Satisfiability and Constraints

Ian Gent
University of St Andrews

Getting my thanks in early ...

"A project of this scope and importance could not be achieved without the aid and assistance of many people......

"... or rather it could but it would be dumb to do it that way when there are so many people around willing to give their aid."

- Peter Schickele

- So apart from Pierre, Justin and Christian for inviting me
    - and **you** for coming
    - I want to thank ...

Thanks to ...

- Dharini Balasubramaniam
- Jim Caldwell
- Dave Clark
- Tony Cohn
- Michelle Cope
- Joe Culberson
- Lakshitha de Silva
- Jeremy Frank
- Enrico Giunchiglia
- Pete Gregory
- Kevin Hammond
- Warwick Harvey
- Holger Hoos
- Sophie Huczynska
- Rob Irving
- Chris Jefferson
- Bill Johnston
- Tom Kelsey
- Lars Kotthoff
- Steve Linton
- Inês Lynce
- Ewan MacIntyre
- David Manlove
- Iain McDonald
- Paul McKay
- Ian Miguel
- Neil Moore
- Massimo Narizzano
- Peter Nightingale
- Justin Pearson
- Karen Petrie
- Patrick Prosser
- Andrea Rendl
- Colva Roney-Dougal
- Andrew Rowley
- Josh Singer
- Alan Smaill
- Barbara Smith
- Kevin Smyth
- Kostas Stergiou
- Armando Tacchella
- Armagan Tarim
- Neven Tomov
- Judith Underwood
- Toby Walsh
- Wu Wei

Special thanks to...

- Chris Jefferson
- Neil Moore
- Peter Stuckey
- for letting me recycle their materials

Topics in this Series

- Why SAT & Constraints?
- SAT basics
- Constraints basics
- Encodings between SAT and Constraints
- Watched Literals in SAT and Constraints
- Learning in SAT and Constraints
- Lazy Clause Generation + SAT Modulo Theories

Topics in this Series

- Why SAT & Constraints?
- SAT basics
- Constraints basics
- Encodings between SAT and Constraints
- Watched Literals in SAT and Constraints
- Learning in SAT and Constraints
- Lazy Clause Generation + SAT Modulo Theories

Why SAT & Constraints?

- Seems pretty simple
- SAT does some things well
- Constraints does other things well
- Let’s get the best of both worlds
- But one tiny problem ....

One tiny problem...

- SAT and Constraints are

  THE SAME THING

The SAME THING?

- How can we have talks on SAT + CP hybridisation
- If they are the same thing?
- And are they the same thing?

The Same Thing?

- We'll see definitions in a bit which show ...
- SAT is just a **special case** of Constraints
- It's easy to encode Constraints to SAT
    - Possibly at some cost to be explained
    - But often at no cost
- Really there should be no difference, right?
- Let's start again ...

SAT & Constraints Hybrids

- 6 hours of lectures about how they are almost but not quite the same thing
- Watch in excitement as we discuss minor differences in optimisation choices
- And then shout at Pierre, Justin & Christian for inviting me

What’s going on?

- Of course previous slide is not true
  - I hope the bit about shouting at organisers is not true, I know the other bits are false
- Best analogy to me is *electroweak* theory in physics

Electroweak Interaction

- *In particle physics, the electroweak interaction is the unified description of two ... fundamental interactions of nature: electromagnetism and the weak interaction. Although these two forces appear very different at everyday low energies, the theory models them as two different aspects of the same force.*

Wikipedia

SAT & Constraints

- *In AI search, the constraint satisfaction problem is the unified description of two ... fundamental problems of search: boolean satisfiability and the constraint problem. Although these two problems appear very different in everyday examples, the theory models them as two different aspects of the same problem.*

Me

SAT & Constraints

- SAT is a special case of Constraints
- BUT ...
  - constraint solvers not great at SAT
    - because they are not engineered only to be SAT solvers
    - and doing so would make them bad at other constraint problems

The Real Story

- Constraint solvers are brilliant at
  - propagating complicated constraints very fast
- SAT solvers are brilliant at
  - propagating one very simple constraint even faster

Constraint Solvers

- Propagate complicated constraints like
  - all-different, global cardinality, sequence, element, table, ...
  - do so using very smart algorithms
  - and very smart implementations of them
- Constraints represented *implicitly*
  - *i.e. not* a list of tuples
- Constraints often *tight*
  - e.g. in all-different only about $1/e^n$ of tuples are allowed

SAT

- Propagates exactly one constraint
  - the *clause*
  - with very smart propagation algorithm
- Constraints represented explicitly
  - as list of literals in the clause
- Constraint is very loose

Constraints & SAT

- Any constraint problem can be encoded in SAT
  - BUT
    - it might not propagate as well
    - it might use a lot of space
    - or both
    - could be prohibitively expensive

Constraints and SAT: the real story

- Constraints and SAT are two sides of the same coin
- So if we get a great idea in one we want to exploit it in the other
- This has often happened
  - both ways
  - I'll talk about some of them
- But we can't do this simplistically
  - Lots of work to get great ideas across

Topics in this Series

- Why SAT & Constraints?
- SAT basics
- Constraints basics
- Encodings between SAT and Constraints
- Watched Literals in SAT and Constraints
- Learning in SAT and Constraints
- Lazy Clause Generation + SAT Modulo Theories

Satisfiability

- What it is
- DPLL
  - Davis Putnam Logemann Loveland
  - Unit Propagation
- Why bother?

SAT solving in 1869

- Not a typo - 1869
- Logic Piano by William Stanley Jevons
- First automated reasoning machine in history?
- Up to 4 boolean variables
- Using truth tables

image wikipedia

Incredibly important historical problem

- Cook’s theorem (1971) says
  - SAT is NP-complete
- One of the most important results in theoretical computer science
- Key amazing result is that ...
  - NP-completeness exists
- SAT a good choice as basis
  - circuits encode naturally into it

Boolean SATisfiability

- I'll restrict attention to SAT in *clause form*
- We have a set of boolean variables ***V***
  - can take values true/false (or 1/0)
- A set of clauses **C**
  - each contains a set of literals
    - literal is a negated or non-negated variable
- Seek an assignment of values 0/1 to **V**
  - such that every clause
  - contains a negated literal -*x* where *x assigned to 0*
  - *OR* a non-negated literal *x* where *x assigned to 1*

Davis-Putnam

- The best complete algorithm for SAT is Davis-Putnam
  - first work by Davis-Putnam 1961
  - current version by Davis-Logemann-Loveland 1962
- variously called DP/DLL/DPLL or just Davis-Putnam
- I will present a slight variant omitting "Pure literal" rule
- A recursive algorithm
  - Two stopping cases
    - an empty set of clauses is trivially satisfiable
    - an empty clause is trivially unsatisfiable
      - there is no way to satisfy the clause

My story about Bob Dylan

- And how St Andrews awarded an honorary degree to ...
- A key figure from the 1960s
- Whose work was inspirational
- And has affected my life in very deep ways
- And I was furious I missed the graduation

My Story about Hilary Putnam

My story about Don Loveland

Algorithm DPLL(clauses)

- 1. If clauses is empty clause set, *Succeed*
- 2. If clauses contains an empty clause, *Fail*
- 3. If clauses contains a unit clause (*literal*)
  - return result of DPLL(clauses[literal])
    - *clauses[literal] means unit propagate clauses with value of literal*
- 4. Else *heuristically* choose a variable *u*
  - *heuristically* choose a value *v*
  - 4.a. If DPLL(clauses[u:=v]) succeeds, *Succeed*
  - 4.b. Else return result of DPLL(clauses[u:= not v])

unit propagation

- *clauses[literal] means unit propagate clauses with value of literal*

- What does this mean?
- When assigning *x = 1*
- For every clause in the problem
  - if the clause contains *x,* delete the clause
    - because it is guaranteed satisfied
  - if the clause contains -*x,* delete the literal from the clause
    - because it cannot satisfy the clause
- If this results in any **unit clause** *y (or -y)*
  - *i.e. clause containing only the literal y (or -y)*
  - then assign *y* to 1 (or 0) and repeat

unit propagation

- *clauses[literal] means unit propagate clauses with value of literal*

- What does this mean?
- When assigning *x = 0*
- For every clause in the problem
  - if the clause contains -*x*, delete the clause
    - because it is guaranteed satisfied
  - if the clause contains *x*, delete the literal from the clause
    - because it cannot satisfy the clause
- If this results in any **unit clause** *y (or -y)*
  - *i.e. clause containing only the literal y (or -y)*
  - then assign *y* to 1 (or 0) and repeat

unit propagation

- *clauses[literal] means unit propagate clauses with value of literal*

- **Simple Theorem**
  - After unit propagation, solutions of *clauses[-x]* are exactly the solutions of *clauses* where *x=0*
  - And similarly for *clauses[x]* and *x=1*
  - And there are no unit clauses in *clauses [literal]*

Complexity of unit propagation

- Naive algorithm for unit propagation of x
  - Iterate through clauses
    - If the clause contains *x*, delete it
    - If the clause contains *-x*, delete *-x*
  - If any clause is unit
    - Pick one, assign variable, GOTO start

Naive = very bad indeed

- Not always but in this case
- I have implemented this algorithm
  - ... and published papers with it
  - ... which shows how much I know [knew]
- Problem?

Naive = very bad indeed

- Problem?
- If there are *n* variables, *m* clauses, each containing *k* literals
  - This takes O(*nmk*) time
    - O(*mk*) per assignment, up to *n* of them
- Shouldn’t really take more than O(*mk*) time total
  - Which is as good as we can possibly do
  - Though it’s important to do much better!

O(*mk*) unit propagation

- One way to get O(mk) unit propagation
- Index each occurrence of each literal
  - e.g. doubly linked list per literal
  - Adds O(1) only per literal, no problem
- Data clause needs is...
  - Doubly linked list of literals in it
  - Again adds O(1) per literal

O(*mk*) unit propagation

- To unit propagate assignment of *x = 0*
- For each element of list of literals of *x*
  - Unstitch *x* from the clause it is in
  - If clause unit, add assignment to the queue of unit clauses
- For each element of list of literals of -*x*
  - Delete the clause it is in
  - Unstitch other literals in the clause from their lists
- If queue of unit clauses not empty, assign next element

O(*mk*) unit propagation

- To unit propagate assignment of *x* = *1*
- For each element of list of literals of -*x*
  - Unstitch *x* from the clause it is in
  - If clause unit, add assignment to the queue of unit clauses
- For each element of list of literals of *x*
  - Delete the clause it is in
  - Unstitch other literals in the clause from their lists
- If queue of unit clauses not empty, assign next element

O(*mk*) unit propagation

- Why is this O(*mk*) ?

O(*mk*) unit propagation

- Why is this O(*mk*) ?
- Because it touches each literal at most once
  - every time a literal is looked at
  - it is deleted from the clause
  - or the clause is deleted
  - and either way it is never touched again
- And there are O(*mk*) of them

O(*mk*) unit propagation

- Why is this better than O(*mk*) - sort of ?

O(*mk*) unit propagation

- Why is this better than O(*mk*) - sort of ?
- Because we might not touch all literals
  - or even a large percentage
- We only touch literals which are either
  - negated or positive version of assignment
  - in a clause which is deleted
- In extreme case this can be O(1)

Backtracking

- We will search millions/billions of nodes
- It’s vital to be able to backtrack efficiently
- Last algorithm has this property
  - Put each d-l-l change onto stack
  - On backtracking restitch them via dancing links
  - O(1) work per literal change
    - so in fact no extra work in big-O terms

But wait there’s more

- We’ll see later on that we can do better still!
- And not even touch every literal that is assigned
  - which is almost magical!
- This is watched literals
  - and one of the areas of fruitful transfer SAT to CP

Why bother?

- SAT is becoming more and more important
- Especially since late 90s/early 00s revolution in solving speed
- Millions of clauses is no inhibition
- Best application area is Computer Aided Verification
- In 2009 ...

CAV Award 2009

- *"The 2009 CAV (Computer-Aided Verification) award was presented to seven individuals who made major advances in creating high-performance Boolean satisfiability solvers. This annual award recognizes a specific fundamental contribution or series of outstanding contributions to the CAV field."*

2009 CAV award announcement

CAV Award 2009

– *Conor F. Madigan, Kateeva, Inc.*

– *Sharad Malik, Princeton University*

– *João P. Marques-Silva, University College Dublin, Ireland*

– *Matthew W. Moskewicz, University of California, Berkeley*

– *Karem A. Sakallah, University of Michigan*

– *Lintao Zhang, Microsoft Research*

–*Ying Zhao, Wuxi Capital Grou*

Topics in this Series

- Why SAT & Constraints?
- SAT basics
- Constraints basics
- Encodings between SAT and Constraints
- Watched Literals in SAT and Constraints
- Learning in SAT and Constraints
- Lazy Clause Generation + SAT Modulo Theories

Constraints Basics

- Definitions
- Basic Theory of Propagation
- Example (Element constraint)
- Practice
- Differences between Theory and Practice

History of Constraints

- Less long lasting and illustrious than SAT
- Oh, wait a minute ...

Constraint solving in 1869

- Not a typo - 1869
- Logic Piano by William Stanley Jevons
- First automated reasoning machine in history?
- Up to 4 boolean variables
- Using truth tables

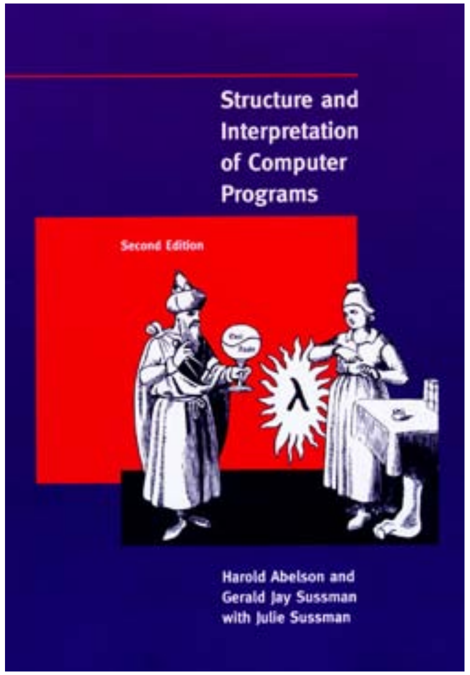
image wikipedia

History

- Ok, not really so illustrious except by stealing SAT’s history
- Goes back to 60s slightly, 70s definitely
    - e.g. classic paper by Stallmann & Sussmann, 1976

History

- Ok, not really so illustrious except by stealing SAT's history
- Goes back to 60s slightly, 70s definitely
  - e.g. classic paper by Stallmann & Sussmann, 1976
  - Yes, *that* Sussmann
  - And *that* Stallmann



Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and Gerald Jay Sussman with Julie Sussman

MIT Press, creative commons

GNU

Definitions

- A constraint satisfaction problem contains..
  - V, a set of variables
  - each with domain D, a set of integers
  - C, a set of constraints
- What is a constraint?

History

- Key work in 70s and 80s
    - Consistency in Networks of Relations, Mackworth, 77
    - Backtrack free search, Freuder, 82
- But didn’t really come into its own until the 90s
    - development of powerful commercial solvers
    - and powerful propagation algorithms

Constraint

- A constraint acts on a subset of V
- The constraint *defines*
  - For each tuple $<x_1,x_2,...,x_n>$
  - with each x_i in current domain of v_i
  - whether the tuple is *allowed* or not

Solution

- Solution to a constraint satisfaction is ...
- An assignment of a value to each variable
  - from its current domain
- such that
  - the tuple thus defined for each constraint
    - is allowed

The usual stuff...

- Every paper starts with this stuff
  - unless we’ve run out of space ...
- But I’ve added one variant
  - “The constraint *defines*”
  - **not** “The constraint *is*”
- Because we only rarely list the tuples

Constraints and SAT

- Nowhere have I said that ...
  - constraints have a limited arity
  - variables can't be boolean
- Which means we can have
  - a 0/1 variable for each SAT variable
  - A *k-ary* constraint for each *k*-clause
  - With the set of allowed tuples being
    - all but the single tuple disallowed by the clause
  - But it does not need to be stored in this hideous way
- So yes, SAT is just a special case of Constraints

SAT and Constraints

- We can encode Constraints into SAT
- e.g. have one SAT variable for each value of each variable in the Constraint problem
    - some clauses to say exactly one value is allowed
- Have one clause for each disallowed tuple in each constraint
- Bingo, any solution to SAT problem is solution to Constraint problem

No big surprise

- Whole point of NP-completeness is encoding between problems
- But these are really close encodings...
  - SAT just is a Constraint Problem
  - Encoding into SAT is very natural
    - Though not ideal in some ways
    - E.g. exponential blowup if constraint represented implicitly
- Encodings is another rich area
  - More from Constraints to SAT this time
  - And some rather cool encodings
    - which can actually be seen as encoding algorithms

Constraint propagator

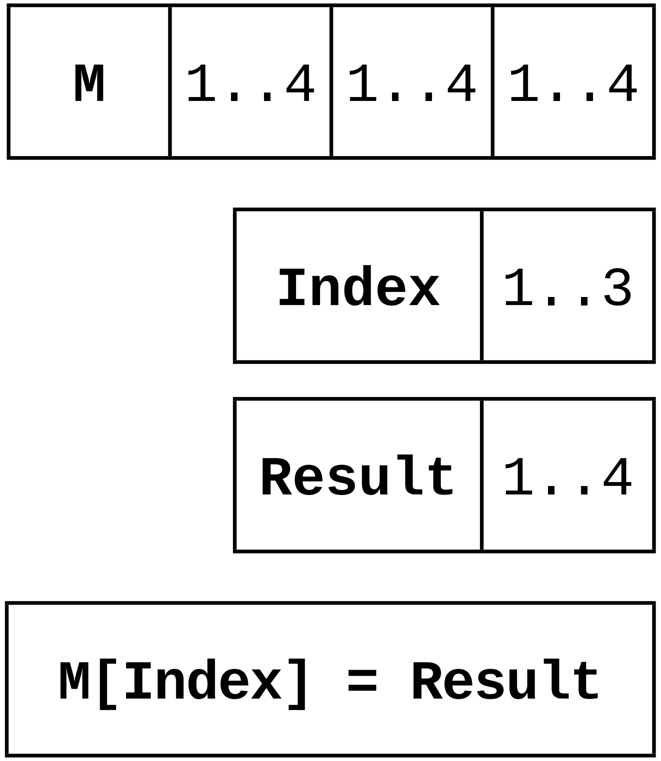
- propagates an individual constraint
- deducing values which can be removed
- because ...
  - just based on this constraint
  - and current state of the domains
  - no allowed tuple involves that value

Constraint propagator

- Works in one of 2 ways
  - The **table** constraint
    - actually have the list of allowed tuples
    - GAC-Schema, GAC2001, ... propagates it
  - **All other** constraints
    - a specialised function propagates it
    - deduces which values can be removed
    - using special purpose algorithm

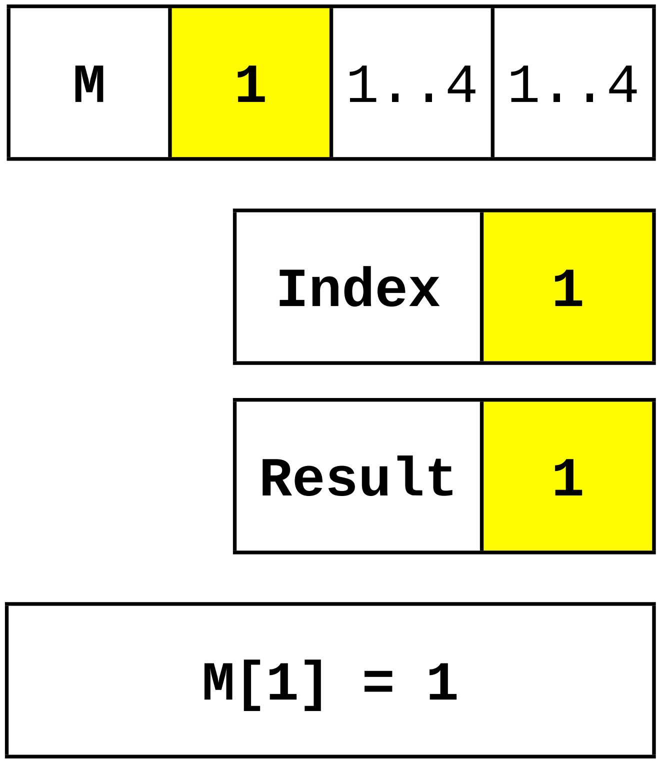
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



Elementary Example

- Vector **M** of variables

`M`	`2`	`1..4`	`1..4`

- **Index** into that vector

`Index`	`1`

- **Result** variable

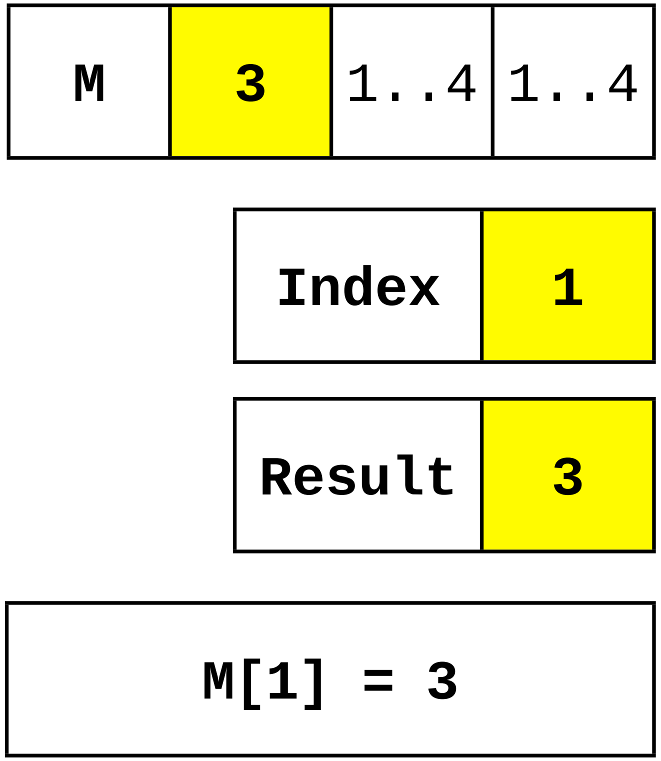
`Result`	`2`

- M[Index] = Result

```
M[1] = 2
```

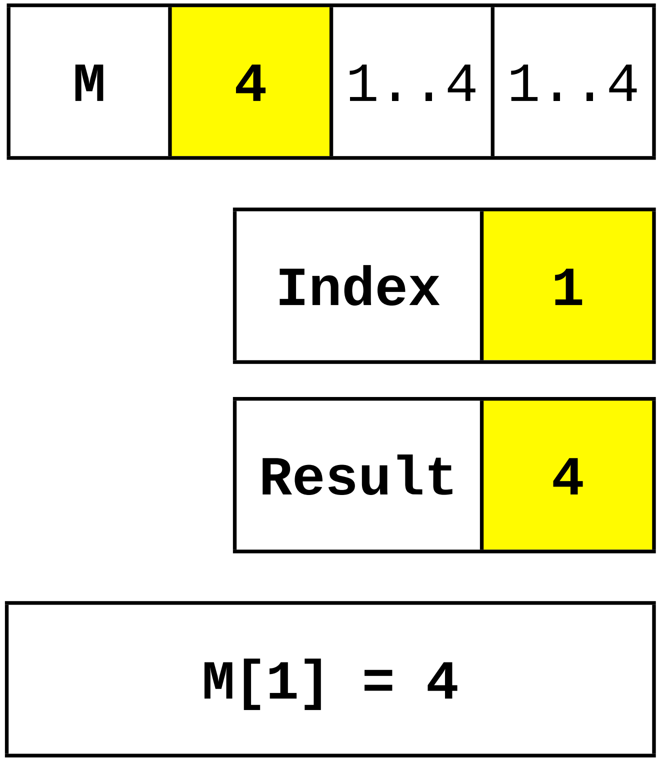
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



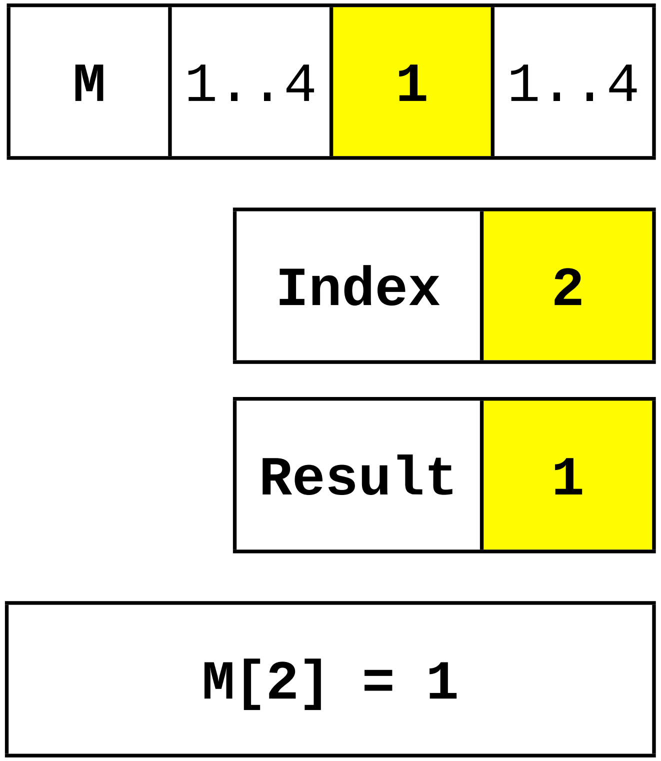
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



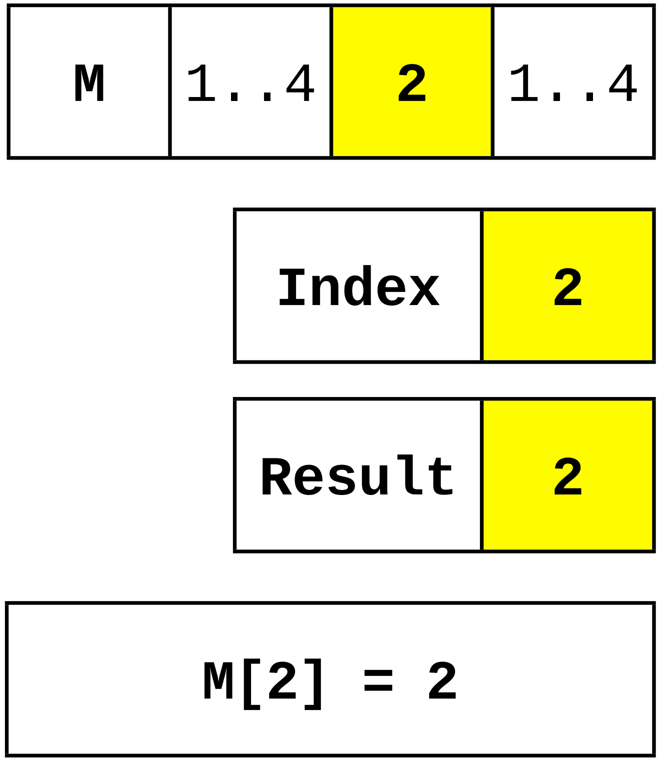
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



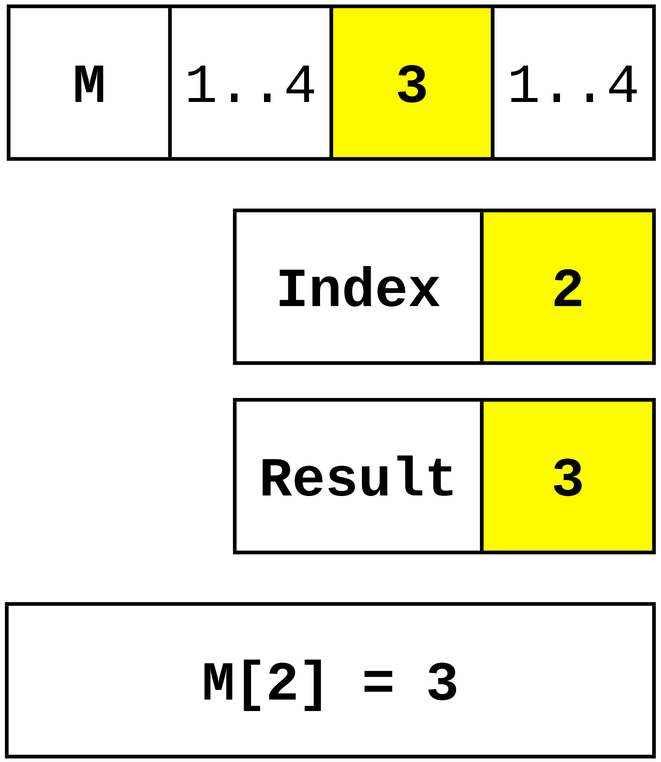
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



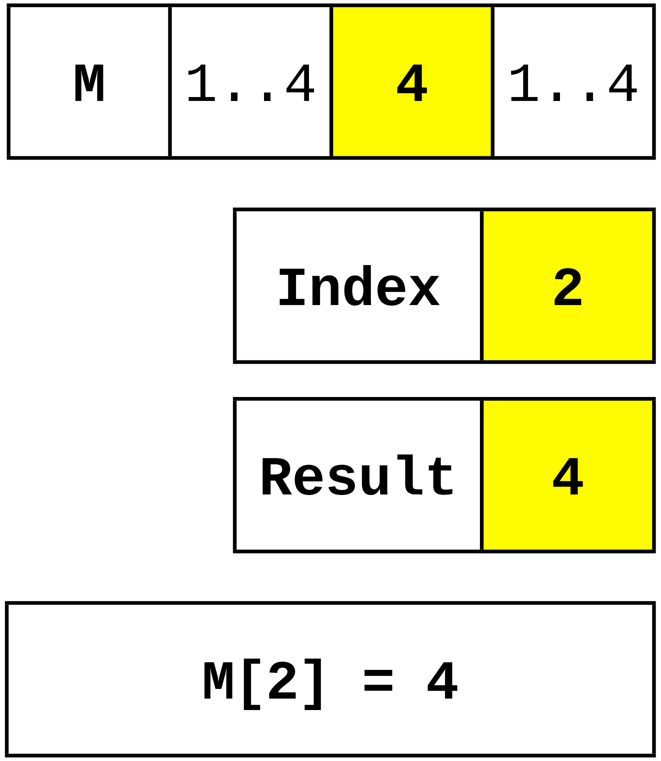
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



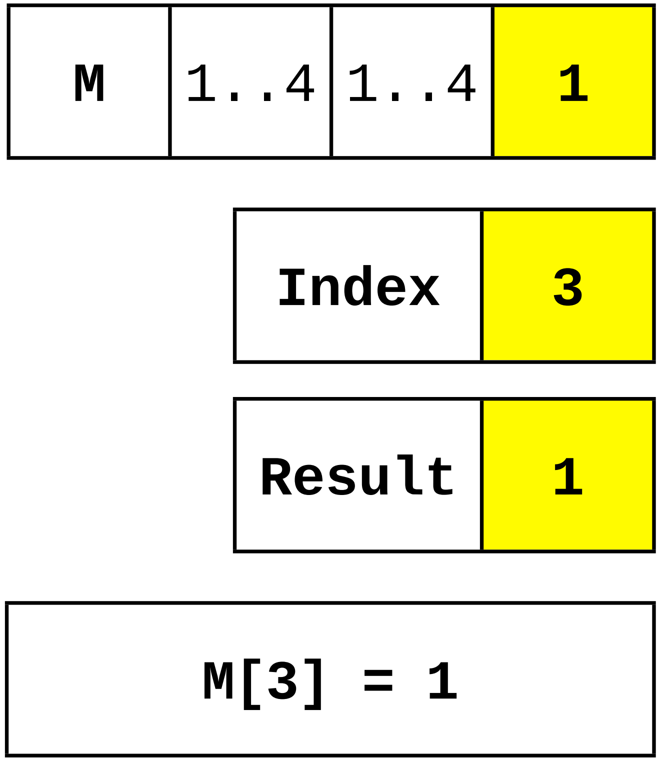
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



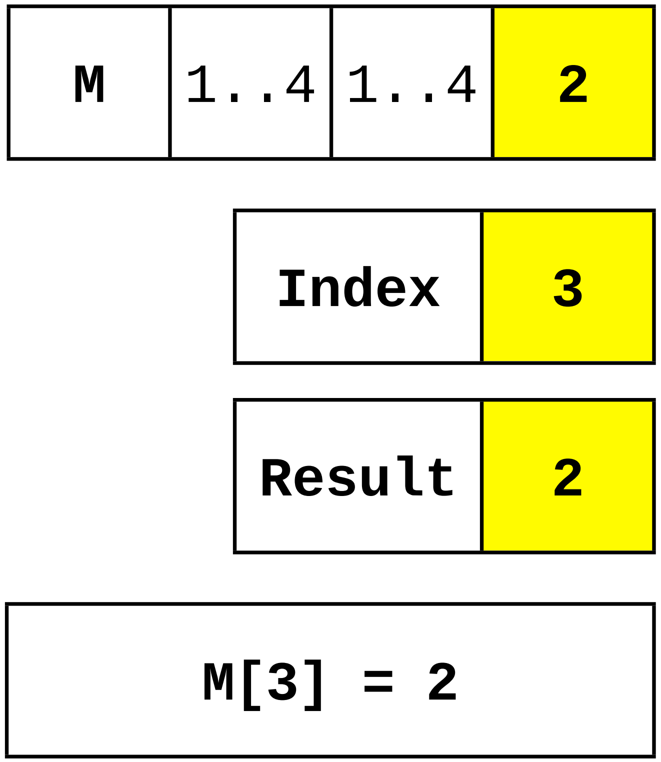
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



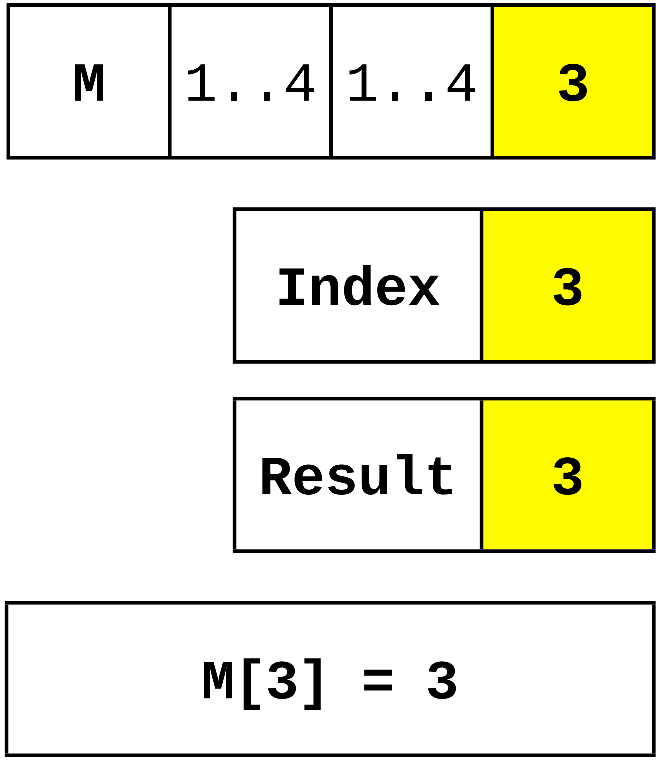
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



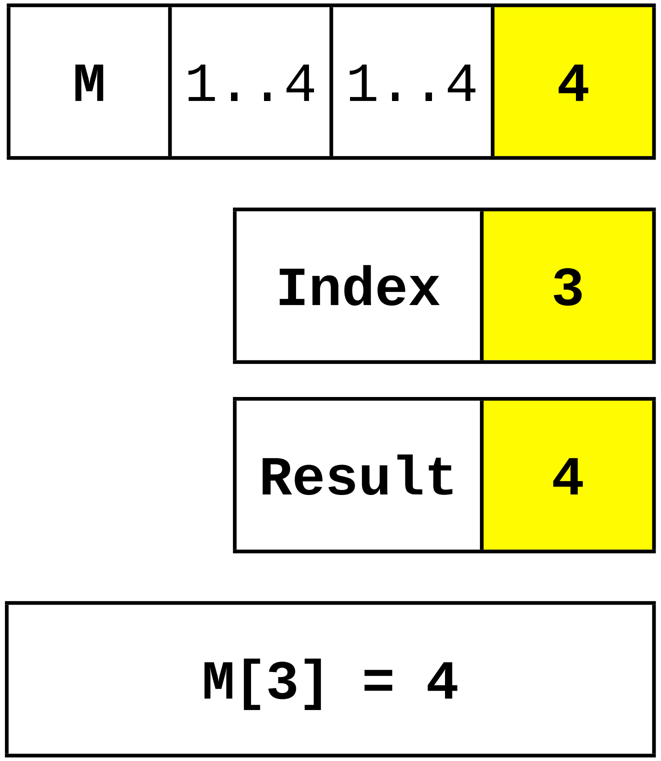
Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result



Elementary Example

- Vector **M** of variables
- **Index** into that vector
- **Result** variable
- M[Index] = Result

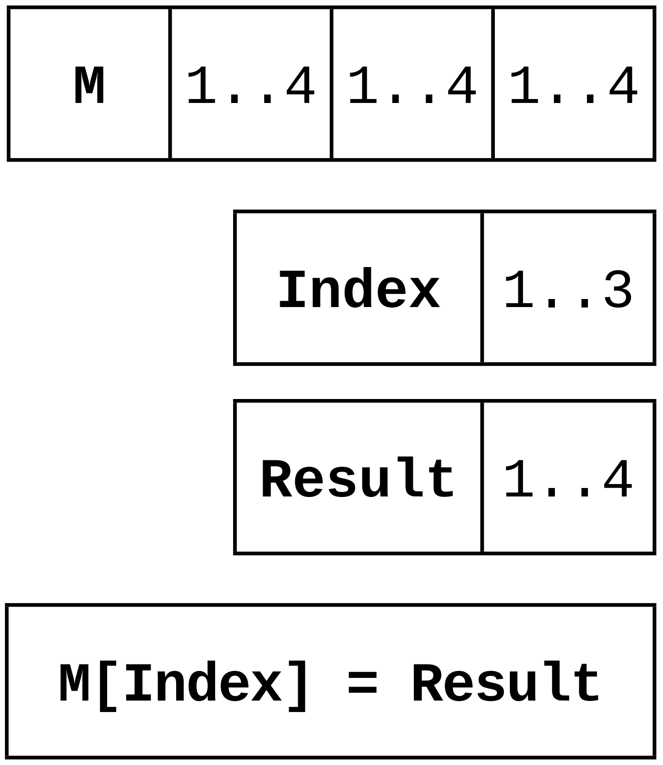


Element	Example
• Vector **M** of variables	`M` `1..4` `1..4` **`4`**
• **Index** into that vector	**`Index`** **`3`**
• **Result** variable	**`Result`** **`4`**
• M[Index] = Result	**`M[3] = 4`**

Element Example

- This is the ***element*** constraint
- Very useful expressively
  - e.g. function application
  - f(g(i)) = `z`
    - i `constant`
    - `z` variable
    - Arrays `F`, `G` for f, g

```
element(F,G[i],z)
```



One in 116,179,193,109,431

```
0 0 2 2 2 2 2 7 7 7
0 0 2 2 2 2 3 7 7 7
2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2
4 4 2 2 2 2 2 2 2 2
5 5 2 2 2 2 2 3 3 3
5 5 2 2 2 2 2 3 2 3
2 2 2 2 2 2 2 2 2 2
5 5 2 2 2 2 3 3 2 3
4 4 2 2 2 4 4 2 4 2
```

- An example of a *semigroup*
  - mathematicians study these
  - various algebraic constraints
- Enumerated by Minion
  - Distler/Kelsey/Kotthoff
  - 72.9 CPU *years*
  - 50,000 found per CPU *second*

Element

M_1	1	2	3	4
M_2	1	2	3	4
M_3	1	2	3	4

Index	1	2	3

Result	1	2	3	4

Example

`M`	`1..4`	`1..4`	`1..4`

`Index`	`1..3`

`Result`	`1..4`

```
M[Index] = Result
```

Element Example

M_1	1	2	3	4
M_2	1	2	3	4
M_3	1	2	3	4

Index	1	2	3

Result	1	2	3	4

- No propagation possible
- Every value in some solution of the constraint
- So let's remove some values

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- No propagation possible
- Every value in some solution of the constraint
- So let’s remove some values

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 4?**

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 4?**
- Could be ...
  - M[2] = 4
  - Index = 2

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 2?**

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 2?**
- None of three possible ways work

Element Example

M_1	1	🟥	3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 2?**
- None of three possible ways work

Element Example

M_1	1		3	
M_2	1	🟥	3	4
M_3	1	2	3	4

Index	1	🟩 2	

Result		🟨 2		4

- **Result = 2?**
- None of three possible ways work

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	🟥

Result		2		4

- **Result = 2?**
- None of three possible ways work

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		2		4

- **Result = 2?**
- None of three possible ways work

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		🟥		4

- **Result = 2?**
- None of three possible ways work
- Remove 2 from domain of Result

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		🟥		4

- **Result = 2?**
- None of three possible ways work
- Remove 2 from domain of Result

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result				4

- **Index = 1?**

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result				4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result	🟥	🟥		4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.

Element Example

M_1	1	🟥	3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		🟥		4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		🟥	🟥	4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.

Element Example

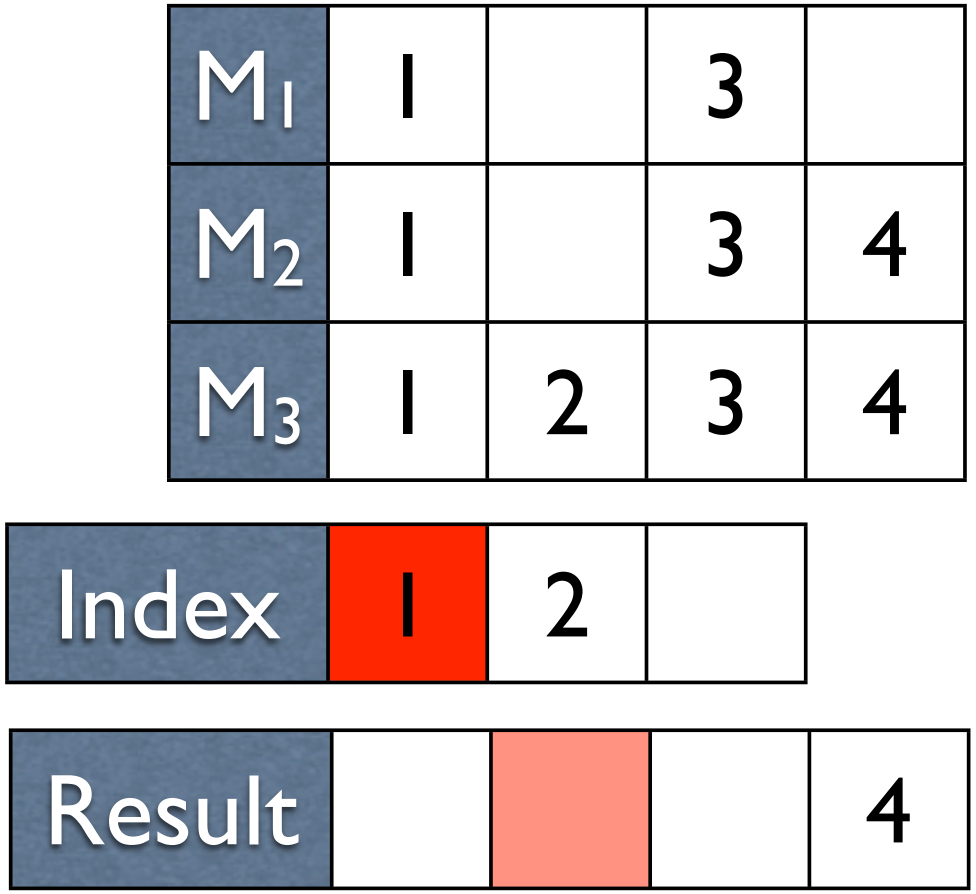
M_1	1		3	🟥
M_2	1		3	4
M_3	1	2	3	4

Index	1	2	

Result		🟥		4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.

Element Example



M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	1	2		
Result				4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.
- Remove 1 from domain of Index

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.
- Remove 1 from domain of Index

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **Index = 1?**
- No.
- No value of M[1] is the same as a value of Result.
- Remove 1 from domain of Index

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 1?**

Element Example

M_1	1		3	
M_2	1		3	4
M_3	1	2	3	4

Index	🟥	2	

Result	🟥	🟥		4

- **M[2] = 1?**
- No.
- Because M[2] = Result
- Result can't be 1

Element Example

M_1	1		3	
M_2	🟥 1		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 1?**
- No.
- Remove 1 from domain of M[2]

Element Example

M_1	1		3	
M_2	🟥		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 1?**
- No.
- Remove 1 from domain of M[2]

Element Example

M_1	1		3	
M_2	🟥		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 1?**
- No.
- Remove 1 from domain of M[2]

Element Example

M_1	1		3	
M_2	🟥		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 3?**

Element Example

M_1	1		3	
M_2	🟥		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥	🟥	4

- **M[2] = 3?**
- No.
- Because M[2] = Result
- Result can’t be 3

Element Example

M_1	1		3	
M_2	🟥		3	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 3?**
- Remove 3 from domain of M[2]

Element Example

M_1	1		3	
M_2	▨		🟥	4
M_3	1	2	3	4

Index	▨	2	

Result		▨		4

- **M[2] = 3?**
- Remove 3 from domain of M[2]

Element Example

M_1	1		3	
M_2	🟥		🟥	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		4

- **M[2] = 3?**
- Remove 3 from domain of M[2]

Element Example

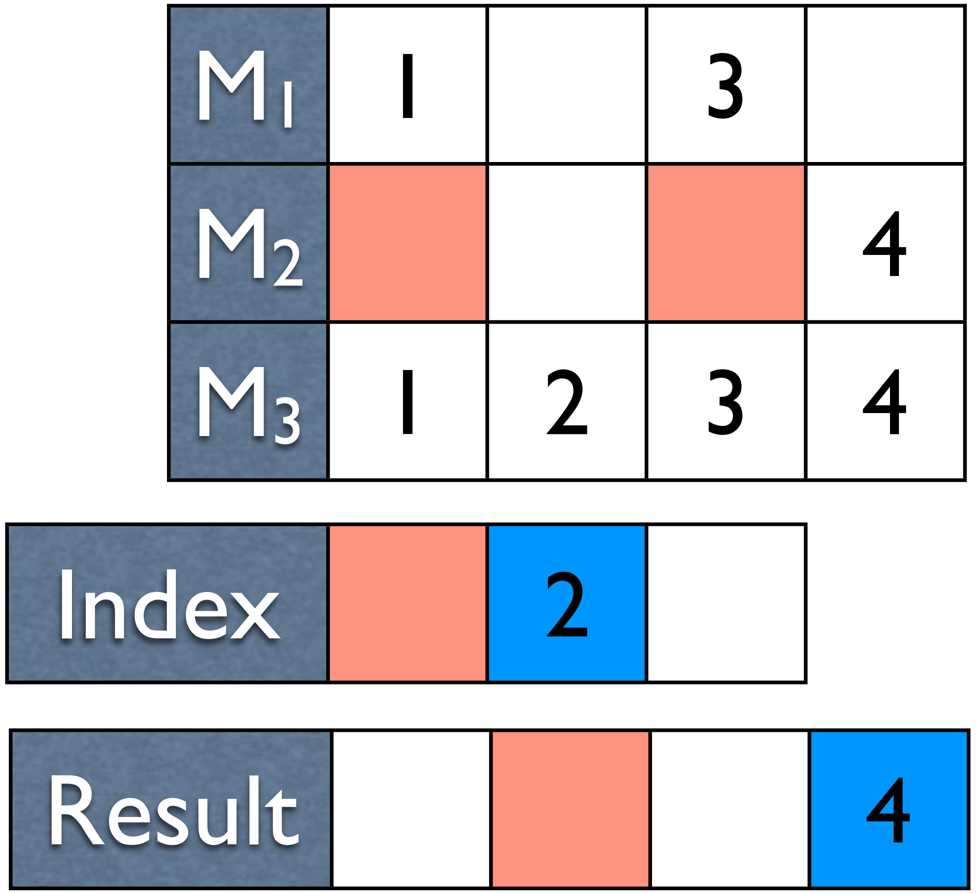
M_1	1		3	
M_2	🟥		🟥	4
M_3	1	2	3	4

Index	🟥	2	

Result		🟥		🟦 4

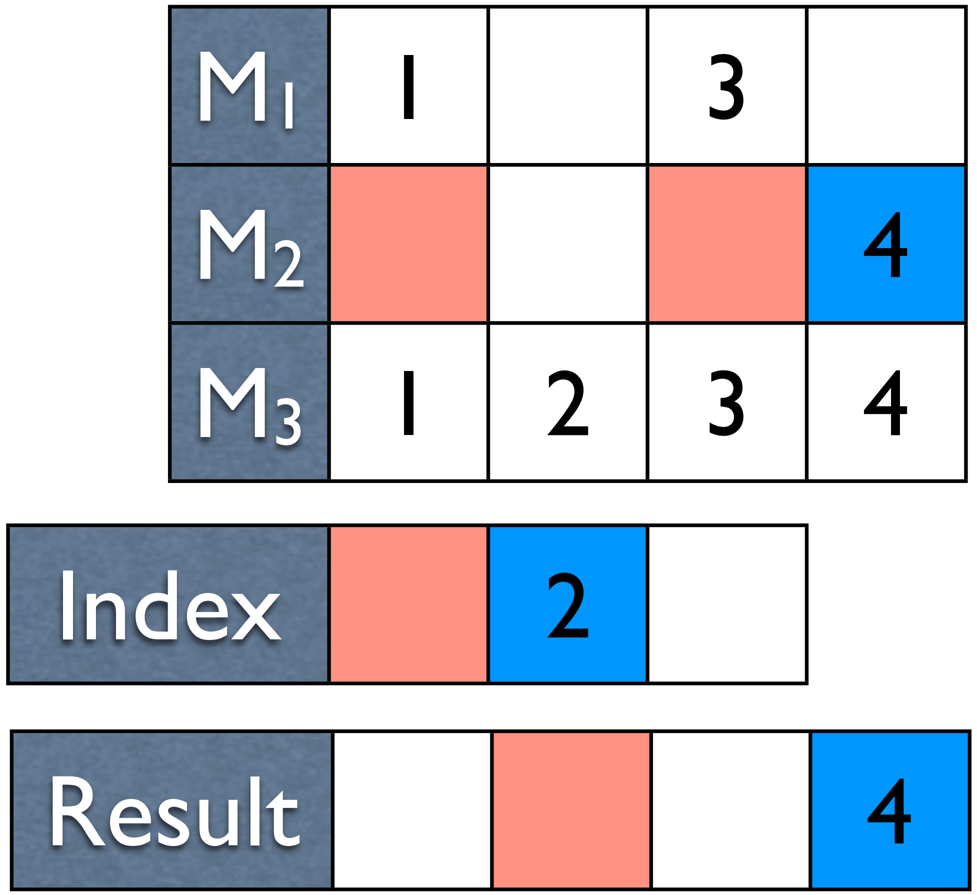
- In doing all this, we have ...
- assigned **Result = 4**

Element Example



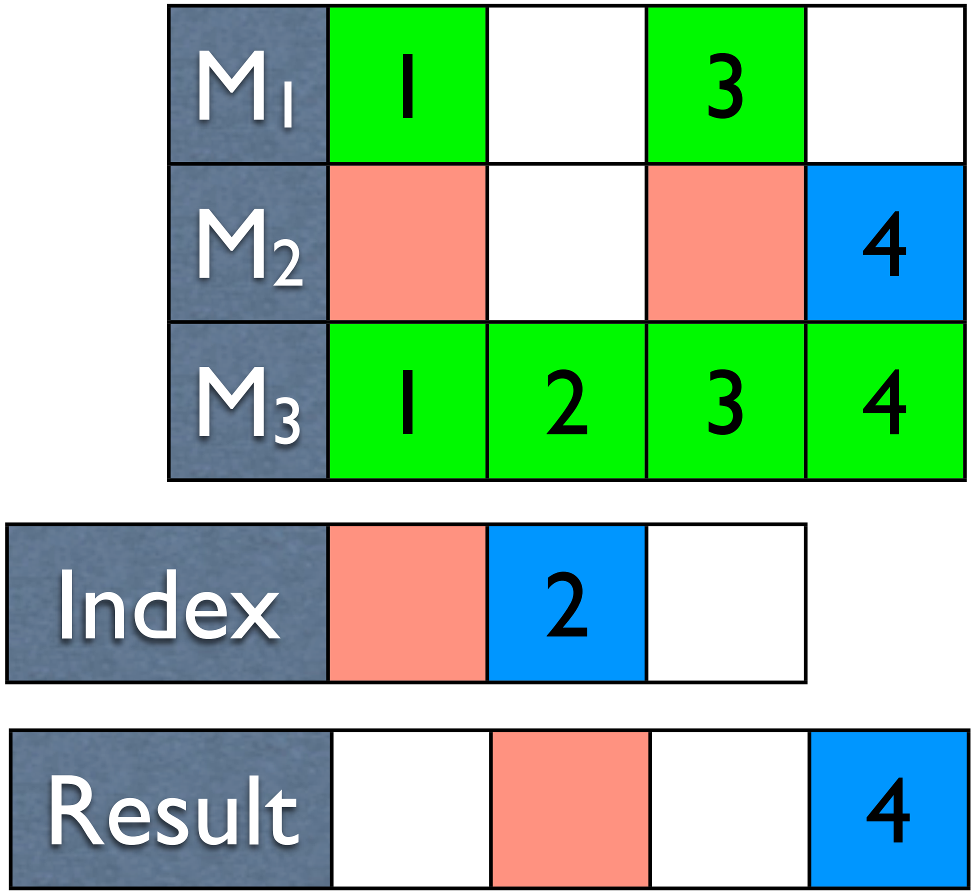
- In doing all this, we have ...
- assigned **Result = 4**
- assigned **Index = 2**

Element Example



- In doing all this, we have ...
- assigned **Result = 4**
- assigned **Index = 2**
- assigned **M[2]=4**

Element Example

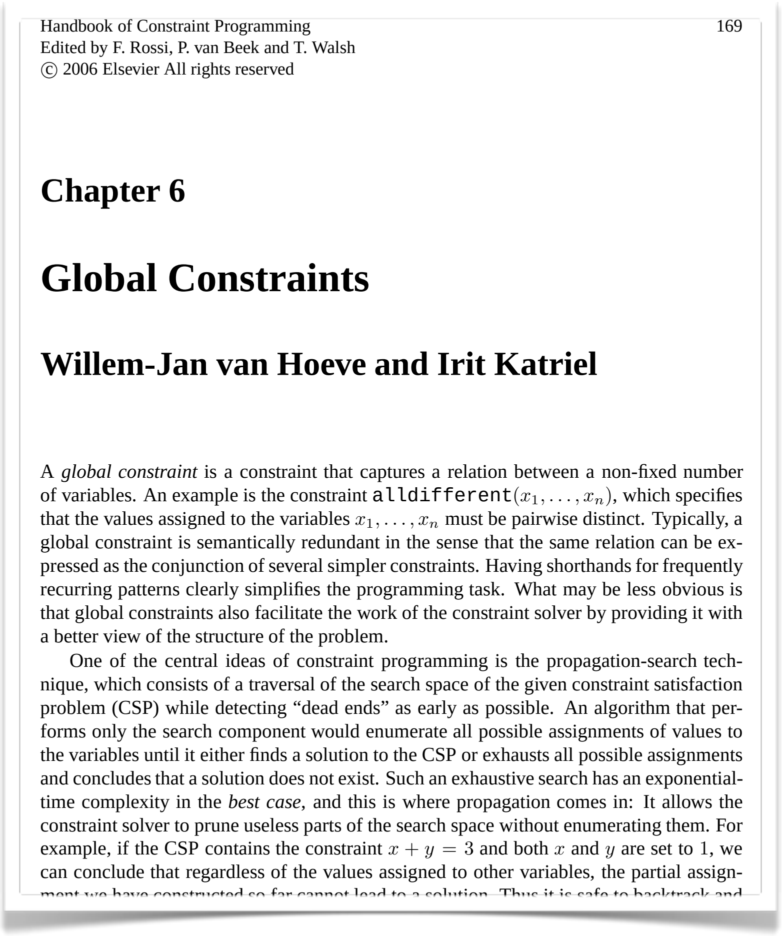


- In doing all this, we have ...
- assigned **Result = 4**
- assigned **Index = 2**
- assigned **M[2]=4**

And proved that all values of M[1] and M[3] are ok.

Some Key Constraints

- table (list of tuples)
- sum
  - knapsack
- element
- all-different
- global cardinality constraint
  - gcc with costs


Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh
© 2006 Elsevier All rights reserved

169

Chapter 6

Global Constraints

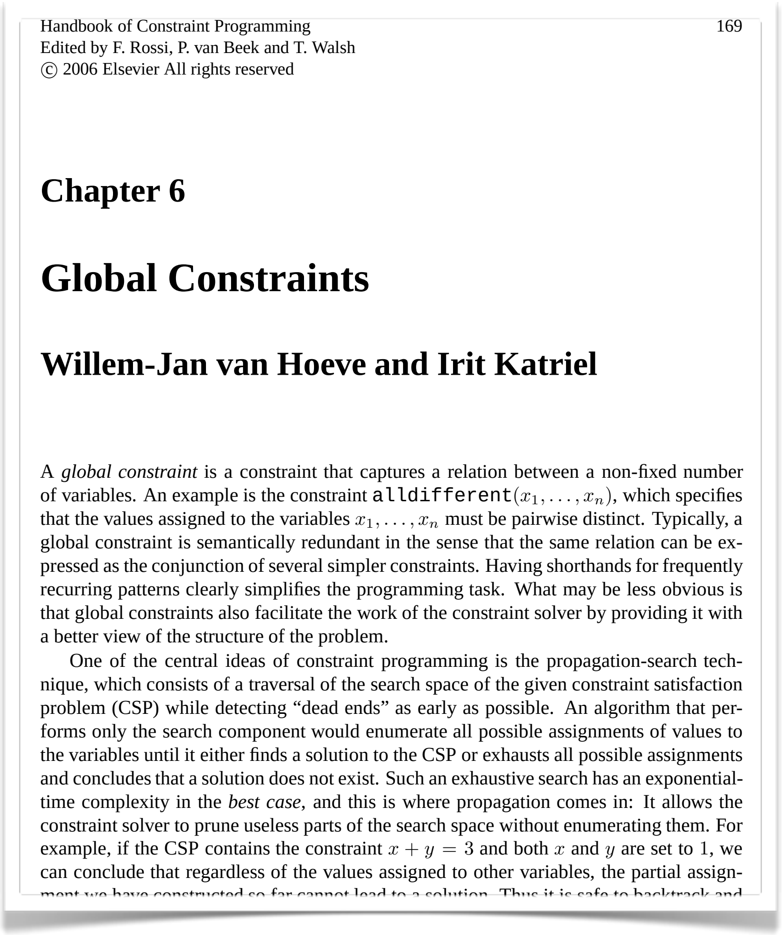
Willem-Jan van Hoeve and Irit Katriel

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables. An example is the constraint `alldifferent`$(x_1, \ldots, x_n)$, which specifies that the values assigned to the variables $x_1, \ldots, x_n$ must be pairwise distinct. Typically, a global constraint is semantically redundant in the sense that the same relation can be expressed as the conjunction of several simpler constraints. Having shorthands for frequently recurring patterns clearly simplifies the programming task. What may be less obvious is that global constraints also facilitate the work of the constraint solver by providing it with a better view of the structure of the problem.

One of the central ideas of constraint programming is the propagation-search technique, which consists of a traversal of the search space of the given constraint satisfaction problem (CSP) while detecting "dead ends" as early as possible. An algorithm that performs only the search component would enumerate all possible assignments of values to the variables until it either finds a solution to the CSP or exhausts all possible assignments and concludes that a solution does not exist. Such an exhaustive search has an exponential-time complexity in the *best case*, and this is where propagation comes in: It allows the constraint solver to prune useless parts of the search space without enumerating them. For example, if the CSP contains the constraint $x + y = 3$ and both x and y are set to 1, we can conclude that regardless of the values assigned to other variables, the partial assignment we have constructed so far cannot lead to a solution. Thus it is safe to backtrack and

Some Key Constraints

- cumulative (for scheduling)
- regular (language membership)
- circuit
- soft all different
- And there are a few more ...



Handbook of Constraint Programming 169
Edited by F. Rossi, P. van Beek and T. Walsh
© 2006 Elsevier All rights reserved

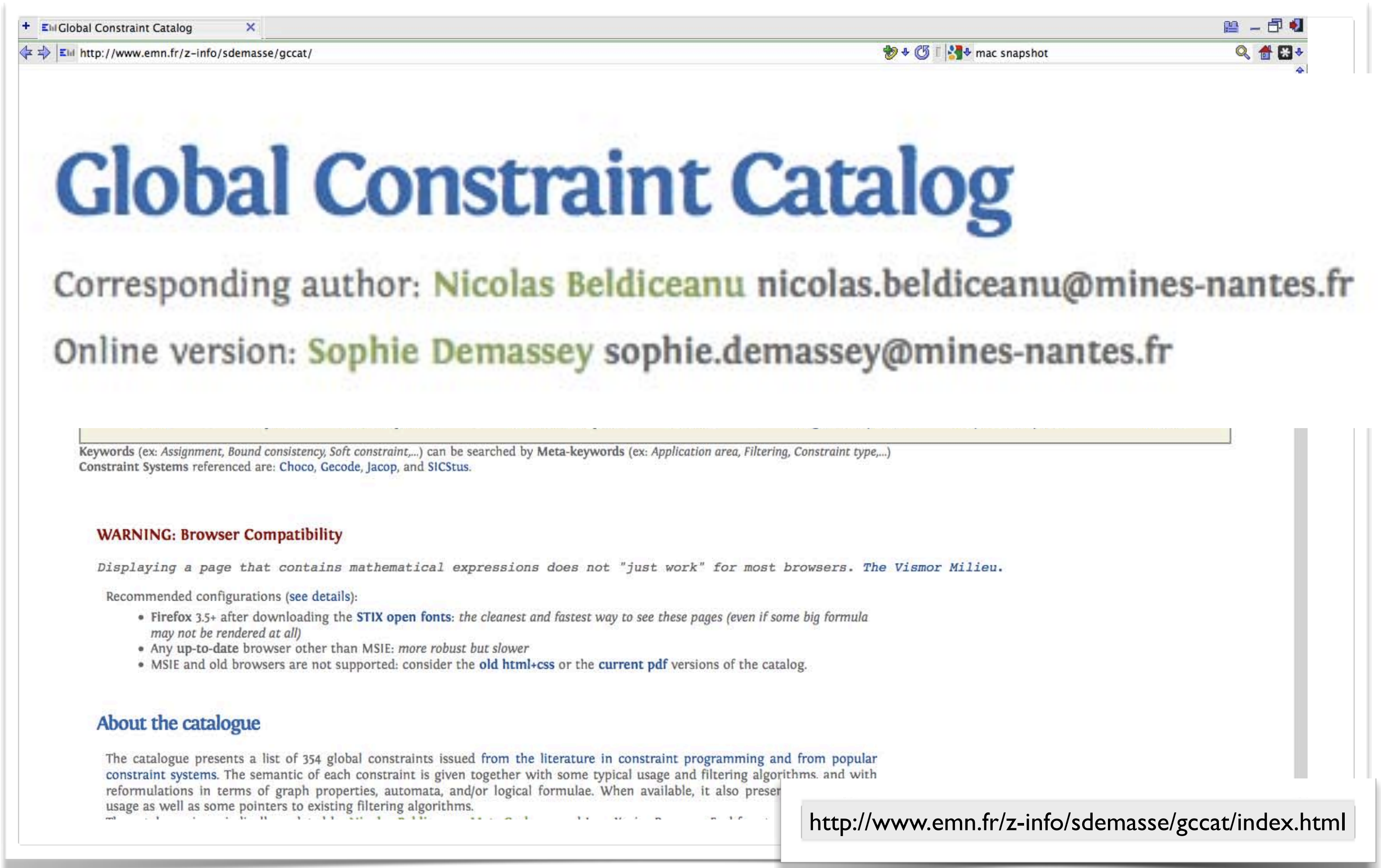
Chapter 6

Global Constraints

Willem-Jan van Hoeve and Irit Katriel

A *global constraint* is a constraint that captures a relation between a non-fixed number of variables. An example is the constraint alldifferent$(x_1, \ldots, x_n)$, which specifies that the values assigned to the variables $x_1, \ldots, x_n$ must be pairwise distinct. Typically, a global constraint is semantically redundant in the sense that the same relation can be expressed as the conjunction of several simpler constraints. Having shorthands for frequently recurring patterns clearly simplifies the programming task. What may be less obvious is that global constraints also facilitate the work of the constraint solver by providing it with a better view of the structure of the problem.

One of the central ideas of constraint programming is the propagation-search technique, which consists of a traversal of the search space of the given constraint satisfaction problem (CSP) while detecting "dead ends" as early as possible. An algorithm that performs only the search component would enumerate all possible assignments of values to the variables until it either finds a solution to the CSP or exhausts all possible assignments and concludes that a solution does not exist. Such an exhaustive search has an exponential-time complexity in the *best case*, and this is where propagation comes in: It allows the constraint solver to prune useless parts of the search space without enumerating them. For example, if the CSP contains the constraint $x + y = 3$ and both x and y are set to 1, we can conclude that regardless of the values assigned to other variables, the partial assignment we have constructed so far cannot lead to a solution. Thus it is safe to backtrack and


Global Constraint Catalog
http://www.emn.fr/z-info/sdemasse/gccat/
mac snapshot

Global Constraint Catalog

Corresponding author: Nicolas Beldiceanu nicolas.beldiceanu@mines-nantes.fr

Online version: Sophie Demassey sophie.demassey@mines-nantes.fr

Keywords (ex: *Assignment, Bound consistency, Soft constraint,...*) can be searched by Meta-keywords (ex: *Application area, Filtering, Constraint type,...*)
Constraint Systems referenced are: Choco, Gecode, Jacop, and SICStus.

WARNING: Browser Compatibility

Displaying a page that contains mathematical expressions does not "just work" for most browsers. The Vismor Milieu.

Recommended configurations (see details):

- **Firefox** 3.5+ after downloading the **STIX open fonts**: *the cleanest and fastest way to see these pages (even if some big formula may not be rendered at all)*
- Any **up-to-date** browser other than MSIE: *more robust but slower*
- MSIE and old browsers are not supported: consider the **old html+css** or the **current pdf** versions of the catalog.

About the catalogue

The catalogue presents a list of 354 global constraints issued from the literature in constraint programming and from popular constraint systems. The semantic of each constraint is given together with some typical usage and filtering algorithms, and with reformulations in terms of graph properties, automata, and/or logical formulae. When available, it also preser
usage as well as some pointers to existing filtering algorithms.

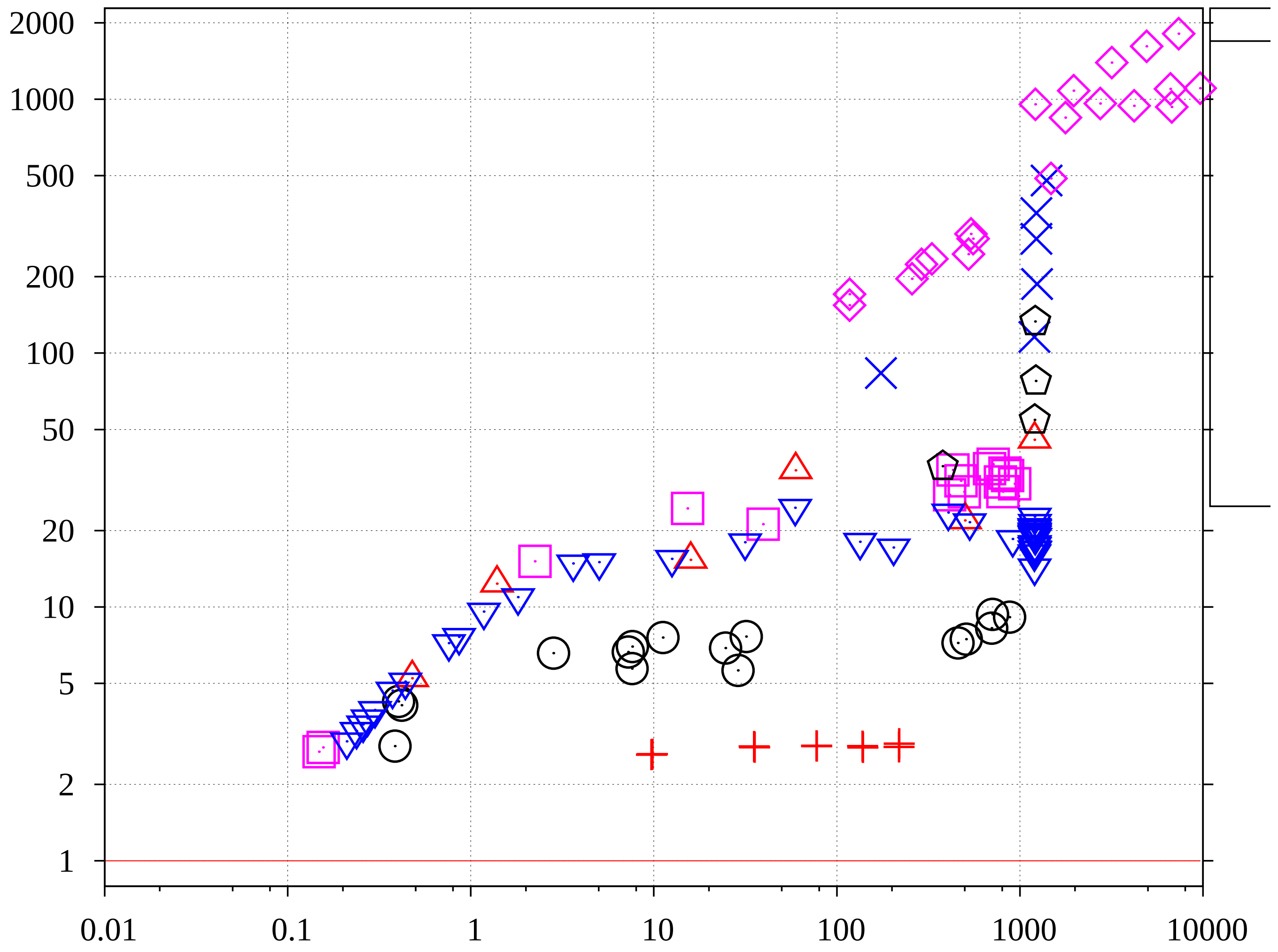
http://www.emn.fr/z-info/sdemasse/gccat/index.html

343 more ...

Propagator algorithms

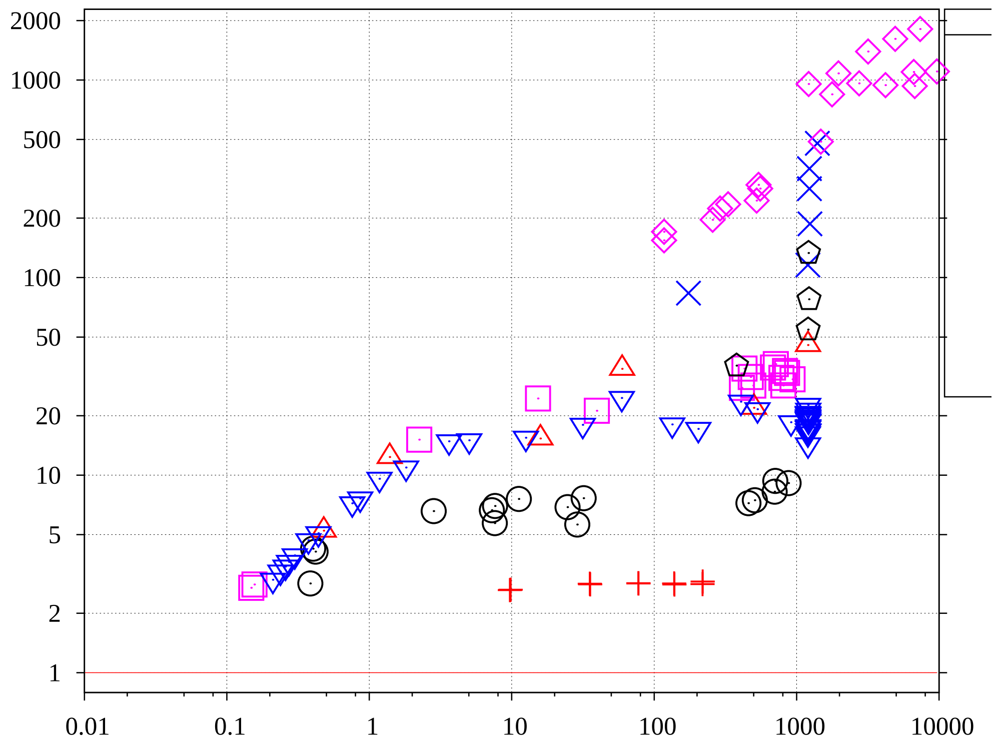
- A specialised propagator
  - Built into a modern constraint solver
  - Can make more deductions than naive method
  - Can be highly optimised
  - Interact with other constraints via propagation

“Can be highly optimised”



“Can be highly optimised”

- all-different examples again
  - x-axis is secs to solve
    - using good implementation of all-different propagator
  - y-axis is *speedup factor*
    - using best optimisations
- We can solve hard problems more than ***1,000*** times faster
  - doing *exactly the same* propagation



Ian P. Gent and Ian Miguel and Peter Nightingale,
Generalised Arc Consistency for the AllDifferent Constraint: An Empirical Survey,
Artificial Intelligence, Volume 172 number 18 pages 1973-2000.

One thing leads to another...

- Key feature of propagation
- Value removal in one constraint
  - means more can be removed by a second
  - and more by a third constraint
  - and now more by the first constraint
  - and now more by a fourth ...

One thing leads to another...

- Looks like this might go on forever...
  - but it won’t.
- Propagation will terminate
- And either prove there is no solution
  - in which case we backtrack
- Or succeed
  - typically a key consistency property holds
  - and we have to branch search

Key Consistency Properties

- GAC
  - “Generalised Arc consistency”
    - or “Domain consistency
  - for any arity constraints
  - there’s an allowed tuple involving every value
    - i.e. all values in relevant domain
    - tuple allowed by constraint

Communicating Constraints

- To have one thing lead to another ...
- We have to propagate effect of one constraint to other constraints ...
- Done by a propagation Queue
- When one value gets removed
  - We put this information on the queue
- When we pop off the queue
  - We notify any constraints that need to know
  - They propagate and either do nothing or remove more values

Constraints Basics II: Practice

"In theory there's no difference between theory and practice.

In practice there is."

Jan L.A. van de Snepscheut

or maybe *Yogi Berra*

Misconceptions about propagation

- Perhaps not too damaging
- But can lead to wrong mental model
  - leading to bad practice
  - and bad research
- And anyway, reality is far more interesting
- I’ve hidden a lot of straw men

First Misconception

- I couldn’t bring myself to put it up even as straw man
- Constraints almost never lists of tuples
  - As in “the usual stuff”
  - Only when using the table constraint
    - though it’s an important constraint
  - If you don’t know this ...
    - ... then you might get complexity wrong
    - since you *don’t* need to read the table to propagate

Second misconception

- Levels of consistency matter
- Very often we go for GAC
  - Or Bounds Consistency or something else
- But **not** if the tradeoff is better elsewhere
  - And we don’t care if we don’t know what consistency it achieves

Third Misconception

- GAC algorithms establish GAC
- Not if there are repeated variables
  - e.g. all-different(x,y,z,x)
- GAC algorithms almost always pretend there are no repeated variables
- And repeated variables can't be banned

All that really matters

- Propagators should not remove values in some allowed tuple
- Propagators should fail if ...
  - all their variables are assigned
    - i.e. domains are now singleton {x}
  - and the resulting tuple is not allowed

Fourth Misconception

- There is a constraint queue
- Almost never single list operated as queue
- Might use a number of queues
  - or some more complex ordering
- E.g. “staging” in all-different
  - do the binary not-equals first
  - only later do more expensive propagator
  - vital to optimising run time

Fifth Misconception

- Constraint solvers have 354 propagators
  - Or at least a good number from catalog
- Of the 11 “key constraints” I listed
  - Minion has 4 of them
- Very few constraints that most solvers have

Sixth Misconception

- There’s only five misconceptions ...

Interlude

- Be careful what you say in your bio ...

  “On a good day he can juggle five balls, and on a very good day figure out some way to get juggling into a technical talk.”

What has juggling got to do with propagation?

What has juggling got to do with propagation?

Almost nothing

What has juggling got to do with propagation?

but not quite...

It’s dealing with some pretty hard constraints

In this case gravity ...

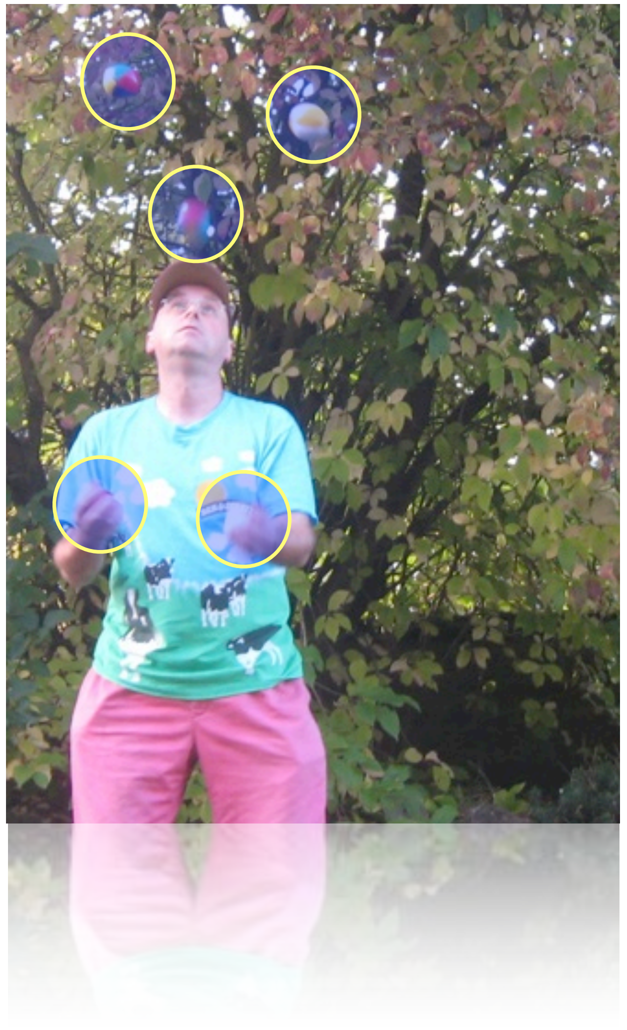
You can have constraints with different numbers of variables ...

This one has five variables

You can have constraints with different numbers of variables ...

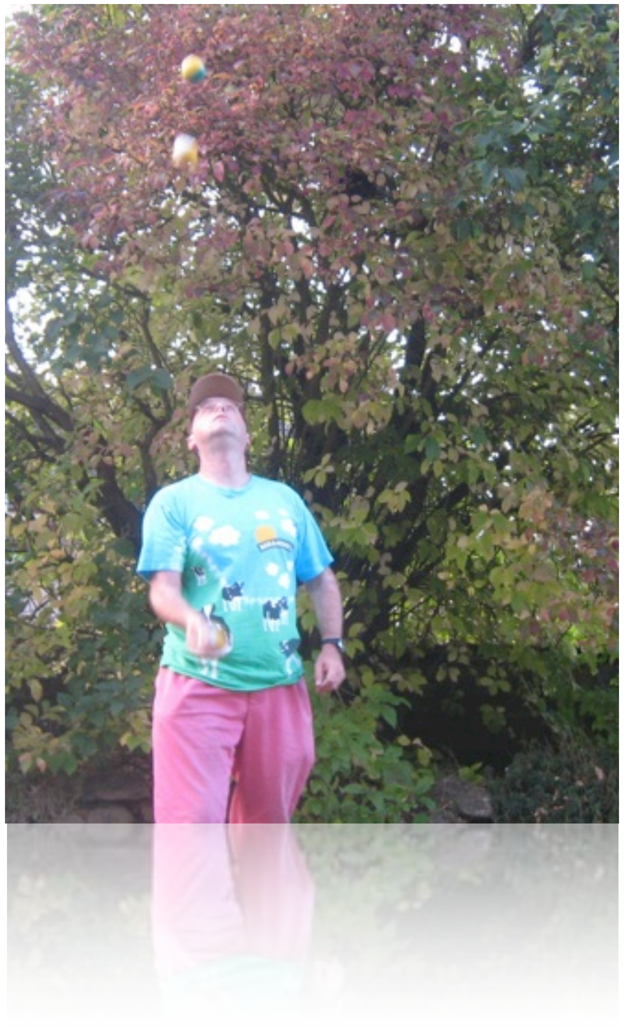
This one has five variables

No really, it does...



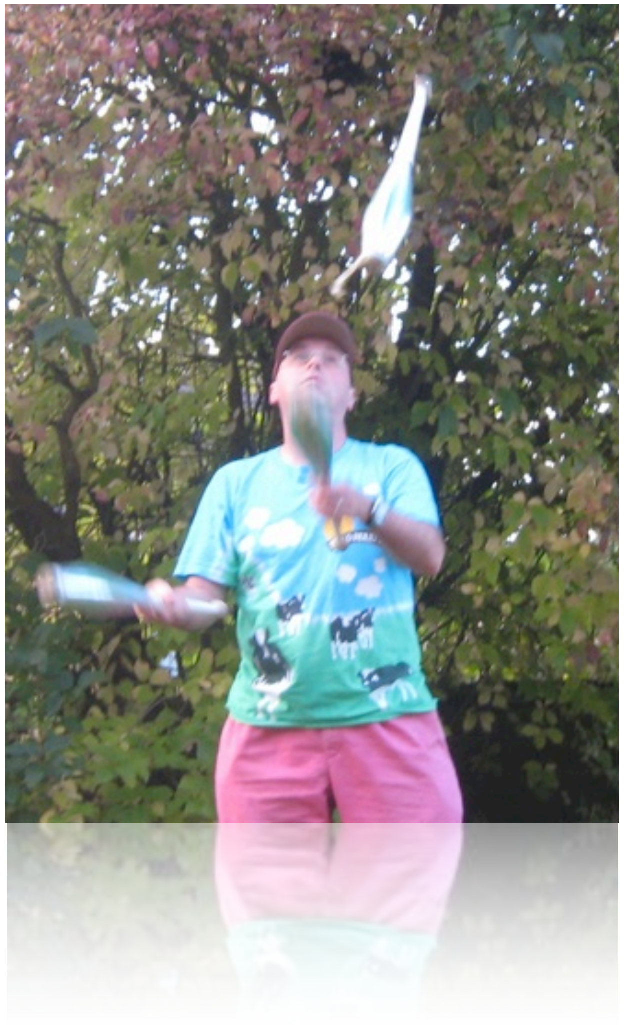
You can have constraints with different numbers of variables ...

This one has three variables



You might have to deal with different kinds of variables

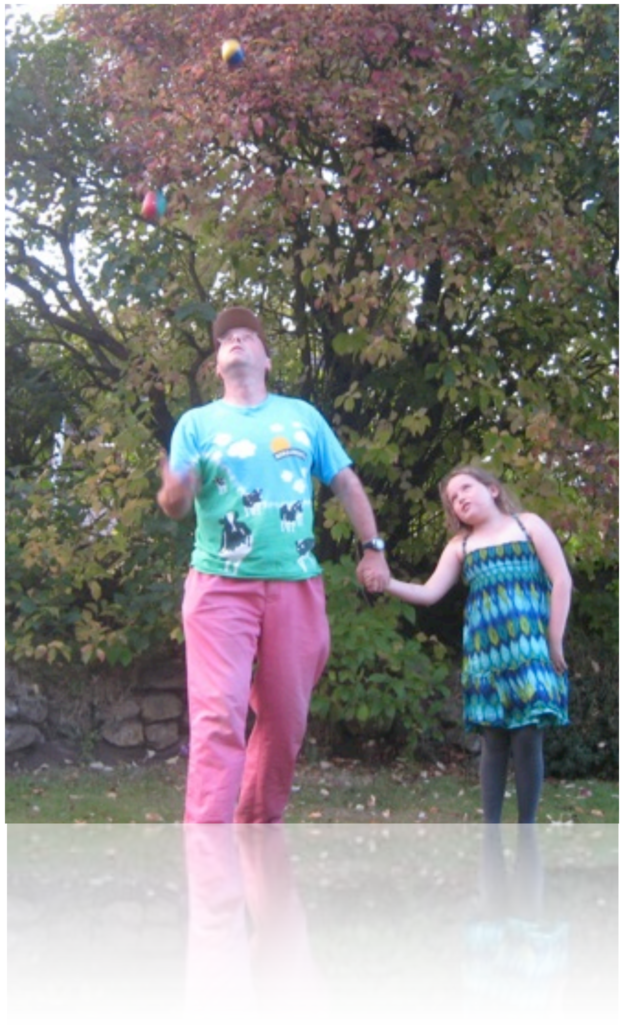
Which makes the propagation different



Different constraints interact with each other restricting solutions

One hand can't be used

And only three variables can be solved (by me)



One thing leads to another ...

You throw a ball ...
And then you have to catch it ...
But to catch it ...
You have to throw the next ball ...
And then you have to catch it ...
But to catch it ...

It looks like it might go on forever ...

But it won’t!

It is guaranteed to stop ...

either with success or failure