Generating Special-purpose Stateless Propagators for Arbitrary Constraints

Ian Gent, Chris Jefferson, Ian Miguel, Peter Nightingale

University of St Andrews
Why is this the way forward?

- 59,049
- 564,092,290
How do you propagate an arbitrary new constraint?

• Hope it is one built in to your solver?
  – But it may not be

• Encode it using existing constraints?
  – Requires modelling time
  – May not achieve GAC

• Use table constraint implementation?
  – Can be exponential during search
  – Definitely not optimised to new constraint

• Write a new special purpose propagator?
  – At best takes programmer time
  – At worst is a significant research project
Instead, we show how to ...

• Generate a special purpose propagator for an arbitrary new constraint
  – Either specified by table or existing propagator
• The generated propagator is ...
  – created by code-generating code
  – guaranteed to achieve GAC every call
  – worst case time of $O(nd)$ per call during search
• Our best experimental results
  – Better than handwritten propagators
  – Achieved 18x speedup on “Oscillating Life”
    • Carrying out identical search to table constraint
  – Easily repaid overheads many times over
But ...

• The overheads can be ...
  – Exponential preprocessing before search
  – Exponential space needed during search
  – 10-ary boolean constraint close to our limit

• Overheads incurred per different table
  – So need constraint to occur many times
    • In a single problem
    • Or over many problems

  – Which does happen
    • Oscillating Life, compile constraint once, run often
• This work is part of the Dominion project
  – Generously funded by EPSRC
  – Follow up to Minion

• *Generates* solver for every instance
  – i.e. is code-generating code
  – Produces a very special purpose solver!
  – Exploit individual features of instance
  – Make implementation choices for this instance
    • Which may not be good in a general purpose solver
• For generating GAC propagators ...
  – We therefore know domain size of all variables
  – And exact set of allowed tuples
  – But we will often see that propagators are reusable

• For technical reasons ...
  – Propagators used in Minion, not Dominion
  – Technical reason being Dominion not existing
Basic Idea

• For *every* state domains of variables can be in
  – Compute what GAC does and store it
    • Either from propagator we are given ...
    • ... or from GAC-table propagator

• Generate code which
  – Looks up state of domains
  – Then does what the store tells it to
Basic Idea

• Constraint of arity $n$, domain size $d$
• There are $2^d - 1$ states of each variable
  – Excluding the empty domain
• So there are $(2^d - 1)^n$ states of all variables
  – e.g. $d = 2$, $n = 10$ gives 59,049 states
• So it will pay to avoid considering all states
• Note that algorithm is *stateless*
  – i.e. it stores no state between invocations
  – All variants in this paper have this property
Why is this the way forward?

• 59,049
  – states to precompute

• 564,092,290
Second idea

• Build a binary decision tree
• Nodes store propagations
  – And next decision (unless leaf)
  – Leaves are sure to be GAC
• Decisions are domain membership
  – i in D(x)? for variable x value I
    • E.g. if domain of x is \{1,3\}
    • Then 1 in D(x), 2 not in D(x), 3 in D(x)
    • Even though 1 or 3 may later be removed
• Proved correct in this paper

Tree for x v y, D(x) = D(y) = \{0,1\}
Binary decision tree

• There are only \( nd \) domain values
  – So max length of branch is \( nd \), as is max run time
• GAC now to be run at each node in this tree
  – Instead of for every possible state of domains
  – If there are \( T \) nodes in tree then cost is
  – \( O( T n d^n) \)
• \( T \) can be as low as 1 for any \( n, d \)
  – E.g. domain wipeout at root
Third Idea

• Some nodes are redundant
  – Nodes where constraint is entailed
  – Other subtrees where no propagation happens

• These nodes can be turned into leaf nodes
  – Reduces space usage
  – Algorithm in paper & implemented
  – Not proved but “obviously” ok
Third Idea

• Some nodes are redundant
  – Nodes where constraint is entailed
  – Other subtrees where no propagation happens

• These nodes can be turned into leaf nodes
  – Reduces space usage
  – Algorithm in paper & implemented
  – Not proved but “obviously” ok
Third Idea

• Some nodes are redundant
  – Nodes where constraint is entailed
  – Other subtrees where no propagation happens
• These nodes can be turned into leaf nodes
  – Reduces space usage
  – Algorithm in paper & implemented
  – Not proved but “obviously” ok
Heuristic

• Variable & value ordering
• Pick variable/value pair which
  – Is in maximum number of disallowed tuples
  – So that result is most likely to be entailed
Experiments

• Three case studies (great detail in paper)
  – Peg Solitaire
  – Low Autocorrelation Binary Sequences
  – Maximum Density Oscillating Still Life
• In each case constraint can be reused
  – So overheads can be amortised over all instances
• Compare against various other techniques
  – Minion GAC-table (using tries)
  – Minion “light table”
  – Encodings using other constraints
Oscillating Life Results
(overhead = 262s)

Run time using Generated propagator (sec)

Speedup

- light-table
- table
- encoded
Why is this the way forward?

- 59,049
  - states to precompute
- 564,092,290
  - nodes in just one search
LABS Results
(overhead = 15.7s)

Speedup vs. Run time using Generated propagator (sec)

- light-table
- table
- encoded
Peg Solitaire Results
(overhead = 16.6s)

![Graph showing speedup versus nodes per second using generated propagator, with two lines representing Encoded (reify) and Encoded (min).]
Experiments: Key Points

• Generated *always* the fastest method
• Generated *much* faster than table/light table
  – Searching identical search space
• Generated repays overhead on hard instances
  – *And* can be reused across all instances in family
• Still faster than encodings
  – Using hand-optimised propagators!
  – And encodings need thought
Relationship to other work

• Two main bodies of related work
  – Improving GAC for table constraint
    • Lhomme/Regin 05
    • Lecoutre/Szymanek 06,
    • Cheng/Yap 06 and 10,
    • Gent/Jefferson/Miguel/Nightingale 07,
    • Katsirelos/Walsh 07
    • Lecoutre/Hemery 07
  – Constructing sets of rules in CHR to establish GAC
    • Apt/Monfroy 01
• But no work (we believe) which generates GAC propagators to be polynomial during search
Conclusions

• We aim to push GAC propagators in new direction
  – Exponential preprocessing / poly during search
  – Dominion is perfect context as we have the full details of the problem before compilation

• Shown can obtain excellent results

• Many ways work can be extended
Why is this the way forward?

- **59,049**
  - states to precompute

- **564,092,290**
  - nodes in just one search
If you have been ...

... thanks for listening
Question

• Why not store the tree in a data structure instead of in code?
  – Avoids need for compiler at run time

• Yes, but compiled code runs even faster
• Even a 20% speedup can make the difference between winning and losing
• But something to consider certainly
Question

• What is the difference between entailed constraint and “no pruning”?

• Because of path to decision in binary tree
  – We know that some values are in domain

• E.g. x=/=y, original domains D(x) = D(y) = {0,1,2,3,4}
  – we might have asked
  – 0 in D(x)? Yes
  – 1 in D(x)? Yes
  – 0 in D(y)? Yes
  – 1 in D(y)? Yes

• It is now guaranteed that no propagation can occur
• But it is still possible that later we will set x = 0, and then propagate y =/= 0
Question

• Can you compute lazily?

• Yes ...
  – Obvious to think of doing GAC when propagations not cached
  – I.e. memoize propagation
  – Gets rid of exponential preprocessing

• No ...
  – Would need very different infrastructure to ours
  – Might increase cost of optimising data structure

• Obvious area of future work
Question

• Can you use BDDs/MDDs/Other data structure?
  
  • No and Yes
  
  • No ...
    – not just by storing the constraint
    – We still need to do all propagations in advance
  
  • Yes ...
    – could compress data structure
    – i.e. which stores all the propagations
Question

• Is your run time really O(nd)?
• Is GAC-table really exponential during search?
• (Because data structure is exponential size)

• No...
  – True, input is much bigger so theoretical runtime is not what we said

• Yes...
  – After setup, exponential state is static and sits in RAM
  – In practice, access to data structure is O(1)
How to make oscillating life run 18 times faster

Minion table constraint

Automatically generated