classes: x ~ y if x and y have the same predecessors and successors.

• Leave out transitive edges and then use adjacency lists.

• Efficiency issue 2: it’s too costly (in terms of time) to maintain the set of enabled vars explicitly (uncompressed).
Example: incrementally maintaining the set of enabled vars

- \([x_1] = [x_2] = \{x_1, x_2\}\), etc.
  i.e. \(\{x_1, x_2\}\) are an equivalence class; they’re equivalent w.r.t what the dependency scheme says about them.

- Initially:
  - no vars are assigned
  - \(w, x_1, x_2\) are the only enabled vars (since they have no predecessors). They are **decision candidates** since they’re also unassigned.
Example: incrementally maintaining the set of enabled vars

- select_dec_var() is called for first time and returns $x_1$.
- Suppose that causes $x_2$ and $y_1$ to be assigned by unit prop.
- Next call to select_dec_var() updates the dependency graph data structure, by processing $x_1, x_2, y_1$ in turn.
Example: incrementally maintaining the set of enabled vars

- Say \( x_1 \) got assigned 1.
- Processing \( x_1 \):
  - Because \( x_2 \in [x_1] \) is not yet assigned, no new vars could be enabled.
  - The graph is not traversed. (if it was traversed, nothing would change).
Example: incrementally maintaining the set of enabled vars

- Say $x_2$ got assigned 0 (by unit prop)
- Processing $x_2$:
  - Now every var in $[x_2]$ is assigned, so must traverse successors of $[x_2]$ (i.e. $[y_2]$ and $[w']$)
  - $w'$ becomes enabled.
  - $y_2$ does not become enabled, since it depends on $w$, which hasn’t yet been assigned.
Example: incrementally maintaining the set of enabled vars

- Say $y_1$ got assigned 0 (by unit prop)
- Processing $y_1$:
  - Because $y_2 \in [y_1]$ is not yet assigned, no new vars could be enabled.
  - The graph is not traversed.
Experimental results

### QBF-EVAL’08 (3326 formulae)

<table>
<thead>
<tr>
<th></th>
<th>$D_{\text{triv}}$</th>
<th>$D_{\text{tree}}$</th>
<th>$D_{\text{std}}$</th>
<th>QuBE6.6-np</th>
<th>QuBE6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>1223</td>
<td>1221</td>
<td>1252</td>
<td>1106</td>
<td>2277</td>
</tr>
<tr>
<td>time</td>
<td>579.94</td>
<td>580.64</td>
<td>572.31</td>
<td>608.97</td>
<td>302.49</td>
</tr>
</tbody>
</table>

### QBF-EVAL’07 (1136 formulae)

<table>
<thead>
<tr>
<th></th>
<th>$D_{\text{triv}}$</th>
<th>$D_{\text{tree}}$</th>
<th>$D_{\text{std}}$</th>
<th>QuBE6.6-np</th>
<th>QuBE6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>533</td>
<td>548</td>
<td>567</td>
<td>458</td>
<td>734</td>
</tr>
<tr>
<td>time</td>
<td>497.12</td>
<td>484.69</td>
<td>469.97</td>
<td>549.29</td>
<td>348.05</td>
</tr>
</tbody>
</table>

### Herbstritt (478 formulae)

<table>
<thead>
<tr>
<th></th>
<th>$D_{\text{triv}}$</th>
<th>$D_{\text{tree}}$</th>
<th>$D_{\text{std}}$</th>
<th>QuBE6.6-np</th>
<th>QuBE6.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>321</td>
<td>357</td>
<td>357</td>
<td>296</td>
<td>395</td>
</tr>
<tr>
<td>time</td>
<td>316.06</td>
<td>248.20</td>
<td>248.07</td>
<td>357.52</td>
<td>173.53</td>
</tr>
</tbody>
</table>

- very good except in comparison to QuBE6.6 with preprocessing

- Authors’ solver DepQBF does not use preprocessing.
- But… in a later 2011 paper, they do preprocessing and give experimental results where they beat QuBE.
# Experimental results

<table>
<thead>
<tr>
<th>QBFEVAL’08 (solved only)</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>solved</td>
<td>1172</td>
<td>1196</td>
<td>1206</td>
</tr>
<tr>
<td>time</td>
<td>23.15</td>
<td>26.68</td>
<td>23.73</td>
</tr>
<tr>
<td>implied/assigned</td>
<td>90.4%</td>
<td>90.7%</td>
<td>88.6%</td>
</tr>
<tr>
<td>backtracks</td>
<td>32431</td>
<td>27938</td>
<td>34323</td>
</tr>
<tr>
<td>sat. cubes/sol.</td>
<td>1.8%</td>
<td>2.9%</td>
<td>1.8%</td>
</tr>
<tr>
<td>learnt constr. size</td>
<td>157</td>
<td>99</td>
<td>150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QBFEVAL’07 (solved only)</th>
<th>501</th>
<th>513</th>
<th>537</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>501</td>
<td>513</td>
<td>537</td>
</tr>
<tr>
<td>time</td>
<td>31.22</td>
<td>34.46</td>
<td>32.76</td>
</tr>
<tr>
<td>implied/assigned</td>
<td>89.0%</td>
<td>89.2%</td>
<td>87.7%</td>
</tr>
<tr>
<td>backtracks</td>
<td>35131</td>
<td>22334</td>
<td>39906</td>
</tr>
<tr>
<td>sat. cubes/sol.</td>
<td>4.0%</td>
<td>10.0%</td>
<td>4.0%</td>
</tr>
<tr>
<td>learnt constr. size</td>
<td>150</td>
<td>101</td>
<td>134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Herbstritt (solved only)</th>
<th>312</th>
<th>308</th>
<th>348</th>
</tr>
</thead>
<tbody>
<tr>
<td>solved</td>
<td>312</td>
<td>308</td>
<td>348</td>
</tr>
<tr>
<td>time</td>
<td>26.86</td>
<td>19.28</td>
<td>24.41</td>
</tr>
<tr>
<td>implied/assigned</td>
<td>96.6%</td>
<td>97.4%</td>
<td>96.2%</td>
</tr>
<tr>
<td>backtracks</td>
<td>26565</td>
<td>1329</td>
<td>26733</td>
</tr>
<tr>
<td>sat. cubes/sol.</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>learnt constr. size</td>
<td>174</td>
<td>306</td>
<td>173</td>
</tr>
</tbody>
</table>

- less-restrictive dependency schemes → more unit prop
- note \(D_{\text{std}}\) did *more* backtracks than \(D_{\text{tree}}\).
Experimental results

**QBF EVAL’08 (3326 formulae)**

<table>
<thead>
<tr>
<th></th>
<th>$D^{\text{triv}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{triv}}$</th>
<th>$D^{\text{tree}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{tree}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DC/d$</td>
<td>13801.0</td>
<td>13801.6</td>
<td>11409.7</td>
<td>11409.0</td>
</tr>
<tr>
<td>$DC$-upd.</td>
<td>3.23</td>
<td>3.16</td>
<td>3.30</td>
<td>3.43</td>
</tr>
<tr>
<td>$\prec$</td>
<td>1</td>
<td>-</td>
<td>6.21</td>
<td>-</td>
</tr>
<tr>
<td>$C$-red.</td>
<td>1.18</td>
<td>-</td>
<td>535.62</td>
<td>-</td>
</tr>
</tbody>
</table>

**Herbstritt (478 formulae)**

<table>
<thead>
<tr>
<th></th>
<th>$D^{\text{triv}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{triv}}$</th>
<th>$D^{\text{tree}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{tree}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DC$-upd.</td>
<td>20.67</td>
<td>20.67</td>
<td>20.16</td>
<td>20.16</td>
</tr>
</tbody>
</table>

**Pan (384 formulae) \cup Sorting-Networks (84 formulae)**

<table>
<thead>
<tr>
<th></th>
<th>$D^{\text{triv}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{triv}}$</th>
<th>$D^{\text{tree}} \times D^{\text{std}}$</th>
<th>$D^{\text{std}} \times D^{\text{tree}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DC/d$</td>
<td>75.81</td>
<td>93.87</td>
<td>117.50</td>
<td>109.66</td>
</tr>
<tr>
<td>$DC$-upd.</td>
<td>86.89</td>
<td>86.90</td>
<td>120.03</td>
<td>119.98</td>
</tr>
</tbody>
</table>

- $DC/d$ is (sum of $|\text{Decision Candidates}_i|$ over all decisions $i$) / (# of decisions) (or rather, the average of those terms)

- $DC$-upd: avg time cost per assigning/unassigning variables when updating $DC$ before decision and during backtracking.

- $\prec$: average time cost of dependency checks for unit prop and constraint reduction

- $C$-red: Note they did not optimize constraint reduction
Summary of contributions

- demonstrating that using non-trivial dependency schemes can be useful in practice, despite the substantial extra overhead
- Even if the solver is highly modular w.r.t to the dependency scheme
- showing how dependency schemes can improve unit propagation and constraint learning