Constraint Programming

A technology to tackle combinatorial optimization problems

What is Constraint Programming

- Our definition

  Solving a combinatorial problem

  Taking into account the problem structure

- Programming with Constraints
  - A declarative programming paradigm where

    Relations between variables are stated as constraints

- Technology for solving combinatorial problems
  - Finite domain propagation

Why Constraint Programming

- Imagine you own a small print shop
- Running your business requires
  - Accepting customer orders
  - Splitting each order into jobs
  - Assigning workers to machines
  - Scheduling tasks for each job
  - Packing orders for delivery

Why Constraint Programming

- Running your business requires
  - Accepting customer orders
    - Capacity constrained optimization problem
  - Splitting orders into jobs
    - Lot sizing problem
  - Assigning workers to machines
    - Assignment problem
  - Scheduling tasks for each job
    - Resource constrained scheduling problem
  - Packing orders for delivery
    - Packing problem

Why Constraint Programming

- Solving each of these separately is an optimization problem
  - But solving each separately will be far from globally optimal
- How can we solve all together.
  - Only if we take into account the problem structure
  - And use a technology that can take advantage of it

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Constraint Satisfaction Problem

- "Find an object from a finite set which satisfies a number of constraints"
- Sounds easy
  - Test each constraint on each object
  - If one satisfies all constraints, finish.
- But
  - There are MANY of them

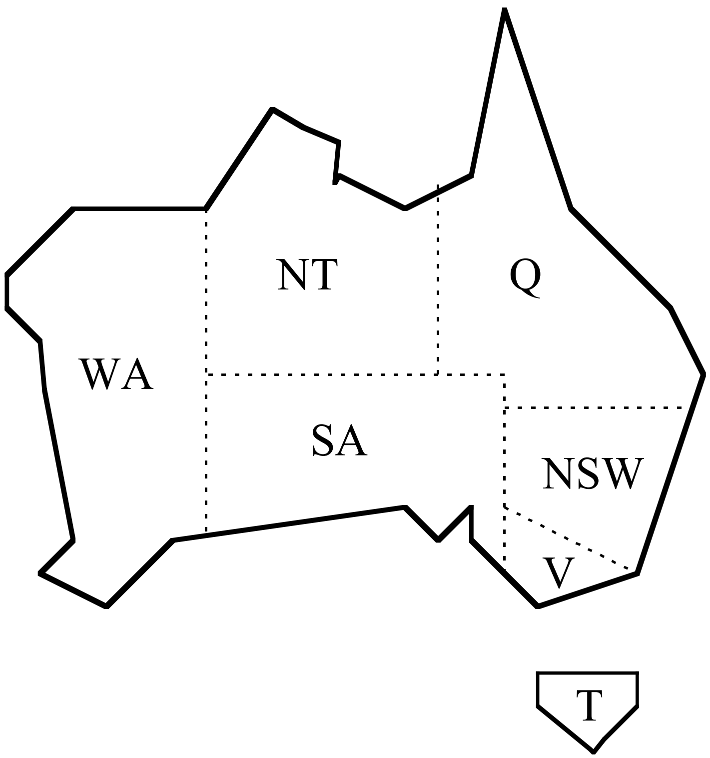
Map Colouring

A classic CSP is the problem of coloring a map so that no adjacent regions have the same color

Can the map of Australia be colored with 4 colors ?

Can the map of Australia be colored with 3 colors ?

Can the map of Australia be colored with 2 colors ?

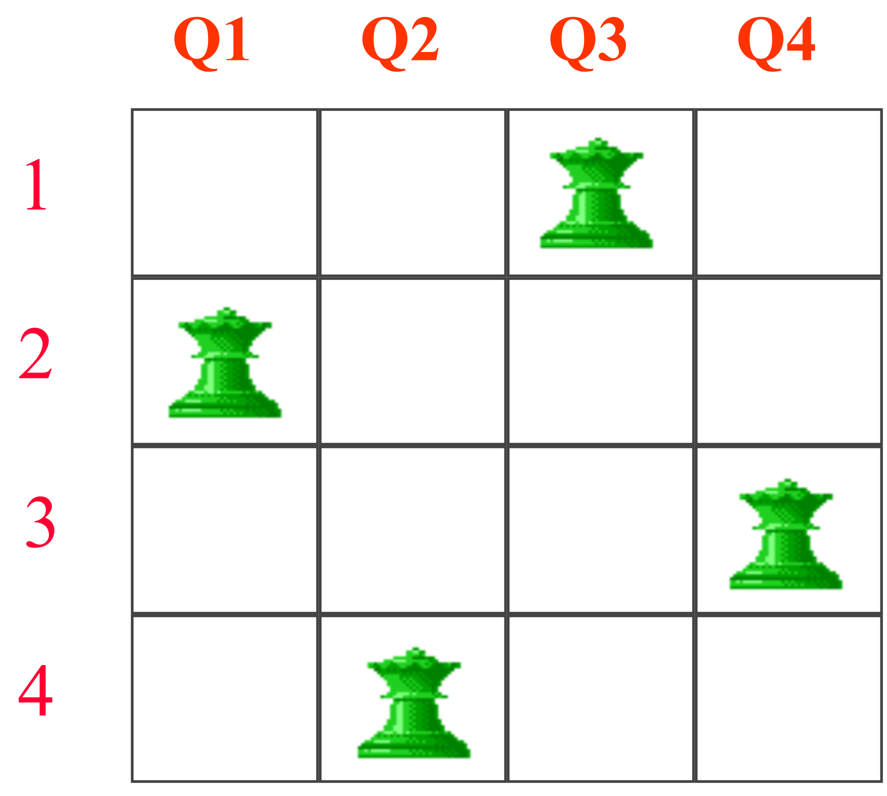


4-Queens

Place 4 queens on a 4 x 4 chessboard so that none can take another.

Four variables Q1, Q2, Q3, Q4 representing the row of the queen in each column. Domain of each variable is {1,2,3,4}

One solution! -->



Sudoku

- How many ways can you fill a Sudoku board with numbers 1-9?

- How many Sudoku puzzles are there?

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

6,670,903,752,021,072,936,960

Combinatorial Optimization

- “Find an optimal object from a set of objects”
- Sounds easy
  - Evaluate each object using the scoring function
  - Remember the best
- But
  - The objects are only specified “intensionally”
    - Only those objects satisfying some constraints
  - There are MANY of them

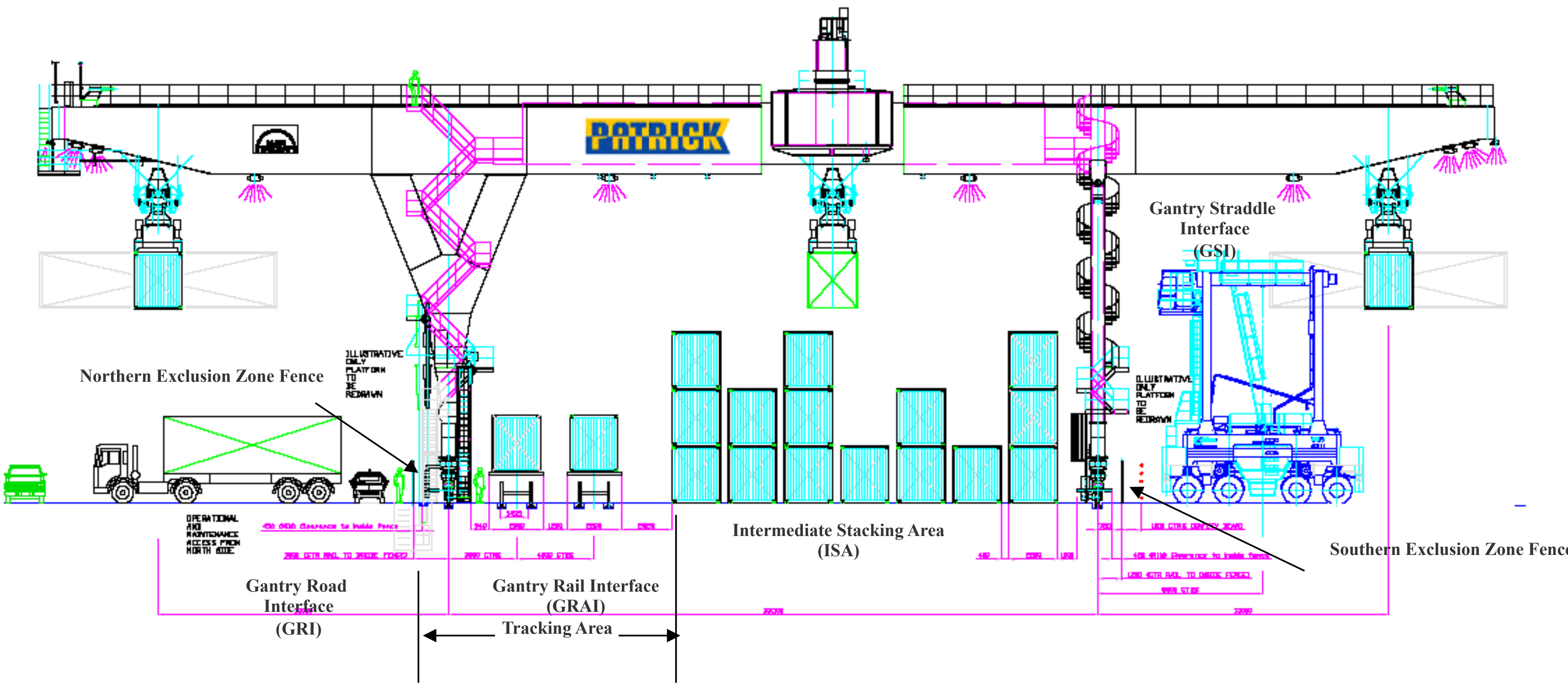
Smuggler's Knapsack

A smuggler with a knapsack with capacity 9, needs to choose items to smuggle to make a maximum profit

object	*profit*	*size*
whiskey	15	4
perfume	10	3
cigarettes	7	2

What is the best set of items you can come up with?

Gantry Crane Planning Example



System Specification: gantry crane planning example

- Where should containers be placed ready for loading/straddling?
- In what order should the gantries pick up the containers?
- What planning should be done for trains/trucks which haven't arrived yet?
- How can we enable the gantries to unload all the trains and all the trucks?

Importance

- Combinatorial Optimization is everywhere
  - Scheduling
  - Rostering
  - Packing
  - Routing
  - Allocating (e.g. water)
  - Planning
- Finding good or optimal solutions can save time, money and reduce environmental impact.

The Holy Grail for Constraint Programming

- Model Problems Naturally
  - constraints
  - solution properties
- Solve them efficiently
  - overcome combinatorial explosion
- Compile
  - Natural models to efficient solutions

Technology for Constraint Solving

- Local search
  - Simulated annealing
  - Tabu search
- Population search
  - Genetic algorithms
  - Beam search
- Mixed integer programming
- Finite domain propagation

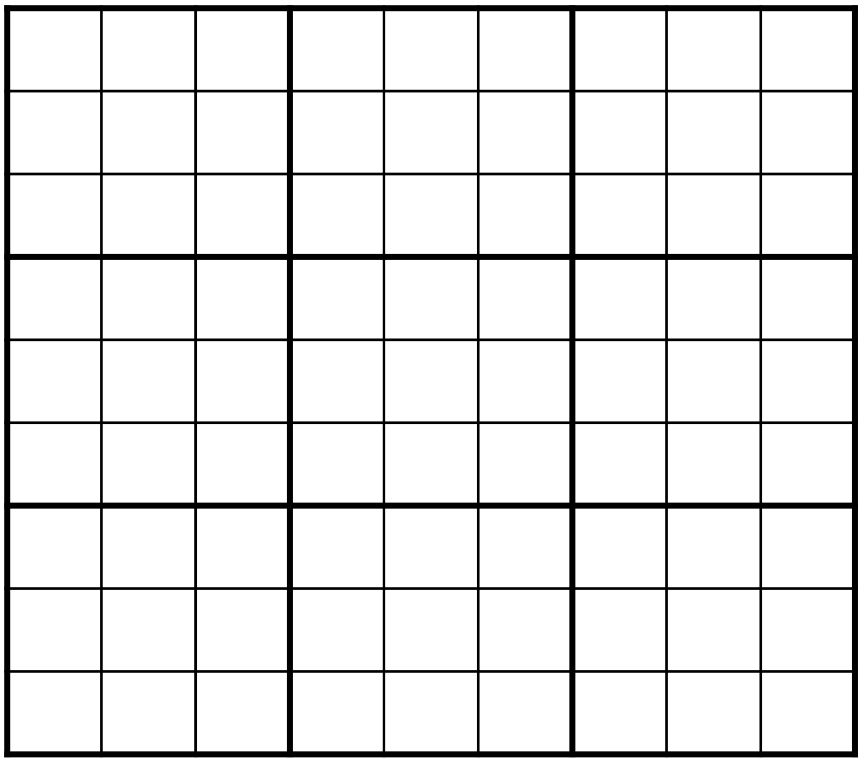
Why is Constraint Solving Hard?

- Write down solutions to the following (integer) constraints or claim unsatisfiability
  - $x = 5, y = 6$
  - $x = 3, y = 4, x = 5$
  - $y = x+2, z = y - x+2, u = 2*y + z$
  - $y = x+2, z = y - x + 2, x = z+1$
  - $y = x+2, z = y - x+2, x \geq z+1, y \leq z - 1$
- The problem is conjunction

Finite Domain Propagation

- Overcoming conjunction
  - Treat each constraint separately
  - Communicate inferences via variables
- A weak inference method
- Add to that
  - Search (guess bits of solution)
  - Engineering (to make the inference fast)
  - Learning (to remember what you already did)

Sudoku



- 81 variables
  - Each cell in table
- Each cell takes 1..9
- Each row, each column, and each 3x3 square contain the numbers 1..9
  - No repeats
  - Each number used
  - Assignment subproblem!

Propagation

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

- What goes in the green cell?
- Reason about the column

Propagation

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

- What goes in the green cell?
- Reason about what numbers cannot go in the other cells in the square?

Propagation

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

- What can go in the green cell?
- Reason about the row and then the column.

Propagation

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

- What can go in the green cell?
- Reason about the row and column

Propagation

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

- What goes in the green cell?
- Reason about the row

Propagation

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

- Any other fixed variables?

Propagation

- Examine each constraint in turn
- Reduce the domains of variables in the constraint
- Repeat until no further reduction

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Domains

- Record for each variable *X* its domain
  - set of possible values, denoted *D(X)*
- Usually *D(X)* is finite, but it might be very large
  - All 32 bit integers
  - All 64 bit floating point numbers between 0 and 1
- Essentially
  - Variables *X* represents a choice
  - The domain *D(X)* represents the possible choices for *X*
- Failed domain: $D(X) = \{\}$ for some *X*.

Valuations

- A valuation θ is a mapping of variables to values: e.g. $\{ X \text{->} 3, Y \text{->} 4 \}$
  - $\theta(X) = 3, \theta(Y) = 4$
  - $vars(\theta) = \{X,Y\}$
- We say a valuation $\theta \in D$ if
  - $\theta(X) \in D(X)$ for each $X \in vars(\theta)$
- A solution is a valuation which satisfies each constraint in the problem
- Valuation domain $D_\theta(X) = \{ \theta(X) \mid X \in vars(\theta) \}$

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Constraints

- A constraint *c* is a set of valuations (its solutions) over a set of variables *vars(c)*
  - $X \neq Y$:
    - *{ { X-> 1, Y -> 2}, { X -> 1, Y -> 3}, {X -> 2, Y -> 1}, {X -> 2, Y -> 3}, {X -> 3, Y -> 1}, {X -> 3, Y -> 2} }*
    - or *{ { X -> red, Y -> yellow}, {X -> red, Y -> blue }, … }*
  - $X = Y + 1$
    - *{ { X -> 2, Y -> 1}, {X -> 3, Y -> 2} }*

Propagators

- A propagator f for constraint c is a function from domains to domains: $D' = f(D)$
- Monotonically decreasing: $f(D)(X) \subseteq D(X)$
- Correct for c: never removes a value which occurs in a solution of c from D
  - $\theta \in D$ and $\theta \in c$ implies $\theta \in f(D)$
- Checking for c: if all variables in c are fixed then it returns a failed domain unless this is solution.
  - $f(D_\theta) = D_\theta$ iff θ is a solution of c

Propagators

- Propagator for $X = Y + 1$
- $f(D)(X) = D(X) \cap [min(D(Y))+1 .. max(D(Y))+1]$
- $f(D)(Y) = D(Y)$
- Correct, even though it never modifies $D(Y)$
- Is it checking?

Domain Propagators

- The strongest propagator for a constraint c removes all values that don't take part in a solution of c in domain D
  - $f(D(X)) = D(X) \cap \{ \theta(X) \mid \theta \in c, \theta \in D \}$
- The strongest propagator for c is called the domain propagator for c
- Write down the domain propagator for the constraint $X \neq Y$
  - $f(D)(X) = D(X) - \{d\}, D(Y) = \{d\}$
  - $f(D(X) = D(X)$, otherwise
  - Y is symmetrically defined

Linear Propagators

- Linear constraints are the most common constraint used in modelling
  - $\Sigma\, a_i\, X_i = b$ or $\Sigma\, a_i\, X_i \leq b$
- What is the result of the domain propagation of
  - $X = 3Y + 5Z$
  - $D(X) = [2..7], D(Y) = [0..2], D(Z) = [-1..2]$
  - Solutions: $(3,1,0), (5,0,1), (6,2,0)$
  - $D'(X) = \{3,5,6\}, D'(Y) = \{0,1,2\}, D'(Z) = \{0,1\}$

Linear Propagators

- The complexity of linear equation $\Sigma\, a_i X_i = b$ domain propagation is?
  - Linear *O(n)*
  - Sorting *O(n log n)*
  - Quadratic *O(n*n)*
  - NP-hard
- For linear inequality $\Sigma\, a_i X_i \leq b$ propagation it is?
  - Linear *O(n)*
  - Sorting *O(n log n)*
  - Quadratic *O(n*n)*
  - NP-hard

Bounds Propagators

- A bounds propagator only examines and sets upper and lower bounds of variable domains
- Advantage only deal with $2n$ pieces of information
- Write down a bounds propagator for the constraint $X = abs(Y)$
  - $D'(X) = D(X) \cap [0 .. m]$ where
    - $m = max(max(D(Y)),-min(D(Y))$
  - $D'(Y) = D(Y) \cap [-max(D(X)) .. max(D(X)]$
- Is this the strongest bounds propagator possible?

Linear Bounds Propagators

- The complexity of linear equation $\Sigma\, a_i X_i = b$ strongest bounds propagation is?
  - Linear $O(n)$
  - Sorting $O(n \log n)$
  - Quadratic $O(n*n)$
  - NP-hard
- The complexity of linear inequality bounds propagation is
  - Linear!

Linear Inequality

- To propagate the general linear inequality

$$\sum_{i=1..n} a_i x_i \leq b$$

- Use propagation rules (where $a_i > 0$)

$$x_i \leq \frac{b - \sum_{j=1..n, j \neq i} a_j \min(D, x_j)}{a_i}$$

Linear Equation

- To propagate the general linear inequality

$$\sum_{i=1..n} a_i x_i = b$$

- Use propagation rules (where $a_i > 0$)

$$x_i \le \frac{b - \sum_{j=1..n, j \ne i} a_j \min(D, x_j)}{a_i}$$

$$x_i \ge \frac{b - \sum_{j=1..n, j \ne i} a_j \max(D, x_j)}{a_i}$$

Linear Bounds Propagators

- Implement linear equation $\Sigma a_i X_i = b$ propagator as
  - $\Sigma a_i X_i \leq b$
  - $\Sigma a_i X_i \geq b$
- What is the result of the bounds propagation of
  - $X = 3Y + 5Z$
  - $D(X) = [2..7], D(Y) = [0..2], D(Z) = [-1..2]$
  - Smallest value of $3Y + 5Z = -5$, largest 16
  - Smallest value of $X - 5Z = -8$, largest 12
  - Smallest value of $X - 3Y = -4$, largest 7
  - $D'(X) = [2..7], D'(Y) = [0..2], D'(Z) = [0..1]$
  - Domain $D'(X) = \{3,5,6\}, D'(Y) = [0..2], D'(Z) = [0..1]$

Exercise: $X = Y \times Z$

- Suppose
  - $D(X) = [\ 0..\ 5\ ]$, $D(Y) = [\ -2\ ..\ 3]$, $D(Z) = [\ 1..6\ ]$
- What domain would a domain propagator return?
- What about
  - $D(X) = [\ 3..\ 5\ ]$, $D(Y) = [\ -2\ ..\ 3]$, $D(Z) = [\ 2..6\ ]$

Propagation Strength

- Propagators should be
  - Strong: remove as many values as possible, and
  - Efficient: execute quickly
- But in the end efficiency is much more important
- Almost no propagators are
  - the strongest possible (domain propagators)
  - or even the strongest possible bounds propagator!

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Propagation Engine

- Propagation repeatedly applied propagators $f \in F$ until all at fixpoint $f(D) = D$

isolv(Fo, Fn, D)

    F := $Fo \cup Fn$; Q := Fn

    while ($Q \neq \{\}$)

        f := choose(Q) % select next propagator to run

        Q := $Q - \{f\}$; D' := $f(D)$;

        Q := $Q \cup$ new(f,F,D,D') % add affected props

        D := D'

    return D

Propagation Engine

- choose(Q)
  - typically a FIFO queue
  - pick the propagator in the queue longest
    - Don't add the same propagator twice!
- new(f,F,D,D')
  - return propagators f' in F where $f'(D') \neq D'$
  - simplest version
    - Add propagators for constraints whose variables have changed domain
    - $\{ f \mid vars(f) \cap \{ X \mid D(X) \neq D'(X) \} \neq \{\} \}$

Propagation Example

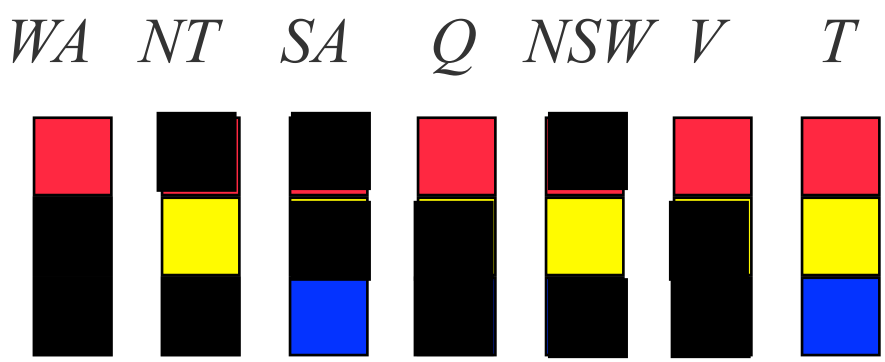
Queue Q given by boxed propagators

$WA = red$ | $NT = yellow$

$WA \neq NT$ | $WA \neq SA$ | $NT \neq SA$

$NT \neq Q$ | $SA \neq Q$ | $SA \neq NSW$

$SA \neq V$ | $Q \neq NSW$ | $NSW \neq V$



Have we found a solution?

Whats Wrong with Propagation?

- Every propagator that makes a change puts itself back on the queue
  - We would expect it to make no new change
- Most propagators wake up and make no change to domains
  - Intrinsic to propagation, but can we improve it?

Idempotence

- A propagator is idempotent if
  - $f(D) = f(f(D))$
- An idempotent propagator does not need to put itself back on the queue.
- Actually most propagators are not idempotent because of domain holes
- E.g. $X = abs(Y)$, $D(X) = \{0,2,4\}$, $D(Y) = \{-3,1\}$
  - $D' = f(D)$, $D'(X) = \{0,2\}$, $D'(Y) = \{-3,1\}$
  - $D'' = f(D')$, $D''(X) = \{0,2\}$, $D''(Y) = \{1\}$
- Dynamic idempotence: propagator returns whether it is idempotent when executed

Events

- Some domain changes will not cause a propagator to change domains
- Only wake up when an event of interest occurs
  - *fix(X)*: *X* becomes fixed
  - *lbc(X)*: lower bound of *X* changes
  - *ubc(X)*: upper bound of *X* changes
  - *dmc(C)*: the domain of *X* changes
- What events should wakeup $X \neq Y$?

Propagation Redundancy

- Sometimes we can tell that
  - $f(D) = D$
  - For all future domains D
- The usual case is redundancy
  - $D \models c$
  - All solutions of D are solutions of c
- For example:
  - once $X \neq Y$ propagates it is redundant

Propagation Example

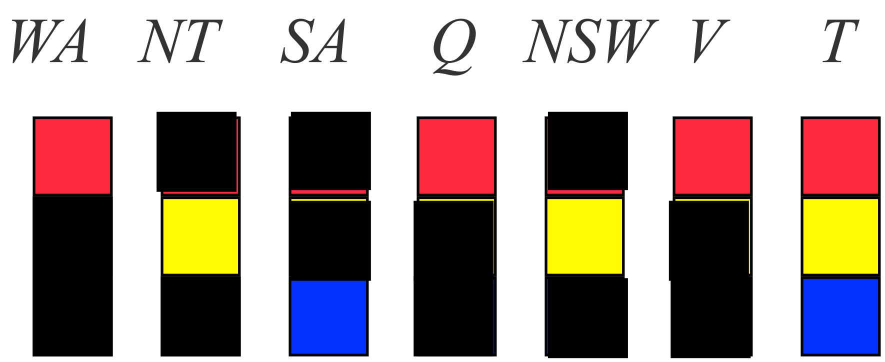
Queue *Q* given by boxed propagators

$WA = red$ $NT = yellow$

$WA \neq NT$ $WA \neq SA$ $NT \neq SA$

$NT \neq Q$ $SA \neq Q$ $SA \neq NSW$

$SA \neq V$ $Q \neq NSW$ $NSW \neq V$



11 propagations versus 21

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Propagation Solving

- A propagation solver only determines
  - Failure with a failed domain
  - Solution when $|D(X)| = 1$ for all X
- Mostly neither case holds.
- We need to add more information
  - By guessing
- Search
  - Usually we split the domain of a variable in two!

Search

search(*Fo*,*Fn*,*D*)

D := isolv(*Fo*,*Fn*,*D*)

if (*D* is a false domain) **return** false domain *D*

if ($|D(X)| = 1$ forall X) **return** *D*

(*c1*, *c2*) := choose(*D*) where $D \models c1 \vee c2$

D1 := search($Fo \cup Fn$, { *prop(c1)* }, *D*))

if (*D1* is not a false domain) **return** *D1*

D2 := search($Fo \cup Fn$, { *prop(c2)* }, *D*))

if (*D2* is not a false domain) **return** *D2*

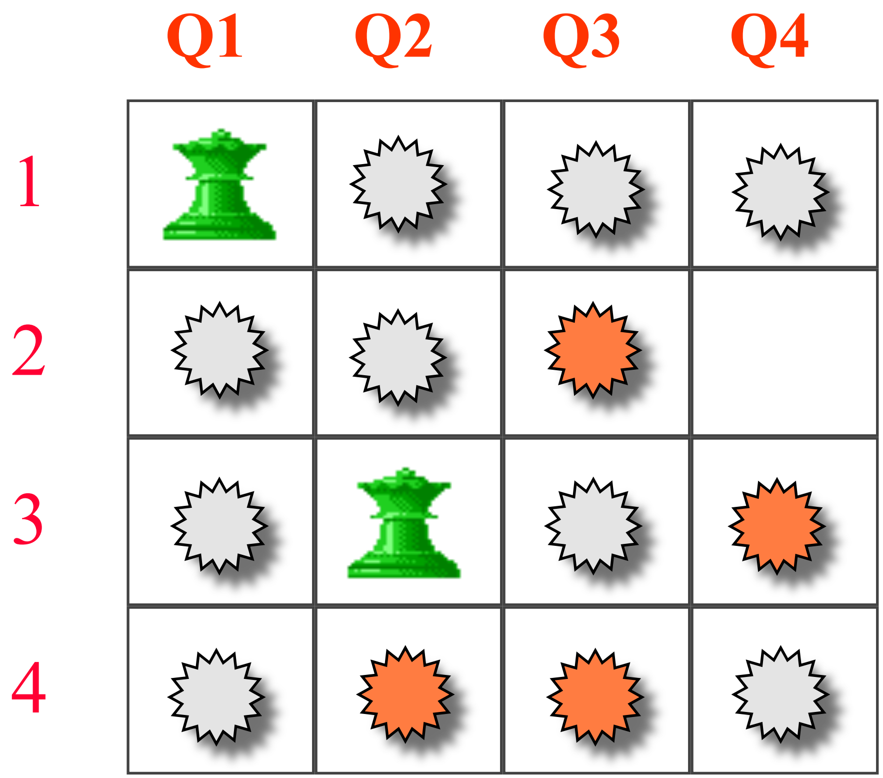
return false domain

Search Choice

- The choice of how to split the search is crucial
- Usually we choose a variable X with $|D(X)| > 1$
- And then choose a value $d \in D(X)$ and add
  - $X = d \lor X \neq d$
  - This is called labelling
- Or choose the $d \in D(X)$ and add
  - $X \leq d \lor X \geq d+1$
  - This is called domain splitting
  - But usually $d = min(D(X))$

Search -- Example

Therefore, we need to choose another value for Q2.

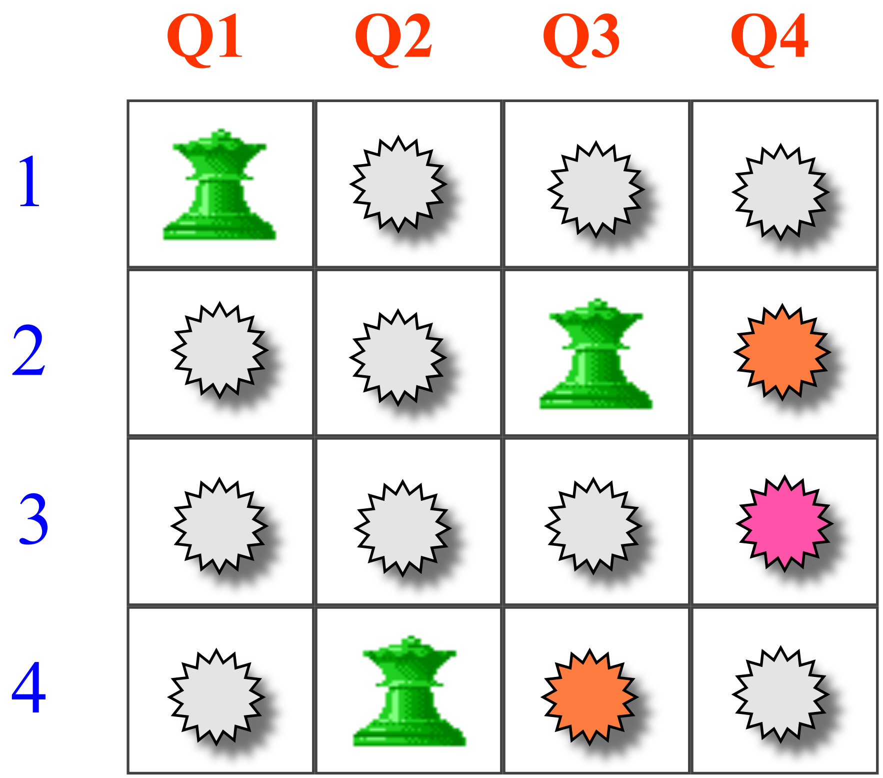


Search-- Example

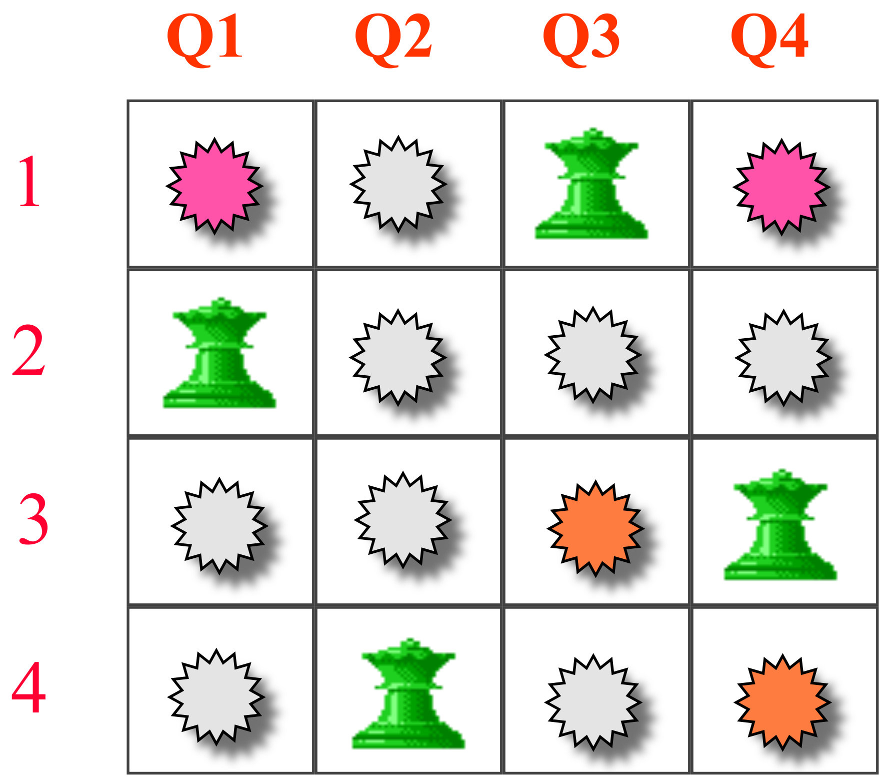
backtracking,

Find another value of Q1?

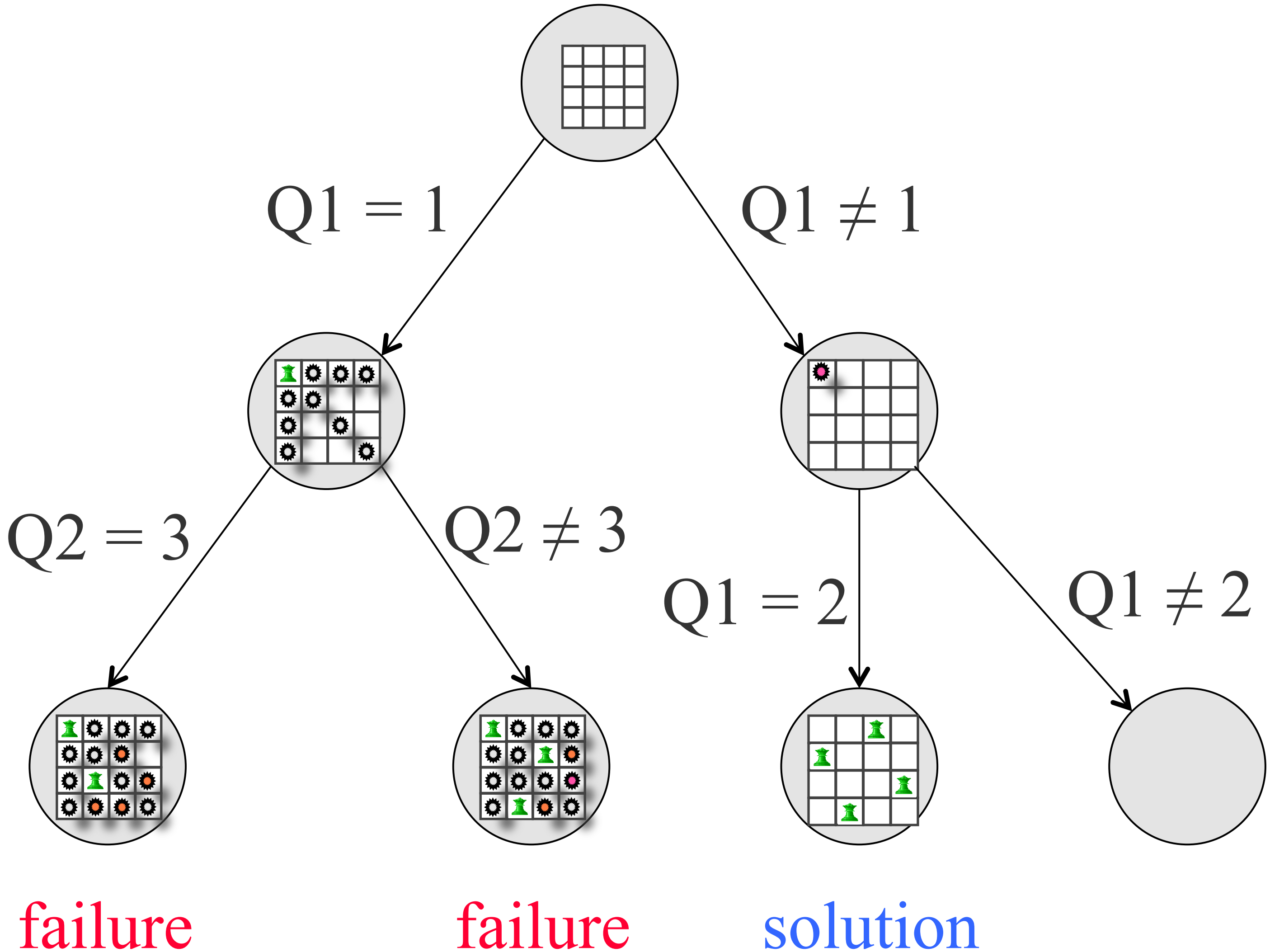
Yes, Q1 = 2



Search -- Example

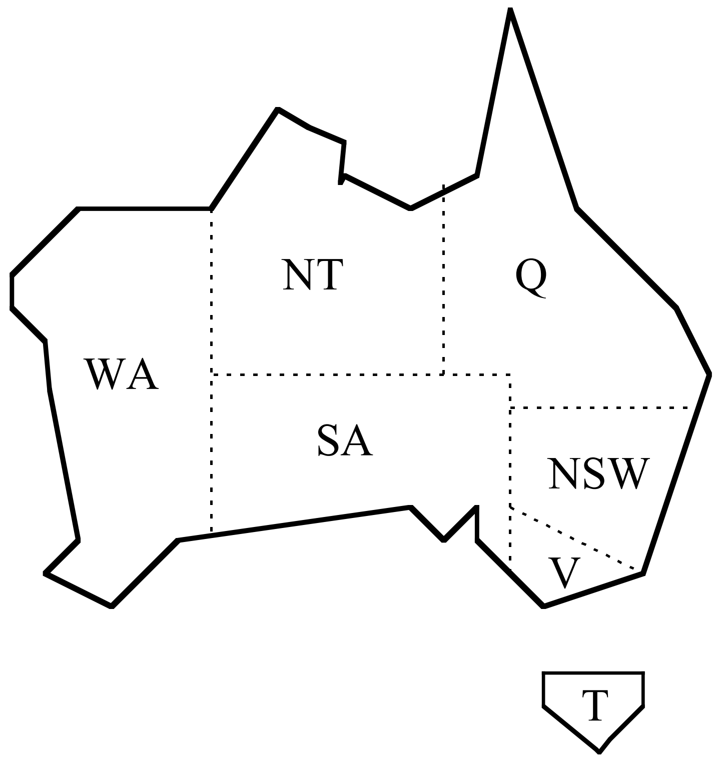


Search Tree



Search Tree Exercise

- Var: value order
- *NSW =r =y =b*
- *NT =b =r =y*
- *Q =r =y =b*
- *T =r =y =b*
- *V =r =y =b*
- *SA =r =y =b*
- *WA =r =y =b*



Programmed Search

- One the advantages of propagation solving
- The user can specify the search strategy
  - Allows them to add knowledge of where solutions lie
- The right search strategy can make an exponential difference
- Not all variables need to be labelled
  - Some will be fixed by the constraints and the rest of the search

Choices for Search Strategy

- Labelling search:
  - `int_search`(*Vars*, *Varchoice*, *Valchoice*, `complete`)
  - Choose a variable (can make an exponential difference)
    - `input_order`: in the order given e.g. *Vars = NSW, NT, …*
    - `first_fail`: choose variable *X* where |*D(X)*| is smallest
    - `smallest`: choose variable *X* where *min(D(X))* is smallest
    - `largest`: choose variable *X* where *max(D(X))* is largest
  - Choose a value (only moves solutions earlier)
    - `indomain_min`: select least possible value
    - `indomain_max`: select greatest possible value
    - `indomain_median`: select median value from domain
    - `indomain_random`: select a random value from domain

Playing with Search Strategies

- `nqueens.mzn` is a model for placing *n* queens on an *n* x *n* chessboard so none can take another
  - Available from summer school website (Exercises)
- We can run the model (for *n = 8*) like this
  - `minizinc -s -D “n = 8;” nqueens.mzn`
- It prints out a solution and the number of choices required to find it (amount of search) using default search
- We can add a programmed search strategy by changing
  - `solve satisfy;` to
  - `solve :: int_search(q,` *Varchoice*, *Valchoice*, `complete) satisfy;`
- Experiment with `nqueens.mzn` to find the most robust search strategy as *n* increases!

Playing with Search Strategies

- We can run the model (for *n = 8*) like this
  - `minizinc -s -D "n = 8;" nqueens.mzn`
- Change search using
  - `solve :: int_search(q,` *Varchoice*, *Valchoice*, `complete) satisfy;`
  - *Varchoice:* `input_order, first_fail, smallest, largest`
  - *Valchoice*: `indomain_min, indomain_max, indomain_median, indomain_random`
- Experiment with `nqueens.mzn` to find the most robust search strategy as *n* increases!

Finished Quickly

- You can find all solutions using
  - `minizinc -a -s -D “n = 8;” nqueens.mzn`
- Compare different *Valchoices* for finding all solutions for $n = 8$
  - Notice anything?

More Advanced Search

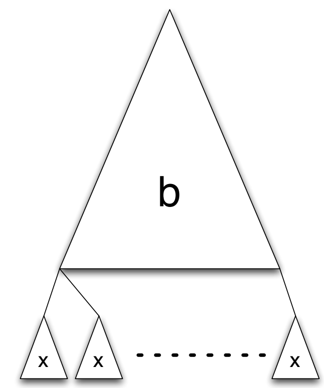
- Programmed search is an important part of CP
- Dynamic variable selection strategies:
  - dom_w_deg, impact, activity, regret, …
- Restarts:
  - Geometric, Luby, …
- Different ways to explore the search tree
  - Limited discrepancy search, breadth first, best first, …

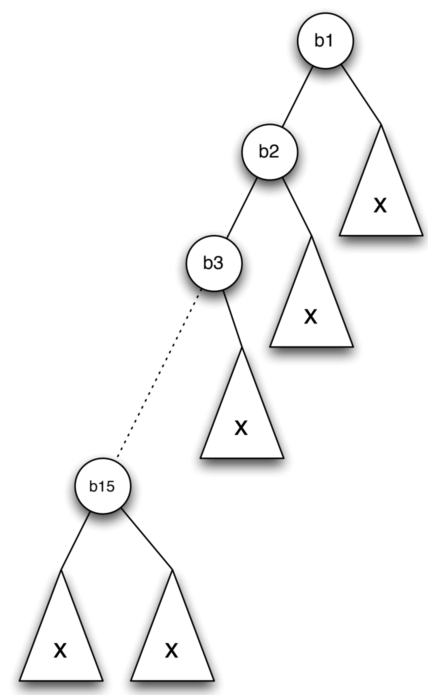
Dom_w_deg

- Domain / weighted degree
  - degree in the number of constraints the var is in
- dom_w_deg: choose a variable with minimum
  - domain size / sum of failures by constraints it is in
- Each variable gets a fail count
  - (= number of constraints it appears in initially)
- Each time a constraint detects failure
  - increment fail count for all variables involved
- Choose the variable with minimum
  - domain size / failcount

Dom_w_deg

- Why does it work
  - Concentrates on variables that are causing failure
- Imagine 15 Boolean vars *b* that are easy to solve and 4 integers *x* with no solution
- Searching with first fail
  - always chooses Booleans
  - then tries to solve integer problem
  - 491504 choices to fail
- Dom_w_deg
  - First branches chooses Booleans
  - On backtracking always chooses *x*s
  - 182 choices to fail

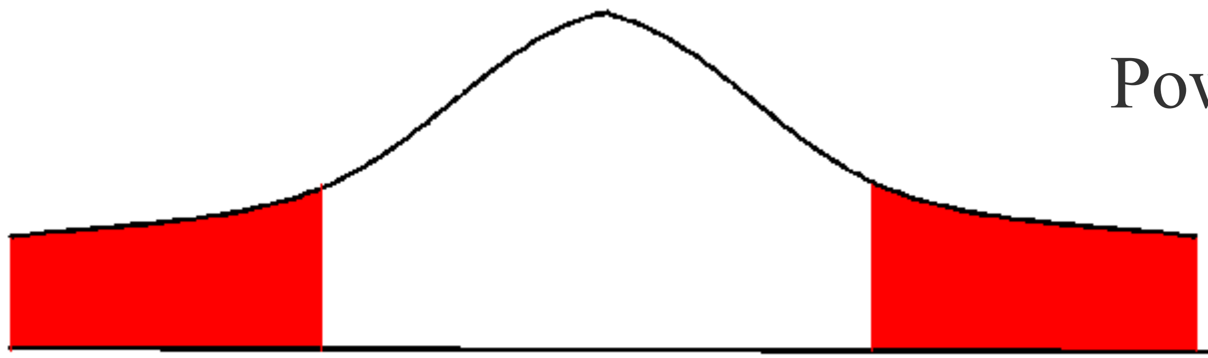




Dom_w_deg

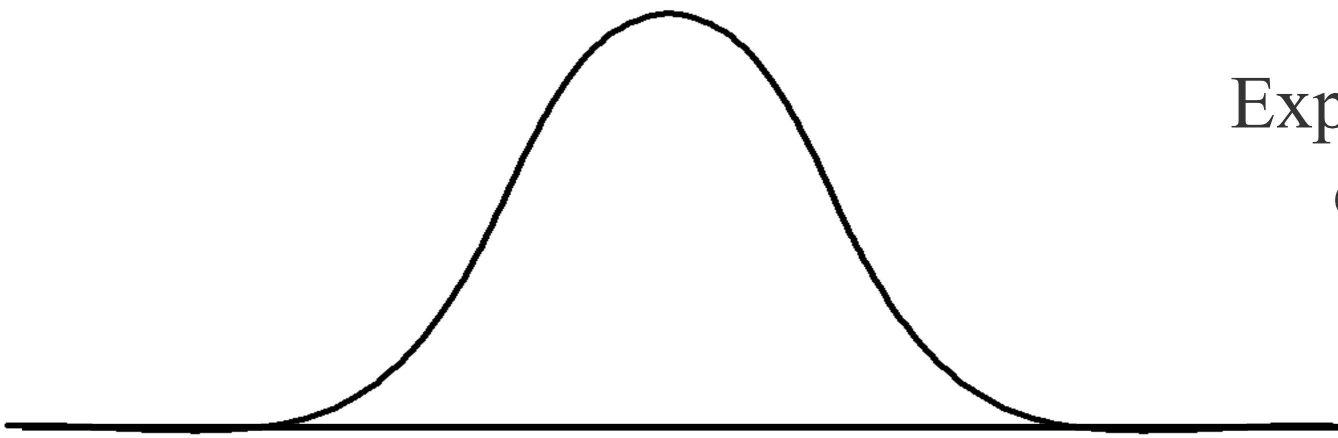
- If you are interested try the search strategy exercise using also
  - `dom_w_deg` as a *Varchoice*
- Note `dom_w_deg` is a poor approximation to the powerful search strategy
  - Activity based search!

Restarts + Heavy tails



Power series decay

HEAVY TAILED DISTRIBUTION
(infinite mean & variance)

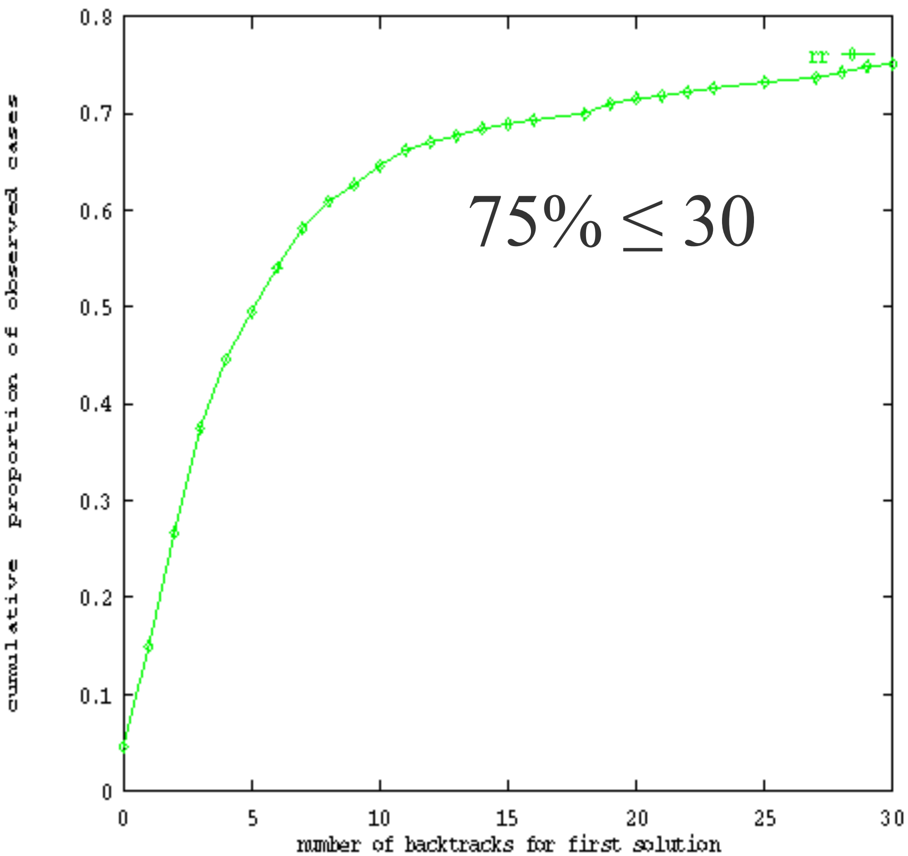


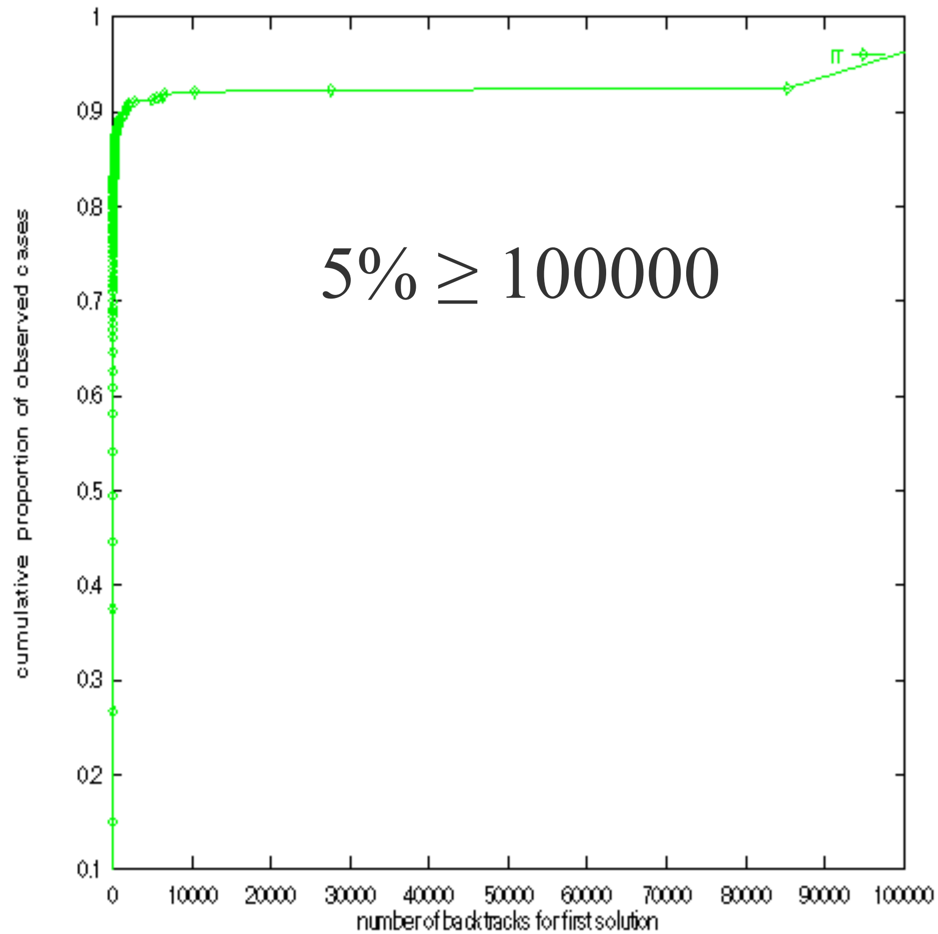
Exponential decay

Standard Distribution
(finite mean &
variance)

Heavy Tailed Behaviour

Searching for solutions to Quasigroup completion problems





Heavy-Tailed Behavior

Restarts

- If 75% finish in 30 backtracks
  - after 50 backtracks why not start again
    - trying a different search
    - here the variable and value selection is random
  - you might be in one of the 5% that require > 100,000
- Restarting conquers heavy tailed behaviour

Restart Strategies

Policy for when to restart

- Constant restart – after using *L* resources
- Geometric restart
  - restart after using *L* resources, with new limit αL
- Luby restart
  - 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8, …
  - "universally optimal" for randomized algorithms:
    - no worse than a log factor slower than optimal policy
    - not bettered by more than a constant factor by other universal policies

Restarts

- Restarts are ubiquitous in default search strategies
- Combined with dynamic variable selection strategies they have another advantage
  - A bad choice at the top requires exponential search to undo
  - Restarts avoid this, by throwing away the choice.

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Optimization for CSPs

- So far only looked at finding a solution: this is ***satisfiability***
- However often we want to find an ***optimal*** solution:
  One that minimizes/maximizes an objective function *o*.
- Because the domains are finite we can use a solver to build a simple optimizer *for minimization*

retry_int_opt(*F*, *D*, *f*, *best_so_far*)

  D2 := search(*F*,{},*D*)

  if (*D2* is a false domain) **return** *best_so_far*

  let θ be the solution corresponding to *D2*

  return retry_int_opt($F \cup \{ prop(o < \theta(o))\}, D, f, \theta$)

Retry Optimization Example

Smugglers knapsack problem (optimize profit)

minimize $-15W - 10P - 7C$ subject to

$$\underbrace{4W + 3P + 2C \leq 9}_{\textit{capacity}} \quad \wedge \quad \underbrace{15W + 10P + 7C \geq 30}_{\textit{profit}}$$

$$-15W - 10P - 7C < -31 \wedge -15W - 10P - 7C < -32$$

$$D(W) = [0..9], D(P) = [0..9], D(C) = [0..9]$$

No next solution! $D(W) = [0..0], D(P) = [1..1], D(C) = [3..3]$

$D(W) = [1..1], D(P) = [1..1], D(C) = [1..1]$

Corresponding solution $\theta = \{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$

Return best solution $\theta = \{W \mapsto 1, P \mapsto 1, C \mapsto 1\}$

$$\theta(o) = -31$$

$$\theta(o) = -32$$

Backtracking Optimization

- Since the solver may use backtracking search anyway combine it with the optimization
- At each step in backtracking search, if *best* is the best solution so far add the constraint $o < best(o)$
- Very similar to branch-and-cut methods
  - Use consistency techniques instead of linear relaxation

Backtracking Optimization (Ex.)

Smugglers knapsack problem

capacity *profit*

$$4W + 3P + 2C \le 9 \quad \wedge \quad 15W + 10P + 7C \ge 30$$

$$-15W - 10P - 7C < -31$$

Current domain:

$$D(W) = [0..0], D(P) = [1..1], D(C) = [3..3]$$

after bounds consistency

$W = 0$

$P = 1$

(0,1,3)

Solution Found: add constraint

Backtracking Optimization (Ex.)

Smugglers knapsack problem

capacity *profit*

$$4W + 3P + 2C \le 9 \quad \wedge \quad 15W + 10P + 7C \ge 30$$

$$-15W - 10P - 7C < -31 \wedge$$

$$-15W - 10P - 7C < -32$$

Initial bounds consistency

$W = 0$

$P = 1$ — **(0,1,3)**

$P = 2$ — *false*

$P = 3$ — *false*

$W = 1$ — **(1,1,1)**

$W = 2$ — *false*

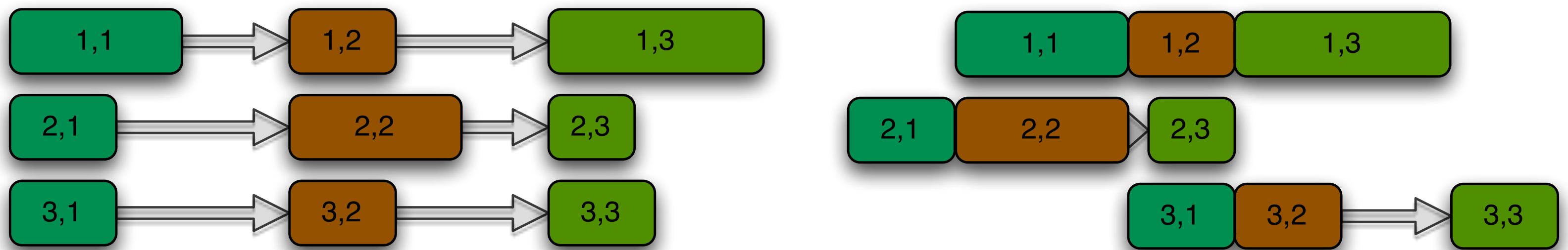
Return last sol (1,1,1)

Search and Optimization

- Programmed search is even more important for optimization
  - Finding a good solution early reduces the search space!

Jobshop Scheduling Exercise

- Scheduling tasks in order, so that only one task is on each machine at any one time



- Aim is to minimize completion time of all tasks
- Challenging problem: some 10x10 problems were unsolved only 10 years ago

Optimization Search Exercise

- You can run a 5x5 jobshop problem as
  - `minizinc -a -s jobshop.mzn`
  - `jobshop.mzn` available from school website
- Modify the search by replacing
  - `solve minimize t_end;` by
  - `solve ::` *Searchstrategy* `minimize t_end;`
- Using
  - `int_search(s,` *Varchoice*, *Valchoice*, `complete)`
  - `int_search([t_end],input_order,`*Valchoice*, `complete)`
  - `seq_search`([*IntSearch1*,*IntSearch2*])
- Find the search strategy requiring least choices to prove optimality

Overview

- Constraint Satisfaction and Optimization Problems
- Domains and Valuations
- Constraints and Propagators
- Propagation Engines
- Search
- Optimization by Satisfaction
- Global Constraints

Global Constraints

- One of the principal advantages of propagation solving
- A global constraint captures an important subproblem:
  - `alldifferent`: assignment subproblem
  - `cumulative`: resource allocation problem
- Each global constraint is implemented by (possibly several)
  - propagators
- A good implementation of a global constraints has
  - strong propagation (ideally domain propagation)
  - fast propagation
- Usually global propagators are not idempotent

Alldifferent

- *alldifferent*([V_1,...,V_n]) holds when each variable V_1,..,V_n takes a different value
- Not needed for expressiveness. *alldifferent*([*X*, *Y*, *Z*]) is equivalent to $X \neq Y \wedge X \neq Z \wedge Y \neq Z$
- But propagation doesn handle disequalities well
  - E.g. $D(X) = \{1,2\}$, $D(Y) = \{1.2\}$, $D(Z) = \{1,2\}$
- But there is a solution
  - Specialized propagator for alldifferent.

Alldifferent Propagator

Simple propagator for *alldifferent*($[V_1,\ldots,V_n]$)

f(D)

let $W = \{V_1,\ldots,V_n\}$

while (exists $V \in W$ where $D(V) = \{d\}$)

$W := W - \{V\}$

foreach ($V' \in W$)

$D(V') := D(V') - \{d\}$

$DV := \cup_{V \in W} D(V)$

if ($|DV| < |W|$) **return** *false* domain

return D

- Wakes up on $fix(V_i)$ events, idempotent
- More efficient but hardly propagates more than disequalities

Alldifferent Example

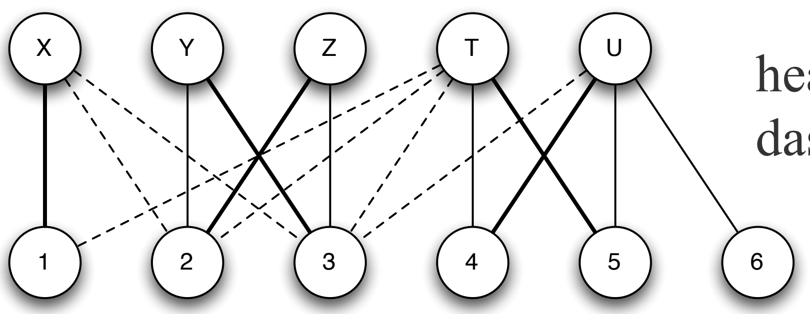
- *alldifferent([X,Y,Z])*
- $D(X) = \{1,2\}$, $D(Y) = \{1,2\}$, $D(Z) = \{1,2\}$

- $DV = \{1,2\}$, $W = \{X,Y,Z\}$
- $|DV| < |W|$ hence detects unsatisfiability.

- Note that the disequations do not!

Alldifferent Propagator

- Domain consistent propagator for *alldifferent*
  - First important global propagator $O(n^{2.5})$
  - Based on maximal matching, wakes on *dmc()* events
- *alldifferent*($[X,Y,Z,T,U]$)
- $D(X) = \{1,2,3\}$, $D(Y) = \{2,3\}$, $D(Z) = \{2,3\}$, $D(T) = \{1,2,3,4,5\}$, $D(U) = \{3,4,5,6\}$

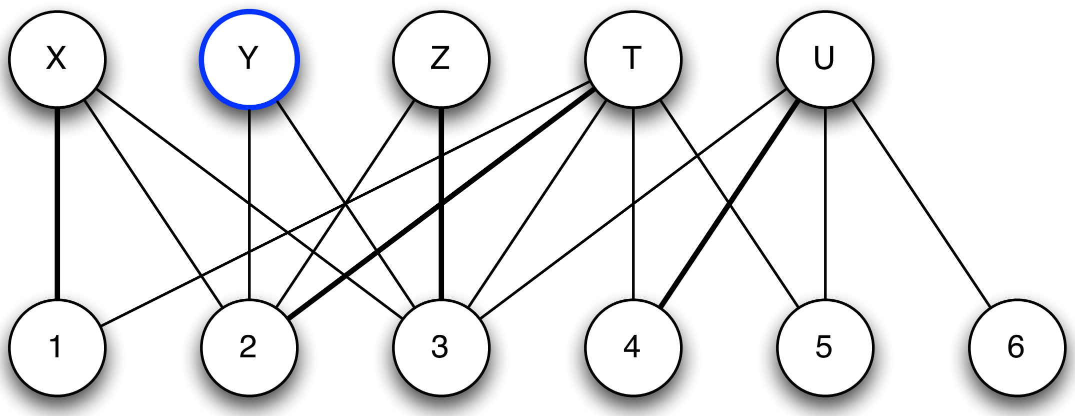


heavy = maximal matching
dashed = cant be in max matching

- $D'(X) = \{1\}$, $D'(Y) = \{2,3\}$, $D'(Z) = \{2,3\}$, $D'(T) = \{4,5\}$, $D'(U) = \{4,5,6\}$

Maximal Matching

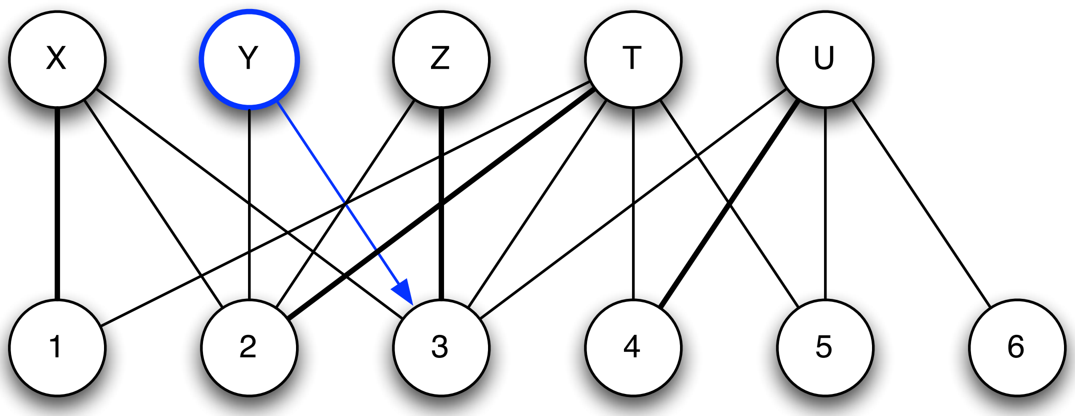
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

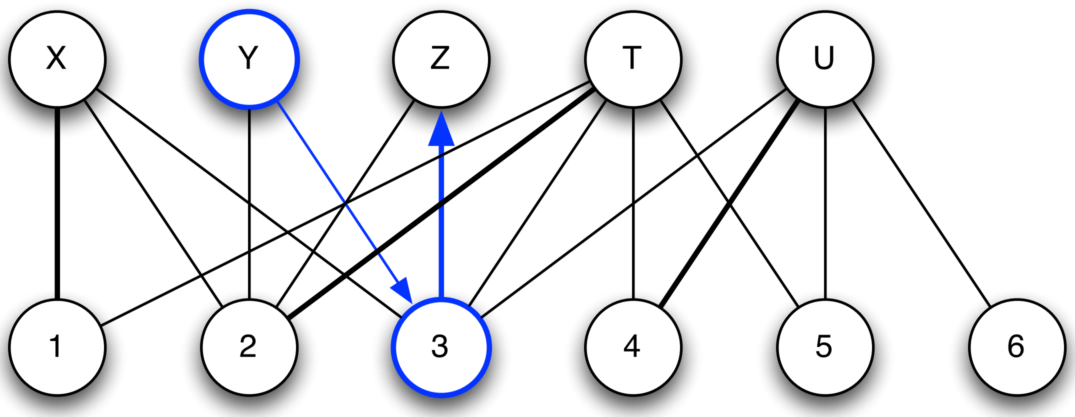
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

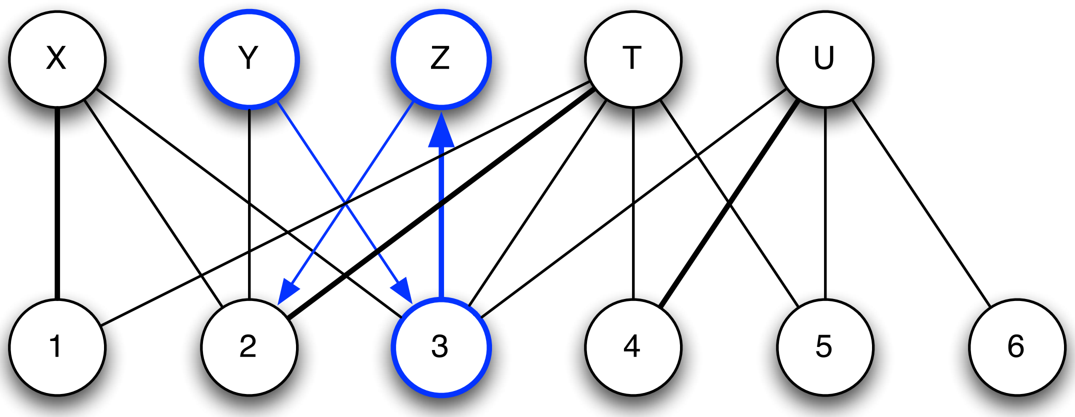
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

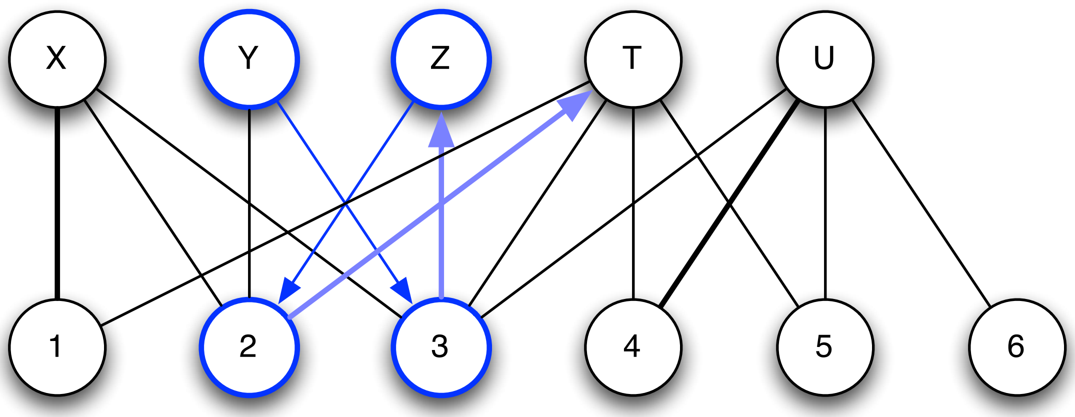
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

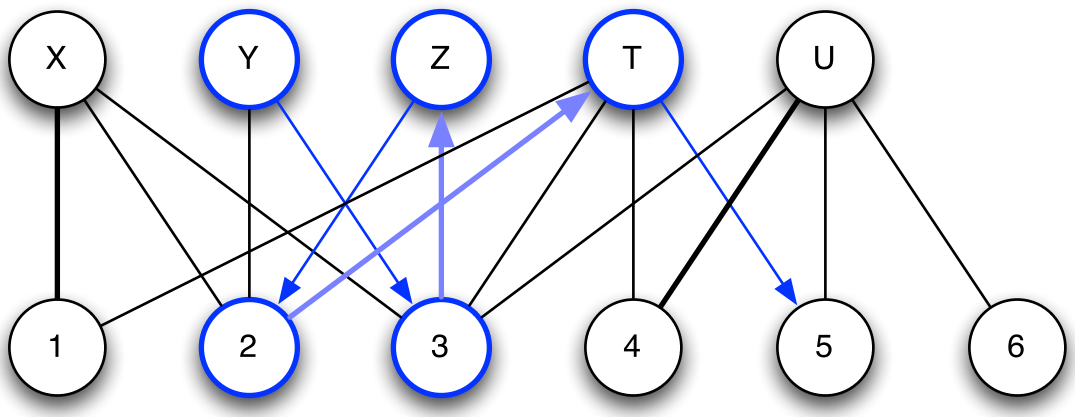
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

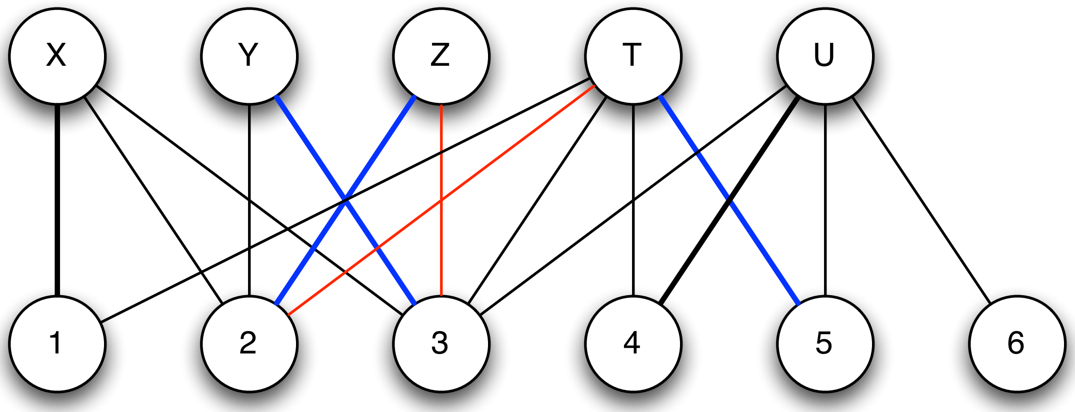
- Start with a given partial matching
- Choose an unmatched variable



- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Maximal Matching

- Start with a given partial matching
- Choose an unmatched variable

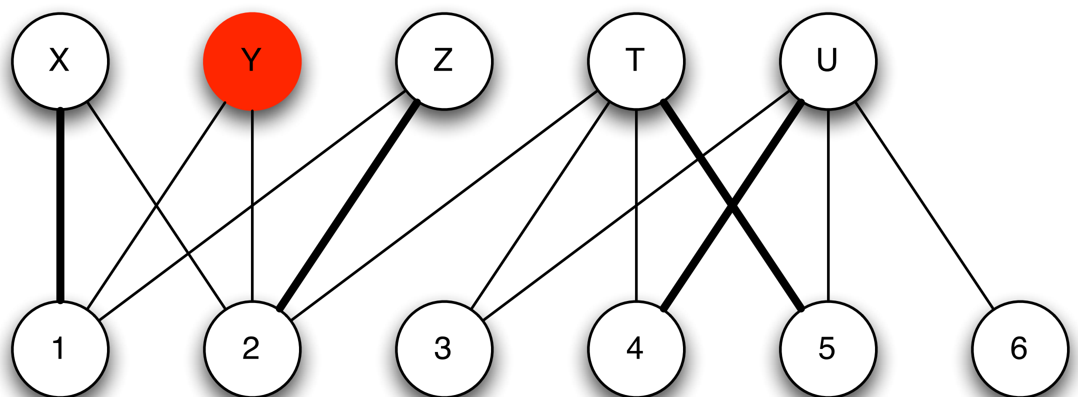


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

Failure

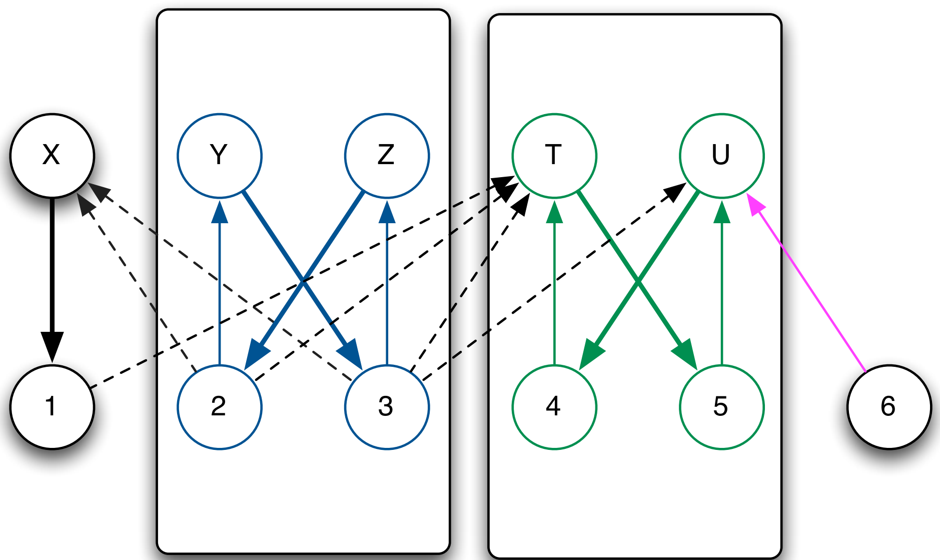
- If not every variable is matched in the maximal matching then the alldifferent constraint cannot be satisfied.

$$alldifferent([X,Y,Z,T,U])$$
$$D(X) = \{1,2\}, D(Y) = \{1,2\}, D(Z) = \{1,2\},$$
$$D(T) = \{2,3,4,5\}, D(U) = \{3,4,5,6\}$$



Propagation

- Keep edges which are reachable from unmatched nodes (pink + green)



- Keep edges in an SCC or in matching, delete rest
- $D'(X) = \{1\}$, $D'(Y) = \{2,3\}$, $D'(Z) = \{2,3\}$, $D'(T) = \{4,5\}$, $D'(U) = \{4,5,6\}$

Alldifferent

- Given the domain $D(X) = \{2,4\}$, $D(Y) = \{1,3,5,6\}$, $D(Z) = \{1,2,3\}$, $D(T) = \{2,4\}$, $D(U) = \{1,2,3,4\}$
- What is the result of propagating *alldifferent([X,Y,Z,T,U])*?
- Draw the matching graph and work it out!

Alldifferent Propagator

- bounds consistent propagator for *alldifferent*
  - Most common implementation $O(n \log n)$
  - Based on maximal matching, wakes on *lbc(), ubc()* events
- Usually as fast as the naïve first propagator

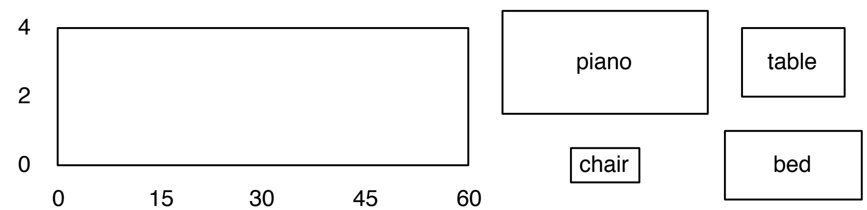
Cumulative

- *cumulative*($[S_1,\ldots,S_n], [D_1,\ldots,D_n], [R_1,\ldots,R_n], L$) schedule n tasks with start times S_i and durations D_i needing R_i units of a single resource where L units are available at each moment.
- Very complex propagator
- Many different implementations
  - Different complexities
  - None implement strongest bounds or domain propagation!

Cumulative Example

Bernd is moving house. He has 4 people to do the move and must move in one hour. He has the following furniture: piano must be moved before bed

Item	Time	No. of people
piano	30 min	3
chair	10 min	1
bed	20 min	2
table	15 min	2



How can we model this?

$D(P) = D(C) = D(B) = D(T) = [0..60]$, $P + 30 \leq B$,

$P + 30 \leq 60$, $C + 10 \leq 60$, $B + 15 \leq 60$, $T + 15 \leq 60$,

$cumulative([P,C,B,T], [30,10,20,15], [3,1,2,2], 4)$

Cumulative Timetable Propagator

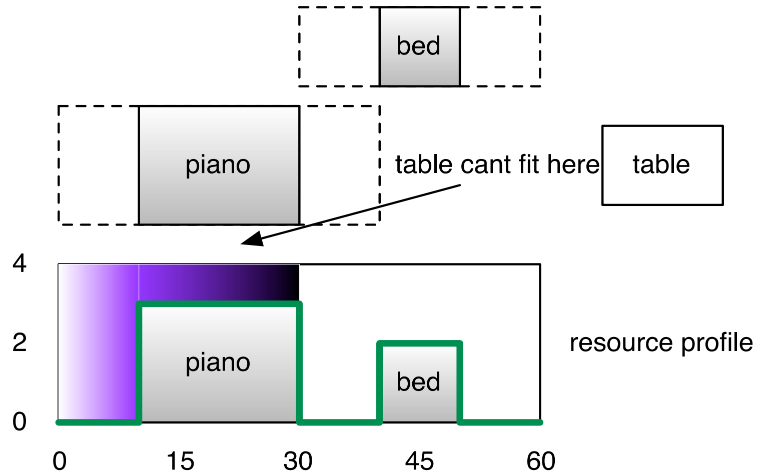
- Determine the parts where a task must be running
- The resource profile adds up these parts
- Use profile to move other tasks

Example: after initial bounds

$D(P) = [0..30]$, $D(C) = [0..50]$, $D(B) = [0..40]$, $D(T) = [0..45]$

Propagating $P + 30 \leq B$

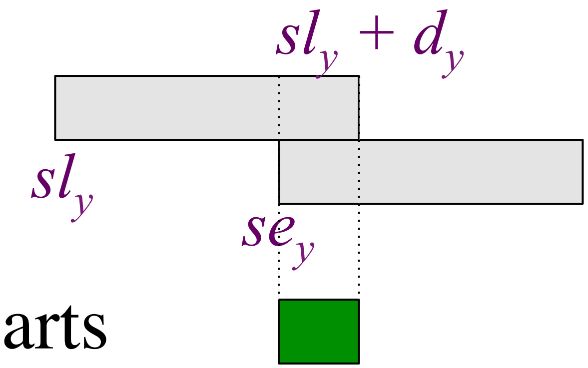
$D(P) = [0..10]$, $D(C) = [0..50]$, $D(B) = [30..40]$, $D(T) = [0..45]$



$D(P) = [0..15]$, $D(C) = [0..50]$,
$D(B) = [30..40]$, $D(T) = [30..45]$

Compulsory Parts

- A task y with earliest start time se_y, latest start time sl_y, and duration d_y
  - compulsory part: sl_y .. $se_y + d_y$



- Profile = sum of compulsory parts
- Failure: at time t profile goes over resource bound
- Propagation
  - If resources for task x don't fit at time $sl_x \leq t < sl_x + d_x$
    - move sl_x to $t + 1$
  - similarly move se_x back to $t\text{-}d_x$ if $se_x \leq t < se_x + d_x$

Cumulative by Decomposition

- We can implement cumulative using simpler constraints
  - $B_{it} \Leftrightarrow (S_i \geq t \wedge S_i + D_i < t)$
  - Task i is active at time t
  - At all times t, $\Sigma_{i \in 1..n} B_{it} \times R_i \leq L$
- Decomposition propagates like timetable
  - But $O(n\ t_{max})$ where n is number of tasks and t_{max} is maximum time horizon
  - Versus $O(n^2)$ for the global propagator
- Very many Boolean vars introduced $O(n\ t_{max})$

Cumulative exercise

- `rcpsp.mzn` is a classic cumulative resource problem
- We can try different implementations of `cumulative`
  - Cumulative by decomposition: `minizinc`
  - Cumulative propagator: `mzn-g12fd`
  - Annotate the `cumulative` constraints
    - `:: histogram_filtering`: time-tabling bounds propagator
    - `:: edge_finding_filtering`: edge-finding bounds propagator $O(n^2 * k)$
    - `:: ext_edge_finding_filtering`: extended edge-finding bounds propagator $O(n^2 * k)$
    - `:: energy_feasibility_check`: edge-finding consistency check $O(n^2)$
  - You can annotate with more than one!
    - :: annot1 :: annot2

Cumulative exercise

- Try different cumulative annotations to find the least choice points required for finding the optimal solution to
  - `mzn-g12fd -s -a rcpsp.mzn` *data.dzn*

  using data
  - `B12002.dzn`
  - `J30_10_5.dzn`
- How do they compare against the decomposition
  - `minizinc -s -a rcpsp.mzn` *data.dzn*

Priorities

- Once we have expensive global constraints
  - Need to reconsider which propagator to run next!
- Expensive global constraints should be chosen last
- Priority queue:
  - Pick the least expensive propagator available
  - Typically few priority levels
    - Unary, binary, ternary, linear, quadratic, cubic, veryslow
- E.g. $X \neq Y$ (binary), $X = Y + Z$ (ternary), *alldifferent* domain propagator (cubic)

Staged Propagators

- With priorities we can run more than one propagator for the same constraint
  - Simple *alldifferent* (linear)
  - Bounds *alldifferent* (quadratic)
  - Domain *alldifferent* (cubic)
- Better yet communicate
  - If a higher priority stage notes that the later stage cannot do anything, it is not run
  - These are called staged propagators

Priorities and Staging

- Priorities and Staging increase the amount of propagators executed
  - We need to reach a fixpoint at each level before proceeding
- But they reduce time
  - Better to let cheap propagators determine all information for a slow global before it executes
  - Instead of executing it multiple times!

Summary

- Constraint programming is based on backtracking search
- Reduce the search using propagation
  - incomplete inference but faster
- Optimization in CP is based on a branch & bound with a backtracking search
- Very general approach, not restricted to linear constraints.
- Programmer can add new global constraints and program their propagation behaviour.
- State-of-the-art solutions for many combinatorial optimization problems: scheduling, routing, rostering …
- A good basis for hybridization (the highest level model)

Lazy Clause Generation

- Repeatedly run propagators
- Propagators change variable domains by:
  - removing values
  - changing upper and lower bounds
  - fixing to a value
- Run until fixpoint.

KEY INSIGHT:

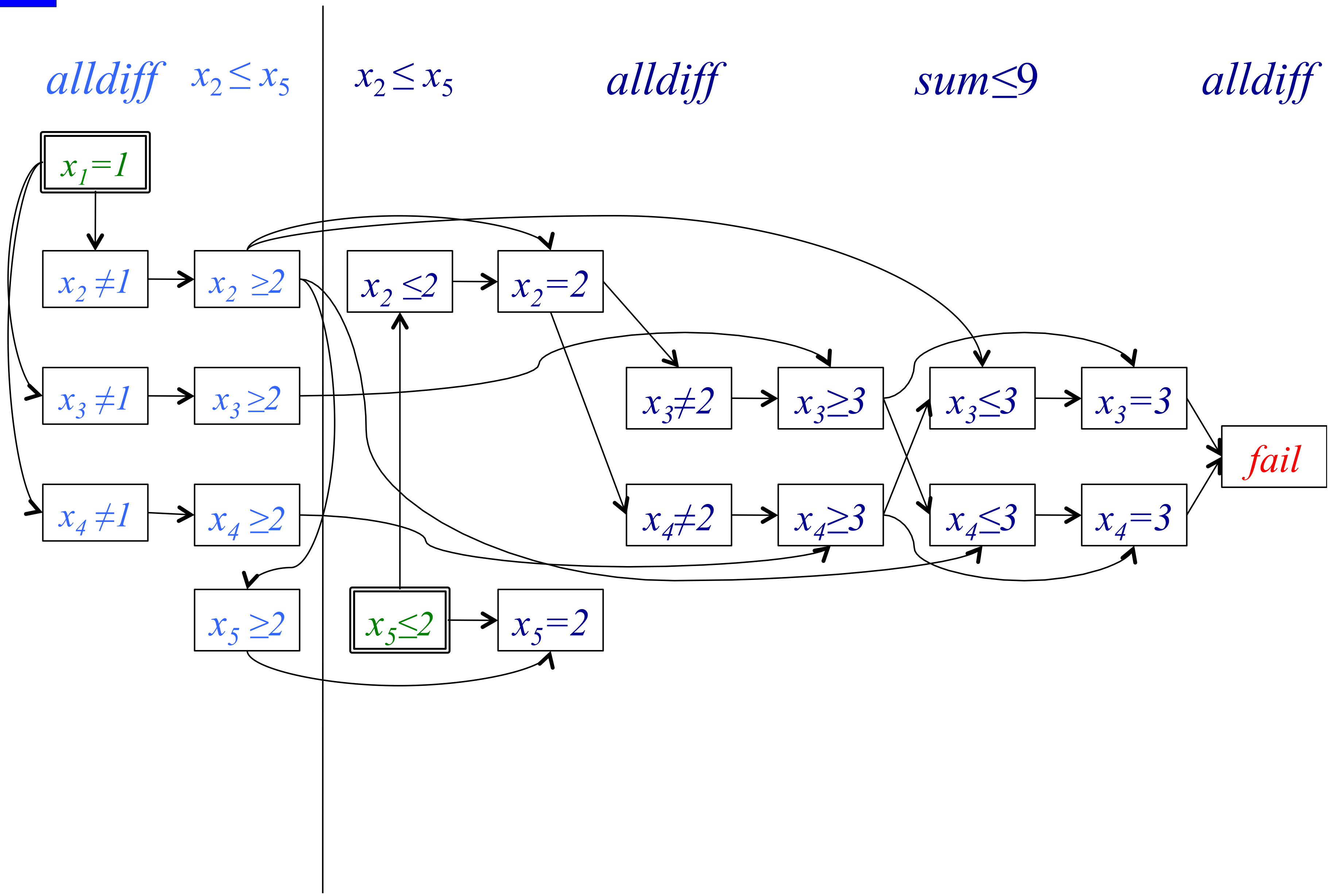
- Changes in domains are really the fixing of Boolean variables representing domains.
- Propagation is just the generation of clauses on these variables.
- FD solving is just SAT solving: conflict analysis for FREE!

Finite Domain Propagation Ex.

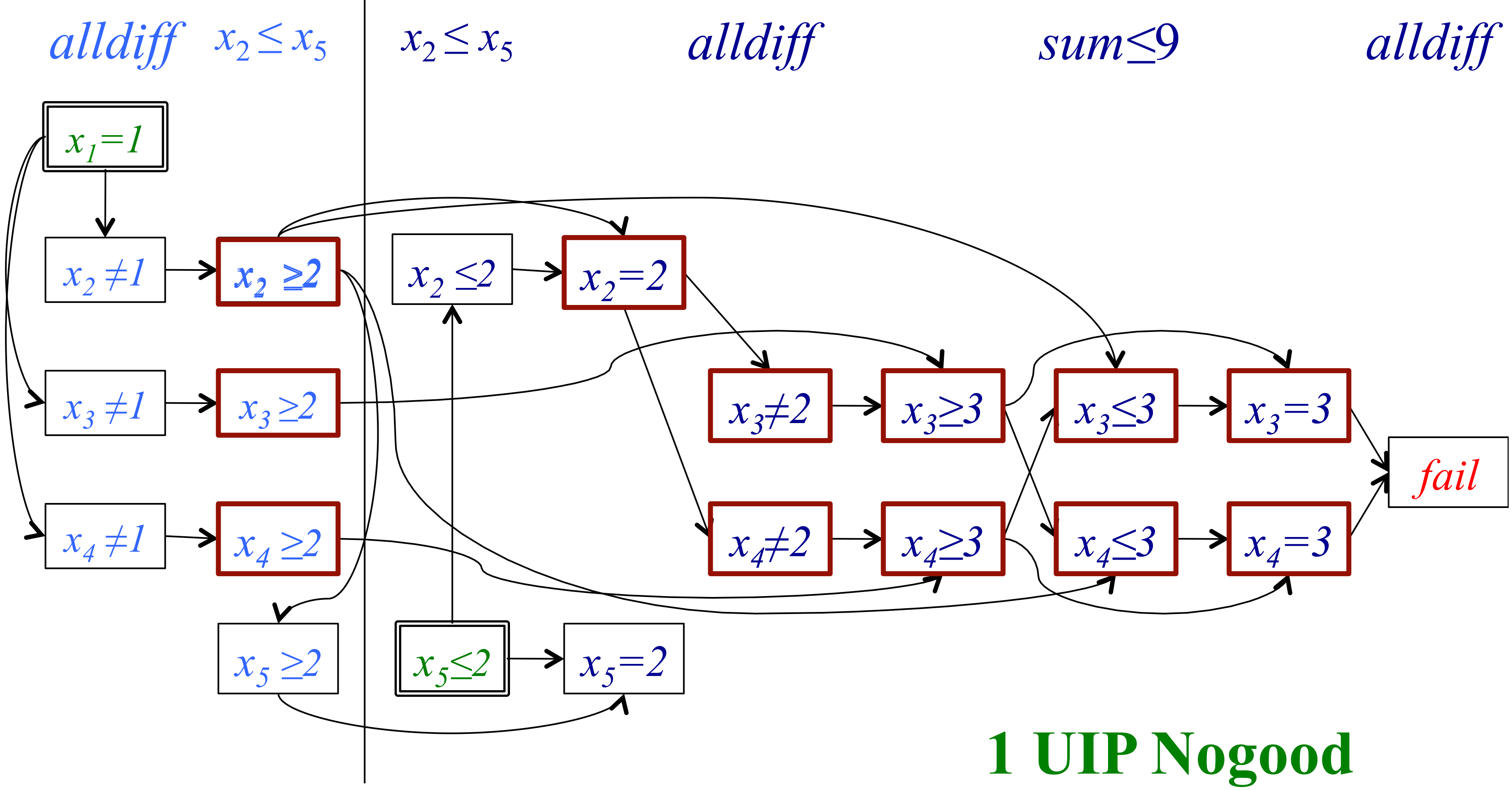
- $D(x_1) = D(x_2) = D(x_3) = D(x_4) = D(x_5) = \{1..4\}$
- $x_2 \leq x_5$, alldifferent$([x_1, x_2, x_3, x_4])$, $x_1+x_2+x_3+x_4 \leq 9$

	$x_1=1$	*alldiff*	$x_2 \leq x_5$	$x_5>2$	$x_2 \leq x_5$	*alldiff*	*sum*≤ 9	*alldiff*
x_1	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*
x_2	*1..4*	*2..4*	*2..4*	*2..4*	*2*	*2*	*2*	*2*
x_3	*1..4*	*2..4*	*2..4*	*2..4*	*2..4*	*3..4*	*3*	✖
x_4	*1..4*	*2..4*	*2..4*	*2..4*	*2..4*	*3..4*	*3*	✖
x_5	*1..4*	*1..4*	*2..4*	*3..4*	*2*	*2*	*2*	*2*

Lazy Clause Generation Ex.



1UIP Nogood Creation

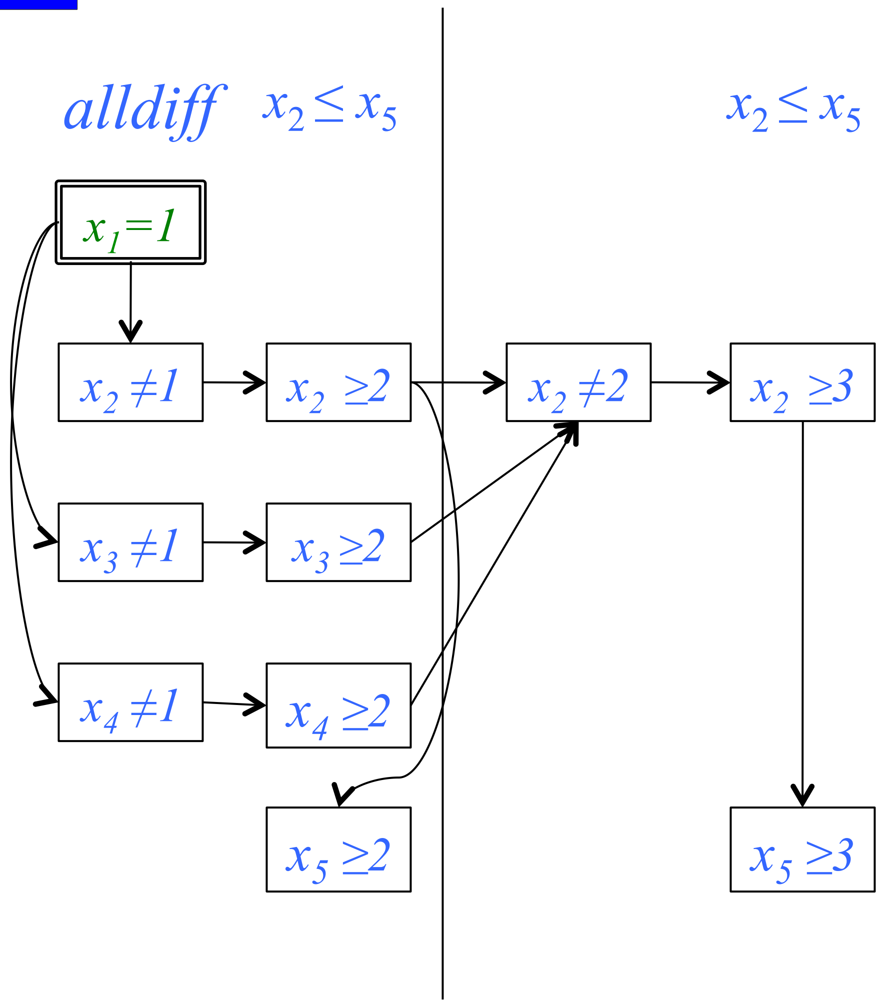


$\{x_2 \geq 2,\ x_3 \geq 2,\ x_4 \geq 2,\ x_2 = 2\} \rightarrow$ *false*

1 UIP Nogood

$\{[[x_2 \leq 1]], [[x_3 \leq 1]], [[x_4 \leq 1]], \neg [[x_2 = 2]]\}$

Backjumping



- Backtrack to second last level in nogood
- Nogood will propagate
- Note stronger domain than usual backtracking
  - $D(x_2) = \{3..4\}$

$\{x_2 \geq 2, x_3 \geq 2, x_4 \geq 2, x_2 = 2\} \rightarrow$ *false*

Whats Really Happening

- A high level "Boolean" model of the problem
- Clausal representation of the Boolean model is generated "as we go"
- All generated clauses are redundant and can be removed at any time
- We can control the size of the active "Boolean" model

Activity-based search

- An excellent default search!
- Weak at the beginning (no meaningful activities)
- Need hybrid approachs
  - Hot Restart:
    - Start with programmed search to “initialize” meaningful activities.
    - Switch to activity-based after restart
  - Alternating
    - Start with programmed search, switch to activity-based on restart
    - Switch search type on each restart
- Much more to explore in this direction

Strengths + Weaknesses

- Strengths
  - High level modelling
  - Learning avoids repeating the same subsearch
  - Strong autonomous search
  - Programmable search
  - Specialized global propagators (but requires work)
- Weaknesses
  - Optimization by repeated satisfaction search
  - Overhead compared to FD when nogoods are useless

LCG Exercise

- Try the three previous exercises before using
  - `mzn-g12lazy`
  - `mzn-g12cpx`

  instead of `minizinc` or `mzn-g12fd`
- What do you notice?

Symbols

Symbols: ∈ ∞ ∪ ⊆ ∩ ⇔ θ