AUTOMATED REFORMULATION OF CONSTRAINT MODELS IN SAVILE ROW

Peter Nightingale

WITH THANKS TO…

Thanks to co-authors (on the Savile Row paper, here at CP 2014):

Ozgur Akgun, Ian Gent, Chris Jefferson, Ian Miguel

And:

Andrea Rendl (author of Tailor)

Hakan Kjellerstrand, Andras Salamon for donating models

James Wetter for interesting conversations about AC-CSE

Glenna Faith Nightingale for putting up with me, plotting data

OUTLINE

Demo of Savile Row

What is automated reformulation?

A brief survey of work to date

Introducing Savile Row and the Essence' language

Reformulations in Savile Row

Chaining optimisations – examples

DEMO OF SAVILE ROW

A short demo – using Savile Row to solve Equidistant Frequency Permutation Arrays

EQUIDISTANT FREQUENCY PERMUTATION ARRAYS

$q=3$, $d=4$, $\lambda=2$

5 codewords: $v=5$

Video demo

4
differences

2 of each
symbol in
each
codeword

c1	1	1	2	2	3	3
c2	1	2	1	3	2	3
c3	1	2	3	1	3	2
c4	1	3	2	3	1	2
c5	1	3	3	2	2	1

AUTOMATED REFORMULATION – THE GOAL

What is a reformulation?

- Mapping from model to model preserving solutions
- Mapping preserving at least one solution (symmetry breaking, variable elimination)
- Mapping preserving at least one optimal solution (dominance breaking)

Automated Reformulation – Given a naïve model, automatically improve it via a sequence of reformulations

We can get inspiration from two main sources:

- Compilers (we may have exhausted this source)
- Automating techniques used by expert modellers

Typically stronger reformulations can be done on problem instances

- This talk focuses on instances

BRIEF SURVEY

I could divide up modelling as follows:

- Choosing viewpoints, channelling between them
- Stating (re-stating) constraints for effective propagation
- Symmetry breaking, dominances
- Adding implied constraints
- Defining search strategy

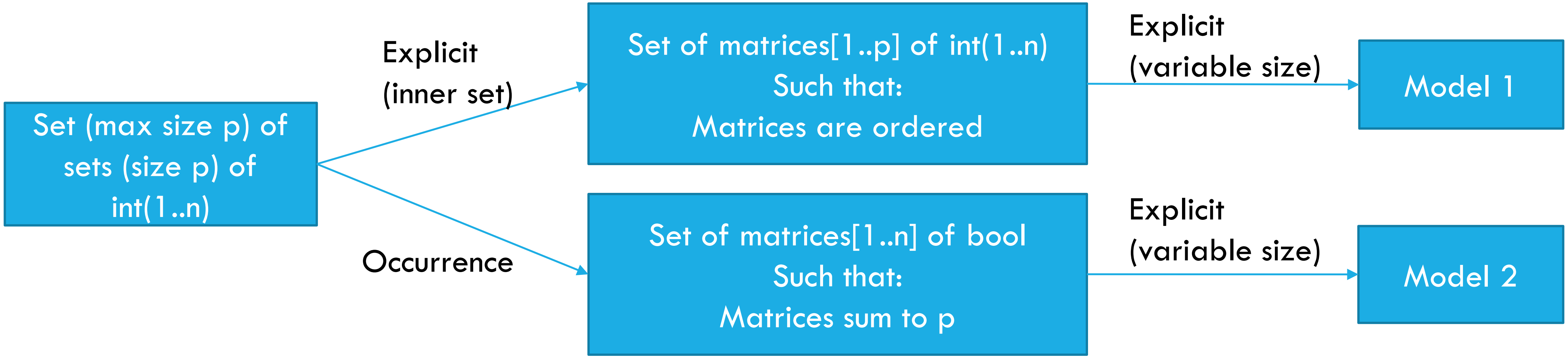
Probably not exhaustive

Survey of automated or semi-automated reformulations under these headings

BRIEF SURVEY – VIEWPOINTS

Choosing viewpoints, channelling between them – Some – CONJURE can generate multiple models based on different viewpoints (Ozgur Akgun, Ian Miguel et al).

Works from a language with nested types.



BRIEF SURVEY – VIEWPOINTS

Choosing viewpoints, channelling between them – Some – CONJURE can generate multiple models based on different viewpoints (Ozgur Akgun, Ian Miguel et al).

Race the resulting models on a set of instances

- Recommends a portfolio of ‘good’ models

Replaces intuition or insight with systematic exploration of known refinements

Really should be called automated modelling – starts with an abstract specification language

BRIEF SURVEY – CONSTRAINTS

Stating (re-stating) constraints for effective propagation – Not a great deal

Aggregation (CGRASS, IBM CP Optimizer Presolve) – Collect constraints into globals, e.g. collect at-most, at-least into a Global Cardinality Constraint

Common Sub-expression Elimination (Andrea Rendl's Tailor, IBM Presolve, Minizinc with Functions) – Connect constraints together by finding identical or equivalent expressions

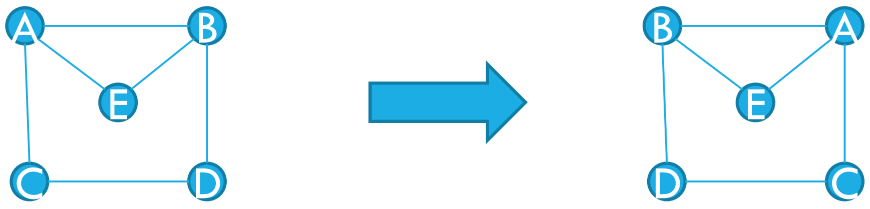
Model Seeker (Beldiceanu, Simonis) – Induces a model (using global constraints) from a set of example solutions

MiniZinc Globalizer (Leo et al) – Interactive, suggests global constraints to replace constraint sets – relies on model structure for scopes, e.g. matrix rows/columns

BRIEF SURVEY – SYMMETRY

Symmetry breaking, dominances – Yes

Graph automorphism-based approaches e.g. SHATTER (Aloul, Markov, Sakallah) – add constraints to break the symmetry in the constraint network



Dominance detection and breaking (Chu & Stuckey)

Dynamic symmetry breaking SBDD, SBDS – avoids generating the constraints but we would still need to find the symmetries

BRIEF SURVEY – IMPLIED CONSTRAINTS

Adding implied constraints – Some

Automatic Generation of Implied Constraints (Charnley, Colton, Miguel) – Generate solutions of small instances, hypothesise implied constraints about the problem class with HR, prove/disprove with Otter

Learning implied global constraints (Bessiere, Coletta, Petit) – Find parameters for implied (parameterised) global constraints

CGRASS (Frisch, Miguel, Walsh) – Forward-chaining of a set of rules deriving new constraints

BRIEF SURVEY – SEARCH STRATEGY

Defining search strategy – May not be necessary

SAT, SMT solvers have excellent heuristics, restarts, conflict learning

In CP: DOM/WDEG, impact, restarts, …

BRIEF SURVEY

Very brief survey – please don't be offended if your paper was not there!

Aspects of modelling:

1. Choosing viewpoints, channelling between them – Some
2. Stating (re-stating) constraints for effective propagation – Not a great deal
3. Symmetry breaking, dominances – Yes
4. Adding implied constraints – Some
5. Defining search strategy – May not be necessary

Some aspects better researched than others

We will be looking mainly at 2. & 4.

INTRODUCING SAVILE ROW

Modelling tool

Reads the evolving language Essence' (Essence-Prime)

- I'll show some examples later

Produces solver output for:

- Minion
- Gecode (via Flatzinc)
- An almost-flat, instance-level subset of Minizinc
- Experimental output for Dominion solver synthesiser
- In next non-bugfix release: SAT

INTRODUCING SAVILE ROW

Implemented in Java – works on Linux, Mac, Windows

Savile Row 1.6 available under GPL 3

http://savilerow.cs.st-andrews.ac.uk/

WHY THE NAME?

Tailoring: optimising and specialising a model for a particular solver

(name by Andrea Rendl)

Savile Row is a street of bespoke tailors in London

SIGNS

Need more signs for future releases

Any suggestions?

(no prize for the Hollywood sign)

THIS TUTORIAL

Focusses on a particular point in an automated modelling toolchain

After class parameters are known

Before general flattening, specialising for particular solvers

- Of course Savile Row does do general flattening etc

Finite domain CP

- Although many of the reformulations I'll talk about are relevant elsewhere
- Set variables and continuous variables

SAVILE ROW AND ESSENCE'

THE ESSENCE’ LANGUAGE

Essence’ is similar to other modelling languages e.g. OPL

The Essence’ model file represents a problem class

The parameter file contains the instance data

Contains only the types Integer, Boolean, and type constructor Matrix – no abstract types such as Set, Multiset, Function

Quantifiers forAll, exists, sum to construct expressions

Matrix comprehensions to build matrices

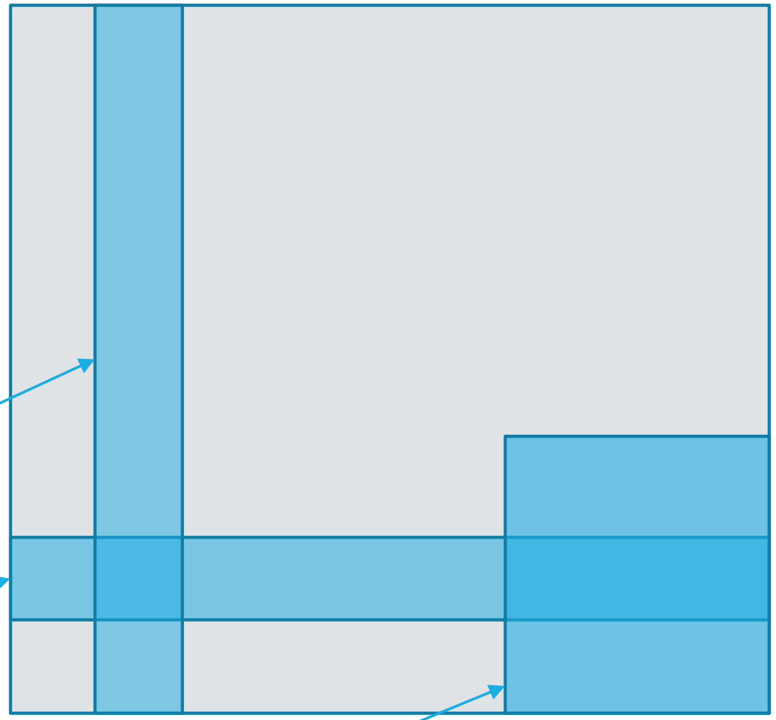
The usual infix operators: +, *, /, =, <, /\, \/, etc

Various functions and global constraints

ESSENCE' EXAMPLES – KNAPSACK

```
language ESSENCE' 1.0
given maxWeight : int
given values : matrix indexed by [int(1..numEntries)] of int(1..)
given weights : matrix indexed by [int(1..numEntries)] of int(1..)
find x : matrix indexed by [int(1..numEntries)] of bool
maximising sum i : int(1..numEntries) . x[i]*values[i]
such that
   ( sum i : int(1..numEntries) . x[i]*weights[i] ) <= maxWeight
```

ESSENCE' EXAMPLES – SUDOKU



```
language ESSENCE' 1.0
find M : matrix indexed by [int(1..9), int(1..9)] of int(1..9)
such that
forAll row : int(1..9) .  allDiff(M[row,..]),
forAll col : int(1..9) .  allDiff(M[..,col]),
forAll i,j : int(1,4,7) .  allDiff( [ M[k,l] | k : int(i..i+2), l : int(j..j+2) ] )
$ Also needs the clues
```

Quantifiers and matrix comprehensions

ESSENCE' EXAMPLES – CAR SEQUENCING

Just the essential constraints – slide along the sequence seq, and gcc

```
forAll option : int(1..numoptions) .
    forAll windowStart : int(1..numcars-windowSize[option]+1) .
        (sum pos : int(windowStart..windowStart+windowSize[option]-1) .
            seq[pos] in toSet([ class | class : int(1..numclasses), optionsRequired[class, option]])
        )<=optMax[option],
gcc(seq, [ i | i : int(1..numclasses)], numberPerClass)
```

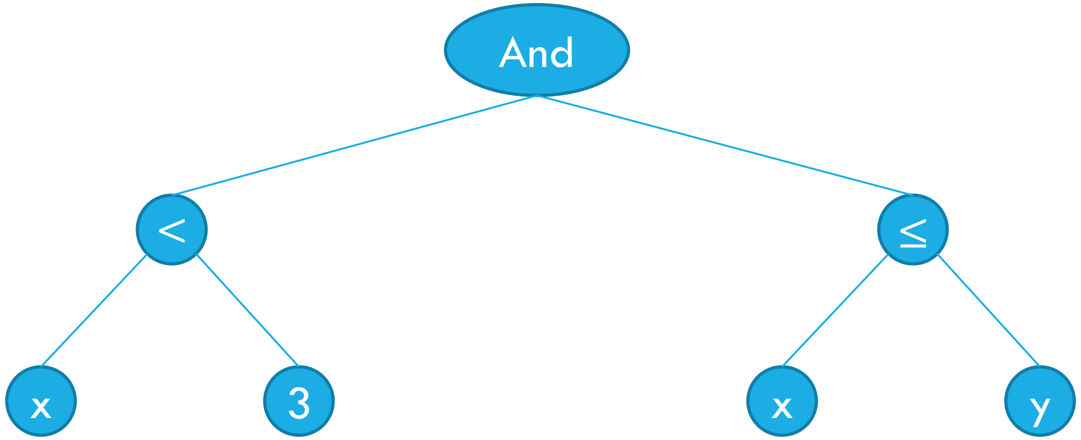
toSet([…]) is just a set of integers (car classes) constructed from a matrix parameter

SAVILE ROW – ABSTRACT SYNTAX TREE

Savile Row works on an abstract syntax tree

Ordinary ‘rules’ walk the tree, replacing subtrees

More complex reformulations scan the whole tree, then replace subtrees later



THE BARE MINIMUM

Minimal tailoring to a typical solver:

1. Deal with undefinedness – SR implement the relational semantics
2. Substitute in the parameters
3. Unroll all quantifiers and matrix comprehensions
4. Flatten any non-flat expressions, e.g. X[allDiff([p,q,r])]=5 becomes X[a]=5 and (a <-> allDiff([p,q,r]). Definition of flat varies by solver.
5. Output to solver language

Some assumptions here. Any expression can be extracted when flattening; matrices are indexed from 0 and one-dimensional.

THE BARE MINIMUM

Suppose we have this expression:

```
forAll i,j : int(1..100)  .  i<j -> M[i,j] ≠ M[j,i]
```

Unrolls to:

1<1 –> M[1,1] ≠ M[1,1],

1<2 –> M[1,2] ≠ M[2,1], etc

Flattens to:

a <–> (1<1),

b <–> (M[1,1] ≠ M[1,1]),

a –> b

Terrible model – lots of useless auxiliary variables (in red)

CONSTANT EVALUATION

The minimal tailoring doesn't do constant evaluation (evaluating constant sub-expressions)

```
forAll i,j : int(1..100)  .  i<j -> M[i,j] ≠ M[j,i]
```

Unrolls to:

$1<1 \rightarrow M[1,1] \neq M[1,1],$

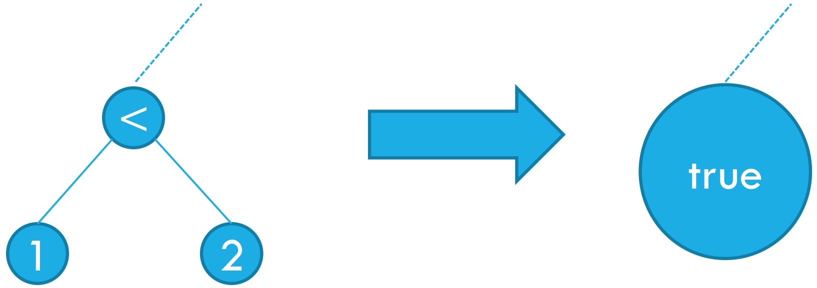
$1<2 \rightarrow M[1,2] \neq M[2,1]$, etc

Evaluate all constant sub-expressions:

false $\rightarrow M[1,1] \neq M[1,1],$

true $\rightarrow M[1,2] \neq M[2,1]$, etc

Better, but not good enough



CONSTANT FOLDING

Implication (and many others) can be further simplified when containing constants

Evaluate all constant sub-expressions:

false –> M[1,1] ≠ M[1,1],

true –> M[1,2] ≠ M[2,1], etc

Apply rules about implication-with-constants:

true,

M[1,2] ≠ M[2,1], etc

Just need to remove that true from the And.

BASIC TRANSLATION

Basic (nearly minimal) tailoring to a typical solver:

1. Deal with undefinedness
2. Substitute in the parameters
3. Unroll all quantifiers and matrix comprehensions
4. Constant folding
5. Flatten any non-flat expressions, e.g. X[allDiff([p,q,r])]=5 becomes X[a]=5 and (a <-> allDiff([p,q,r]). Definition of flat varies by solver.
6. Output to solver language

Does no model optimisation

MATRICES

Essence' allows holey matrices

find M : matrix indexed by [int(1..5, 10..15), int(2,4,6,8)] of int(1..10)

Two ways of accessing M:

- Matrix dereference M[X,Y] – X and Y can be any expression, with or without decision variables.
- Matrix slice M[Z,..] – Take row Z from M. Z is any expression without decision variables.

Fun: M[..,..][..,6][M[1,2]]

How to slice with a decision variable expression Z:

[M[Z,i] | i : int(2,4,6,8)]

MATRICES

Dealt with in multiple stages

1. Shift indices to 0-based contiguous ranges and adjust all matrix derefs and slices
2. Matrix derefs become element functions
   - M[x,y] becomes element(flatten(M), 4x+y)
3. Atomise into individual decision variables
   - element([M_00_00, M_00_01, …], 4x+y)

Last step allows, for example, unifying two variables in the matrix, tightening domains of individual variables from the matrix

AUXILIARY VARIABLES

Flattening (and other processes we will see later) create new auxiliary variables

Potential Danger – increase the number of solutions, increase search

To avoid problems, there are some conventions about aux variables

1. Always a function of primary variables
   - Possibly via other auxiliary variables
   - Cannot have multiple values after primary variables assigned, assuming at least FC
2. Branched after the primary variables
   - Assuming the solver will let us specify this

OPTIMISATIONS

Now we have the basics, we can start to optimise

1. Simplifiers
2. Variable deletion
3. Domain filtering

SIMPLIFIERS

Every expression type in Savile Row has a simplifier

- A function that takes an expression and returns a smaller, simpler expression

The simplifier performs constant evaluation as an absolute minimum

- If all children of the expression are constant, then the expression must become a constant

Simplifiers are loosely analogous to propagators

- Constant evaluation analogous to checking satisfaction – both evaluate a constraint on constants to true or false

Given expression **E**, simplifiers can

- Examine bounds of the children of **E**
- Compare children of **E** for syntactic equality (e.g. allDiff becomes false if two children are same expression)
- Get full domain of any children that are decision variables

SIMPLIFYING LEX-ORDERING

x+y	p	z	q	p	r

<=lex

x+y	q	z	p	r	q

One example to illustrate simplifiers

SIMPLIFYING LEX-ORDERING

~~x+y~~	p	~~z~~	q	p	r

<=lex

~~x+y~~	q	~~z~~	p	r	q

One example to illustrate simplifiers

Get rid of syntactically equal pairs

SIMPLIFYING LEX-ORDERING

~~x+y~~	p	~~z~~	~~q~~	p	~~r~~

<=lex

~~x+y~~	q	~~z~~	~~p~~	r	~~q~~

One example to illustrate simplifiers

Some pairs must be equal because of pairs to the left – Rule 1 [Frisch & Harvey]

SIMPLIFYING LEX-ORDERING

~~x+y~~	p	~~z~~	~~q~~	~~p~~	~~r~~

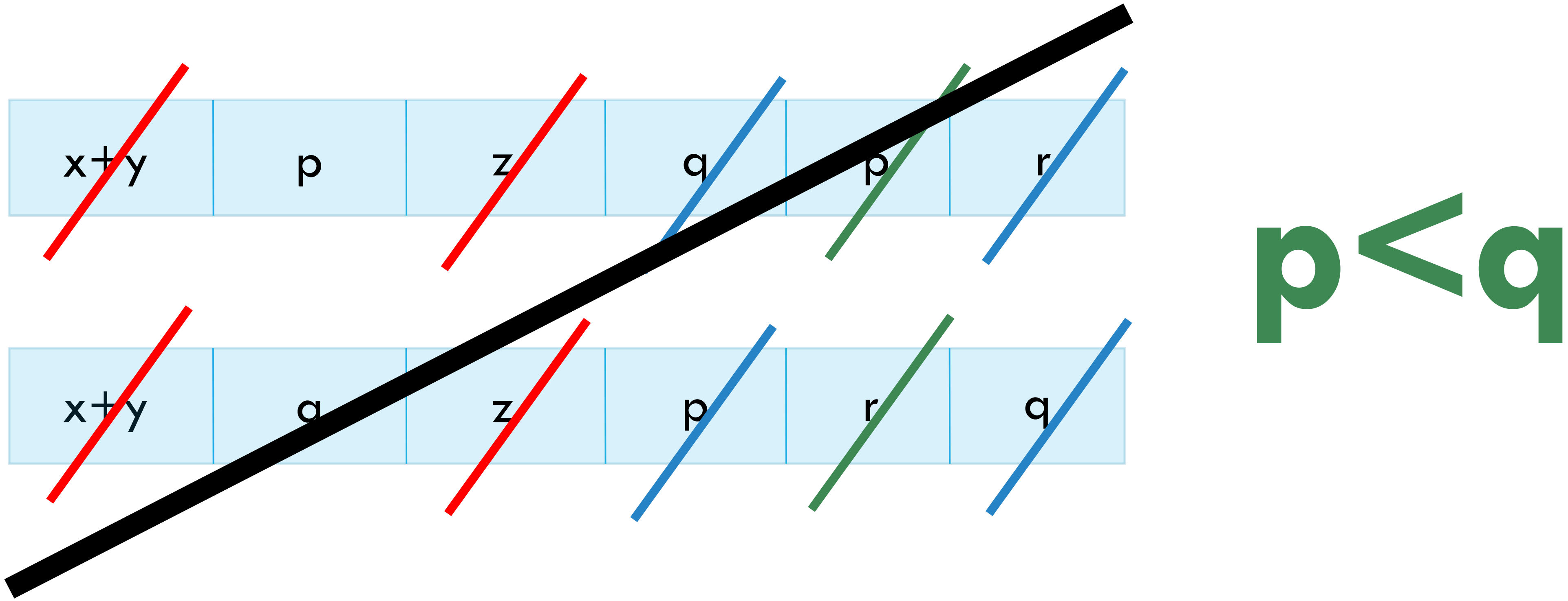
<lex

~~x+y~~	q	~~z~~	~~p~~	~~r~~	~~q~~

One example to illustrate simplifiers

Suppose p : {6..10} and r : {1..5} – pair p,r cannot satisfy constraint

SIMPLIFYING LEX-ORDERING



One example to illustrate simplifiers

Only one pair left – replace with <

SIMPLIFIERS

Savile Row runs simplifiers repeatedly until no further changes are made

- Similar to the propagation loop of a propagating constraint solver

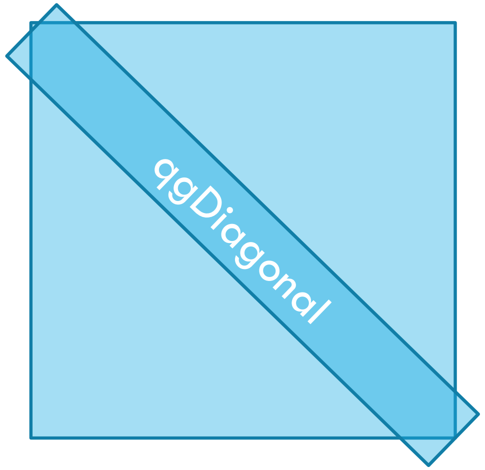
No need for idempotence

From now on, “simplify” means run all simplifiers to exhaustion

VARIABLE DELETION

Suppose someone wrote this:

```
find quasiGroup : matrix indexed by [nDomain, nDomain] of nDomain
find qgDiagonal : matrix indexed by [nDomain] of nDomain
such that
    forAll i : nDomain  .  qgDiagonal[i] = quasiGroup[i,i] ,
    allDiff(qgDiagonal),
    …
```



The allDiff is an implied constraint in the Quasigroup 3 Idempotent problem

VARIABLE DELETION

Need to unify equal variables in this case

- Triggered by =, <-> (iff on Boolean variables) and special type ToVariable
- Intersect the two domains
- Keep the first in the default search order

find quasiGroup : matrix indexed by [nDomain, nDomain] of nDomain

such that

    allDiff([quasiGroup[1,1], quasiGroup[2,2], … , quasiGroup[n,n]]),

    …

(not valid Essence' – actually done after quasiGroup atomised)

VARIABLE DELETION

Replace a variable with a constant

- Triggered by a constraint e.g. x=5 or singleton domain

Unify negated boolean variables

- x <-> !y
- Requires target solver to support negation views (a.k.a. mappers) everywhere – Minion

What else could be done?

With views, could delete any x where x=view(y)

DOMAIN FILTERING

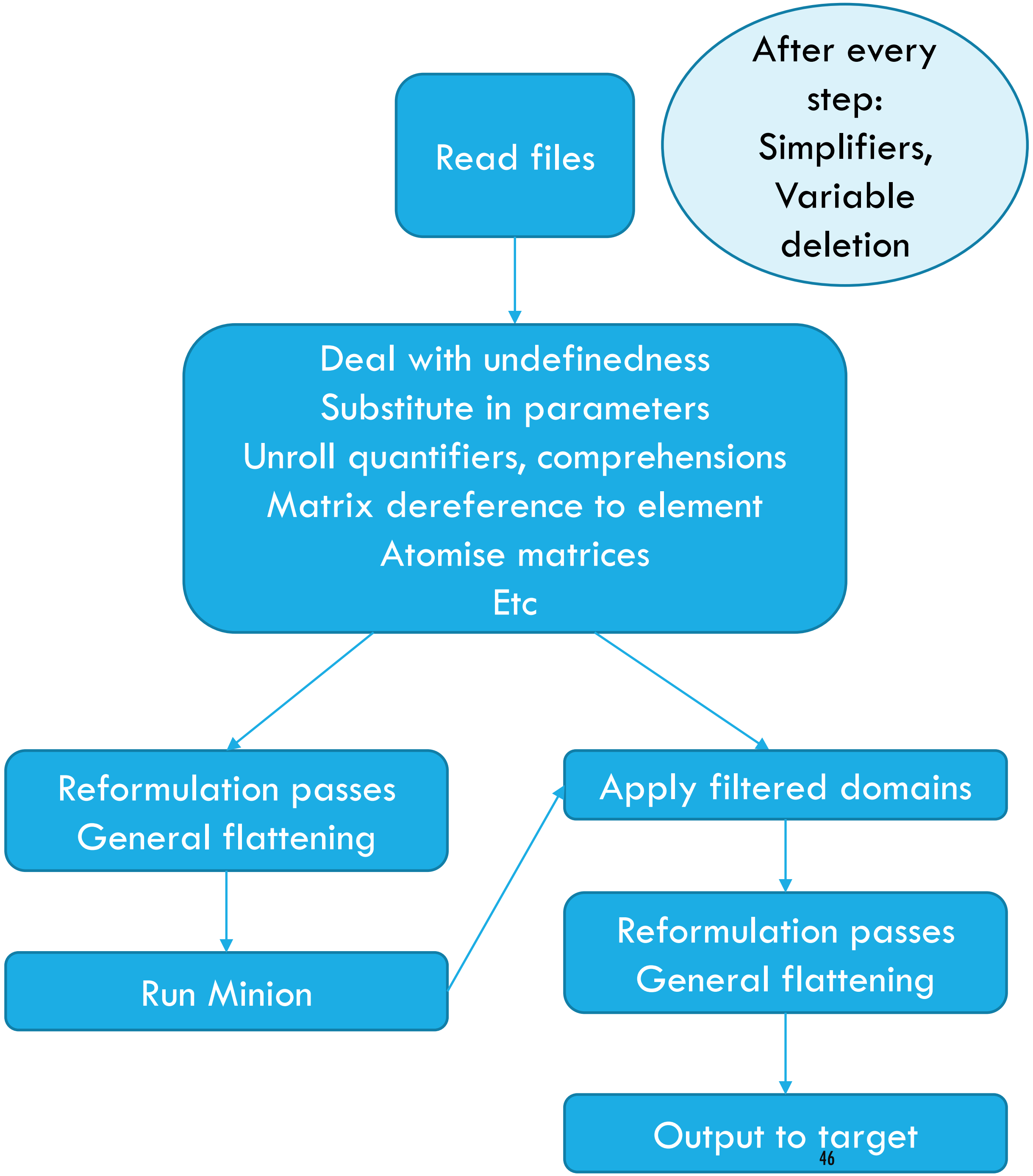
It can be useful to tighten domains of decision variables

- Interacts with variable deletion – can cause variables to be assigned
- Interacts with simplifiers – causes tighter bounds on expressions, constants in place of variables
- Causes tighter bounds for auxiliary variables
- Feeds into optimisation passes we will see later

Savile Row delegates to an external solver – Minion – avoids duplicating functionality

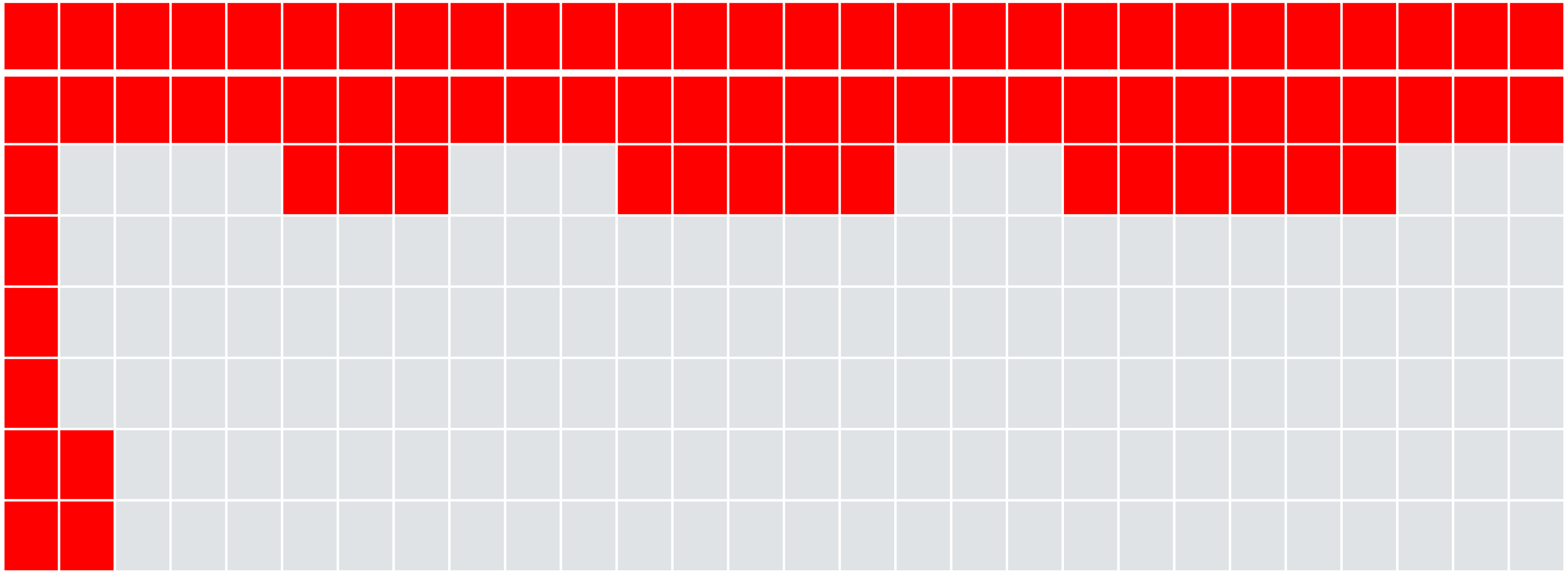
DOMAIN FILTERING

1. Do first part of tailoring process – stops after atomising matrices, before flattening
2. Continue tailoring to Minion
3. Run Minion to perform SAC-Bounds – SAC applied to bounds only
4. Read filtered domains
5. Revert model to the output of step 1.
   Apply filtered domains
6. Continue tailoring to target solver



DOMAIN FILTERING – BIBD

Boolean 2-d matrix – lex-ordering rows and columns – v=8, k=4, l=6



Removes about one-third of the variables – some constraints thrown away, others simplified

OPTIMISATIONS SO FAR

Put the ingredients in, crank the handle…

1. Tighten the formulation of the constraints, remove true constraints
2. Remove redundant variables
3. Tighten variable domains

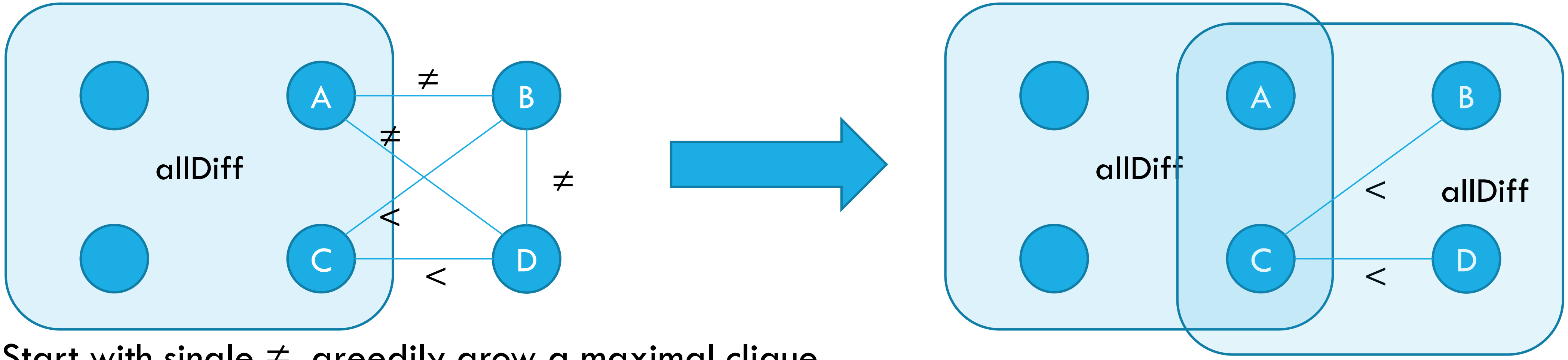
… Some tightening up of the model. No really substantial reformulations yet.

BUT these reformulations lay the groundwork for other more substantial ones

savilerow –O0 –deletevars –reduce-domains (simplifiers always on)

AGGREGATION

AllDifferent created from ≠, <, other allDifferent constraints



Start with single ≠, greedily grow a maximal clique

A, B, C, D may be arbitrary expressions – not just variables

Leads to implied sum constraints – more later

AGGREGATION

Collect AtMost, AtLeast into Global Cardinality Constraint

atmost([w,x,y,z], [2,2,2], [1,2,3]) – at most 2 occurrences each of 1, 2 and 3

atleast([w,x,y,z], [1,1,1], [1,2,3]) – at least 1 occurrence each of 1, 2 and 3

Are collected into:

gcc([w,x,y,z], [1,2,3], [aux1, aux2, aux3]),

New variables aux1…aux3 in {1..2}

Similarly to allDifferent, leads to implied sum constraints

AGGREGATION – GOLOMB RULER

Naïve model:

```
find ruler : matrix indexed by [int(1..ticks)]
        of int(0..ticks**2)
ruler[1]=0,
forAll i : int(2..ticks) .
    ruler[i-1] < ruler[i],
forAll i,j,k,l : TICKS .
    ((i < j) /\ (k < l) /\ ((j>l) \/ (i>k))) ->
    (ruler[j] - ruler[i] != ruler[l] - ruler[k])
```

AGGREGATION – GOLOMB RULER

For ticks=5, Savile Row produces this:

ruler2 in {1..19}, ruler3 in {3..22},

ruler4 in {6..24}, ruler5 in {10..25}, …

allDiff([ruler2, ruler3, ruler4, ruler5, aux8, aux9, aux10, aux11, aux12, aux13])

ruler2<ruler3, …

ruler3-ruler2=aux8, …

Domain filtering, variable deletion, aggregation

GLOBAL CONSTRAINTS TO IMPLIED CONSTRAINTS

Focused on cardinality constraints so far:

allDiff([x,y,z]), where x,y,z in {1..5}

$x+y+z \geq 6$, $x+y+z \leq 12$

GCC([x,y,z], [1,2,3], [a,b,c]), where x,y,z in {1..3} and a,b,c in {0..2}

$x+y+z \geq 4$, $x+y+z \leq 8$, $a+b+c = 3$

Little or no use alone

- $a+b+c=3$ adds propagation for Minion

Overlap with other sums

Feeds into Associative-Commutative CSE – described later

COMMON SUB-EXPRESSION ELIMINATION

Suppose the same expression appears in more than one place – very common

Variables x[1..10] in 0..10

$x[1] + x[2] + \ldots + x[10] < 10$,

$x[1] + x[2] + \ldots + x[10] = 10$

Nothing so far can discover the conflict – 24,301 left-branches Minion, 24,302 fails Gecode

Extract $x[1] + x[2] + \ldots + x[10]$, introduce aux1

$aux1 < 10$, $aux1 = 10$ 0 left-branches Minion, 1 fail Gecode

COMMON SUB-EXPRESSION ELIMINATION

One of the main topics of Andrea Rendl's PhD

Algorithm (implemented in Tailor) has two main parts:

1. Normalisation
   - Sort all commutative operators +, *, =, etc
   - Have only one form of asymmetric comparisons <, ≤, <lex
   - Constant folding/simplifiers applied uniformly
2. Flattening
   - During general flattening, re-use same auxiliary variable for identical expressions
   - Hash table of all expressions flattened so far

IDENTICAL CSE

In Savile Row, Identical CSE is a separate optimisation pass, not an addition to general flattening

1. Normalisation
2. Insert every non-trivial expression into hash table
3. Expressions with two or more occurrences: extract and replace with new aux variable

With some sensible assumptions, Identical CSE cannot worsen propagation

Can obtain tighter bounds on auxiliary variables, improve propagation

IDENTICAL CSE

Let's look at this example again

Variables x[1..10] in 0..10

x[1] + x[2] + … + x[10] < 10,

x[1] + x[2] + … + x[10] = 10

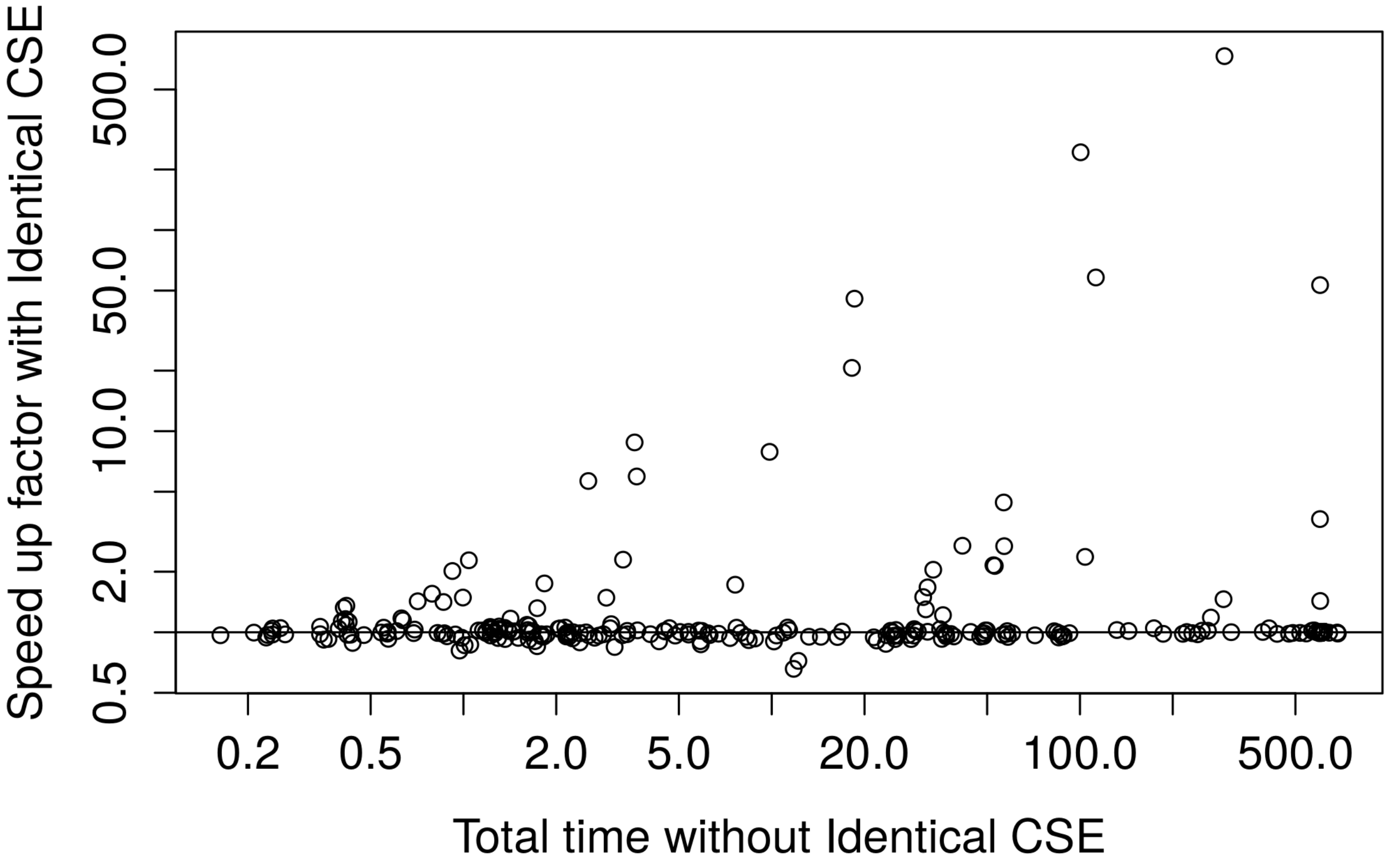
Identical CSE algorithm extracts x[1] + x[2] + … + x[10], introduces aux1

aux1 < 10, aux1 = 10

Tailor's algorithm misses this case – no need to flatten either original constraint

- Separate pass vs part of general flattening

IDENTICAL CSE



41 problem classes, 423 instances

Variable deletion, simplifiers with/without Identical CSE

Three groups:

1. No difference
2. Speeds up propagation by reducing # aux variables – 2-3x faster on Knights Tour
3. Strengthens propagation e.g. on a naïve Golomb Ruler with no allDiff. Hundreds of times faster

COMMON SUB-EXPRESSION ELIMINATION

Interesting results so far

How can we push CSE further?

COMMON SUB-EXPRESSION ELIMINATION

Interesting results so far

How can we push CSE further?

Extend the equivalence to match non-identical expressions

Active CSE [Rendl et al, thesis & SARA 2009]

- Applies a set of transformations

ACTIVE CSE

Algorithm sketch. For a given expression e and transform t that applies to e:

1. Apply t to make e'
2. Simplify and normalise e'
3. Check for occurrences of e' elsewhere in the model
4. e replaced with aux, e' replaced with t(aux) – e.g. !aux for boolean negation

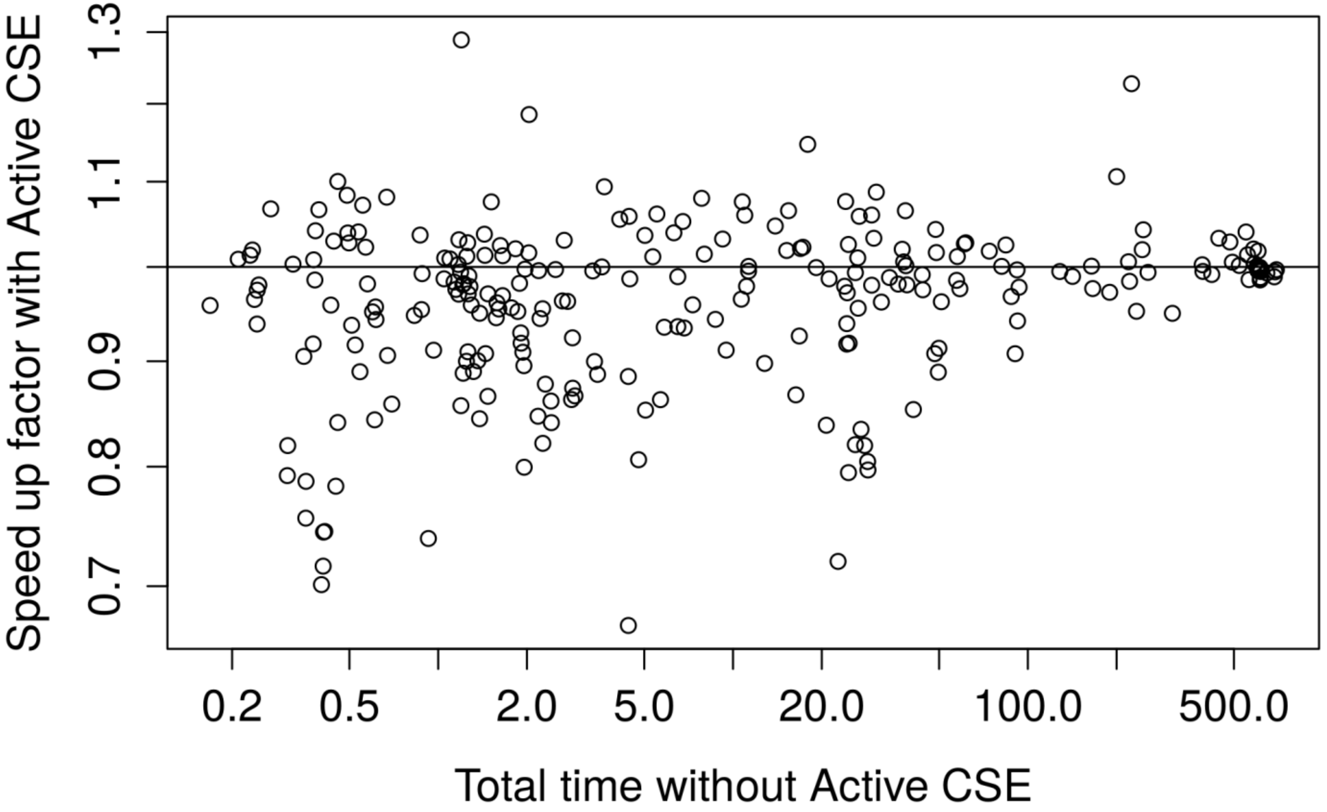
Boolean negation: $(x=y,\ x\neq y)$, $(x<y,\ y\leq x)$, $(a \vee b,\ !a \wedge !b)$

Unary minus: $(x+y+z,\ -x-y-z)$

Multiply by 2 or -2: $(33x-67y,\ 134y-66x)$

Simplifiers are vital – turn $!(x<y)$ into $y\leq x$

ACTIVE CSE



41 problem classes, 423 instances

Active CSE vs Identical CSE

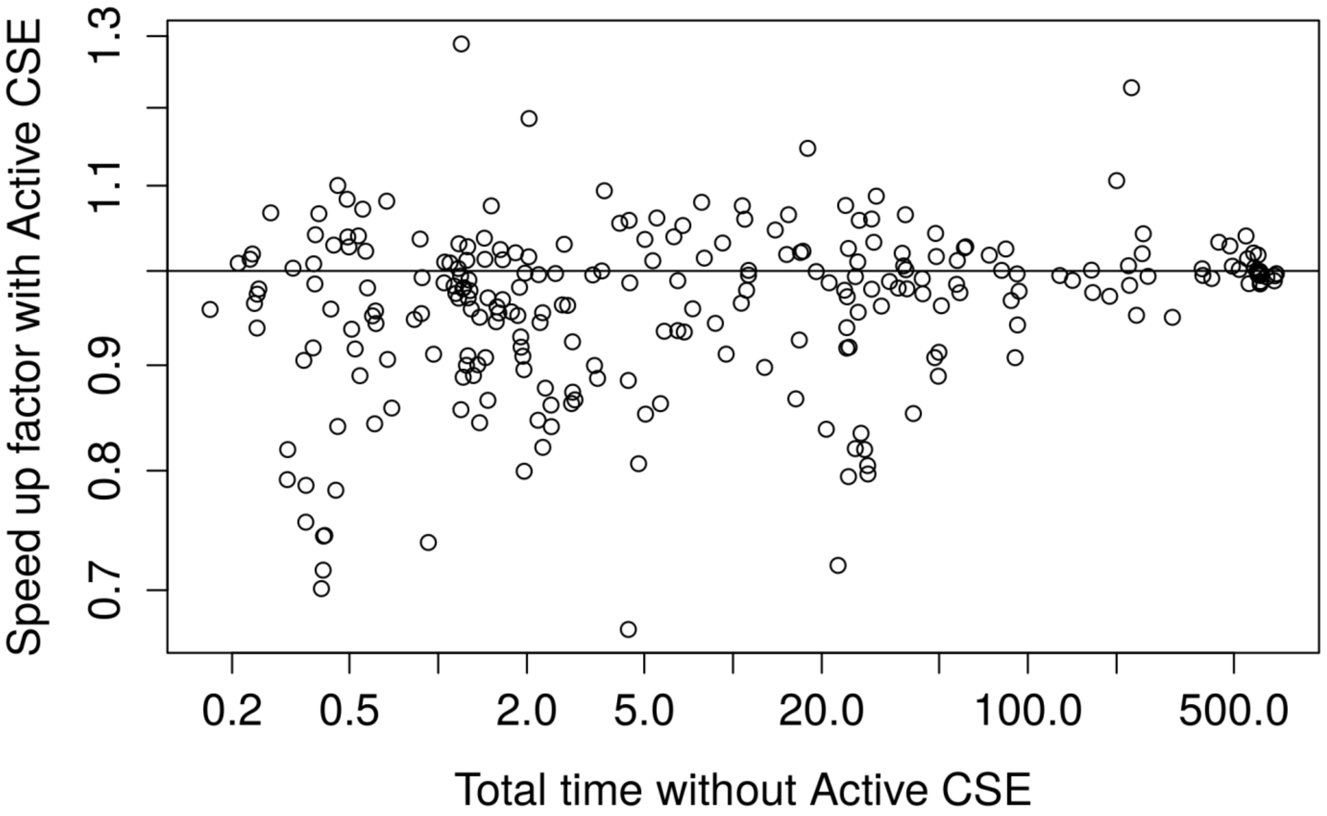
Very rarely a benefit

Best case: pegSolitaireState, 1.8x solver speed-up, same search

3=moves[5] matches 3≠moves[5]

Clearly performing the ‘active’ transformations takes some time

ACTIVE CSE



41 problem classes, 423 instances

Active CSE vs Identical CSE

I suspect hand-written models might explain this

- Cut and paste of expressions
- Simply expressing the same thought in the same way throughout a model

Theoretically more robust than Identical CSE, therefore switched on by default in Savile Row – this may change

COMMON SUB-EXPRESSION ELIMINATION

How can we push CSE further?

Reorder expressions to create identical sub-expressions

Associative-Commutative CSE

- Numerical CSP, interval propagation [Araya, Trombettoni & Neveu, CP 2008]
- Finite domain CSP [Nightingale et al, CP 2014]

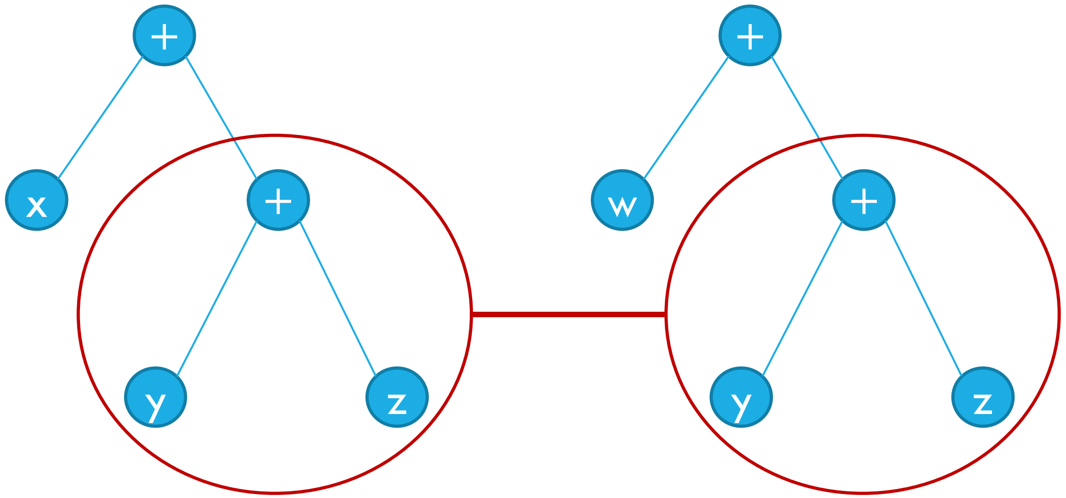
ASSOCIATIVE-COMMUTATIVE CSE

We have already sorted associative-commutative expressions

- x+y+z matches z+x+y with Identical CSE

But we cannot yet match arbitrary overlaps

- A binary tree representation would allow matching prefixes (left-branching) or postfixes (right-branching) in the sorted order… GNU C++ compiler does this
- …but Savile Row represents AC operators using non-binary trees

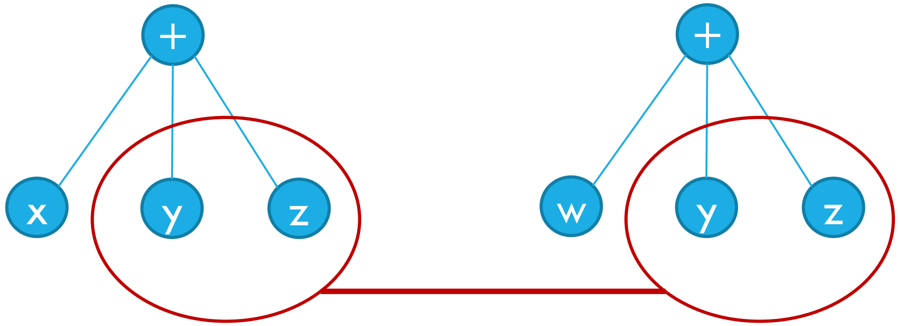


ASSOCIATIVE-COMMUTATIVE CSE

Treat an AC expression as a set of terms

Find a subset common to two or more AC expressions

Extract the common subset everywhere and replace with aux



Can improve propagation dramatically

With some sensible assumptions, never reduces propagation

ASSOCIATIVE-COMMUTATIVE CSE

Example: Knapsack problem

```
given maxWeight : int
given values : matrix indexed by [int(1..numEntries)] of int(1..)
given weights : matrix indexed by [int(1..numEntries)] of int(1..)
find x : matrix indexed by [int(1..numEntries)] of bool
maximising sum i : int(1..numEntries) . x[i]*values[i]
such that
   ( sum i : int(1..numEntries) . x[i]*weights[i] ) <= maxWeight
```

The two sums overlap where values[i]=weights[i]

ASSOCIATIVE-COMMUTATIVE CSE

Conflicting AC-CSs

w+x+y, w+x+z, x+y+z

w+x+y, w+x+z, x+y+z

I-CSE [Araya et al] extracts all AC-CSs between two expressions

- Makes copies of original expressions – potential big slowdown

X-CSE uses heuristic ordering

- Extracts AC-CS with most occurrences first
- Never copies original expressions – can be more efficient in finite-domain context

Genuine choice – difficult to know right answer

ASSOCIATIVE-COMMUTATIVE CSE

X-CSE algorithm implemented in Savile Row

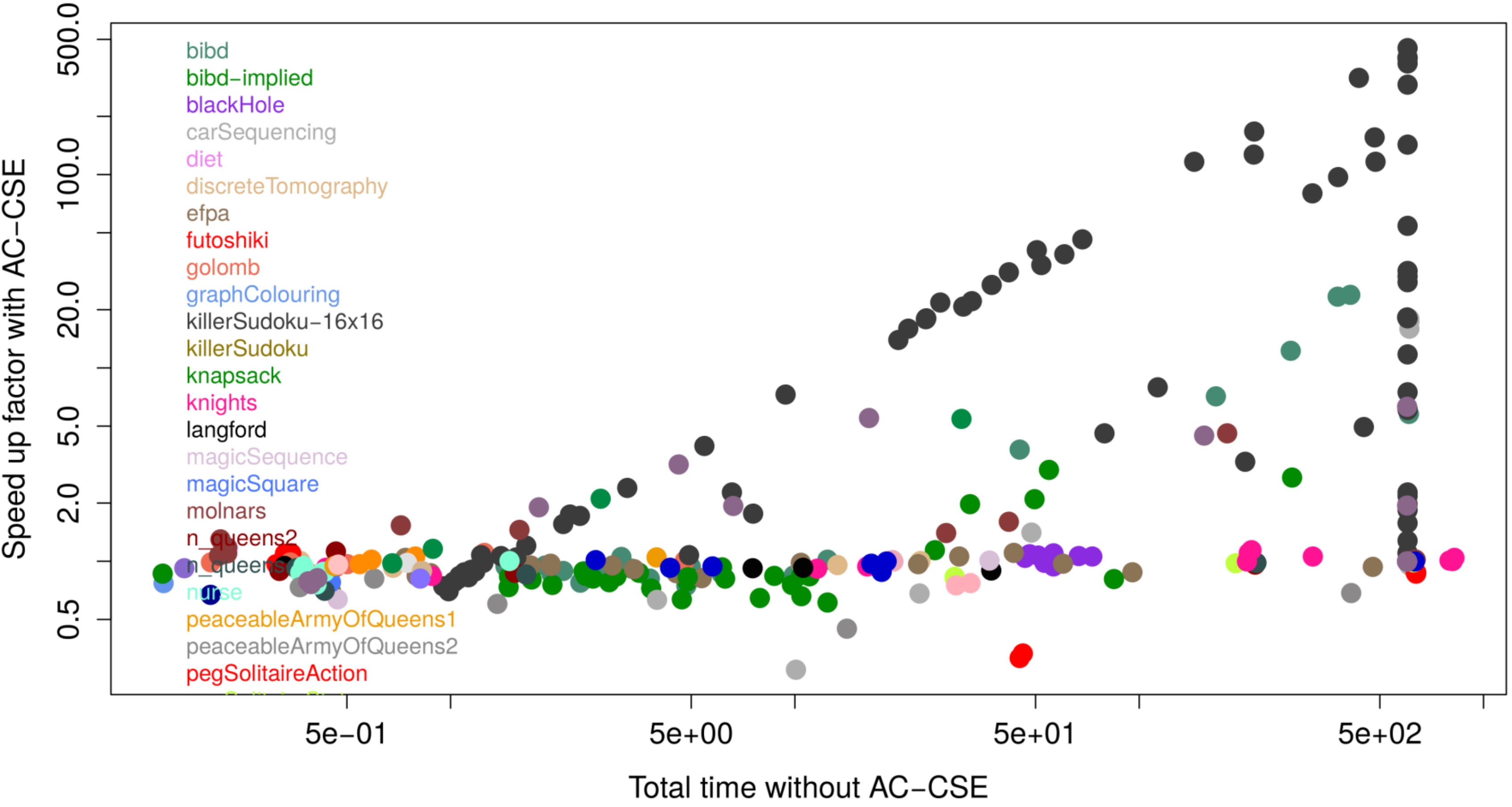
Sum, Product (seen to be useful)

And, Or (apparently not useful – at least on our benchmarks)

Just switch on optimisation level 3 (highest)

savilerow –O3 …

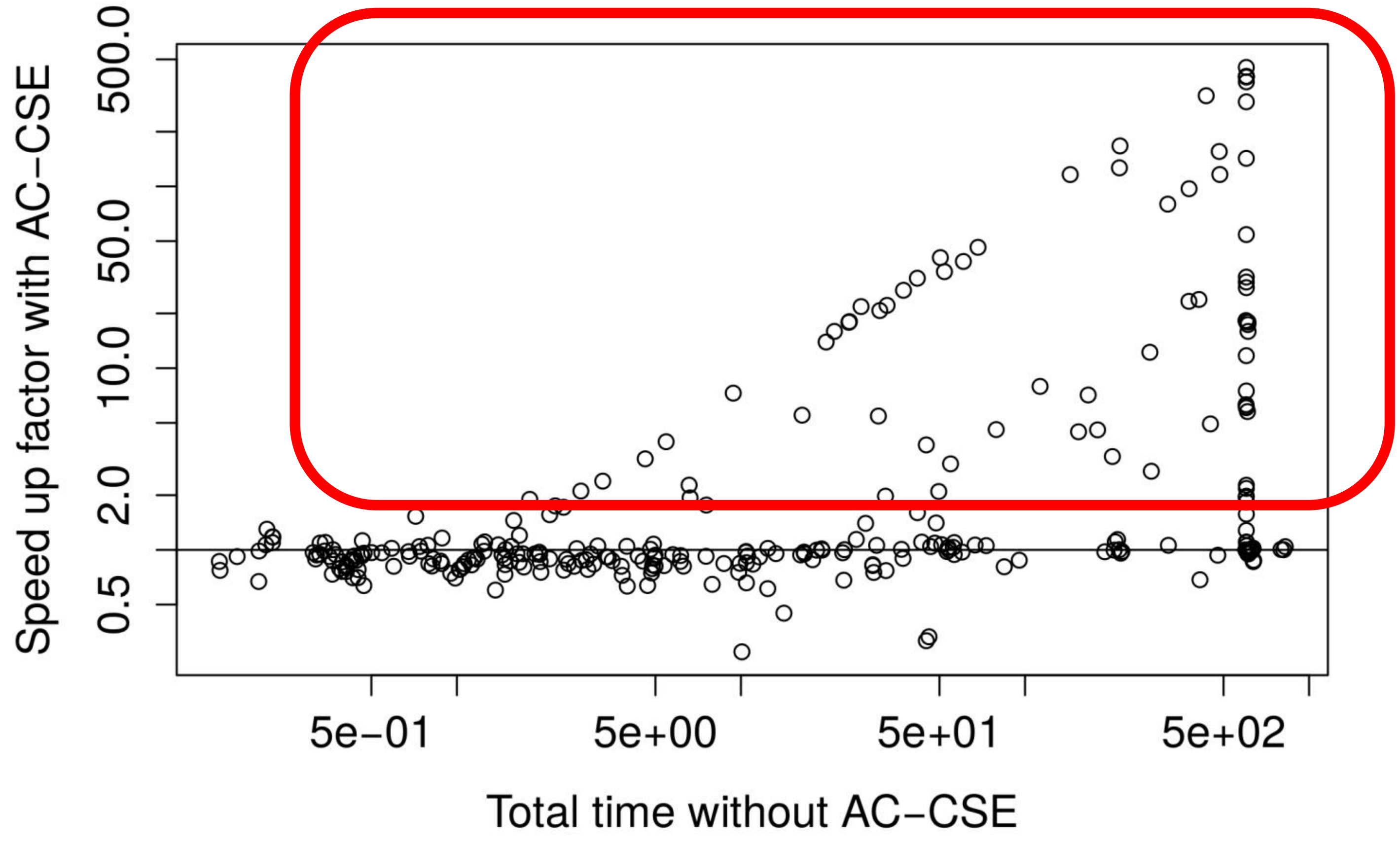
ASSOCIATIVE-COMMUTATIVE CSE



X-CSE+Active CSE vs Active CSE

- Switching on X-CSE switches on implied sum constraints from AllDiff

ASSOCIATIVE-COMMUTATIVE CSE



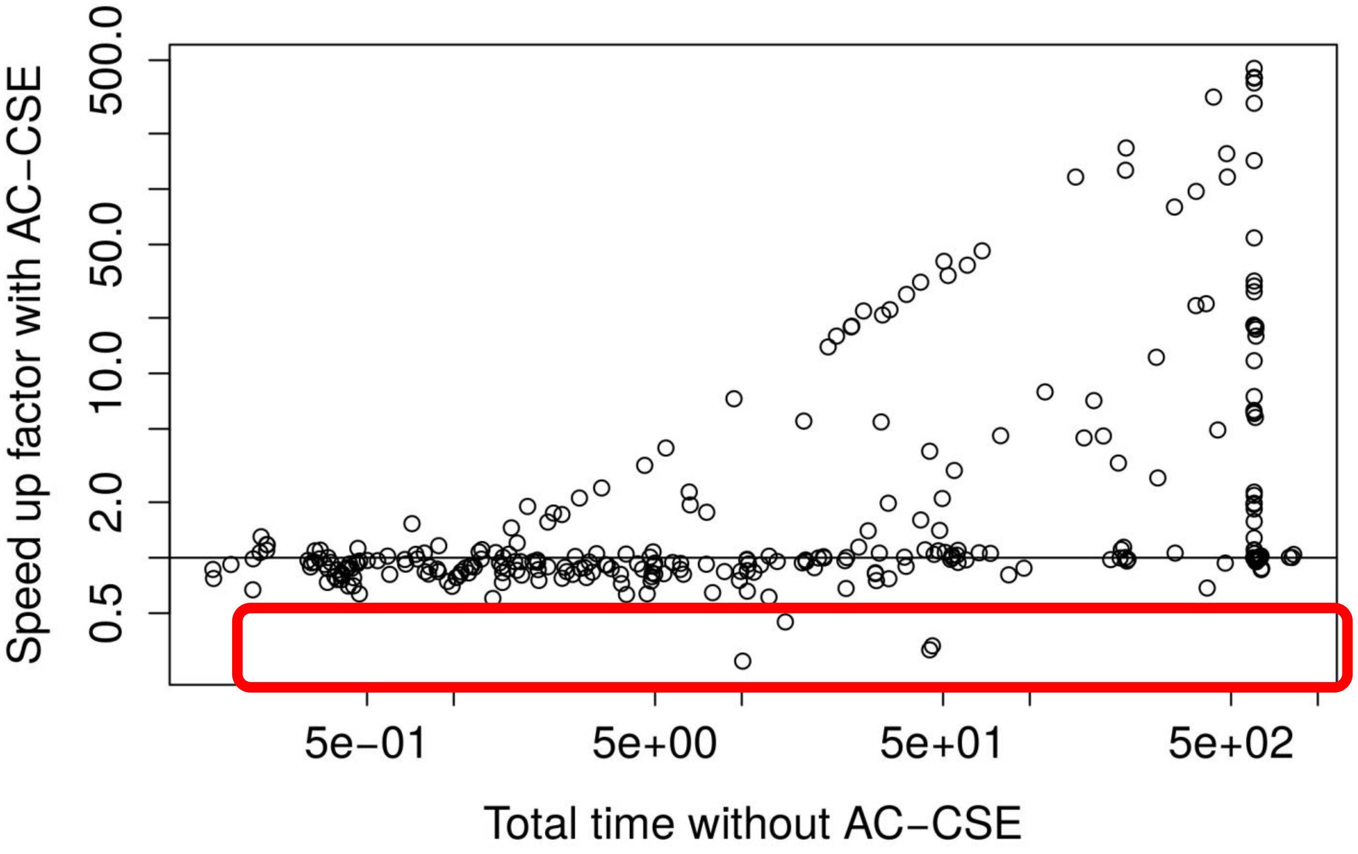
X-CSE+Active CSE vs Active CSE

- Switching on X-CSE switches on implied sum constraints from AllDiff

Winners by 2x:

- Car sequencing (simple model)
- Killer Sudoku – most instances by far
- Waterbucket puzzle
- BIBD
- Molnars problem
- SONET

ASSOCIATIVE-COMMUTATIVE CSE



X-CSE+Active CSE vs Active CSE

- Switching on X-CSE switches on implied sum constraints from AllDiff

Losers (by 2x or more):

- Car sequencing again!
  - One very easy instance: X-CSE takes a long time, saves no search
- PeacableArmyQueens2
- PegSolitaireAction – 2 instances

Never increases search – increases Minion time, Savile Row time

FURTHER INSPIRATION FROM COMPILERS?

Chris Jefferson and I looked through a list of compiler optimisations

Most not relevant – a few are:

Loop-invariant Code Motion – Done, by Identical CSE, after quantifiers unrolled

CSE – Done

Constant folding – Done

Dead-Store elimination – Not done in Savile Row

- Would be equivalent to finding functional variable, removing it and its defining constraint
- Chris Mears mentioned this at ModRef

FORWARD-CHAINING REFORMULATIONS

FORWARD-CHAINING REFORMULATIONS

We have seen several reformulations

These can be useful individually

Much more interesting when one feeds another

Two detailed examples of this happening

BIBD

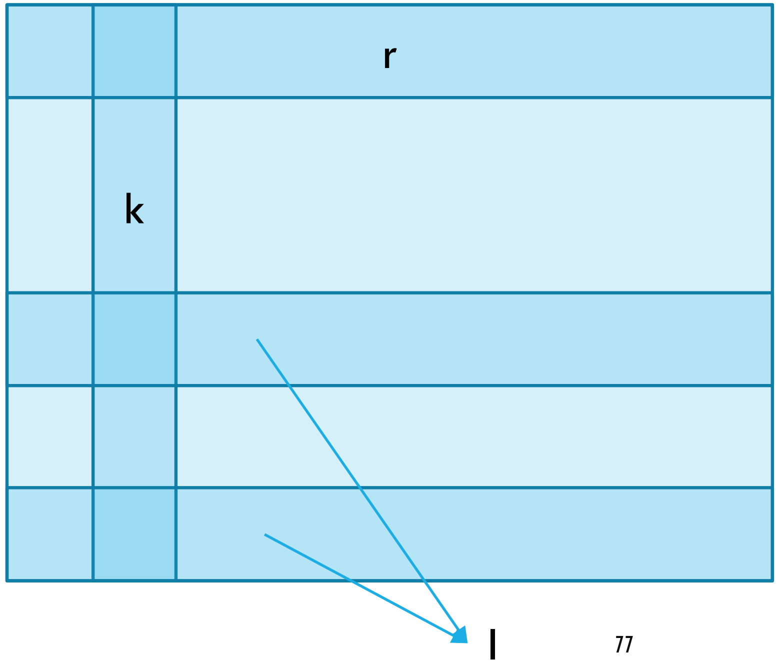
A familiar benchmark

Usually modelled with a matrix of boolean variables:

```
given v, k, l : int
letting b be (l*v*(v-1))/(k*(k-1))
letting r be (l*(v-1))/(k-1)
find bibd: matrix indexed by [int(1..v), int(1..b)] of bool
```

BIBD

```
such that

forAll block : int(1..b) .

    (sum object : int(1..v).  bibd[object, block]) =  k,   $ column sum

forAll object : int(1..v) .

    (sum block : int(1..b). bibd[object, block]) = r,   $ row sum

$ scalar product of two rows is l

forAll object1, object2 : int(1..v) .  (object1 < object2) ->

    ((sum block : int(1..b).

        bibd[object1,block] * bibd[object2, block]) = l),
```



BIBD

```
$ Row and column symmetry breaking
  forAll row: int(1..v-1) .
        bibd[row,..] <=lex bibd[row+1,..],
  forAll col: int(1..b-1) .
       bibd[..,col] <=lex bibd[..,col+1]
```

BIBD

Naïve model (except the symmetry breaking)

Not obvious (to me at least) how this model can be automatically improved

- No CSEs
- Overlap between row sums & scalar products – but not clear how to exploit that

Feel free to interject – how can this model be improved?

BIBD – DOMAIN FILTERING

BIBD v=8, k=4, l=6

The combination of symmetry-breaking and problem constraints causes domain filtering:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0					1	0	0				1	1	1	0	0				1	1	1	0	0	0			
0																											
1																											
1																											
1	1																										
1	1																										

BIBD – SIMPLIFIERS

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0					1	0	0				1	1	1	0	0				1	1	1	0	0	0			
0																											
1																											
1																											
1	1																										
1	1																										

These values are subbed into the relevant constraints

Scalar product 0*bibd[5,2] + … + 1*bibd[5,15] + … + 1*bibd[5,28] = l

Becomes bibd[5,15] + … + bibd[5,28] = l

BIBD – AC-CSE

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0					1	0	0				1	1	1	0	0				1	1	1	0	0	0			
0																											
1							r-1-l													l							
1																											
1	1																										
1	1																										

Scalar product $\quad bibd[5,15] + \ldots + bibd[5,28] = l$

Row sum $\quad bibd[5,2] + \ldots + bibd[5,28] = r-1$

Effectively splits row into two pieces, summing to l and r-1-l

BIBD – AC-CSE

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0					1	0	0				1	1	1	0	0				1	1	1	0	0	0			
0																											
1																											
1																											
1	1																										
1	1																										

Step 1: Splits row into two pieces, summing to l and r-1-l

Step 2: Red parts also sum to l – extract these to aux vars

BIBD – AC-CSE

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1
0					1	0	0				1	1	1	0	0				1	1	1	0	0	0			
0																											
1																											
1																											
1	1																										
1	1																										

Step 1: Splits row into two pieces, summing to l and r-1-l

Step 2: Red parts also sum to l – extract these to aux vars

Step 3: Green parts common to scalar product and row sum – extract these

BIBD – RESULTS

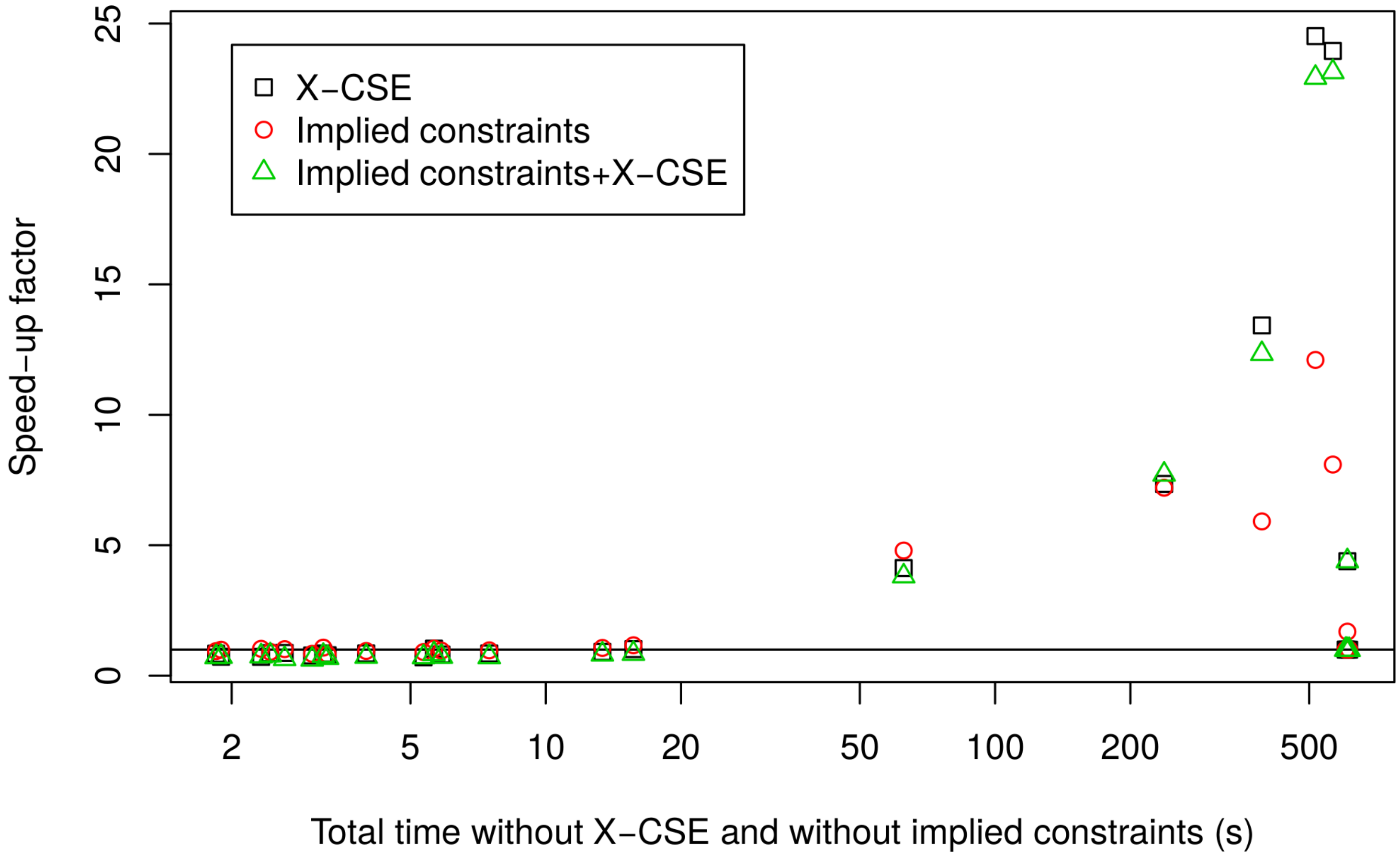
Cross-constraint communication

Each row sum (rows 4-8) joined to 3 scalar product constraints

Convincingly beats naïve model

Sophisticated model introduces implied constraints [Frisch, Jefferson, Miguel, ECAI 04]

Savile Row beats sophisticated model, and can improve it.



BIBD – SUMMARY

We need to do these steps in order:

1. Add lex-ordering constraints (manual)
2. SAC-Bounds domain filtering
3. Sub in assigned values and simplify sum of products
4. AC-CSE

KILLER SUDOKU

9x9 grids too easy, we did 16x16

16x16 matrix, each cell takes value in {1..16}

Rows, columns and 4x4 subsquares: allDifferent

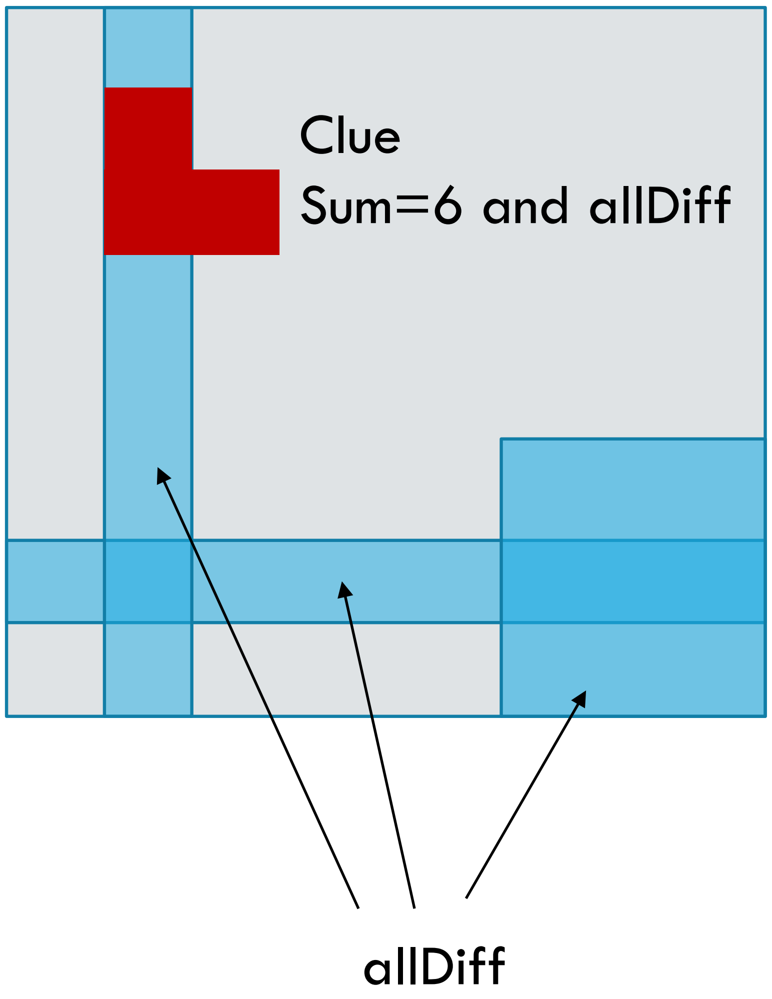
Clues are contiguous sets of cells

- The sum is given as part of the clue
- Cells within the clue are allDifferent

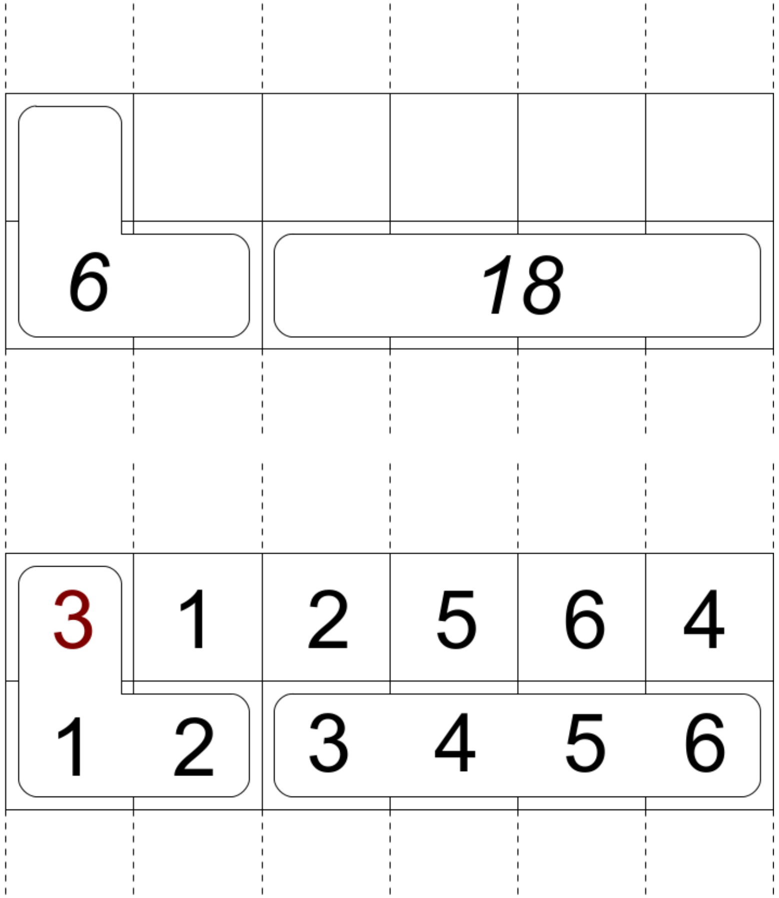
In the example (right) the clue must contain values 1,2,3

Entire matrix covered by non-overlapping clues

Model is exactly the above constraints



KILLER SUDOKU



Rows/columns/subsquares are a permutation

Introduce sum constraints from AllDifferent

For each row, column and subsquare X:

sum(X) = 136 (for 16x16 case)

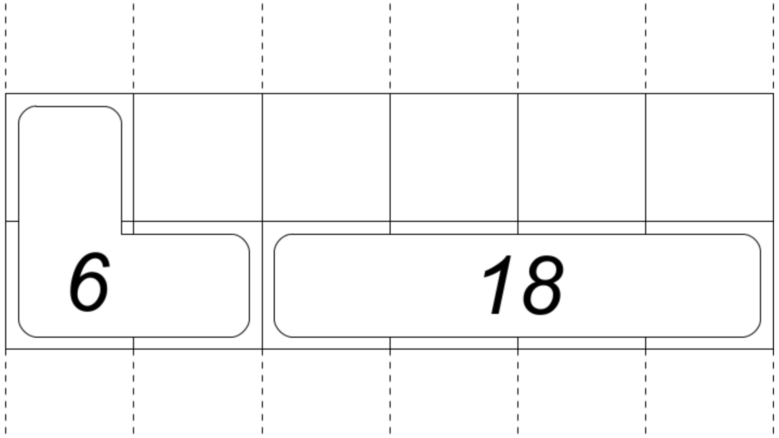
Suppose we had 6x6 Killer Sudoku (left)

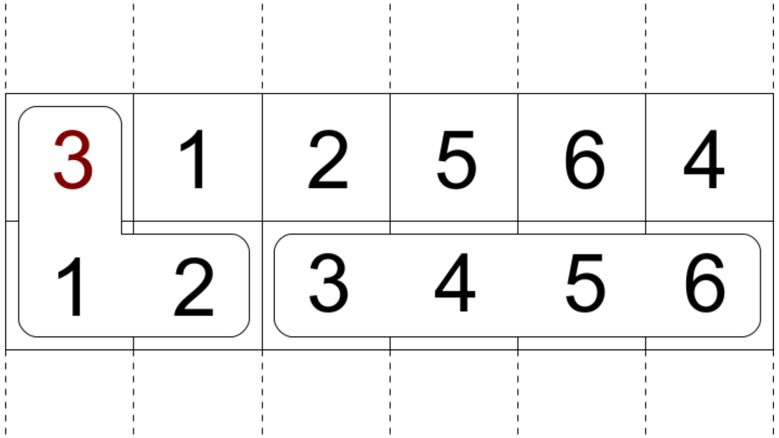
sum(X) = 21

For each clue, we also get useless sum≤a and sum≥b

- Removed by Identical CSE followed by simplifiers

KILLER SUDOKU





New sums on rows/columns/subsquares intersect with clues

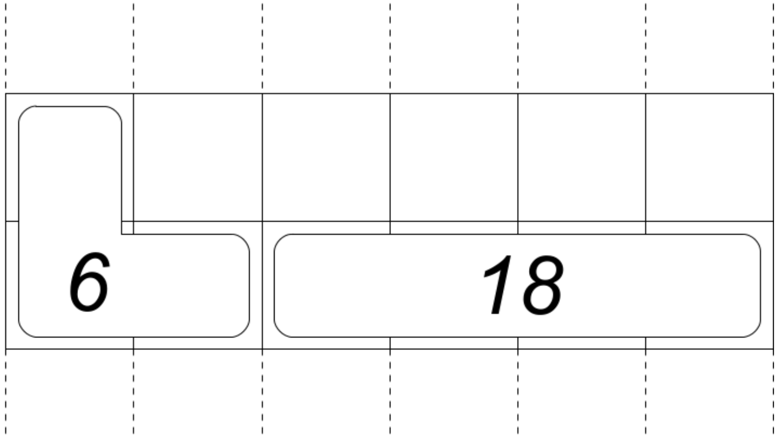
In example (left), suppose two rows are k[1,..] and k[2,..]

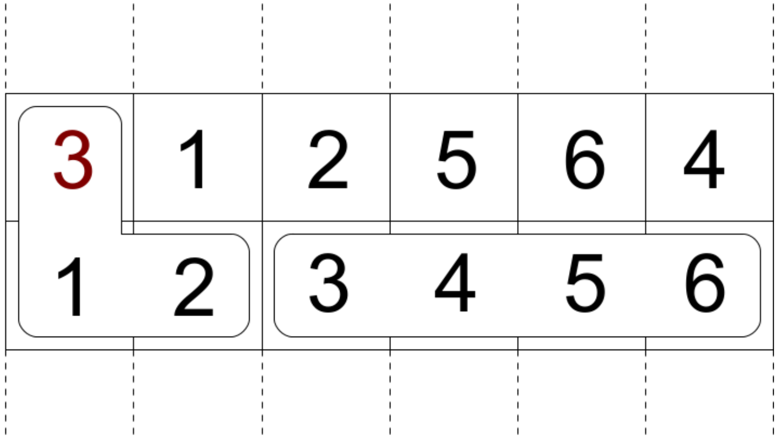
AC-CSE connects clues to rows

k[2,3] + … + k[2,6] is common to the 18 clue and the row sum

k[2,1] + k[2,2] is common to the 6 clue and the row sum

KILLER SUDOKU





k[2,3] + … + k[2,6] = aux1

k[2,1]+k[2,2] = aux2

aux1=18, aux2+k[1,1]=6

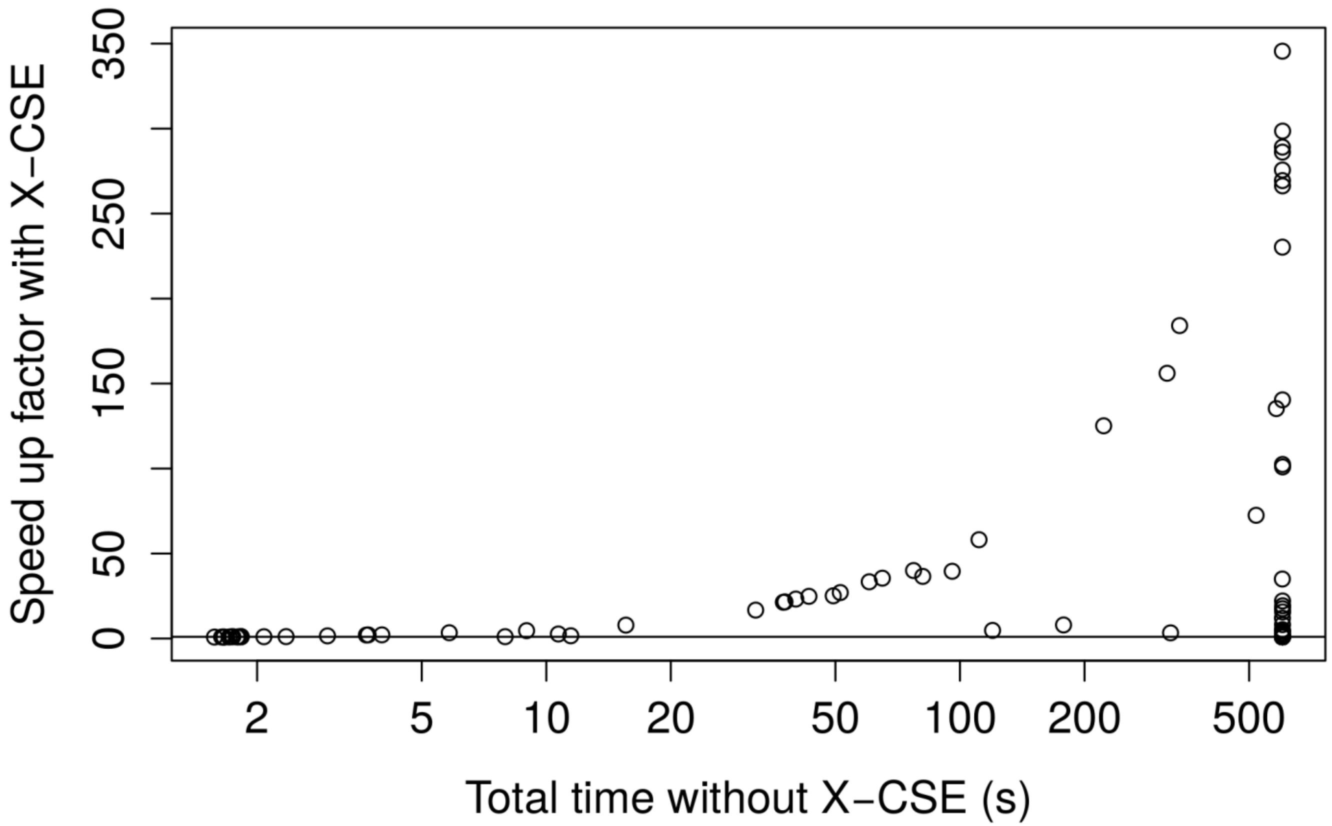
aux1+aux2=21

aux1 replaced with 18 (variable deletion)

aux2 becomes 3 (simplifier, then var deletion)

k[1,1] becomes 3 (simplifier, then var deletion)

KILLER SUDOKU – RESULTS



Some hard problems made almost trivial

Peak instance:

Without X-CSE Savile Row took 2.26s

Minion timed out at 600s

2,774,028 nodes

With X-CSE Savile Row took 1.62s

Minion took 0.13s, 2 nodes

savilerow –O3 killer.eprime …

KILLER SUDOKU – SUMMARY

We need to do these steps in order:

1. Add implied sum to all AllDifferent constraints
2. Apply AC-CSE
3. Variable deletion (interleaved with simplifiers)

IMMINENT

SAT encoding

The first iteration, but good enough to beat static variable ordering sometimes

- Thanks to student Patrick Spracklen

Automatic variable symmetry breaking

Calls a graph automorphism solver then adds lex-ordering

- Thanks to student Saad Atiher and Chris Jefferson

Both need a little more testing – should appear very soon

CONCLUSIONS

I hope I have convinced you that reformulations are an interesting research topic

Most interesting when one reformulation feeds valuable input into another

Try Savile Row for yourself:

http://savilerow.cs.st-andrews.ac.uk/

THE OTHER TUTORIAL – CSPLIB

Completely re-written website

Editor-in-chief: Chris Jefferson

Website maintainer: Bilal Hussain

http://csplib.org/

Contribute new problems here on Github:

http//github.com/csplib/csplib/