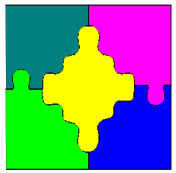
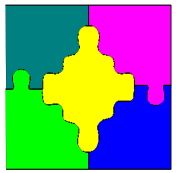


Finite Domain Propagation

A very general method to tackle discrete combinatorial optimization problems



Solving Discrete Problems

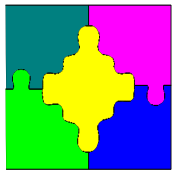
Linear programming solves *continuous* problem —problems over the real numbers.

Discrete problems are problems over the integers

Discrete problems are harder than continuous problems.

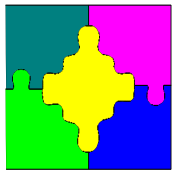
We will look at the following discrete solving methods

- Finite Domain Propagation
- Integer Programming
- Network Flow Problems
- Boolean Satisfiability
- Local Search



Overview

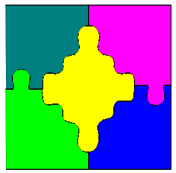
- Constraint satisfaction problems (CSPs)
- A backtracking solver
- Node and arc consistency
- Bounds consistency
- Propagators and Propagation Solving
- Consistency for sets
- Combining consistency methods with backtracking
- Optimization for arithmetic CSPs



Constraint Satisfaction Problems

- A ***constraint satisfaction problem (CSP)*** consists of:
  - A ***conjunction of primitive constraints*** C over variables $X_1,\ldots, X_n$
  - A ***domain*** D which maps each variable X_i to a set of possible values $D(X_i)$
- It is understood as the constraint

  $C \wedge X_1 \in D(X_1) \wedge \ldots \wedge X_n \in D(X_n)$
- CSPs arose in Artificial Intelligence (AI) and where one of the reasons consistency techniques were developed.

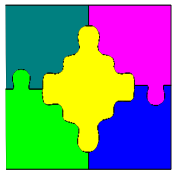


Constraint Satisfaction Problems

- Many of the problems we looked at in

  Modelling and Advanced Modelling

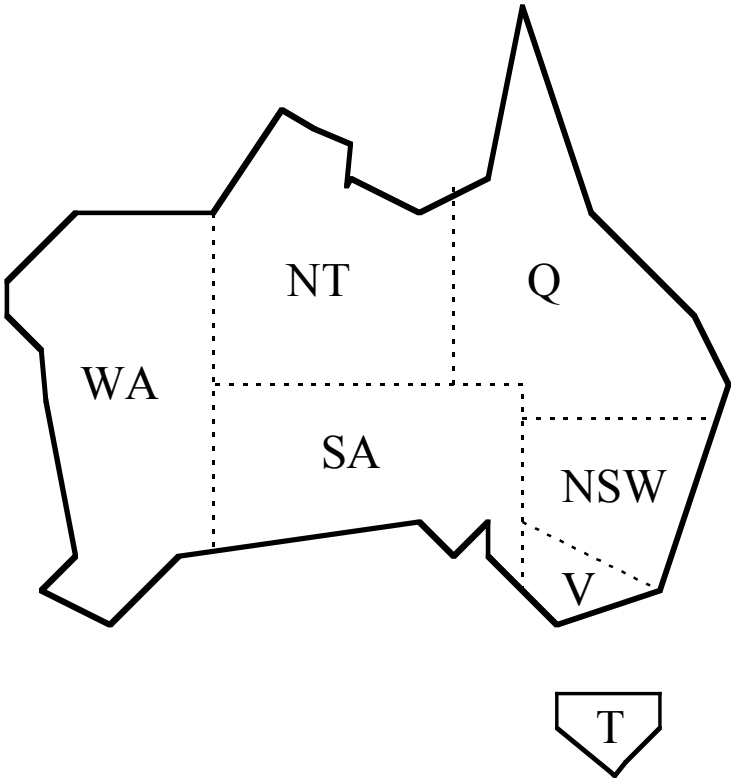
  were constraint satisfaction problems
  - graph coloring (Australia)
  - stable marriage
  - send-more-money
  - n-queens
- Those with solve satisfy



Map Colouring

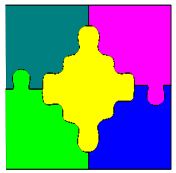
A classic CSP is the problem of coloring a map so that no adjacent regions have the same color

Can the map of Australia be colored with 3 colors ?



$$WA \neq NT \wedge WA \neq SA \wedge NT \neq SA \wedge$$
$$NT \neq Q \wedge SA \neq Q \wedge SA \neq NSW \wedge$$
$$SA \neq V \wedge Q \neq NSW \wedge NSW \neq V$$

$$D(WA) = D(NT) = D(SA) = D(Q) =$$
$$D(NSW) = D(V) = D(T) =$$
$$\{red, yellow, blue\}$$

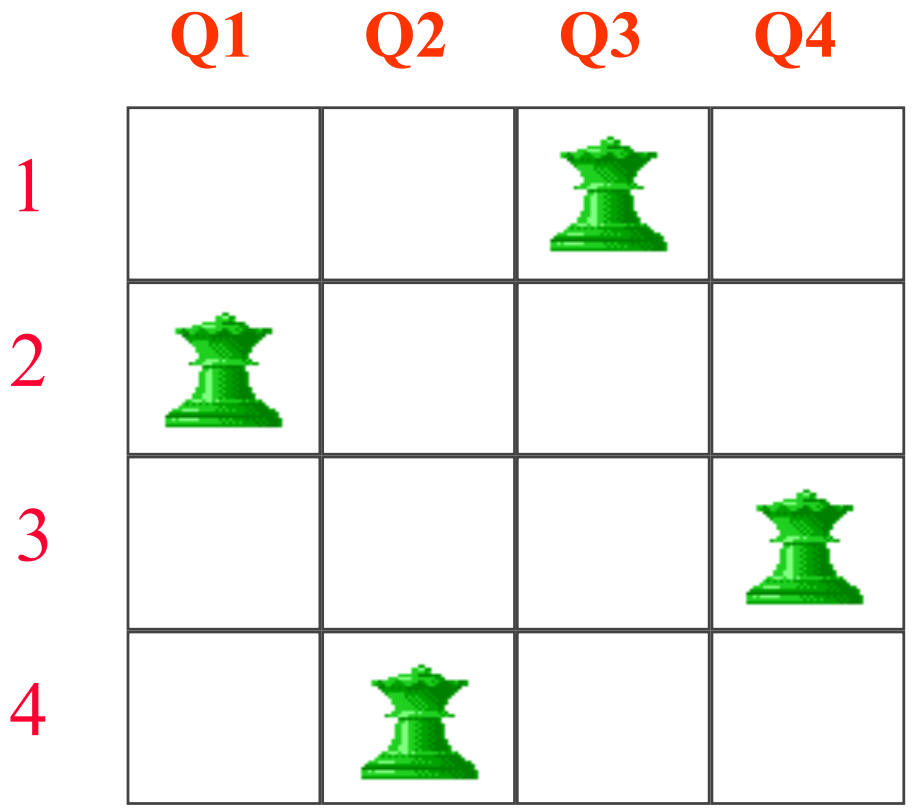


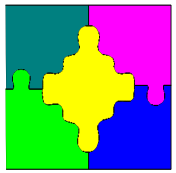
4-Queens

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! -->





4-Queens (Cont.)

The constraints:

Not on the same row

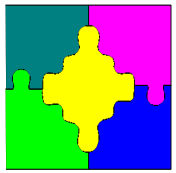
$$Q1 \neq Q2 \wedge Q1 \neq Q3 \wedge Q1 \neq Q4 \wedge$$
$$Q2 \neq Q3 \wedge Q2 \neq Q4 \wedge Q3 \neq Q4 \wedge$$

Not diagonally up

$$Q1 \neq Q2 + 1 \wedge Q1 \neq Q3 + 2 \wedge Q1 \neq Q4 + 3 \wedge$$
$$Q2 \neq Q3 + 1 \wedge Q2 \neq Q4 + 2 \wedge Q3 \neq Q4 + 1 \wedge$$

Not diagonally down

$$Q1 \neq Q2 - 1 \wedge Q1 \neq Q3 - 2 \wedge Q1 \neq Q4 - 3 \wedge$$
$$Q2 \neq Q3 - 1 \wedge Q2 \neq Q4 - 2 \wedge Q3 \neq Q4 - 1$$



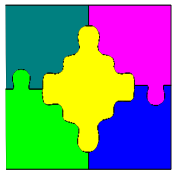
Smuggler's Knapsack

A smuggler with a knapsack with capacity 9, needs to choose items to smuggle to make a profit of at least 30

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

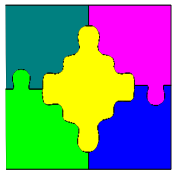
$$4W + 3P + 2C \leq 9 \wedge 15W + 10P + 7C \geq 30$$

What should be the domains of the variables?



Simple Backtracking Solver

- The simplest way to solve CSPs is to enumerate all possible solutions and try each in turn
- The ***backtracking solver***:
  - Iterates through the values for each variable in turn
  - Checks that no primitive constraint is false at each stage
- Assume *satisfiable(c)* returns *false* when primitive constraint *c* with no variables is unsatisfiable
- We let *vars(C)* return the variables in a constraint *C*.



Partial Satisfiable

The function partial_satisfiable(C) checks whether constraint C is unsatisfiable due to it containing a primitive constraint with no variables which is unsatisfiable

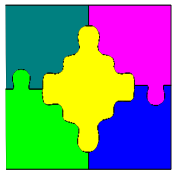
partial_satisfiable(C)

for each primitive constraint c in C

if *vars(c)* is empty

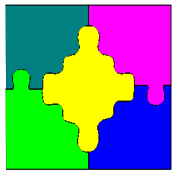
if *satisfiable(c) = false* **return** *false*

return *true*



Simple Backtracking Solver

```
back_solve(C,D)
    if vars(C) is empty return  partial_satisfiable(C)
    choose x in vars(C)
    for each value d in D(x)
        let C1 be C with x replaced by d
        if partial_satisfiable(C1) then
            if back_solve(C1,D) then return true
    return false
```



Backtracking Solve

$X < Y \wedge Y < Z \qquad D(X) = D(Y) = D(Z) = \{1,2\}$

$X < Y \wedge Y < Z$

unsatisfiable

Choose X

$X = 1$ $\qquad$ $X = 2$

$1 < Y \wedge Y < Z$ $\qquad$ $2 < Y \wedge Y < Z$

Choose Y

$Y = 1$ $\qquad$ $Y = 2$ $\qquad$ $Y = 1$ $\qquad$ $Y = 2$

$1 < 1 \wedge 1 < Z$ $\qquad$ $1 < 2 \wedge 2 < Z$ $\qquad$ $2 < 1 \wedge 1 < Z$ $\qquad$ $2 < 2 \wedge 2 < Z$

Choose Z

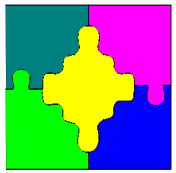
$Z = 1$ $\qquad$ $Z = 2$

$1 < 2 \wedge 2 < 1$ $\qquad$ $1 < 2 \wedge 2 < 2$

partial_satisfiable *false*

partial_satisfiable *false*

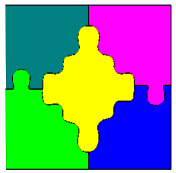
partial_satisfiable *false*



Consistency Methods

- Unfortunately the worst-case time complexity of simple backtracking is ***exponential***
- Instead we can use ***fast*** (polynomial time) but ***incomplete*** constraint solving methods:
  - Can return *true*, *false* or *unknown*
- One class are the ***consistency methods***:
  - Find an equivalent CSP to the original one with smaller variable domains
  - If any domain is empty the original problem is unsatisfiable
  - (**Similar to tightening constraints in Branch&Cut later!**)
- Consistency methods are ***local***-- they examine each primitive constraint in isolation
- We shall look at consistency techniques:

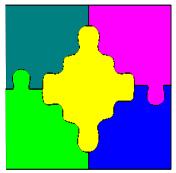
  Node ***Arc*** ***Bounds*** ***Domain***



Node Consistency

- Primitive constraint c is ***node consistent*** with domain D
  - if $|vars(c)| \neq 1$ or
  - if $vars(c) = \{x\}$ then

    for each d in $D(x)$, $\{x \rightarrow d\}$ is a solution of c
- A CSP is ***node consistent*** if each primitive constraint in it is node consistent



Node Consistency Examples

This CSP is not node consistent

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

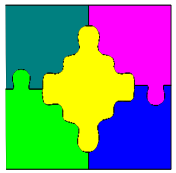
$$D(X) = D(Y) = D(Z) = \{1,2,3,4\}$$

This CSP is node consistent

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = \{1,2,3,4\},\ D(Z) = \{1,2\}$$

The map coloring and 4-queens CSPs are node consistent.
Why?



Achieving Node Consistency

node_consistent(C,D)

for each primitive constraint c in C

D := node_consistent_primitive(c, D)

return D

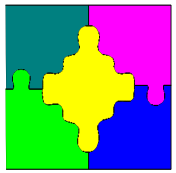
node_consistent_primitive(c, D)

if $|vars(c)| = 1$ **then**

let $\{x\} = vars(c)$

$D(x) := \{ d \in D(x) \mid \{x \rightarrow d\}$ is a solution of $c\}$

return D



Achieving Node Consistency- Example

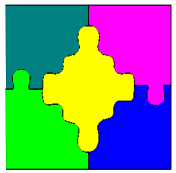
Let C and D be

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = D(Z) = \{1,2,3,4\}$$

Then node_consistent(C,D) returns domain D

where $D(X) = D(Y) = \{1,2,3,4\},\ D(Z) = \{1,2\}$



Arc Consistency

- A primitive constraint c is ***arc consistent*** with domain D
  - if $|vars\{c\}| \neq 2$ or
  - $vars(c) = \{x,y\}$ and

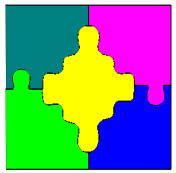
    for each d in $D(x)$ there is an e in $D(y)$ such that

    $\{ x \rightarrow d, y \rightarrow e\}$ is a solution of c

    and for each e in $D(y)$ there is a d in $D(x)$ such that

    $\{ x \rightarrow d, y \rightarrow e\}$ is a solution of c

  A CSP is ***arc consistent*** if each primitive constraint in it is arc consistent



Arc Consistency Examples

This CSP is node consistent but not arc consistent

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = \{1,2,3,4\},\ D(Z) = \{1,2\}$$

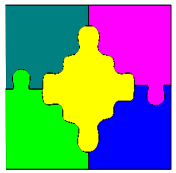
For example the value 4 for X and $X < Y$.

The following equivalent CSP is arc consistent

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = D(Z) = \varnothing$$

The map coloring and 4-queens CSPs are also arc consistent.



Achieving Arc Consistency

- arc_consistent_primitive(c, D)

  if $|vars(c)| = 2$ **then**

  let $\{x,y\} = vars(c)$

  $D(x) := \{ d \in D(x) \mid \text{exists } e \in D(y)$

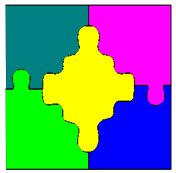
  $\{x \rightarrow d, y \rightarrow e\}$ is a solution of $c\}$

  $D(y) := \{ e \in D(y) \mid \text{exists } d \in D(x)$

  $\{x \rightarrow d, y \rightarrow e\}$ is a solution of $c\}$

  return D

- Removes values which are not arc consistent with c



Achieving Arc Consistency (Cont.)

- arc_consistent(C,D)

  Repeat

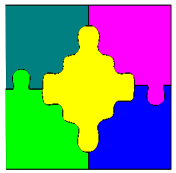
  for each primitive constraint c in C

  D := arc_consistent_primitive(c,D)

  until no domain changes in D

  return D

- Note that iteration is required
- The above is a very naive algorithm
  Faster algorithms are described in "Handbook of Constraint Programming", Elsevier, 2006. (Chapter 2)



Achieving Arc Consistency (Ex.)

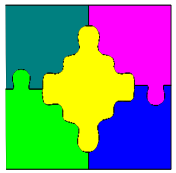
- Let C and D be given by

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = \{1,2,3,4\},\ D(Z) = \{1,2\}$$

Exercise:

Trace what arc_consistent(C,D) will do. Assume that the constraints are processed left to right.



Achieving Arc Consistency (Ex.)

- Let C and D be given by

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

$$D(X) = D(Y) = \{1,2,3,4\},\ D(Z) = \{1,2\}$$

- Consider arc_consistent(C,D)

-- calls arc_consistent_primitive($X<Y,D$) which updates D to

$$D(X) = \{1,2,3,\},\ D(Y) = \{2,3,4\},\ D(Z) = \{1,2\}$$

-- calls arc_consistent_primitive($Y<Z,D$) which updates D to

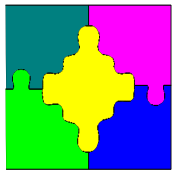
$$D(X) = \{1,2,3,\},\ D(Y) = \varnothing,\ D(Z) = \varnothing$$

-- calls arc_consistent_primitive($Z\leq 2,D$) which doesn't change D

-- calls arc_consistent_primitive($X<Y,D$) which updates D to

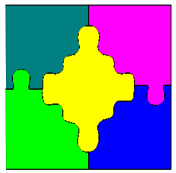
$$D(X) = \varnothing, D(Y) = \varnothing,\ D(Z) = \varnothing$$

-- calls arc_consistent_primitive 5 more times until it recognises that D remains unchanged



Using Node and Arc Consistency

- We can build constraint solvers using these consistency methods
- Two important kinds of domain
  - ***False domain***: some variable has an empty domain
  - ***Valuation domain***: every variable has a domain with a single value



Node and Arc Constraint Solver

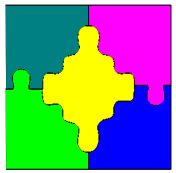
D := node_consistent(C,D)

D := arc_consistent(C,D)

if D is a false domain **then return** *false*

if D is a valuation domain **then return** *satisfiable*
(C,D)

return *unknown*

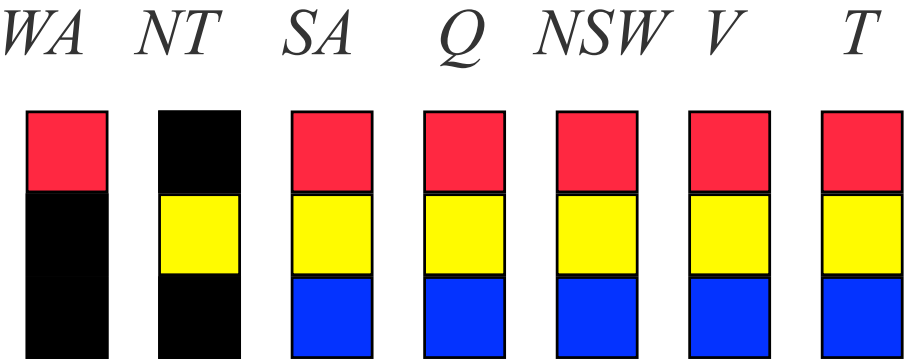


Node and Arc Solver Example

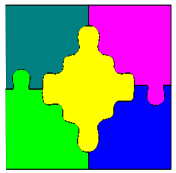
Colouring Australia: with constraints

$$WA = red \wedge NT = yellow$$

Node consistency



$$WA \neq NT \quad WA \neq SA \quad NT \neq SA$$
$$NT \neq Q \quad SA \neq Q \quad SA \neq NSW$$
$$SA \neq V \quad Q \neq NSW \quad NSW \neq V$$

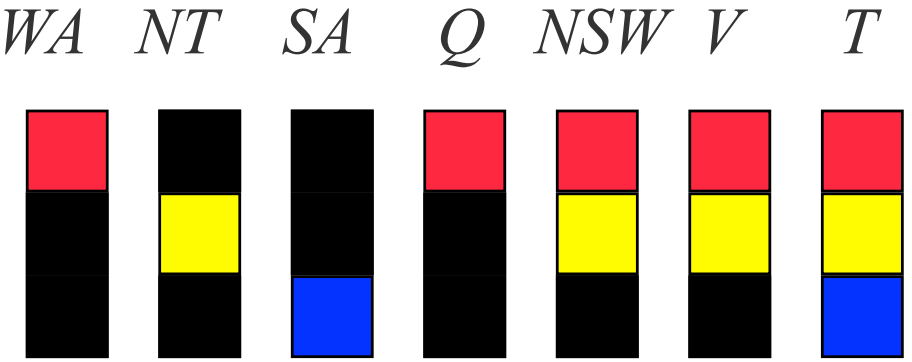


Node and Arc Solver Example

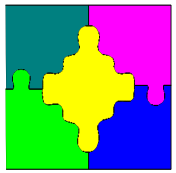
Colouring Australia: with constraints

$$WA = red \wedge NT = yellow$$

Arc consistency



$WA \neq NT$	$WA \neq SA$	$NT \neq SA$
$NT \neq Q$	$SA \neq Q$	$SA \neq NSW$
$SA \neq V$	$Q \neq NSW$	$NSW \neq V$

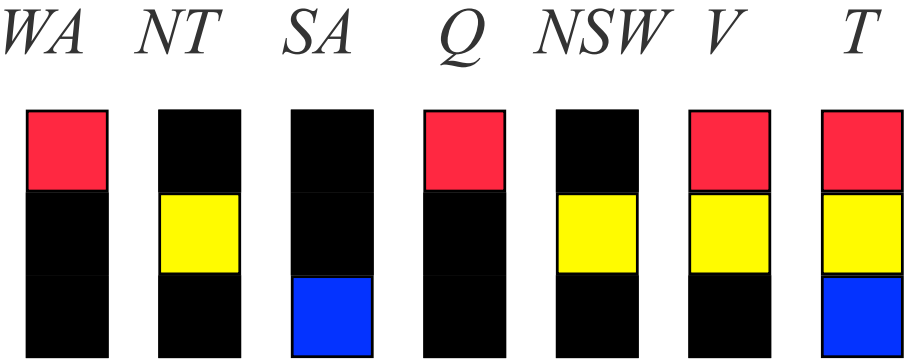


Node and Arc Solver Example

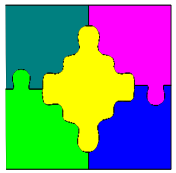
Colouring Australia: with constraints

$$WA = red \wedge NT = yellow$$

Arc consistency



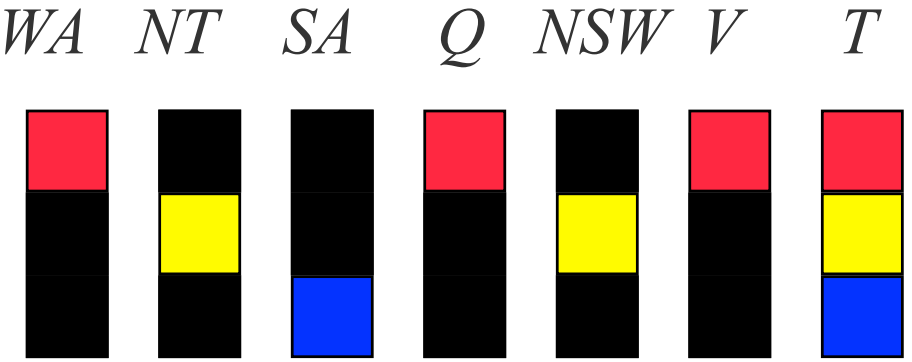
$WA \neq NT$	$WA \neq SA$	$NT \neq SA$
$NT \neq Q$	$SA \neq Q$	$SA \neq NSW$
$SA \neq V$	$Q \neq NSW$	$NSW \neq V$



Node and Arc Solver Example

Colouring Australia: with constraints

$$WA = red \wedge NT = yellow$$

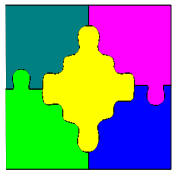


Arc consistency

Answer:

unknown

$WA \neq NT$	$WA \neq SA$	$NT \neq SA$
$NT \neq Q$	$SA \neq Q$	$SA \neq NSW$
$SA \neq V$	$Q \neq NSW$	$NSW \neq V$

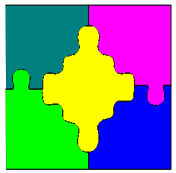


Combined Consistency with BT Search

- We can ***combine*** consistency methods with the backtracking solver
- Apply node and arc consistency before starting the backtracking solver and after each variable is given a value
- This ***reduces*** the number of values that need to be tried for each variable.

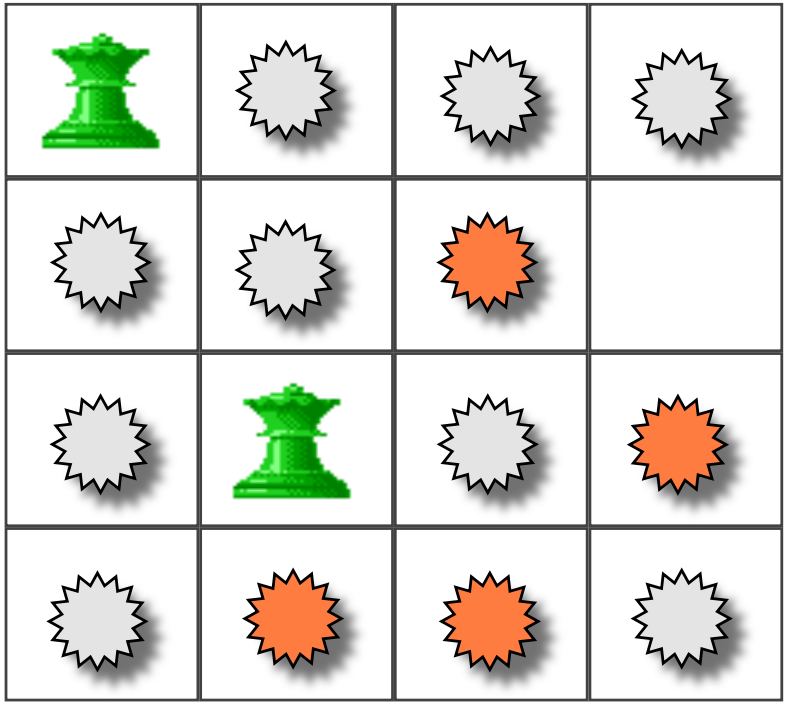
  I.e. it reduces the ***search space***

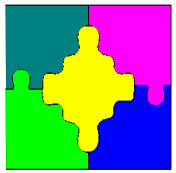
This is the finite domain programming approach



BT+Consistency -- Example

Therefore, we need to choose another value for Q2.



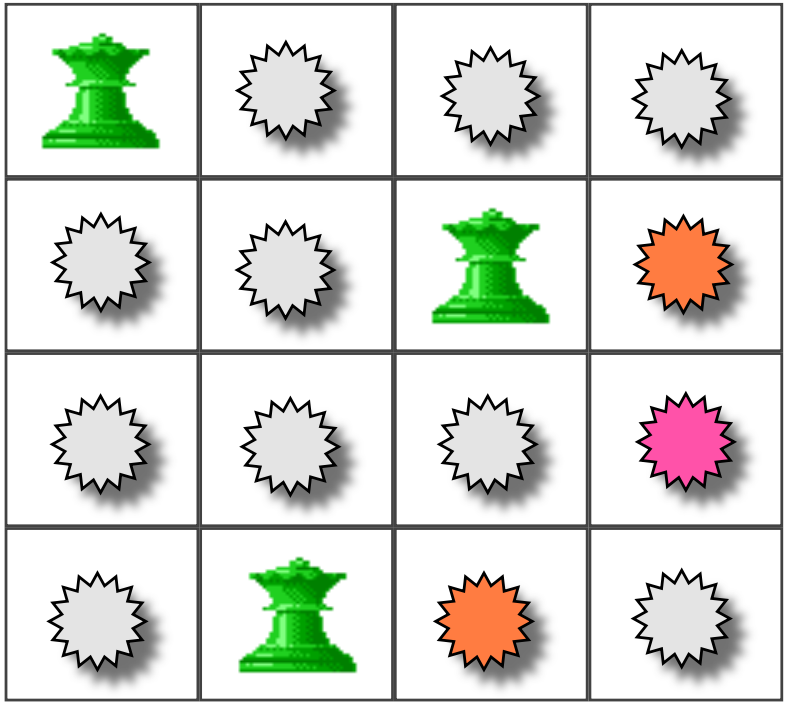


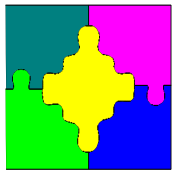
BT+Consistency -- Example

backtracking,

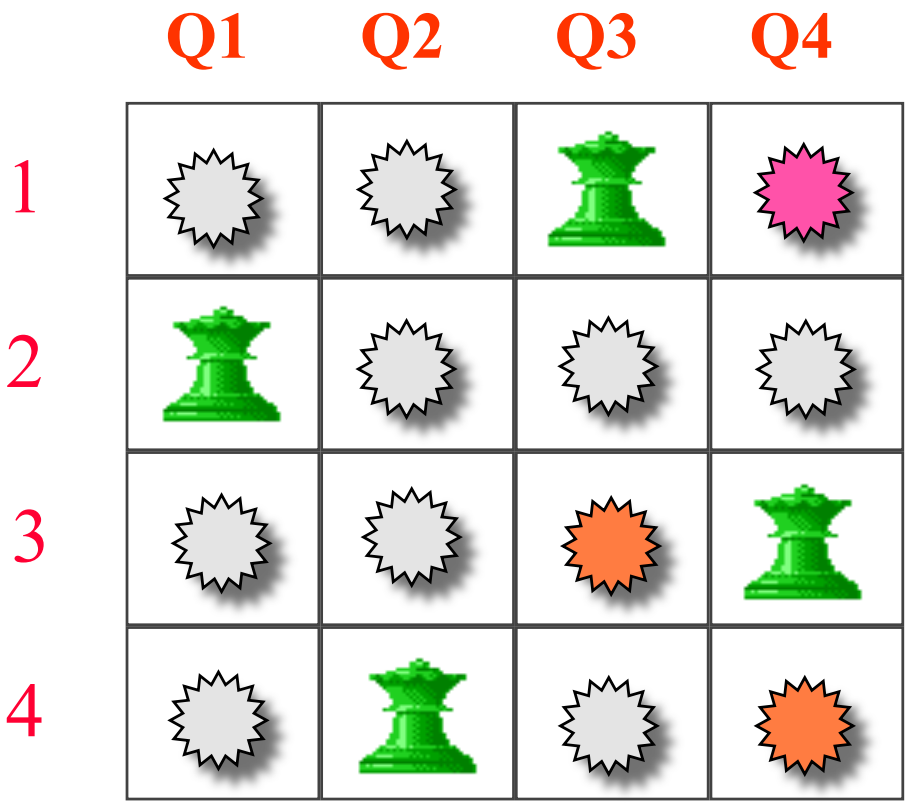
Find another value of Q1?

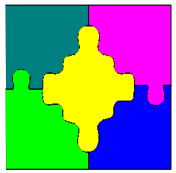
Yes, Q1 = 2





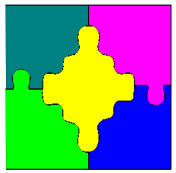
BT+Consistency -- Example





Domain Consistency

- What about primitive constraints with more than 2 variables? E.g. X=3Y+5Z.
- ***Domain consistency*** for *c*
  - extend arc consistency to arbitrary number of variables
  - also called *generalized arc* or *hyper-arc* consistency
- $D'(x_i) = \{ d_i \text{ in } D(x_i) \mid \{ x_i \rightarrow d_i \} \text{ is a soln of } c \}$
- Strongest possible propagation! (for single cons.)
- But ***NP-hard*** for linear equations (so it's probably exponential)
- So whats the answer ?

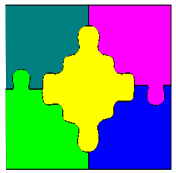


Bounds Consistency

- Use ***bounds consistency***
- This only works for ***arithmetic CSPs*****:** I.e. CSPs whose constraints are over the integers.
- Two ***key*** ideas
  - Use consistency over the ***real numbers***
  - ***map to integer interval arithmetic***
  - Only consider the ***endpoints*** (upper and lower bounds) of the interval
  - Narrow intervals whenever possible
- Let the ***range*** $[l..u]$ represent the set of integers

  $\{l, l+1, ..., u\}$

  and the empty set if $l>u$.
- Define *min(D,x)* to be the smallest element in the domain of x
- Define *max(D,x)* to be the largest



Bounds Consistency

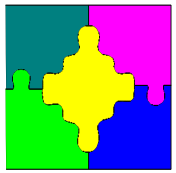
- A primitive constraint c is ***bounds(R) consistent*** with domain D if for each variable x in *vars(c)*
  - There exist real numbers $d_1, ..., d_k$ for remaining vars $x_1, ..., x_k$ such that for each x_i, $\min(D,x_i) \leq d_i \leq \max(D,x_i)$ and

    $\{ x \rightarrow \min(D,x), x_1 \rightarrow d_1, ..., x_k \rightarrow d_k \}$

    is a solution of c
  - and there exist real numbers $e_1, ..., e_k$ for $x_1, ..., x_k$ such that for each x_i, $\min(D,x_i) \leq e_i \leq \max(D,x_i)$ and

    $\{ x \rightarrow \max(D,x), x_1 \rightarrow e_1, ..., x_k \rightarrow e_k \}$

    is a solution of c
- An arithmetic CSP is ***bounds(R) consistent*** if all its primitive constraints are.



Bounds Consistency Examples

$$X = 3Y + 5Z$$

$$D(X) = [2..7], D(Y) = [0..2], D(Z) = [-1..2]$$

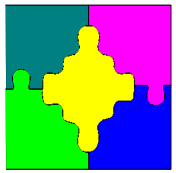
Not bounds consistent, consider *Z=2*, then *X-3Y=10*

But the domain below is bounds consistent

$$D(X) = [2..7], D(Y) = [0..2], D(Z) = [0..1]$$

Compare with the domain consistent domain

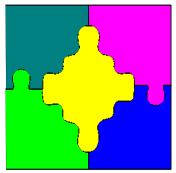
$$D(X) = \{3,5,6\}, D(Y) = \{0,1,2\}, D(Z) = \{0,1\}$$



Achieving Bounds Consistency

- Given a current domain we wish to modify the endpoints of the domains so the result is bounds consistent
- ***Propagation rules*** do this
- The constraint solver generates propagation rules for each primitive constraint.
- These are repeatedly applied until there is no change in the domain
- We do not need to apply a propagation rule until the domains of the variables involved are modified

We now look at how propagation rules are generated for common kinds of primitive constraint.

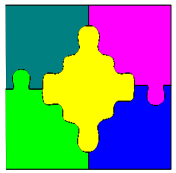


Propagation Rules for X = Y + Z

Consider the primitive constraint $X = Y + Z$ which is equivalent to the three forms

$$X = Y + Z \quad Y = X - Z \quad Z = X - Y$$

Propagation Rules ??



Propagation Rules for X = Y + Z

Consider the primitive constraint $X = Y + Z$ which is equivalent to the three forms

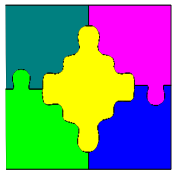
$$X = Y + Z \quad Y = X - Z \quad Z = X - Y$$

Reasoning about minimum and maximum values:

$$X \geq \min(D,Y) + \min(D,Z) \qquad X \leq \max(D,Y) + \max(D,Z)$$

$$Y \geq \min(D,X) - \max(D,Z) \qquad Y \leq \max(D,X) - \min(D,Z)$$

$$Z \geq \min(D,X) - \max(D,Y) \qquad Z \leq \max(D,X) - \min(D,Y)$$



Propagation Rules for X = Y + Z

$$X \geq \min(D,Y) + \min(D,Z) \quad X \leq \max(D,Y) + \max(D,Z)$$

Xmin := max {min(D,X), min(D,Y) + min(D,Z)}

Xmax := min {max(D,X), max(D,Y) + max(D,Z)}

D(X) := { X ∈ D(X) | Xmin ≤ X ≤ Xmax }

$$Y \geq \min(D,X) - \max(D,Z) \quad Y \leq \max(D,X) - \min(D,Z)$$

Ymin := max {min(D,Y), min(D,X) - max(D,Z)}

Ymax := min {max(D,Y), max(D,X) - min(D,Z)}

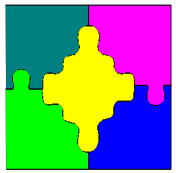
D(Y) := { Y ∈ D(Y) | Ymin ≤ Y ≤ Ymax }

$$Z \geq \min(D,X) - \max(D,Y) \quad X \leq \max(D,X) - \min(D,Y)$$

Zmin := max {min(D,Z), min(D,X) - max(D,Y)}

Zmax := min {max(D,Z), max(D,X) - min(D,Y)}

D(Z) := { Z ∈ D(Z) | Zmin ≤ Z ≤ Zmax }



Achieving Bounds Consistency--Example

$$X = Y + Z$$

$$D(X) = [4..8], D(Y) = [0..3], D(Z) = [2..2]$$

The propagation rules determine that:

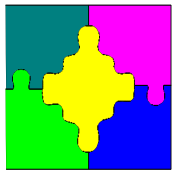
$$(0 + 2 =)\ \ 2 \leq X \leq 5\ \ (= 3 + 2)$$

$$(4 - 2 =)\ \ 2 \leq Y \leq 6\ \ (= 8 - 2)$$

$$(4 - 3 =)\ \ 1 \leq Z \leq 8\ \ (= 8 - 0)$$

Hence the domains can be reduced to

$$D(X) = [4..5], D(Y) = [2..3], D(Z) = [2..2]$$



Linear Inequality

$$4W + 3P + 2C \le 9$$

$$W \le \frac{9}{4} - \frac{3}{4}\min(D,P) - \frac{2}{4}\min(D,C)$$

$$P \le \frac{9}{3} - \frac{4}{3}\min(D,W) - \frac{2}{3}\min(D,C)$$

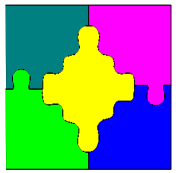
$$C \le \frac{9}{2} - \frac{4}{2}\min(D,W) - \frac{3}{2}\min(D,P)$$

Given initial domain:

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

We determine that $W \le \left\lfloor \frac{9}{4} \right\rfloor, \quad P \le \left\lfloor \frac{9}{3} \right\rfloor, \quad C \le \left\lfloor \frac{9}{2} \right\rfloor$

new domain: $D(W) = [0..2], D(P) = [0..3], D(C) = [0..4]$



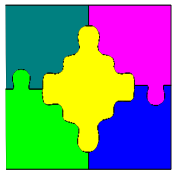
Linear Inequality

- 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

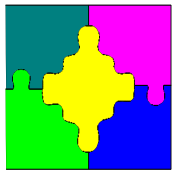
- To propagate the general linear inequality

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

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

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

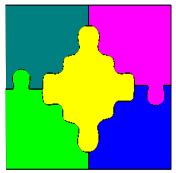
$$x_i \geq \frac{b - \sum_{j=1..n, j \neq i} a_j \max(D, x_j)}{a_i}$$



Linear equation example

- *SEND+MORE=MONEY*
- $9000M + 900O + 90N - 90E + Y + D - 1000S - 10R = 0$
- $9000M \leq -900*0 - 90*0 + 90*9 - 0 - 0 + 1000*9 + 10*9$
- $M \leq 1.1 \Rightarrow M = 1$
- $1000S \geq 9000*1 + 900*0 + 90*0 - 90*9 + 0 + 0 - 10*9$
- $S \geq 8.9 \Rightarrow S = 9$
- $900O \leq -9000*1 - 90*0 + 90*9 - 0 - 0 + 1000*9 + 10*9$
- $O \leq 1$

- Linear equation propagation requires us to revisit the same constraint to take into account its changes



Dis-equations Y≠Z

Dis-equations give ***weak*** bounds propagation rules.

Only when one side takes a fixed value that equals the minimum or maximum of the other is there propagation

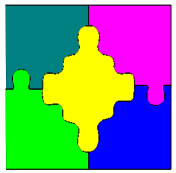
$D(Y) = [2..4], D(Z) = [2..3]$ no propagation

$D(Y) = [2..4], D(Z) = [3..3]$ no propagation

$D(Y) = [2..4], \underline{D(Z) = [2..2]}$ prop $D(Y) = [3..4], D(Z) = [2..2]$

In practice disequations implemented by domain propagation

Exercise: How do you propagate $\sum_{i=1..n} a_i x_i \neq b$



Non-Linear Constraints X=Y×Z

If all variables are positive it is simple enough

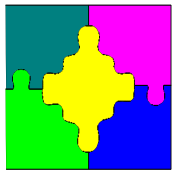
$$X \geq \min(D,Y) \times \min(D,Z) \qquad X \leq \max(D,Y) \times \max(D,Z)$$
$$Y \geq \min(D,X) / \max(D,Z) \qquad Y \leq \max(D,X) / \min(D,Z)$$
$$Z \geq \min(D,X) / \max(D,Y) \qquad Z \leq \max(D,X) / \min(D,Y)$$

Example: $D(X) = [4..8], D(Y) = [1..2], D(Z) = [1..3]$

becomes: $D(X) = [4..6], D(Y) = [2..2], D(Z) = [2..3]$

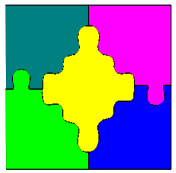
Much harder if some domains are non-positive or span zero

See Marriott&Stuckey, 1998



Exercise: X = Y × Z

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



Other Bounds Consistencies

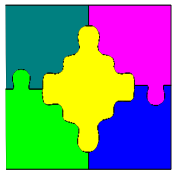
- A primitive constraint c is ***bounds(Z) consistent*** with domain D if for each variable x in *vars(c)*
  - There exist integer numbers $d_1, ..., d_k$ for remaining vars $x_1, ..., x_k$ such that for each x_i, $\min(D,x_i) \leq d_i \leq \max(D,x_i)$ and

    $\{ x \rightarrow \min(D,x), x_1 \rightarrow d_1, ..., x_k \rightarrow d_k \}$

    is a solution of c
  - and there exist integer numbers $e_1, ..., e_k$ for $x_1, ..., x_k$ such that for each x_i, $\min(D,x_i) \leq e_i \leq \max(D,x_i)$ and

    $\{ x \rightarrow \max(D,x), x_1 \rightarrow e_1, ..., x_k \rightarrow e_k \}$

    is a solution of c
- An arithmetic CSP is ***bounds(Z) consistent*** if all its primitive constraints are.
- Note still NP-hard for linear equations!



Bounds Consistency Example

Smugglers knapsack problem (no whiskey available)

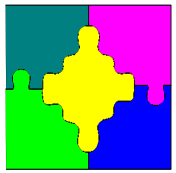
$$\underbrace{4W + 3P + 2C \leq 9}_{capacity} \wedge \underbrace{15W + 10P + 7C \geq 30}_{profit}$$

$$D(W) = [0..0], D(P) = [0..9], D(C) = [0..6]$$

Step 1 examines profit constraint

$$W \geq -102/15 \quad P \geq -12/10 \quad C \geq -60/7$$

no change



Bounds Consistency Example

Smugglers knapsack problem (no whiskey available)

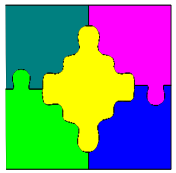
$$\underbrace{4W + 3P + 2C \leq 9}_{capacity} \wedge \underbrace{15W + 10P + 7C \geq 30}_{profit}$$

$$D(W) = [0..0], D(P) = [0..9], D(C) = [0..6]$$

Step 2 examines capacity constraint

$$W \leq 9/4 \quad P \leq 9/3 \quad C \leq 9/2$$

$$D(W) = [0..0], D(P) = [0..3], D(C) = [0..4]$$



Bounds Consistency Example

Smugglers knapsack problem (no whiskey available)

capacity		*profit*
$4W + 3P + 2C \leq 9$	$\wedge$	$15W + 10P + 7C \geq 30$

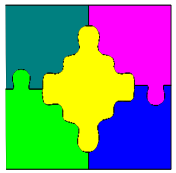
$$D(W) = [0..0], D(P) = [0..3], D(C) = [0..4]$$

Step 3 re-examines profit constraint, because of max(D(P,C)) change

$$W \geq -28/15 \quad P \geq 2/10 \quad C \geq 0/7$$

$$D(W) = [0..0], D(P) = [1..3], D(C) = [0..4]$$

no further change after this



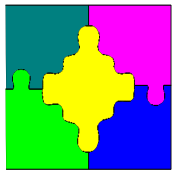
Bounds Consistency Solver

D := bounds_consistent(C,D)

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

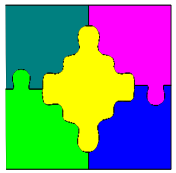
if D is a valuation domain **return** *satisfiable*(C,D)

return *unknown*



BT + Bounds Consistency

- Like arc and node consistency we can ***combine*** bounds consistency methods with the backtracking solver
- Apply bounds consistency before starting the backtracking solver and after each variable is given a value
- This reduces the ***search space***



BT + Bounds Consistency- Example

Smugglers knapsack problem

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

Current domain:

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

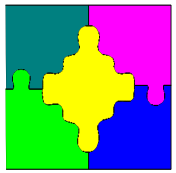
after bounds consistency

$W = 0$

$P = 1$

(0,1,3)

Solution Found: return *true*



BT + Bounds Consistency- Example

Smugglers knapsack problem (whiskey available)

capacity *profit*

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

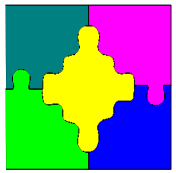
Current domain:

$$D(W) = \{0\}, D(P) = \{3\}, D(C) = \{0\}$$

Initial bounds consistency

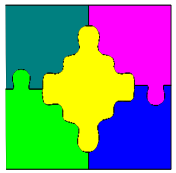
$W = 0$			$W = 1$	$W = 2$
$P = 1$	$P = 2$	$P = 3$	**(1,1,1)**	**(2,0,0)**
(0,1,3)	*false*	**(0,3,0)**		

No more solutions



Propagators

- In reality consistency is just a notion to define propagators
- Propagator *f* for constraint *c*
  - function from domains to domains *D* -> *D*
  - *f* removes values from the the domain that cant be solutions of *c*
  - correctness: solutions of *c* in *D*, and same as in *f(D)*
  - checking: f(D) is false domain when D is an unsatisfiable valuation domain
- Propagator for X=Y × Z
  - typically does not implement any consistency
- Tradeoff
  - strong propagators eliminate as many values as possible
  - fast propagators propagate efficiently



Propagation Solving

- Fixpoint of all propagators
  - assume *D* is a fixpoint for *F0*

isolv(*F0*, *Fn*, *D*)

F := *F0* union *Fn*

Q := *Fn*

while (Q ≠ ⦸)

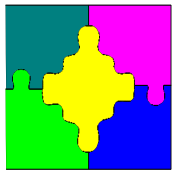
f := choose(*Q*) % select next propagator to run

Q := *Q* – {*f*}; *D*' := *f*(*D*);

Q := *Q* union new(*f*,*F*,*D*,*D*') % add affected props

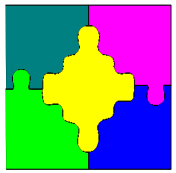
D := *D*'

return *D*



Propagation Solving

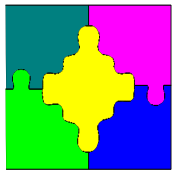
- choose(Q)
  - typically a FIFO queue
  - usually with priority levels
    - do fast propagators first
- new(f,F,D,D')
  - return propagators f' in F where $f'(D') \neq D'$
  - typically propagators are attached to events
    - fix(x): x becomes fixed
    - lbc(x): lower bound of x changed, ubc(x) (upper bound)
    - dmc(x): domain of x changed
  - add propagators for the events that occurred in $f(D)$



Propagation Solving Example

- $x = 2y \wedge x = 3z, D(x) = [0..17], D(y) = [0..9], D(z) = [0..6]$

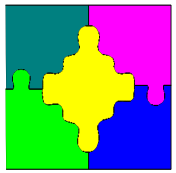
Q	f	$D(x)$	$D(y)$	$D(z)$	new
f1,f2	f1	[0..17]	[0..8]	[0..6]	{f1}
f2,f1	f2	[0..17]	[0..8]	[0..5]	{f2}
f1,f2	f1	[0..16]	[0..8]	[0..5]	{f1,f2}
f2,f1	f2	[0..15]	[0..8]	[0..5]	{f1,f2}
f1,f2	f1	[0..15]	[0..7]	[0..5]	{f1}
f2,f1	f2	[0..15]	[0..7]	[0..5]	{}
f1	f1	[0..14]	[0..7]	[0..5]	{f1,f2}
f2,f1	f2	[0..14]	[0..7]	[0..4]	{f2}
f1,f2	f1	[0..14]	[0..7]	[0..4]	{}
f2	f2	[0..12]	[0..7]	[0..4]	{f1,f2}
f1,f2	f1	[0..12]	[0..6]	[0..4]	{f1}
f2,f1	f2	[0..12]	[0..6]	[0..4]	{}
f1	f1	[0..12]	[0..6]	[0..4]	{}



Idempotence

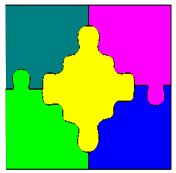
- Many propagators know they are at fixpoint after execution
- Such propagators are termed idempotent
- The usual propagators for $x = 2y$, $x = 3z$ are not idempotent
- Execution with idempotent propagators
- $x = 2y \wedge x = 3z, D(x) = [0..17], D(y) = [0..9], D(z) = [0..6]$

Q	f	$D(x)$	$D(y)$	$D(z)$	new
f1,f2	f1	[0..16]	[0..8]	[0..6]	{f2}
f2	f2	[0..15]	[0..8]	[0..5]	{f1}
f1	f1	[0..14]	[0..7]	[0..5]	{f2}
f2	f2	[0..12]	[0..7]	[0..4]	{f1}
f1	f1	[0..12]	[0..6]	[0..4]	{}



Flattening and Propagation

- In order to understand the behaviour of a propagation solver we need to understand how complex constraint expressions are converted to constraints in the solver
- Flattening a MiniZinc model has three important parts
  - unrolling loops
  - inlining predicate definitions
  - decomposing complex expressions



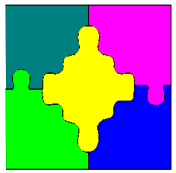
Complex Constraint Expressions

- Complex constraints are translated into ***reified constraints***
- For every primitive constraint $c(x)$ there is a corresponding reified constraint $c_{reify}(x,B)$.
- The Boolean variable ***B*** reflects the truth of constraint $c(x)$.
  I.e. B=*true* if $c(x)$ holds and B=*false* if $\neg c(x)$ holds.
- The constraint

  $F1 \geq F2+K \vee F2 \geq F1+K.$

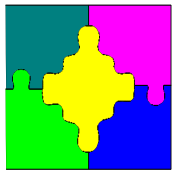
  is translated into

  let { var bool: B1, var bool: B2 } in
  $\geq_{reify}(F1,F2+K,B1) \wedge$
  $\geq_{reify}(F2,F1+K,B2) \wedge$
  exists([B1,B2])



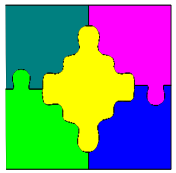
Weak Propagation of Reified Constraints

- Beware that reified constraints don't propagate much
- x = abs(y) versus (x = y \/ x = -y) /\ x >= 0
- $D(x) = \{ -2, -1, 0, 1, 2, 4, 5\}, D(y) = \{ -1, 1, 3, 4 \}$
  - domain consistent abs: $D(x) = \{1,4\}, D(y) = \{-1,1,4\}$
  - bounds consistent abs: $D(x) = \{0,1,2,4\}, D(y) = \{-1,1,3,4\}$
  - reified form b1 ⇔ x = y, b2 ⇔ x = -y, b1 \/ b2
  - $D(b1) = D(b2) = \{0,1\}$, $D(x) = \{0,1,2,4,5\}, D(y)$ same!
- $D(x) = \{ -2, -1, 0, 1, 2, 4, 5\}, D(y) = \{ 1, 3, 4 \}$
  - reified form (no change)



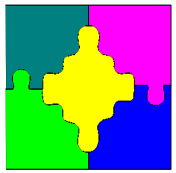
A better decomposition

- Different decompositions can propagate differently
- x >= 0 /\ x >= y /\ x >= -y /\
  (x <= y \/ x <= -y)
- $D(x) = \{ -2, -1, 0, 1, 2, 4, 5\}, D(y) = \{ -1, 1, 3, 4 \}$
  - $D(x) = \{0,1,2,4,5\}, D(y) = \{-1,1,3,4\}$ $x >= 0$
- $D(x) = \{ -2, -1, 0, 1, 2, 4, 5\}, D(y) = \{ 1, 3, 4 \}$
  - $D(x) = \{1,2,4,5\}, D(y) = \{1,3,4\}$ $x >= y$
  - $D(x) = \{1,2,4,5\}, D(y) = \{1,3,4\}$ $x <= -y$ is *false*
  - $D(x) = \{1,2,4\}, D(y) = \{1,3,4\}$ $x <= y$ (*true*)



Global Constraints

- Each global constraint is implemented by (possibly several)
  - propagators
- A good implementation of a global constraints has
  - strong propagation (ideally domain consistent)
  - fast propagation
- Often global propagators are not idempotent

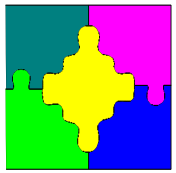


All_different

- *all_different*($[V_1,...,V_n]$) holds when each variable $V_1,..,V_n$ takes a different value
- Not needed for expressiveness. E.g. *all_different*([X, Y, Z]) is equivalent to $X \neq Y \wedge X \neq Z \wedge Y \neq Z$
- But conjunctions of disequations are not handled well by arc (or bounds) consistency.

  E.g. The following domain is arc consistent with the above

  $$D(X) = \{1,2\}, D(Y) = \{1,2\}, D(Z) = \{1,2\}$$
- BUT there is no solution!
- Specialized consistency techniques for *all_different* can find this



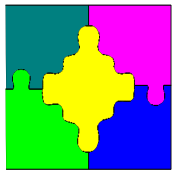
All_different Propagator

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

f(D)

let $W = \{V_1,\ldots,V_n\}$
while exists V in W where $D(V) = \{d\}$
$W := W - \{V\}$
for each V' in W
$D(V') := D(V') - \{d\}$
$DV :=$ union of all $D(V)$ for V in W
if $|DV| < |W|$ **then return** *false*
return D

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



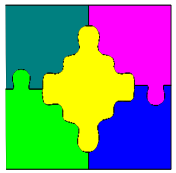
All_different Example

$$all\_different([X, Y, Z])$$

$$D(X) = \{1,2\}, D(Y) = \{1,2\}, D(Z) = \{1,2\}$$

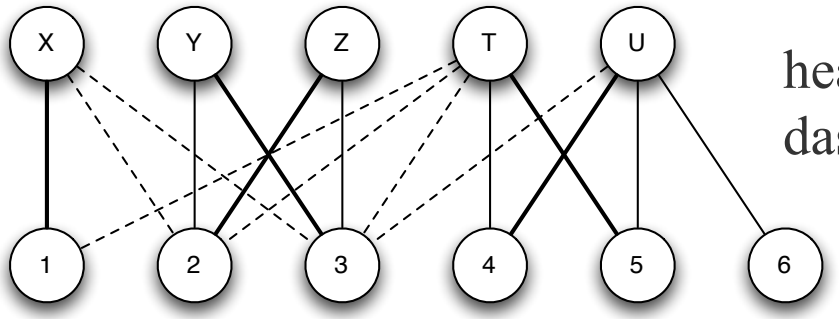
$DV = \{1,2\}$, $W=\{X,Y,Z\}$ so $|DV| < |W|$

hence detects unsatisfiability



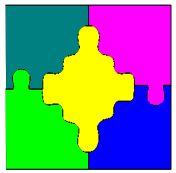
All_different Propagator

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



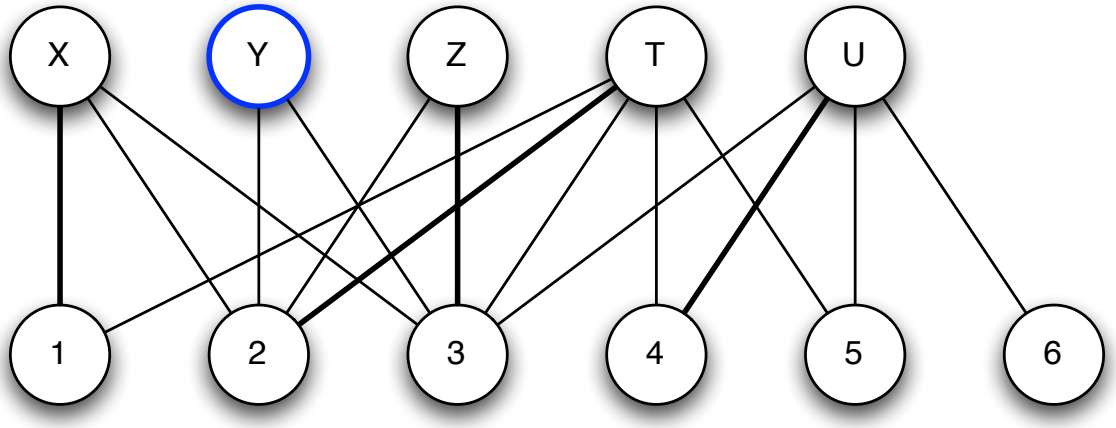
heavy = maximal matching
dashed = cant be in max matching

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

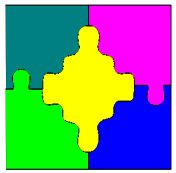


Maximal Matching

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

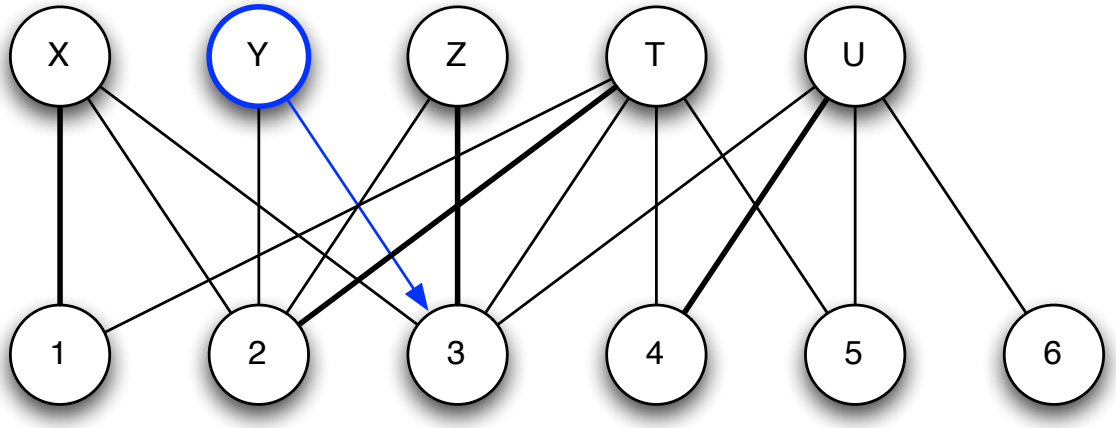


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

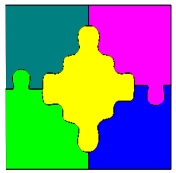


Maximal Matching

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

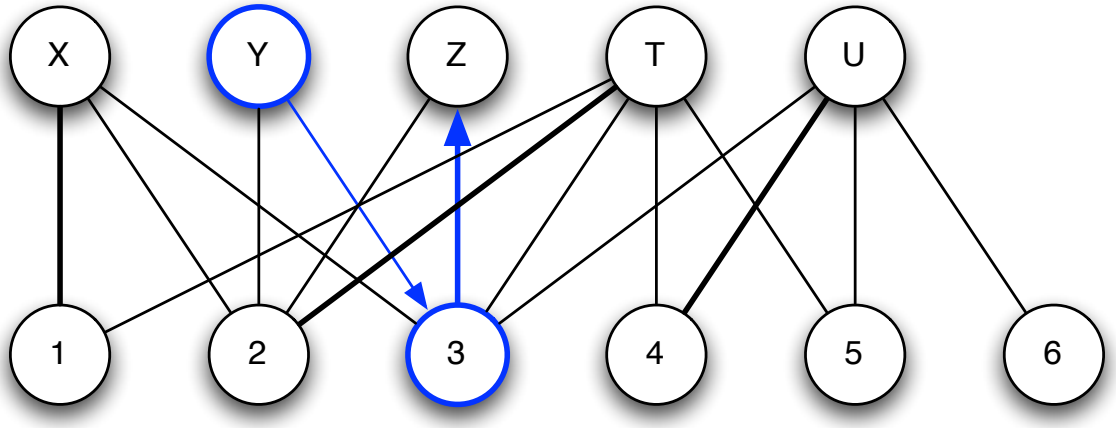


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

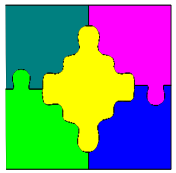


Maximal Matching

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

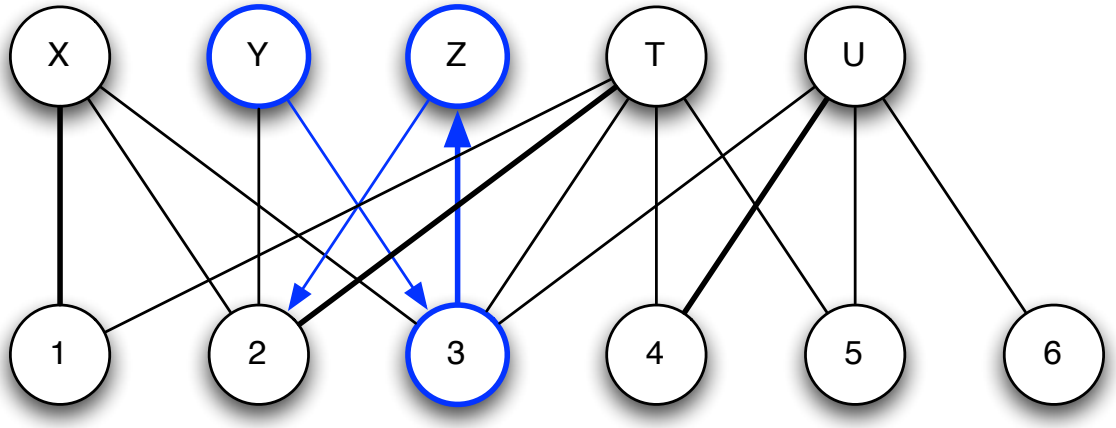


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

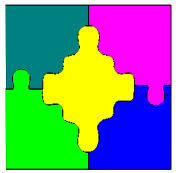


Maximal Matching

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

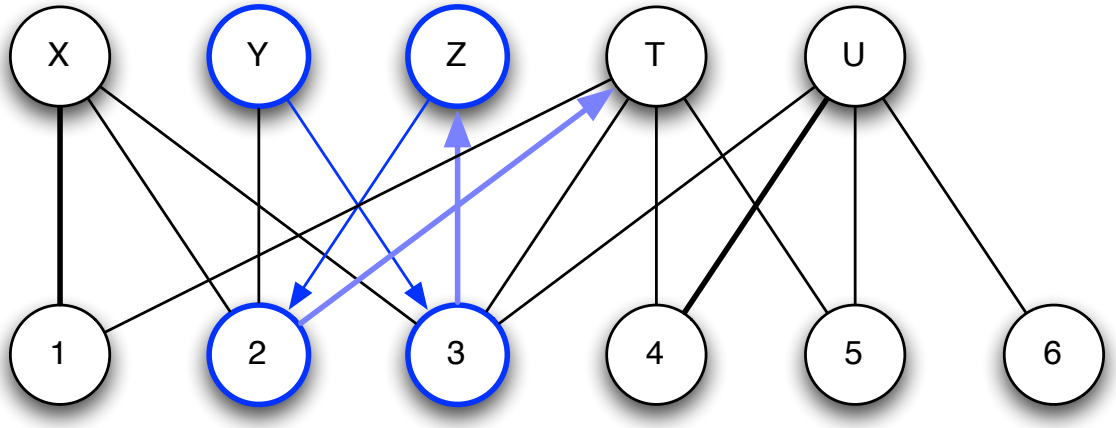


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

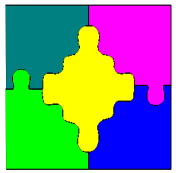


Maximal Matching

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

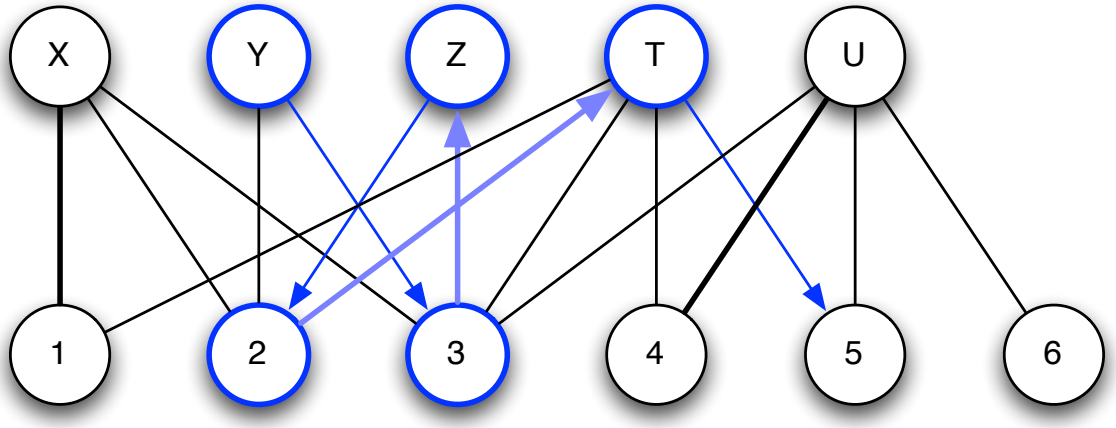


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

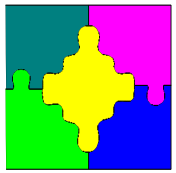


Maximal Matching

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

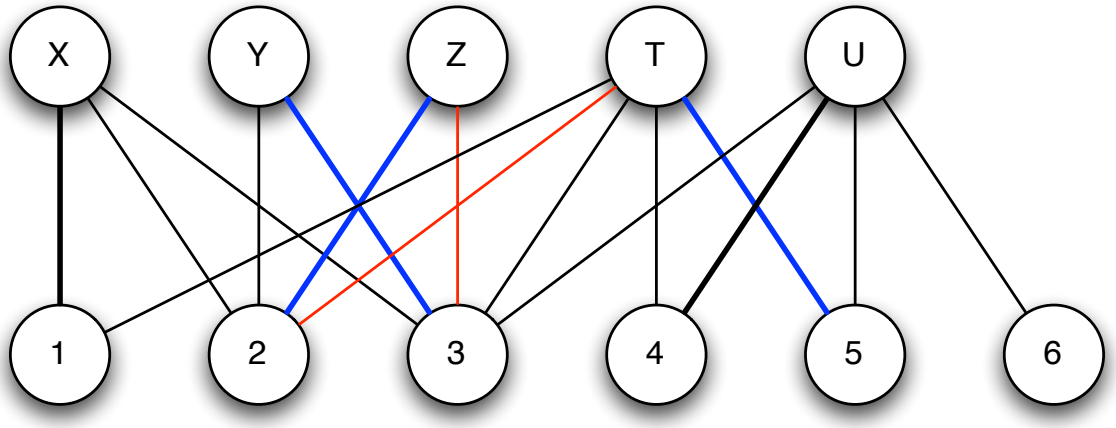


- Search for an alternating path
  - unmatched and matched edges
  - reaching an unmatched value

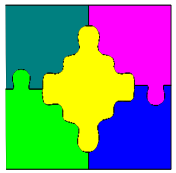


Maximal Matching

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



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



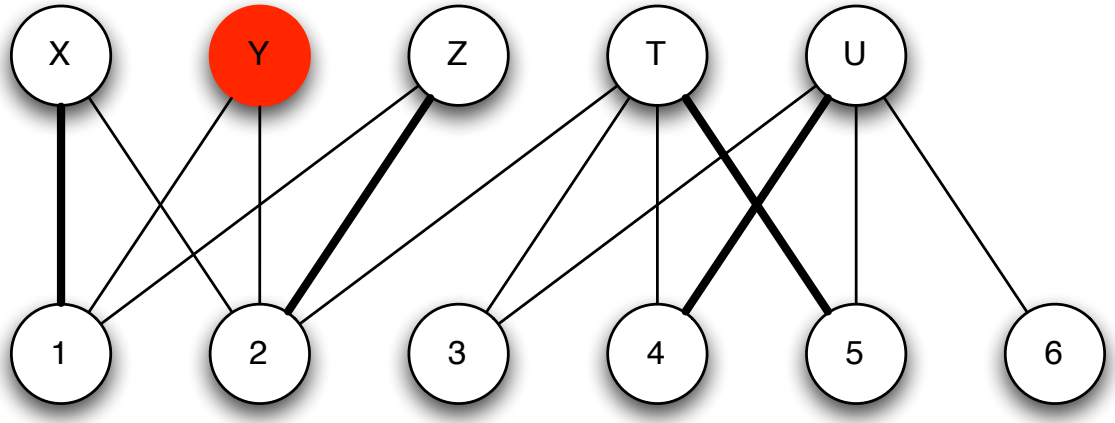
Failure

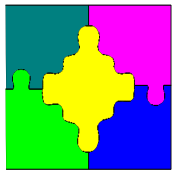
- If not every variable is matched in the maximal matching then the all_different constraint cannot be satisfied.

$$all\_different([X, Y, Z, T, U])$$

$$D(X) = \{1, 2\}, D(Y) = \{1, 2\}, D(Z) = \{1, 2\},$$

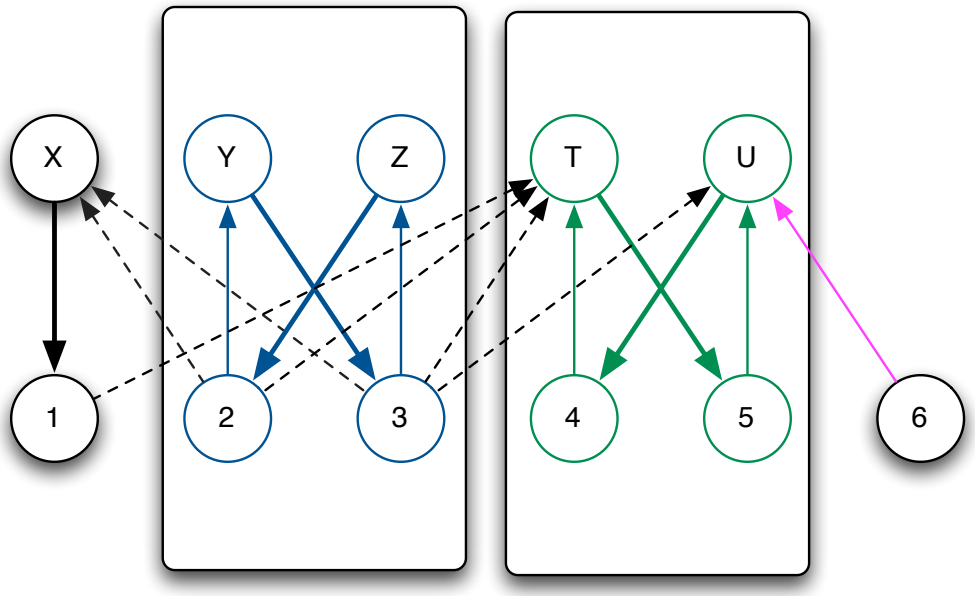
$$D(T) = \{2, 3, 4, 5\}, D(U) = \{3, 4, 5, 6\}$$



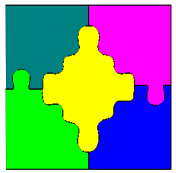


Propagation

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

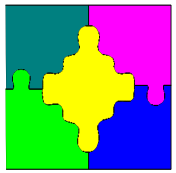


- Keep edges in an SCC or in matching, delete rest
- $D'(X) = \{1\}, D'(Y) = \{2,3\}, D'(Z) = \{2,3\},$ $D'(T) = \{4,5\}, D'(U) = \{4,5,6\}$



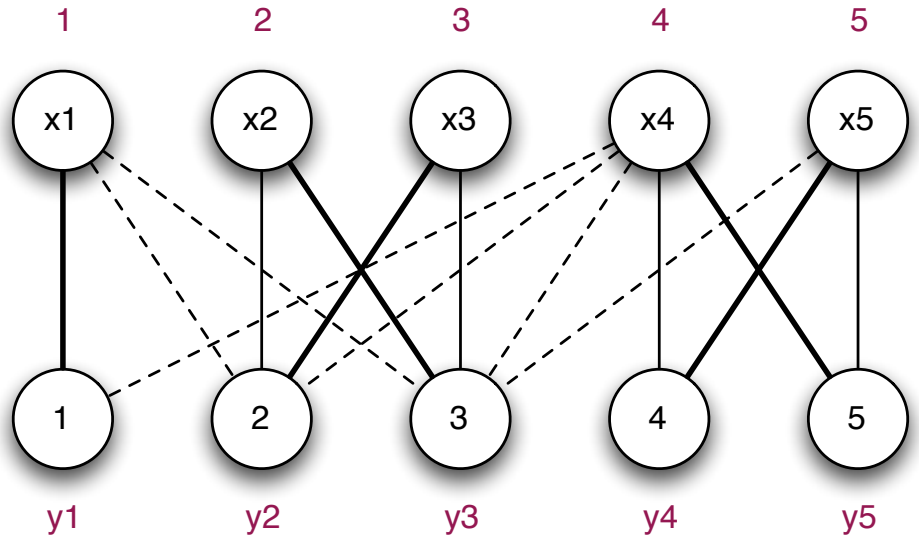
All_different Propagator

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

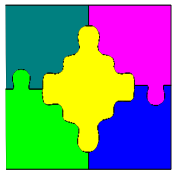


Inverse Propagator

- Same algorithm as all_different (?)
- $x1 = 3 \Leftrightarrow y3 = 1$



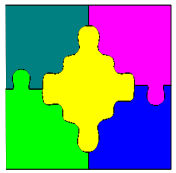
- $D(x4) = \{4,5\} \Leftrightarrow 4 \text{ in } D(y4), 4 \text{ in } D(y5)$
- Wakes on dmc(x) and dmc(y) events
- Implements domain consistency



Inverse by Decomposition

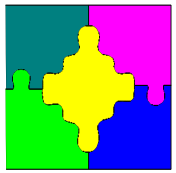
```
predicate inverse(array[int] of var int: f,
                  array[int] of var int: invf) =
    forall(j in index_set(invf))(invf[j] in index_set(f)) /\
    forall(i in index_set(f))(
        f[i] in index_set(invf) /\
        forall(j in index_set(invf))(j == f[i] <-> i == invf[j])
    );
```

- Propagates more than alldifferent decomposition
  - **Exercise**: Why? Give an example where it does
- Large number of constraints + Boolean vars
- Not as strong as global



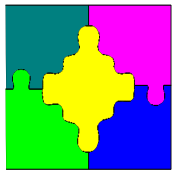
Element propagator

- The expression $x = a[i]$ where i is a variable generates
  - $element(i, [a_1, a_2, \ldots, a_n], x)$
- The *element* propagator
  - ensures $i = j \rightarrow x = a_j$
  - ensures $x = d \rightarrow i = j_1 \vee i = j_2 \vee \ldots \vee i = j_n$
    - where $a[j_1] = a[j_2] = \ldots = a[j_n] = d$
- Wakes up on dmc(x) and dmc(i) events
- Implements domain consistency, idempotent



Element by Decomposition

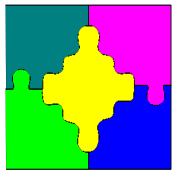
- predicate element(var int:i, array[int] of int:a,
  var int:x) =
  x in a /\ i in index_set(a) /\
  forall(j in index_set(a))(
  i = j -> x = a[i]
  );
- Substantially weaker than domain consistent
- Introduces many Booleans



Cumulative Constraints

- Recall the cumulative constraint
- *cumulative*($[S_1,\ldots,S_n]$, $[D_1,\ldots,D_n]$, $[R_1,\ldots,R_n]$, L)

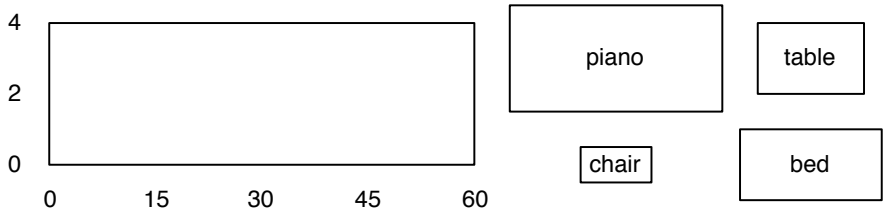
  schedule n tasks with start times S_i and durations D_i needing R_i units of a single resource where L units are available at each moment.
- Very complex propagator
- Many different implementations
  - Different complexities
  - None implement bounds or domain consistency



Cumulative Example

Bernd is moving house again. 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

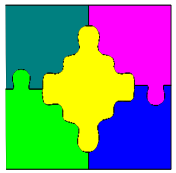


How can we model this?

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

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

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



Cumulative timetable propagator

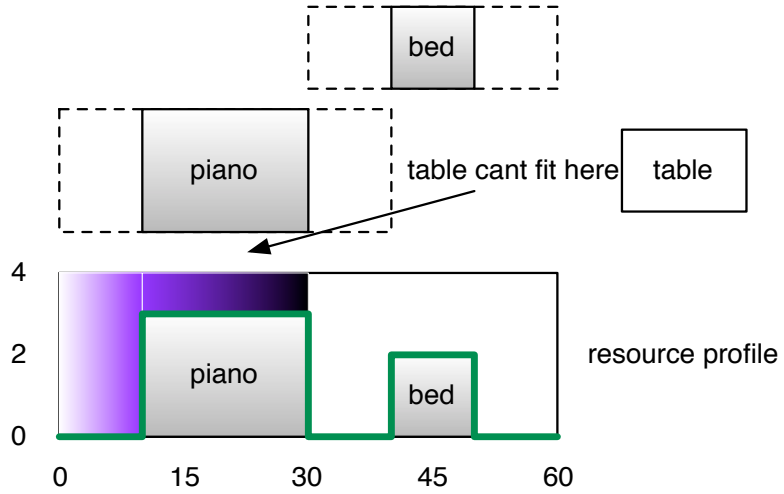
- Determine the parts where a task must be running
- The resource profile adds up these parts
- Use profile to move other tasks

Example: after initial bounds

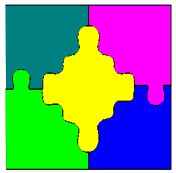
$D(P) = [0..30]$, $D(C) = [0..50]$, $D(B) = [0..40]$, $D(T) = [0..45]$

Propagating P + 30 ≤ B

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

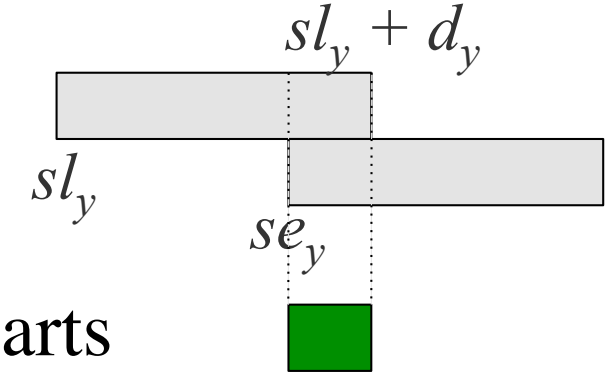


$D(P) = [0..15]$, $D(C) = [0..50]$,
$D(B) = [30..40]$, $D(T) = [30..45]$

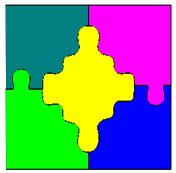


Compulsory Parts

- 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-d_x if $se_x \leq t < se_x + d_x$



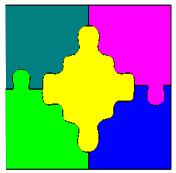
Cumulative by Decomposition

```
predicate cumulative(array[int] of var int: s,
                     array[int] of var int: d,
                     array[int] of var int: r, var int: b) =
  assert(index_set(s) == index_set(d) /\
         index_set(s) == index_set(r),
    "cumulative: the array arguments must have identical index sets",
  assert(lb_array(d) >= 0 /\ lb_array(r) >= 0,
    "cumulative: durations and resource usages must be non-negative",
      let {
        set of int: times =
          min([ lb(s[i]) | i in index_set(s) ]) ..
          max([ ub(s[i]) + ub(d[i]) | i in index_set(s) ])
        }
      in
        forall( t in times ) (
          b >= sum( i in index_set(s) ) (
            bool2int( s[i] <= t /\ t < s[i] + d[i] ) * r[i]
          )
        )
    )
);
```

At each time *t*

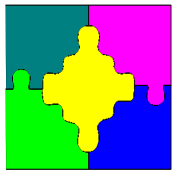
resources used at time *t*

task *i* is active at *t*



Cumulative by Decomposition

- Decomposition has identical propagation to profile based propagator
  - 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 Example with BT

$D(P) = D(C) = D(B) = D(T) = [0..60]$, $P + 30 \leq B$,
$P + 30 \leq 60$, $C + 10 \leq 60$, $B + 15 \leq 60$, $T + 15 \leq 60$,
cumulative($[P,C,B,T]$, $[30,10,20,15]$, $[3,1,2,2]$, 4)

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

Choose P | $P = 0$

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

Choose C | $C = 0$

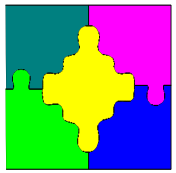
$D(P) = \{0\}$, $D(C) = \{0\}$, $D(B) = [30..45]$, $D(T) = [30..45]$

Choose B | $B = 30$

$D(P) = \{0\}$, $D(C) = \{0\}$, $D(B) = \{30\}$, $D(T) = \{45\}$

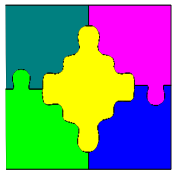
Choose T | $T = 45$

$D(P) = \{0\}$, $D(C) = \{0\}$, $D(B) = \{30\}$, $D(T) = \{45\}$



Programming Propagation

- Modern constraint programming systems, like ECLiPSe or the solver under MiniZinc allow the programmer to add new global constraints and program their propagation rules.
- Usually programmed as rules triggered by events on variable domain
  - Change in lower/upper bound
  - Has fixed value
- Not performed at once but placed in a priority queue



Propagation and Search

- search($F0, Fn, D$)

  D = isolv($F0$,Fn, D)

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

  if (exists x where $|D(x)| > 1$)

    choose $\{c1,..,cm\}$ implied by $C \wedge D$

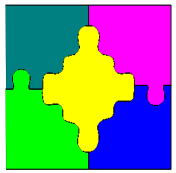
    for i in $1..m$

      if (search($F0$ union Fn, f_{ci}, D)

        return *true*

    return *false*

  return *true*



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 the objective function *f*.
- Because the domains are finite we can use a solver to build a simple optimizer *for minimization*

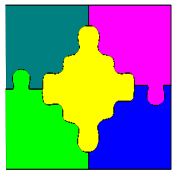
retry_int_opt(C, D, f, *best_so_far*)

$D2$:= solve_satisfaction(C,,D)

if $D2$ is a false domain **then return** *best_so_far*

let *sol* be the solution corresponding to $D2$

return retry_int_opt($C \wedge f < sol(f)$, D, f, *sol*)



Retry Optimization Example

Smugglers knapsack problem (optimize profit)

$$\text{minimize } -15W - 10P - 7C \text{ subject to}$$

capacity *profit*

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

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

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

No next solution!

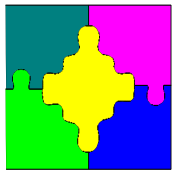
$$D(W) = [0..0], D(P) = [1..1], D(C) = [3..3]$$

Corresponding solution

$$sol = \{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$$

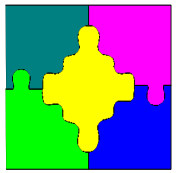
Return best solution

$$sol(f) \equiv -32$$



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 $f < best(f)$
- Very similar to branch-and-cut methods
  - Use consistency techniques instead of linear relaxation



Backtracking Optimization (Ex.)

Smugglers knapsack problem (whiskey available)

capacity *profit*

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

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

Current domain:

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

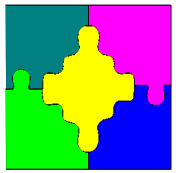
after bounds consistency

$W = 0$

$P = 1$

(0,1,3)

Solution Found: add constraint



Backtracking Optimization (Ex.)

Smugglers knapsack problem (whiskey available)

capacity *profit*

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

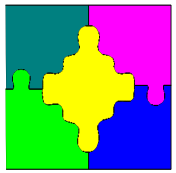
$$-15W - 10P - 7C < -31 \wedge$$

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

Initial bounds consistency

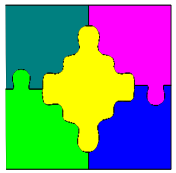
$W = 0$			$W = 1$	$W = 2$
$P = 1$	$P = 2$	$P = 3$	**(1,1,1)**	*false*
(0,1,3)	*false*	*false*		

Return last sol (1,1,1)



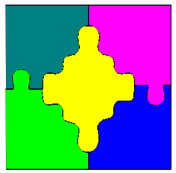
Summary

- Constraint programming techniques are based on backtracking search
- Reduce the search using consistency methods
  - incomplete but faster
  - node, arc, bound, generalized
- Optimization can be 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.



Comparison between CP and MIP

- What are the similarities?
- What are the strengths of MIP?
- What are the strengths of CP?
- Does it make sense to combine them?

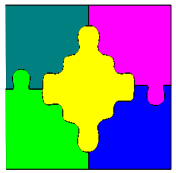


Homework

- Read Chapter 3 of Marriott&Stuckey, 1998
- Solve the Australian Map Colouring problem by hand using simple backtracking, then with arc consistency and backtracking.
- Give propagation rules for constraints of form

$$a_1 X_1 + \ldots + a_n X_n \leq b_1 Y_1 + \ldots + b_m Y_m + c$$

where each $a_{i,}$, $b_i > 0$.



Homework

- Read Chapter 3 of Marriott&Stuckey, 1998
- Solve the Australian Map Colouring problem by hand using simple backtracking, then with arc consistency and backtracking.
- Give propagation rules for constraints of form

  $$b \Leftrightarrow x \leq y + 1$$

- MiniZinc provides decision variables which are sets of integer and normal set operations including cardinality. How would you
  - Represent sets?
  - Program these constraints using propagation rules?