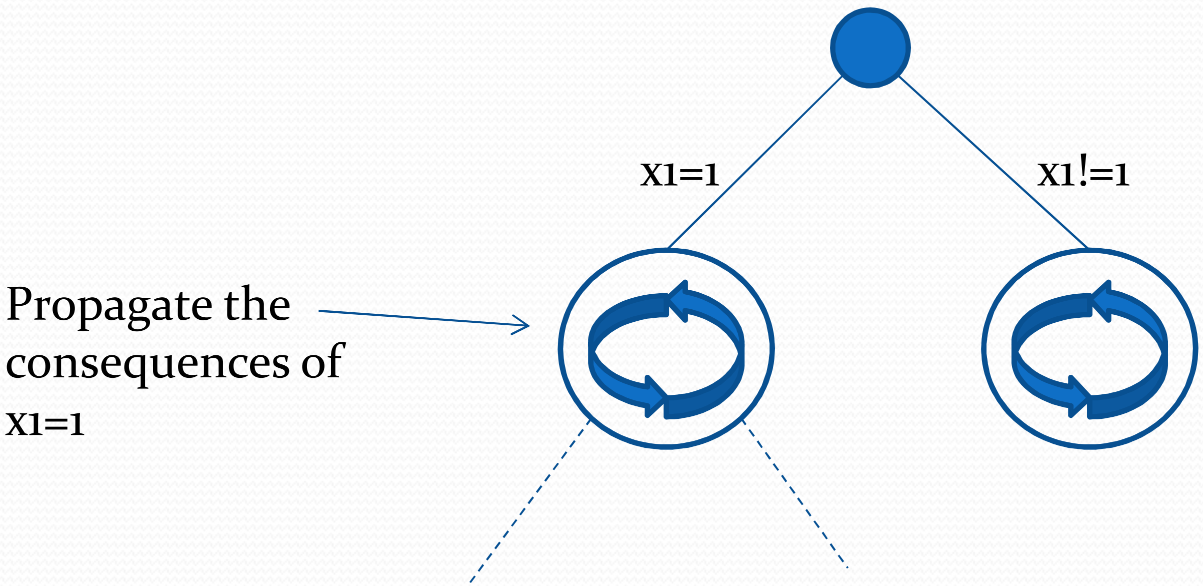
Watched Literals and Generating Propagators in Constraint Programming

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

21st August 13:45-14:10 H1058

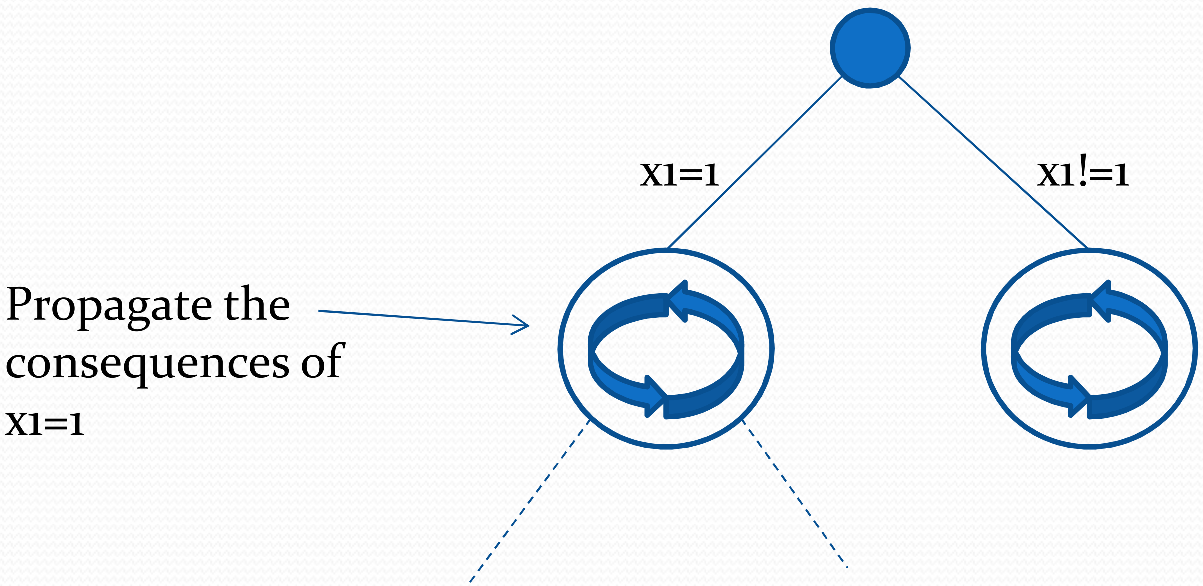
Introduction to Minion

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



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”



Propagation Example

First three rows of a Sudoku

Suppose we look at the first row

1..9	7	1..9	4	8	9	3	6	5
3	5	6	1..9	1..9	1..9	1..9	1..9	1..9
1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9

Propagation Example

First three rows of a Sudoku

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

1,2	7	1,2	4	8	9	3	6	5
3	5	6	1..9	1..9	1..9	1..9	1..9	1..9
1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9

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

1,2	7	1,2	4	8	9	3	6	5
3	5	6	1..9	1..9	1..9	1..9	1..9	1..9
1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9	1..9

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

1,2	7	1,2	4	8	9	3	6	5
3	5	6	1..9	1..9	1..9	1..9	1..9	1..9
4,8,9	4,8,9	4,8,9	1..9	1..9	1..9	1..9	1..9	1..9

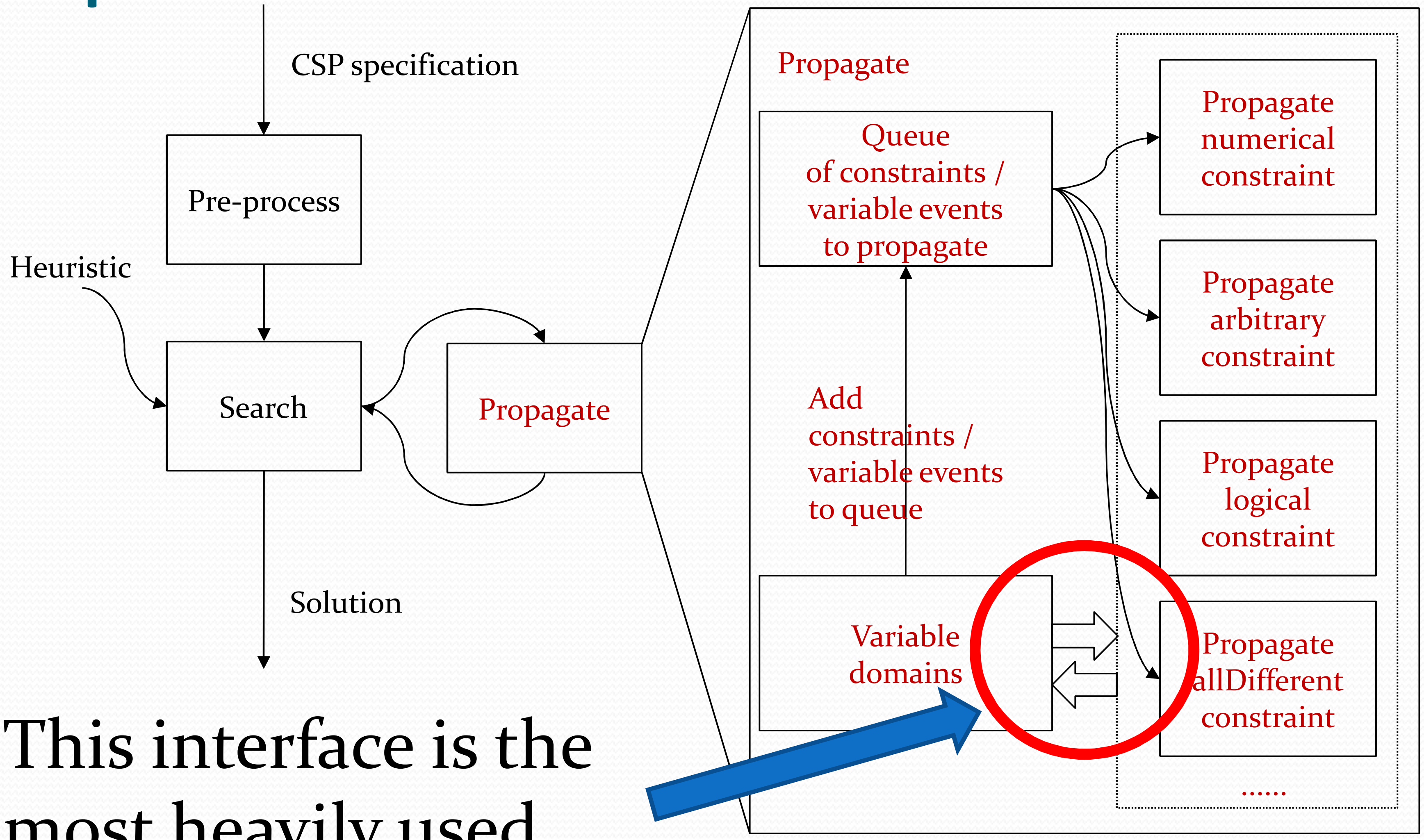
Propagation

- 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



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

	Minion	Minion-virtual funcs
BIBD 10	39	107
Graceful Graph k6p2	68	83
Quasigroup 7-10	162	196
Solitaire 6	22	33

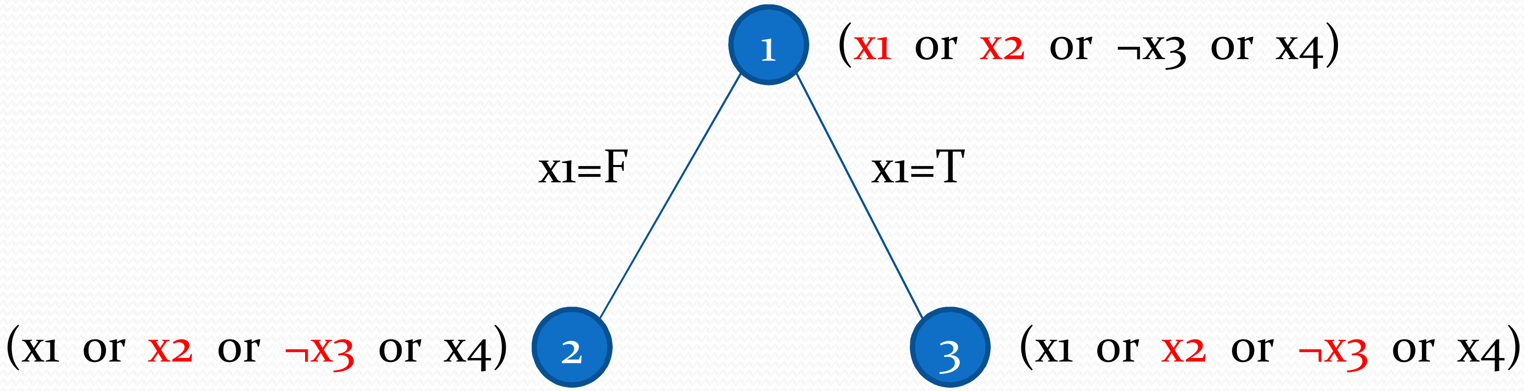
Watched Literals

- Propositional Satisfiability (SAT) solvers introduced watched literals
- All variables are boolean
- Constraints all look like this: $(x1 \;\text{or}\; x2 \;\text{or}\; \neg x3 \;\text{or}\; x4)$
- If $x1=F$, $x2=F$ and $x3=T$, then need to assign $x4=T$

Watched Literals

- Watch two literals: (x_1 or x_2 or $\neg x_3$ or x_4)
- Suppose x_4 is assigned F: don't care (not watched)
  - O(0) work, compared to O(1) with static triggers
- Suppose x_2=F.
  - Update watches: (x_1 or x_2 or $\neg x_3$ 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.

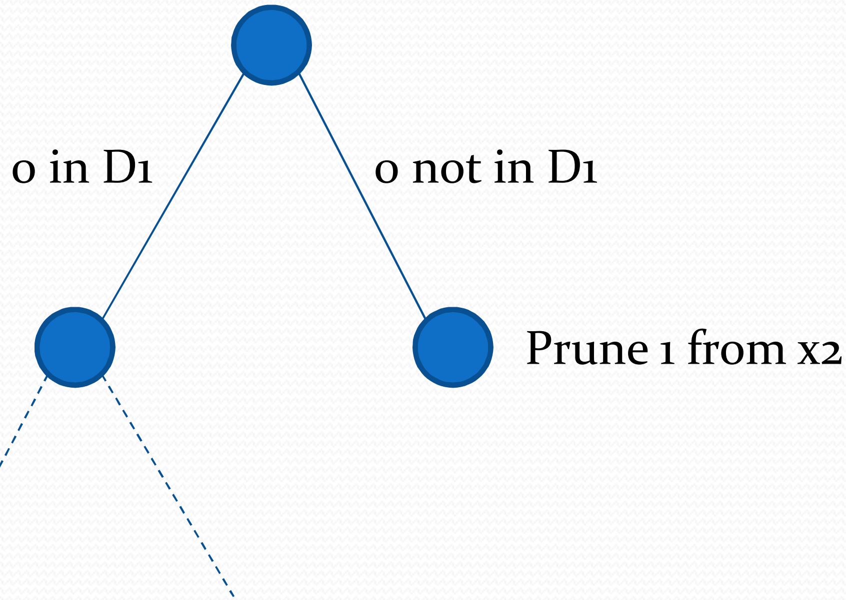
Watched Literals

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

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

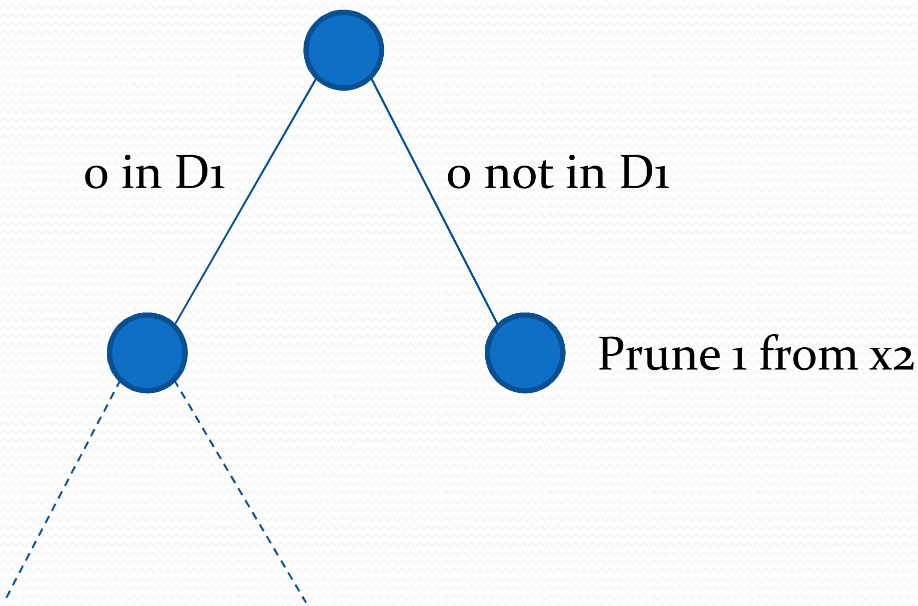
Propagator Generation

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



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)

Starting position	Node rate (per s)		
	Generated	Min	Reified Sumgeq
1	11249	7088	3303
2	6338	4140	3312
4	10986	7514	3926
5	12964	8431	3652
9	11135	7531	3544
10	13456	8886	3920
17	6892	4315	2587

Propagator Generation

- Compared to two table propagators (Oscillating Life)

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

Minion

- Why use it?
  - You have lots of nested Or/And
  - Universal reification (including Or and And)
    - And universal reifyimply (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