Watched Literals and Generating Propagators in Constraint Programming

Peter Nightingale
Ian P. Gent
Chris Jefferson
Ian Miguel
Karen Petrie
Neil Moore

21st August 13:45-14:10 H1058
Introduction to Minion

- Minion is a relatively simple, non-hybrid CP solver (unlike previous talk!)
- Interleaves backtracking search and propagation (reasoning about constraints)

Propagate the consequences of $x_1=1$
Introduction to Minion

- Focus on making the propagation loop efficient and scalable
- Deliberately few options – “model and run”
  - However, very simple search limits “model and run”

Propagate the consequences of \( x_1=1 \)

\[ x_1=1 \quad x_1\neq 1 \]
Propagation Example

First three rows of a Sudoku

Suppose we look at the first row

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1..9</td>
<td>7</td>
<td>1..9</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
<tr>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
</tbody>
</table>
Propagation Example

First three rows of a Sudoku

Suppose we look at the first row – we can delete some values

<table>
<thead>
<tr>
<th>1,2</th>
<th>7</th>
<th>1,2</th>
<th>4</th>
<th>8</th>
<th>9</th>
<th>3</th>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>6</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
<tr>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
</tbody>
</table>
Propagation Example

First three rows of a Sudoku

Suppose we look at the first row – we can delete some values

Move on to the first sub-square
## Propagation Example

First three rows of a Sudoku

Suppose we look at the first row – we can delete some values

Move on to the first sub-square – deletes some values on the bottom row, including values 1,2 as a consequence of the first constraint

<table>
<thead>
<tr>
<th>1,2</th>
<th>7</th>
<th>1,2</th>
<th>4..9</th>
<th>8..9</th>
<th>9..9</th>
<th>3..9</th>
<th>6..9</th>
<th>5..9</th>
</tr>
</thead>
<tbody>
<tr>
<td>3..9</td>
<td>5</td>
<td>6..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
<tr>
<td>4,8,9</td>
<td>4,8,9</td>
<td>4,8,9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
<td>1..9</td>
</tr>
</tbody>
</table>
Propagation

- Propagation is a tight loop
- Constraints read and write variable domains heavily (mostly read)
- Deleting a value from a variable domain triggers other constraints to be executed
- Queue(s) hold variable events or constraints to be propagated
- Managing internal state of constraints
- Lots of efficiency issues
Propagation and Minion

- Brief overview of some research performed with Minion
  - Specialisation of variables
  - Watched Literals
  - Propagator Generation
Specialisation of Variables

Pre-process

CSP specification

Queue of constraints / variable events to propagate

Propagate

Solution

Add constraints / variable events to queue

Variable domains

Propagate

Queue of constraints / variable events to propagate

This interface is the most heavily used
Specialisation of Variables

- Minion has 5 types of variables:
  - Boolean
  - Bounds – just stores upper and lower bound
  - Discrete
  - Sparse Bounds
  - Constant (more useful than it sounds)

- ... And two interfaces:
  - Negated Boolean
  - Reference to any variable type
Specialisation of Variables

- Minion may have been first to reject one-size-fits-all variable representation
- Gave it a brief advantage
- Other systems (ILOG CP, Gecode) have now closed the gap
Specialisation of Variables

- Minion has effectively 6 types of variables
- How to access them from propagators?
  - Through interface with virtual function calls
  - Switch statements
  - Specialise propagators
- Specialising propagators allows inlining, in-place optimisation of the variables’ methods
- Most propagators in Minion compiled 49 times – 7 times each for two sets of variables
## Specialisation of Variables

- Compare specialisation to virtual function calls
- Time (s) for whole solver, not just propagators
- Current version 0.14

<table>
<thead>
<tr>
<th></th>
<th>Minion</th>
<th>Minion-virtual funcs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIBD 10</td>
<td>39</td>
<td>107</td>
</tr>
<tr>
<td>Graceful Graph k6p2</td>
<td>68</td>
<td>83</td>
</tr>
<tr>
<td>Quasigroup 7-10</td>
<td>162</td>
<td>196</td>
</tr>
<tr>
<td>Solitaire 6</td>
<td>22</td>
<td>33</td>
</tr>
</tbody>
</table>
Watched Literals

- Propositional Satisfiability (SAT) solvers introduced watched literals
- All variables are boolean
- Constraints all look like this: \((x_1 \text{ or } x_2 \text{ or } \neg x_3 \text{ or } x_4)\)
- If \(x_1=F, x_2=F\) and \(x_3=T,\) then need to assign \(x_4=T\)
Watched Literals

- Watch two literals: \((x_1 \text{ or } x_2 \text{ or } \neg x_3 \text{ or } x_4)\)
- Suppose \(x_4\) is assigned \(F\): don’t care (not watched)
  - \(O(o)\) work, compared to \(O(1)\) with static triggers
- Suppose \(x_2=F\).
  - Update watches: \((x_1 \text{ or } x_2 \text{ or } \neg x_3 \text{ or } x_4)\)
- Suppose \(x_1=F\). Update: We can’t.
- Assign \(x_3\) to \(F\) to satisfy the constraint.
Watched Literals in Search

- Watched literals are **not backtracked** as search backtracks.
- No cost from copying/trailing/recomputing
- Supports of constraint must be *backtrack stable* to use WLs. Otherwise backtrack them.
Watched Literals

- WLs adapted to constraint programming
- Minion uses WLs for propagating disjunctions of constraints (among other things)
- Generalised pigeon-hole experiment:

<table>
<thead>
<tr>
<th>$&lt; n, p, d &gt;$</th>
<th>Watched OR</th>
<th>Sum</th>
<th>Watched Sum</th>
<th>Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; 100, 5, 2 &gt;$</td>
<td>191,536.22</td>
<td>19,304.05</td>
<td>29,404.22</td>
<td>54,180.04</td>
</tr>
<tr>
<td>$&lt; 100, 10, 2 &gt;$</td>
<td>499,007.21</td>
<td>1,268.15</td>
<td>1,377.21</td>
<td>79,704.14</td>
</tr>
<tr>
<td>$&lt; 100, 20, 2 &gt;$</td>
<td>1,576,413.85</td>
<td>755.48</td>
<td>782.40</td>
<td>87,443.99</td>
</tr>
<tr>
<td>$&lt; 100, 30, 2 &gt;$</td>
<td>1,579,347.99</td>
<td>548.23</td>
<td>564.70</td>
<td>84,170.60</td>
</tr>
<tr>
<td>$&lt; 100, 40, 2 &gt;$</td>
<td>1,461,316.06</td>
<td>424.32</td>
<td>428.23</td>
<td>78,234.20</td>
</tr>
<tr>
<td>$&lt; 100, 50, 2 &gt;$</td>
<td>1,439,796.97</td>
<td>370.62</td>
<td>373.95</td>
<td>76,766.77</td>
</tr>
</tbody>
</table>
Propagator Generation

- Given a constraint, automatically generate a simple (tree) propagator...
- At each node, branches for a literal in/out of domain
- Nodes labelled with deletions

```
 o in D1
     o not in D1
      Prune 1 from x2
```
Propagator Generation

- Very simple, no incremental state, no clever triggering, doesn’t exploit symmetries in the constraint...
- Yet performs surprisingly well on small constraints
Propagator Generation

-Executes in time $O(nd)$
-Compare to $O(d^n)$ (at least) for table constraints
-Cost is moved up-front
  - $O(2^{nd})$ to generate the tree, same space to store it
  - Actual size depends on constraint, heuristic
Propagator Generation

- Beating a hand-crafted propagator! (Peg Solitaire)

<table>
<thead>
<tr>
<th>Starting position</th>
<th>Node rate (per s)</th>
<th>Generated</th>
<th>Min</th>
<th>Reified Sumgeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11249 7088</td>
<td>3303</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6338 4140</td>
<td>3312</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10986 7514</td>
<td>3926</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12964 8431</td>
<td>3652</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11135 7531</td>
<td>3544</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>13456 8886</td>
<td>3920</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>6892 4315</td>
<td>2587</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Propagator Generation

- Compared to two table propagators (Oscillating Life)

<table>
<thead>
<tr>
<th>$n$</th>
<th>period $p$</th>
<th>Generated</th>
<th>Time (s)</th>
<th>Sum Lighttable</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
<td>0.04</td>
<td>0.09</td>
<td>0.20</td>
<td>0.22</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0.08</td>
<td>0.42</td>
<td>1.34</td>
<td>1.26</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0.42</td>
<td>2.38</td>
<td>7.42</td>
<td>6.05</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1.09</td>
<td>6.35</td>
<td>21.55</td>
<td>16.66</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2.34</td>
<td>11.18</td>
<td>40.00</td>
<td>38.15</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0.13</td>
<td>0.67</td>
<td>2.03</td>
<td>2.17</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>0.93</td>
<td>7.02</td>
<td>19.18</td>
<td>24.59</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>11.98</td>
<td>75.29</td>
<td>350.19</td>
<td>225.29</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>124.75</td>
<td>896.97</td>
<td>2779.78</td>
<td>1999.82</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>446.44</td>
<td>3108.18</td>
<td>13929.2</td>
<td>6231.22</td>
</tr>
</tbody>
</table>
Minion

- **Why use it?**
  - You have lots of nested Or/And
  - Universal reification (including Or and And)
    - And universal reify imply (half reification)
  - There is a good static variable order
    - Or DOM/WDEG works well

- **Why not use it?**
  - You need Cumulative, Hamiltonian Circuit
  - You need sophisticated search
Conclusions

- Try it out: minion.sourceforge.net