PART II: Constraint Propagation & Global Constraints

Constraint Solver

- Enumerates all possible variable-value combinations via a systematic backtracking tree search. Ricerca sistematica nell'albero backtracking
  - indovina
  - Guesses a value for each variable.
- During search, examines the constraints to remove incompatible (inconsistent) values dai from the domains of the future (unexplored) variables, via propagation.
  - Shrinks the domains of the future variables.

Search and Propagation

- Search decisions and propagation are interleaved.

Propagation

$\Downarrow$

$X_i \leftarrow v_j$

$\Downarrow$

Propagation

$\Downarrow$

$X_{i'} \leftarrow v_{j'}$

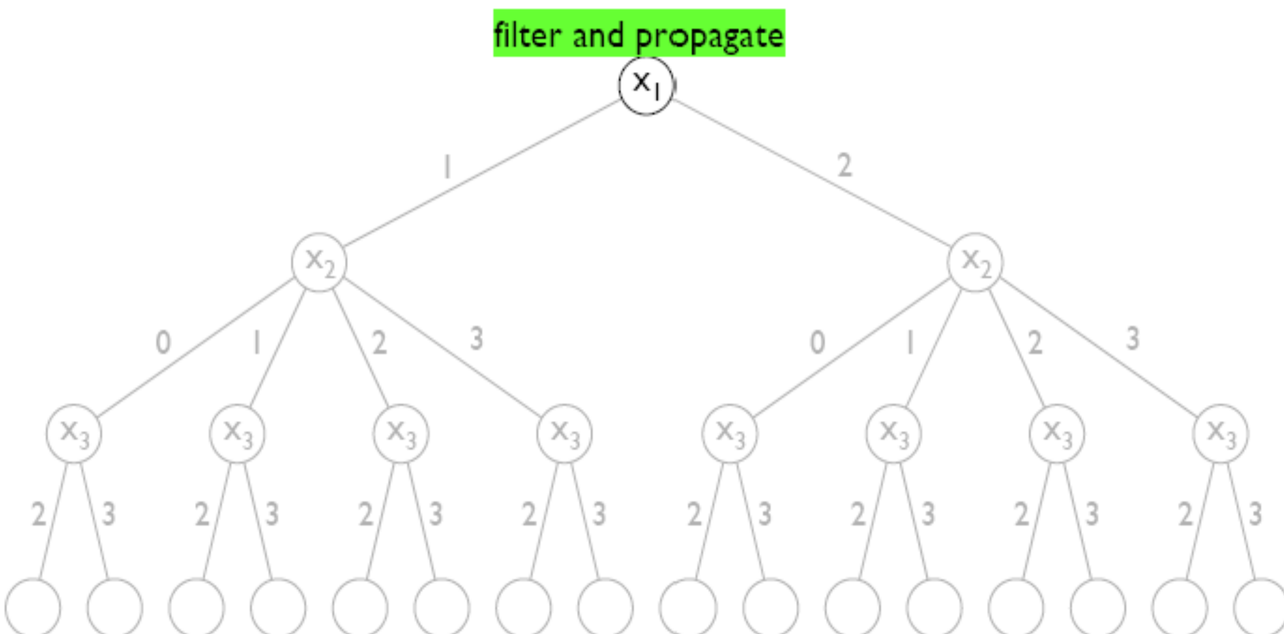
$\Downarrow$

Propagation

Le decisioni di ricerca e la propagazione sono interlacciate.

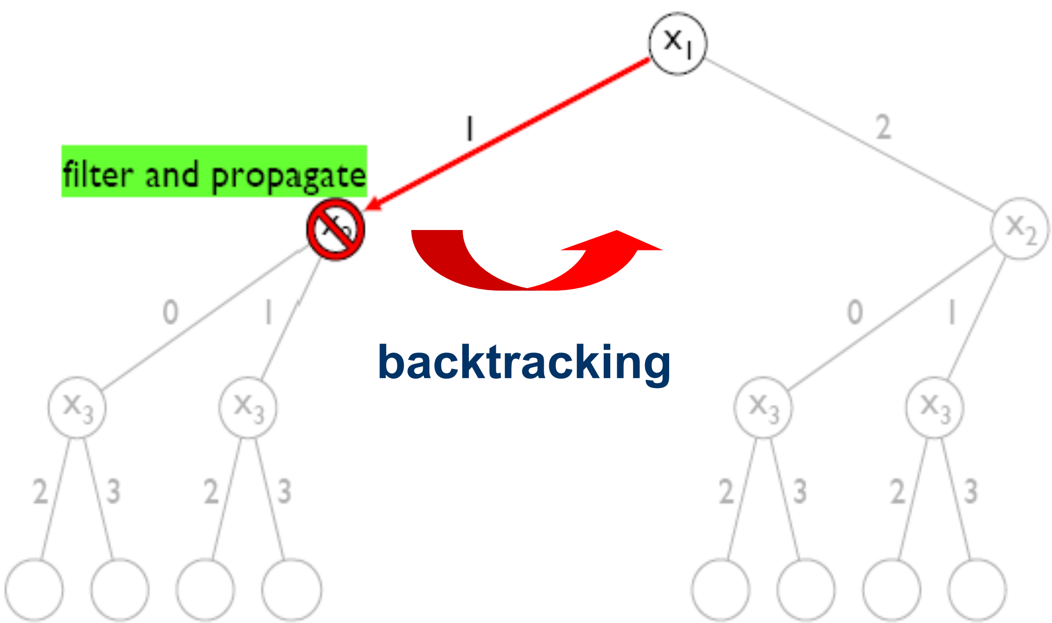
Backtracking Tree Search & Propagation

- $X_1 \in \{1,2\}$ $X_2 \in \{0,1,\cancel{2},\cancel{3}\}$ $X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent($[X_1, X_2, X_3]$)



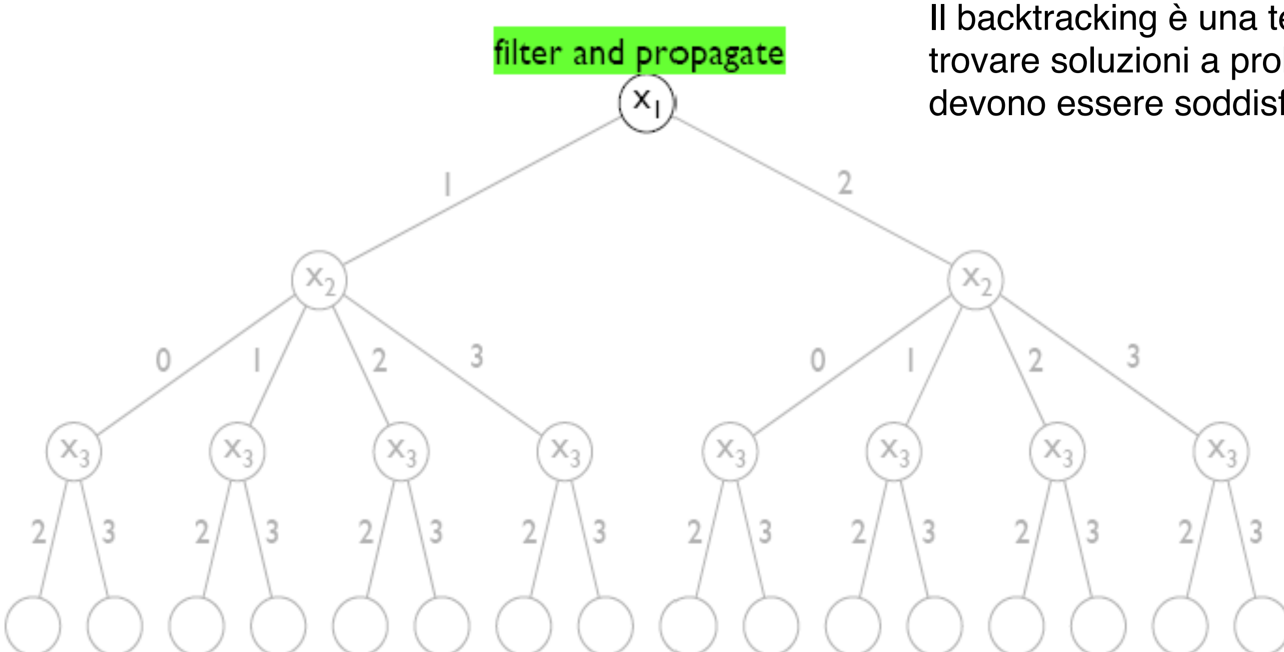
Backtracking Tree Search & Propagation

- $X_1 = 1, \; X_2 \in \{0, \cancel{1}\} \; X_3 \in \{\cancel{2}, \cancel{3}\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent($[X_1, X_2, X_3]$)



Backtracking Tree Search & Propagation

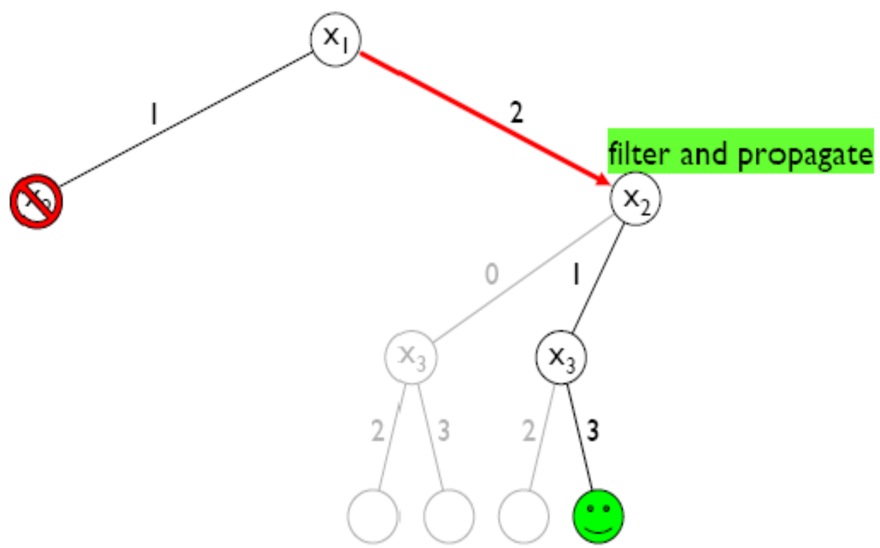
- $X_1 \in \{1,2\}$ $X_2 \in \{0,1,\cancel{2},\cancel{3}\}$ $X_3 \in \{2,3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent($[X_1, X_2, X_3]$)



Il backtracking è una tecnica per trovare soluzioni a problemi in cui devono essere soddisfatti dei vincoli.

Backtracking Tree Search & Propagation

- $X_1 = 2 \;\; X_2 \in \{\cancel{0},1\} \;\; X_3 \in \{\cancel{2},3\}$
- $X_1 > X_2$ and $X_1 + X_2 = X_3$ and alldifferent($[X_1, X_2, X_3]$)



Outline

- Local Consistency
  - Generalized Arc Consistency (GAC)
  - Bounds Consistency (BC)
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation
  - Global Constraints
- Global Constraints for Generic Purposes

Local Consistency

rileva assegnazioni parziali incoerenti.

- A form of inference which detects inconsistent partial assignments.
  - Local, because we examine individual constraints.

Le coerenze locali più diffuse sono basate sul dominio:

- Popular local consistencies are domain-based:
  - Generalized Arc Consistency (GAC).
    - Also referred to as Hyper-arc or Domain Consistency;

limiti

  - Bounds Consistency (BC).

rilevano

  - They detect inconsistent partial assignments of the form $X_i = j$, hence:
    - j can be removed from $D(X_i)$ via propagation;
    - propagation can be implemented easily.

Generalized Arc Consistency (GAC)

- A constraint C defined on k variables $C(X_1,\ldots, X_k)$ gives the set of allowed combinations of values (i.e. allowed tuples).
  - $C \subseteq D(X_1) \times \ldots \times D(X_k)$
  - E.g., $D(X_1) = \{0,1\}$, $D(X_2) = \{1,2\}$, $D(X_3) = \{2,3\}$ C: $X_1 + X_2 = X_3$

$$C(X_1,X_2,X_3) = \{(0,2,2), (1,1,2), (1,2,3)\}$$

Each allowed tuple $(d_1,\ldots,d_k) \in C$ where $d_i \in X_i$ is a support for C.

GAC

- C(X_1,…, X_k) is GAC iff:
  - for all X_i in {X_1,…, X_k}, for all v $\in$ D(X_i), v belongs to a support for C.
- Called Arc Consistency (AC) when k = 2.
- A CSP is GAC iff all its constraints are GAC.

Examples

- $D(X_1) = \{1,2,3\}$, $D(X_2) = \{2,3,4\}$, C: $X_1 = X_2$
  - AC(C)?
    - $1 \in D(X_1)$ and $4 \in D(X_2)$ do not have a support.
    - $X_1 = 1$ and $X_2 = 4$ are inconsistent partial assignments.

- $D(X_1) = \{1,2,3\}$, $D(X_2) = \{1,2\}$, $D(X_3) = \{1,2\}$, C: alldifferent($[X_1, X_2, X_3]$)
  - GAC(C)?
    - $1 \in D(X_1)$ and $2 \in D(X_1)$ do not have support.
    - $X_1 = 1$ and $X_1 = 2$ are inconsistent partial assignments.

Bounds Consistency (BC)

- Defined for totally ordered (e.g. integer) domains.

  rilassa
- Relaxes the domain of X_i from $D(X_i)$ to $[\min(X_i)..\max(X_i)]$.
  - E.g., $D(X_i) = \{1,3,5\} \rightarrow [1..5]$
- A bound support is a tuple $(d_1,\ldots,d_k) \in C$ where $d_i \in [\min(X_i)..\max(Xi)]$.
- $C(X_1,\ldots, X_k)$ is BC iff:

  appartengono
  - For all X_i in $\{X_1,\ldots, X_k\}$, $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

BC

- Disadvantage

  rilevare

  - BC might not detect all GAC inconsistencies in general.
    - We need to search more.
- Advantages

  cercare che in un

  - Might be easier to look for a support in a range than in a domain.

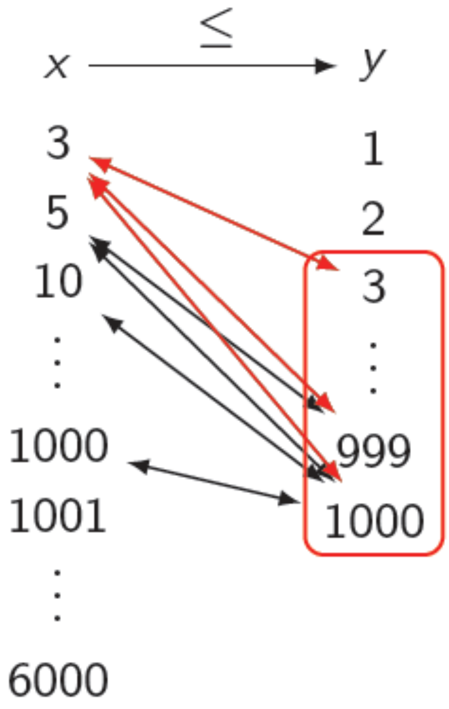
    raggiungere + economico

  - Achieving BC is often cheaper than achieving GAC.
    - Of interest in arithmetic constraints defined on integer variables with large domains.

    monotoni

  - Achieving BC is enough to achieve GAC for monotonic constraints.

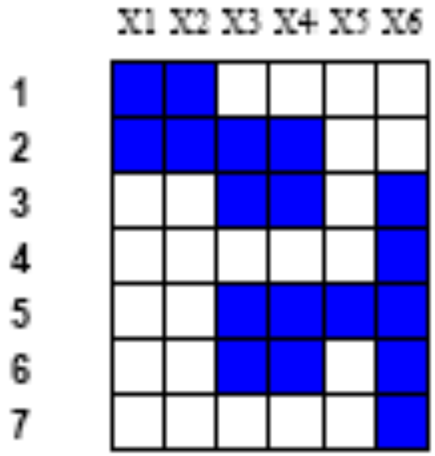
GAC = BC



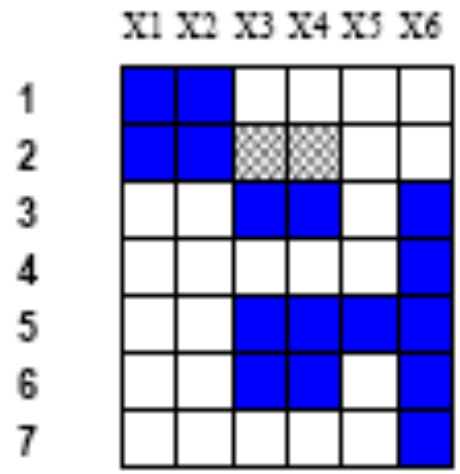
- All values of D(X) ≤ max(Y) are GAC.
- All values of D(Y) ≥ min(X) are GAC.
- Enough to adjust max(X) and min(Y).
  - max(X) ≤ max(Y)
  - min(X) ≤ min(Y)

GAC > BC

- $D(X_1) = D(X_2) = \{1,2\}$, $D(X_3) = D(X_4) = \{2,3,5,6\}$, $X_5 = 5$, $D(X_6) = \{3,4,5,6,7\}$, alldifferent($[X_1, X_2, X_3, X_4, X_5, X_6]$)
- Only $2 \in D(X_3)$ and $2 \in D(X_4)$ have no BC support.



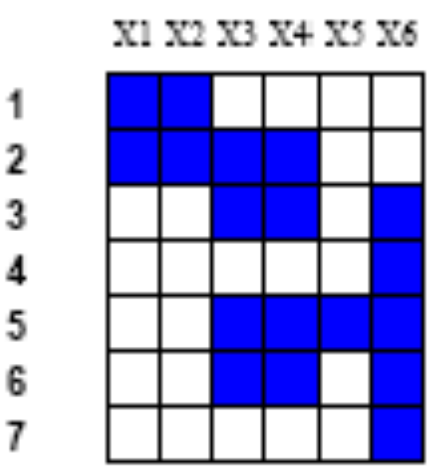
Original



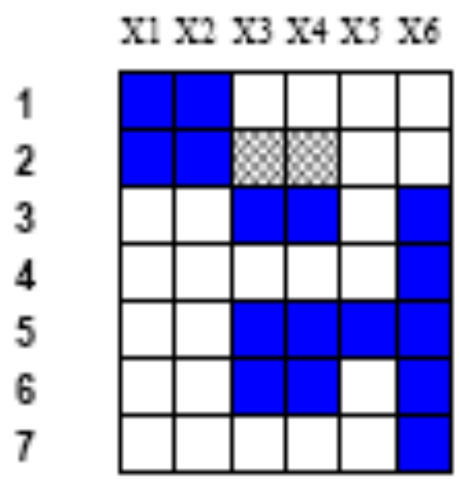
BC

GAC > BC

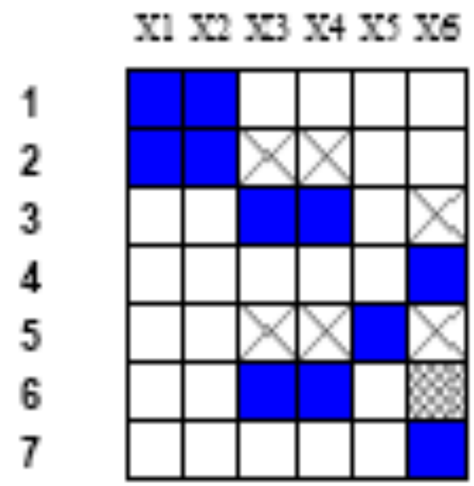
- $D(X_1) = D(X_2) = \{1,2\}$, $D(X_3) = D(X_4) = \{2,3,5,6\}$, $X_5 = 5$, $D(X_6) = \{3,4,5,6,7\}$, alldifferent($[X_1, X_2, X_3, X_4, X_5, X_6]$)
- $\{2,5\} \in D(X_3)$, $\{2,5\} \in D(X_4)$, $\{3,5,6\} \in D(X_6)$ have no GAC support.



Original



BC



GAC

Outline

- Local Consistency
  - Generalized Arc Consistency (GAC)
  - Bounds Consistency (BC)
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation
  - Global Constraints
- Global Constraints for Generic Purposes

Constraint Propagation

- Can appear under different names:
  - constraint relaxation
  - filtering
  - local consistency enforcing, …
- A local consistency notion defines properties that a constraint C must satisfy after constraint propagation.
  comportamento operativo
  - The operational behaviour is completely left open.
  - We can develop different algorithms with different complexities to
    ottenere
    achieve the same effect.
  - The only requirement is to achieve the required property on C.

Propagation Algorithms

ottenere

- A propagation algorithm achieves a certain level of rimuovendo consistency on a constraint C by removing the inconsistent values from the domains of the variables in C.
- The level of consistency depends on C.
  - GAC if an efficient propagation algorithm can be developed.
  - Otherwise BC or a lower level of consistency.

Propagation Algorithms

- When solving a CSP with multiple constraints:
  - propagation algorithms interact;
  - a propagation algorithm can wake up an already propagated constraint to be propagated again!
  - in the end, propagation reaches a fixed-point and all constraints reach a level of consistency;
  - the whole process is referred as constraint propagation.

Example

- $D(X_1) = D(X_2) = D(X_3) = \{1,2,3\}$
  C_1: alldifferent($[X_1, X_2, X_3]$) C_2: $X_2 < 3$ C_3: $X_3 < 3$
- Let's assume:
  - the order of propagation is C_1, C_2, C_3;
  - propagation algorithms maintain (G)AC.
- Propagation of C_1:
  - nothing happens, C_1 is GAC.
- Propagation of C_2:
  - 3 is removed from $D(X_2)$, C_2 is now AC.
- Propagation of C_3:
  - 3 is removed from $D(X_3)$, C_3 is now AC.
- C_1 is not GAC anymore, because the supports of $\{1,2\} \in D(X_1)$ in $D(X_2)$ and $D(X_3)$ are removed by the propagation of C_2 and C_3.
- Re-propagation of C_1:
  - 1 and 2 are removed from $D(X_1)$, C_1 is now AC.

Properties of Propagation Algorithms

- It may not be enough to remove inconsistent values from domains once.
- A propagation algorithm must wake up again when necessary, otherwise may not achieve the desired local consistency property.
- Events that can trigger a constraint propagation:
  - when the domain of a variable changes (for GAC);
  - when the domain bounds of a variable changes (for BC);
  - when a variable is assigned a value;
  - …

Complexity of Propagation Algorithms

- Assume $|D(X_i)| = d$.
- Following the definition of the local consistency property:
  - one time AC propagation on a $C(X_1,X_2)$ takes $O(d^2)$ time.
- We can do better!

Examples

- C: $X_1 = X_2$
  - $D(X_1) = D(X_2) = D(X_1) \cap D(X_2)$
  - Complexity: the cost of the set intersection operation
- C: $X_1 \neq X_2$
  - When $D(X_i) = \{v\}$, remove v from $D(X_j)$.
  - Complexity: O(1)
- C: $X_1 \leq X_2$
  - $\max(X_1) \leq \max(X_2)$, $\min(X_1) \leq \min(X_2)$
  - Complexity: O(1)

Outline

- Local Consistency
  - Generalized Arc Consistency (GAC)
  - Bounds Consistency (BC)
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation
  - Global Constraints
    - Decompositions
    - Ad-hoc Algorithms
- Global Constraints for Generic Purposes

Specialized Propagation

- Propagation specific to a given constraint.
- Advantages
  - Exploits the constraint semantics.
  - Potentially much more efficient than a general propagation approach.

Specialized BC Propagation

- C: $X_1 = X_2 + X_3$
- Observation
  - $\min(X_1)$ cannot be smaller than $\min(X_2) + \min(X_3)$.
  - $\max(X_1)$ cannot be larger than $\max(X_2) + \max(X_3)$.
  - $\min(X_2)$ cannot be smaller than $\min(X_1) - \max(X_3)$.
  - $\max(X_2)$ cannot be larger than $\max(X_1) - \min(X_3)$.
  - X_3 analogous to X_2.
- BC propagation rules
  - $\max(X_1) \leq \max(X_2) + \max(X_3)$, $\min(X_1) \geq \min(X_2) + \min(X_3)$
  - $\max(X_2) \leq \max(X_1) - \min(X_3)$, $\min(X_2) \geq \min(X_1) - \max(X_3)$
  - Similarly for X_3

Example

- $D(X_1) = [4,9]$, $D(X_2) = [3,5]$, $D(X_3) = [2,3]$
  C: $X_1 = X_2 + X_3$

Example

- $D(X_1) = [5,8]$, $D(X_2) = [3,5]$, $D(X_3) = [2,3]$
  C: $X_1 = X_2 + X_3$
- Propagation
  - $\max(X_1) \leq \max(X_2) + \max(X_3)$, $\min(X_1) \geq \min(X_2) + \min(X_3)$

Example

- $D(X_1) = [5,8]$, $D(X_2) = [3,5]$, $D(X_3) = [2,3]$
  C: $X_1 = X_2 + X_3$
- Propagation
  - $\max(X_1) \leq \max(X_2) + \max(X_3)$, $\min(X_1) \geq \min(X_2) + \min(X_3)$
  - $\max(X_2) \leq \max(X_1) - \min(X_3)$, $\min(X_2) \geq \min(X_1) - \max(X_3)$
  - Similarly for X_3

Example

- $X_1 = 5$, $D(X_2) = [3,5]$, $D(X_3) = [2,3]$
  C: $X_1 = X_2 + X_3$
- Propagation
  - $\max(X_1) \leq \max(X_2) + \max(X_3)$, $\min(X_1) \geq \min(X_2) + \min(X_3)$
  - $\max(X_2) \leq \max(X_1) - \min(X_3)$, $\min(X_2) \geq \min(X_1) - \max(X_3)$
  - Similarly for X_3

Example

- $X_1 = 5$, $D(X_2) = [3]$, $D(X_3) = [2]$

  C: $X_1 = X_2 + X_3$
- Propagation
  - $\max(X_1) \le \max(X_2) + \max(X_3)$, $\min(X_1) \ge \min(X_2) + \min(X_3)$
  - $\max(X_2) \le \max(X_1) - \min(X_3)$, $\min(X_2) \ge \min(X_1) - \max(X_3)$
  - Similarly for X_3

Specialized Propagation

- Propagation specific to a given constraint.
- Advantages
  - sfrutta
    Exploits the constraint semantics.
  - Potentially much more efficient than a general propagation approach.
- Disadvantages
  - Limited use.
  - Not always easy to develop one.
- Worth developing for recurring constraints.

Global Constraints

- Capture complex, non-binary and recurring combinatorial substructures arising in a variety of applications.
- incorpora
  Embed specialized propagation which exploits the substructure.

Benefits of Global Constraints

- Modelling benefits
  - Reduce the gap between the problem statement and the model.
  - May allow the expression of constraints that are otherwise not possible to state using primitive constraints (semantic).
- Solving benefits
  - Strong inference in propagation (operational).
  - Efficient propagation (algorithmic).

Global Constraints

- Capture complex, non-binary and recurring combinatorial substructures arising in a variety of applications.
- Embed specialized propagation which exploits the substructure.

Benefits of Global Constraints

- Modelling benefits
  - Reduce the gap between the problem statement and the model.
  - May allow the expression of constraints that are otherwise not possible to state using primitive constraints (semantic).
- Solving benefits
  - Strong inference in propagation (operational).
  - Efficient propagation (algorithmic).

Some Groups of Global Constraints

- Counting
- Sequencing
- Scheduling
- Ordering
- Balancing
- Distance
- Packing
- Graph-based
- …

Counting Constraints

- Restrict the number of variables satisfying a condition or the number of times values are taken.

Alldifferent Constraint

- alldifferent($[X_1, X_2, \ldots, X_k]$) iff
  $X_i \neq X_j$ for $i < j \in \{1,\ldots,k\}$
  - permutation constraint with $|D(X_i)| = k$.
  - alldifferent([3,5,2,1,4])
- Useful in a variety of context, like:
  - puzzles (e.g., sudoku and n-queens);
  - timetabling (e.g. allocation of activities to different slots);
  - scheduling (e.g. a team can play at most once in a week);
  - configuration (e.g. a particular product cannot have repeating components).

Nvalue Constraint

- Constrains the number of distinct values assigned to the variables.
- Nvalue($[X_1, X_2, \ldots, X_k]$, N) iff $N = |\{X_i \mid 1 \leq i \leq k\}|$
  - Nvalue([1, 2, 2, 1, 3], 3).
  - alldifferent when N = k.
- Useful e.g. in:
  - resource allocation (e.g. limit the number of resource types).

Global Cardinality Constraint

- Constrains the number of times each value is taken by the variables.
- gcc($[X_1, X_2, \ldots, X_k], [v_1, \ldots, v_m], [O_1, \ldots, O_m]$) iff

  forall $j \in \{1,\ldots, m\}$ $O_j = |\{X_i \mid X_i = v_j, 1 \leq i \leq k\}|$
  - gcc([1, 1, 3, 2, 3], [1, 2, 3, 4], [2, 1, 2, 0])
  - alldifferent when $O_j \leq 1$.
- Useful e.g. in:
  - resource allocation (e.g. limit the usage of each resource).

Among Constraint

- Constrains the number of variables taken from a given set of values.
- among($[X_1, X_2, \ldots, X_k]$, s, N) iff
  $$N = |\{i \mid X_i \in s, 1 \leq i \leq k\}|$$
  – among([1, 5, 3, 2, 5, 4], {1,2,3,4}, 4)
- among($[X_1, X_2, \ldots, X_k]$, s, l, u) iff
  $$l \leq |\{i \mid X_i \in s, 1 \leq i \leq k\}| \leq u$$
  – among([1, 5, 3, 2, 5, 4], {1,2,3,4}, 3, 4)
- Useful in sequencing problems, as we see next.

Sequencing Constraints

- Ensure a sequence of variables obey certain patterns.

Sequence/AmongSeq Constraint

- Constrains the number of values taken from a given set in any subsequence of q variables.
- sequence(l, u, q, [X_1, X_2, …, X_k], s) iff among([X_i, X_{i+1}, …, X_{i+q-1}], s, l, u) for $1 \leq i \leq k-q+1$
  - sequence(1,2,3,[1,0,2,0,3],{0,1})
- Useful e.g. in:
  - rostering (e.g. every employee has 2 days off in any 7 day of period);
  - production line (e.g. at most 1 in 3 cars along the production line can have a sun-roof fitted).

Scheduling Constraints

- Help schedule tasks with respective release times, duration, and deadlines, using limited resources in a time interval.

Disjunctive Resource Constraint

- Requires that tasks do not overlap in time.
  - Known also as noOverlap constraint.
- Given tasks $t_1, \ldots, t_k$, each associated with a start time S_i and duration D_i:

  disjunctive($[S_1, \ldots, S_k]$,$[D_1, \ldots, D_k]$) iff for all $i < j$

  $$(S_i + D_i \leq S_j) \vee (S_j + D_j \leq S_i)$$

- Useful when a resource can execute at most one task at a time.

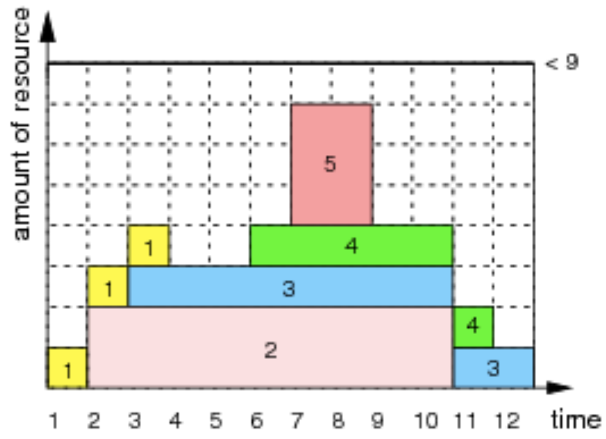
Cumulative Resource Constraint

- Constrains the usage of a shared resource.
- Given tasks $t_1, \ldots, t_k$, each associated with a start time S_i, duration D_i, resource requirement R_i, and a resource with a capacity C:

  cumulative($[S_1, \ldots, S_k]$, $[D_1, \ldots, D_k]$, $[R_1, \ldots, R_k]$, C) iff

  $\sum_{i|S_i \le u < S_i + D_i} R_i \le C$ forall u in D

- Useful when a resource with a capacity can execute multiple tasks at a time.



Ordering Constraints

- Enforce an ordering between the variables or the values.

Lexicographic Ordering Constraint

- Requires a sequence of variables to be lexicographically less than or equal to another sequence of variables.
- lex≤($[X_1, X_2, \dots, X_k]$, $[Y_1, Y_2, \dots, Y_k]$) holds iff:

$$X_1 \leq Y_1 \wedge$$
$$(X_1 = Y_1 \rightarrow X_2 \leq Y_2) \wedge$$
$$(X_1 = Y_1 \wedge X_2 = Y_2 \rightarrow X_3 \leq Y_3) \dots$$
$$(X_1 = Y_1 \wedge X_2 = Y_2 \dots . X_{k-1} = Y_{k-1} \rightarrow X_k \leq Y_k)$$

  - lex≤([1, 2, 4],[1, 3, 3])
- Useful in symmetry breaking.
  - Avoid permutations of (groups of) variables.

Permutation of Variables

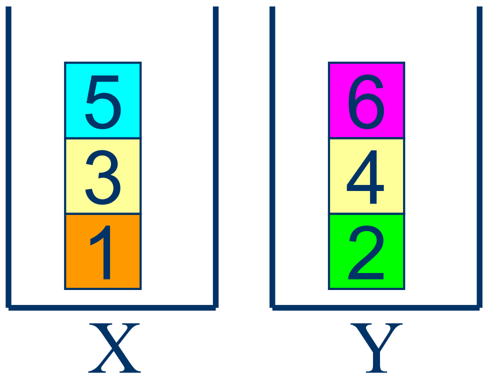
- lex≤([X_1, X_2, …, X_k] , π([X_1, X_2, …, X_k])) for some π.
- E.g., with n-Queens:

```
constraint
     lex_lesseq(array1d(qb), [ qb[j,i] | i,j in 1..n ])
/\   lex_lesseq(array1d(qb), [ qb[i,j] | i in reverse(1..n), j in 1..n ])
/\   lex_lesseq(array1d(qb), [ qb[j,i] | i in 1..n, j in reverse(1..n) ])
/\   lex_lesseq(array1d(qb), [ qb[i,j] | i in 1..n, j in reverse(1..n) ])
/\   lex_lesseq(array1d(qb), [ qb[j,i] | i in reverse(1..n), j in 1..n ])
/\   lex_lesseq(array1d(qb), [ qb[i,j] | i,j in reverse(1..n) ])
/\   lex_lesseq(array1d(qb), [ qb[j,i] | i,j in reverse(1..n) ])
;
```

Permutation of Two Sequences of Variables

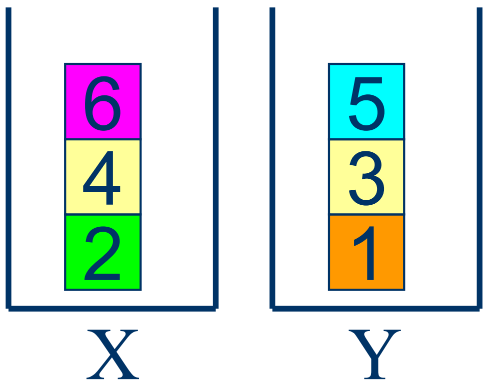
- Assignments of items to two identical bins can be represented by a matrix of Boolean variables:

a)



	i_1	i_2	i_3	i_4	i_5	i_6
X	1	0	1	0	1	0
Y	0	1	0	1	0	1

b)

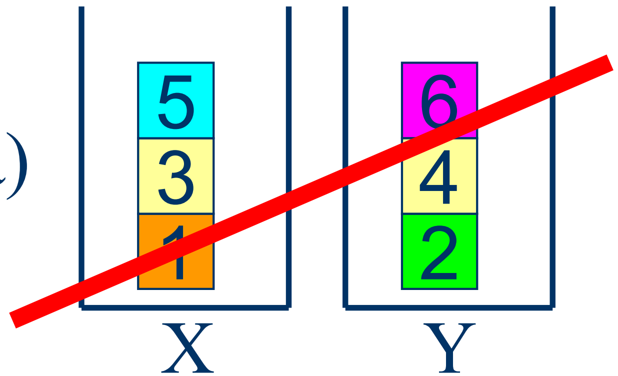


	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	0	1	0	1
Y	1	0	1	0	1	0

Permutation of Two Sequences of Variables

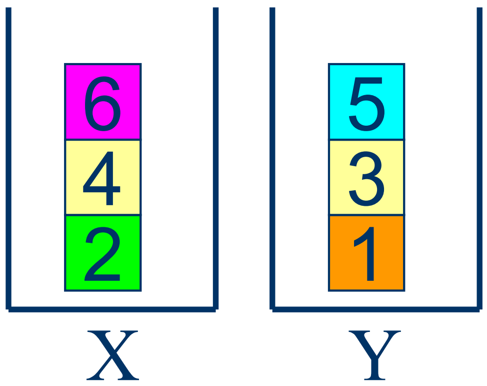
- Need to avoid the symmetric assignments.

a)



	i_1	i_2	i_3	i_4	i_5	i_6
X	1	0	1	0	1	0
Y	0	1	0	1	0	1

b)

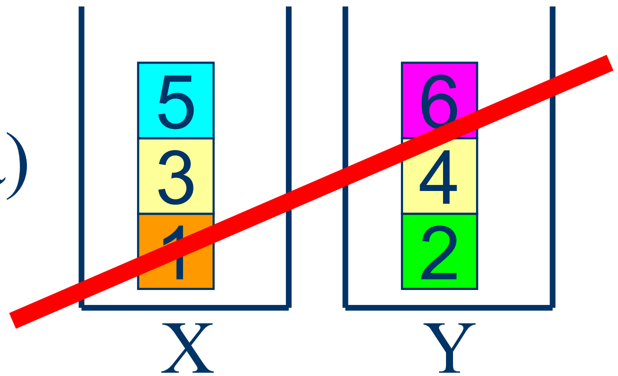


	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	0	1	0	1
Y	1	0	1	0	1	0

Permutation of Two Sequences of Variables

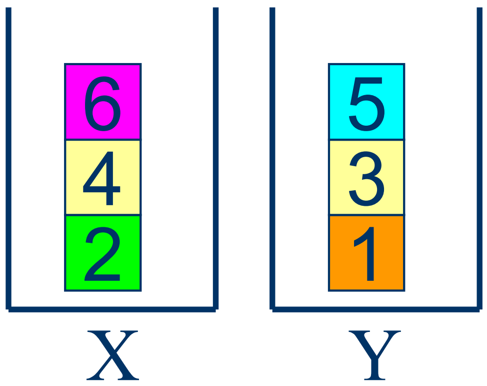
- lex ≤(X , Y).

a)



	i_1	i_2	i_3	i_4	i_5	i_6
X	1	0	1	0	1	0
Y	0	1	0	1	0	1

b)

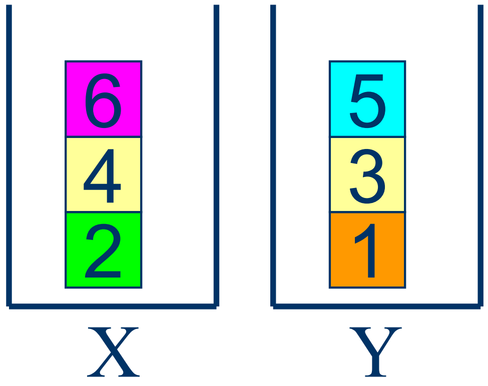


	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	0	1	0	1
Y	1	0	1	0	1	0

Permutation of Two Sequences of Variables

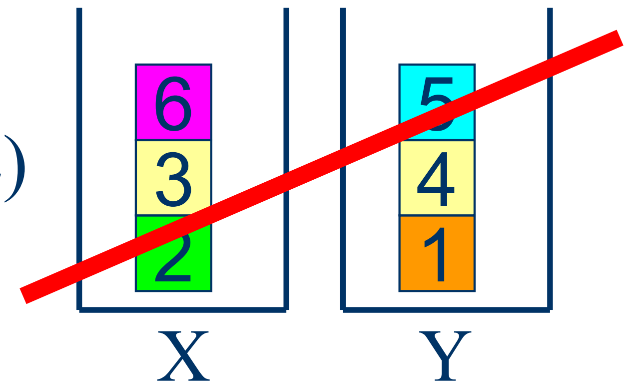
- Need to avoid the symmetric assignments.

b)



	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	0	1	0	1
Y	1	0	1	0	1	0

c)

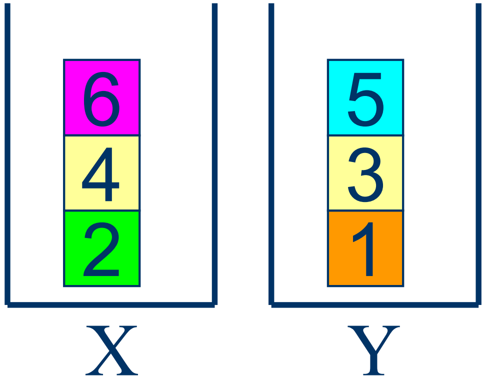


	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	1	0	0	1
Y	1	0	0	1	1	0

Permutation of Two Sequences of Variables

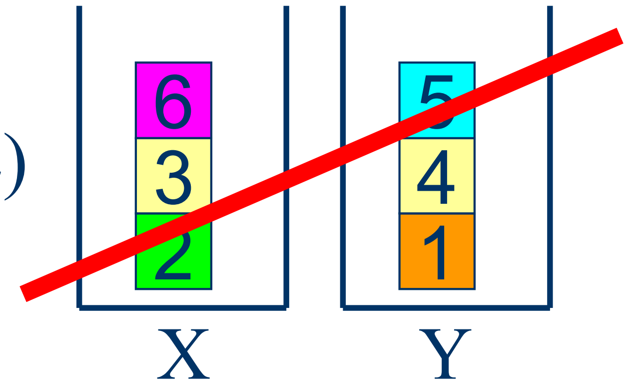
- lex $\leq(i_3, i_4)$.

b)



	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	0	1	0	1
Y	1	0	1	0	1	0

c)



	i_1	i_2	i_3	i_4	i_5	i_6
X	0	1	1	0	0	1
Y	1	0	0	1	1	0

Value Precedence Constraint

- Requires a value to precede another value in a sequence of variables.
- value_precede(v_{j1}, v_{j2}, [X_1, X_2, …, X_k]) holds iff:
  - $\min\{ i \mid X_i = v_{j1} \vee i = k+1\} < \min\{ i \mid X_i = v_{j2} \vee i = k + 2\}$.
  - value_precede(5, 4, [2, 5, 3, 5, 4])
- Useful in symmetry breaking.
  - Avoid permutations of values.

Specialized Propagation for Global Constraints

- How do we develop specialized propagation for global constraints?
- Two main approaches:
  - constraint decomposition;
  - dedicated ad-hoc algorithm.

Constraint Decomposition

- A global constraint is decomposed into smaller and simpler constraints, each of which has a known propagation algorithm.
- Propagating each of the constraints gives a propagation algorithm for the original global constraint.
  - A very effective and efficient method for some global constraints.

A Decomposition of Among

- among([X_1, X_2, …, X_k], s, N)
- Decomposition as a conjunction of logical constrains and a sum constraint.
  - B_i with $D(B_i) = \{0, 1\}$ for $1 \le i \le k$
  - C_i: $B_i = 1 \leftrightarrow X_i \in s$ for $1 \le i \le k$
  - C_{k+1}: $\sum_i B_i = N$
- AC(C_i) for all i and BC($\sum_i B_i = N$) ensures GAC on among.

A Decomposition of Lex

- lex $\leq([X_1, X_2, \ldots, X_k], [Y_1, Y_2, \ldots, Y_k])$
- Decomposition as a conjunction of disjunctions.
  - B_i with $D(B_i) = \{0, 1\}$ for $1 \leq i \leq k+1$ to indicate the vectors have been ordered by position i-1.
  - $B_1 = 0$
  - C_i: $(B_i = B_{i+1} = 0 \text{ AND } X_i = Y_i)$ OR $(B_i = 0 \text{ AND } B_{i+1} = 1 \text{ AND } X_i < Y_i)$ OR $(B_i = B_{i+1} = 1)$ for $1 \leq i \leq k$
- GAC(C_i) for all i ensures GAC on lex $\leq$.

Constraint Decompositions

- May not always provide an effective propagation.
- Often GAC on the original constraint is stronger than (G)AC on the constraints in the decomposition.

A Decomposition of Alldifferent

- alldifferent($[X_1, X_2, \ldots, X_k]$)
- Decomposition as a conjunction of difference constraints.
  - C_{ij}: $X_i \neq X_j$ for $i < j \in \{1,\ldots,k\}$
- AC(C_{ij}) for all $i < j$ is weaker than GAC on alldifferent.
  - E.g., alldifferent($[X_1, X_2, X_3]$) with $D(X_1) = D(X_2) = D(X_3) = \{1,2\}$.
  - alldifferent is not GAC but the decomposition does not prune anything.

A Decomposition of Sequence

- sequence(l, u, q, $[X_1, X_2, \ldots, X_k]$, s)
- Decomposition as a conjunction of among constraints.
  - C_i: among($[X_i, X_{i+1}, \ldots, X_{i+q-1}]$, s, l, u) for $1 \leq i \leq k-q+1$
- GAC(C_i) for all i is weaker than GAC on sequence.
  - E.g., sequence(2, 3, 5, $[X_1, X_2, \ldots, X_7]$, {1}) with $X_1 = X_2 = 1$, $X_6 = 0$, $D(X_i) = \{0,1\}$ for $i \in \{3,4,5,7\}$.
  - sequence is not GAC but the decomposition does not prune anything.

A Decomposition of Sequence

- 1 1 {0,1} {0,1} {0,1} 0 {0,1} q=5, l =2, u =3,v={1}
- 1 1 {0,1} {0,1} {0,1} 0 {0,1} GAC(among)
- 1 1 {0,1} {0,1} {0,1} 0 {0,1} GAC(among)
- 1 1 {0,1} {0,1} {0,1} 0 {0,1} GAC(among)

A Decomposition of Sequence

- 1 1 {0,1} {0,1} {0,1} 0 {0,1} q=5, l =2, u =3,v={1}
- 1 1 {0,1} {0,1} {0,1} 0 {0,1} GAC(among)
- 1 1 {0,1} {0,1} {0,1} 0 {0,1} GAC(among)
- 1 1 {0,1} {0,1} {0,1} 0 {~~0~~,1} GAC(among)

A Decomposition of Lex

- lex $\leq([X_1, X_2, \ldots, X_k], [Y_1, Y_2, \ldots, Y_k])$
- Decomposition as a conjunction of implications
  - $X_1 \leq Y_1$ AND $(X_1 = Y_1 \rightarrow X_2 \leq Y_2)$ AND …

    $(X_1 = Y_1$ AND $X_2 = Y_2$ AND …. $X_{k-1} = Y_{k-1} \rightarrow X_k \leq Y_k)$
- AC on the decomposition is weaker than GAC on lex≤.
  - E.g., lex $\leq([X_1, X_2], [Y_1, Y_2])$ with $D(X_1) = \{0,1\}$, $X_2 = 1$,
    $D(Y_1) = \{0,1\}$, $Y_2 = 0$
  - lex ≤ is not GAC but the decomposition does not prune anything.

Decomposition vs Ad-hoc Algorithm

- Even if a decomposition is effective, may not always provide an efficient propagation.
- Often propagating a constraint via an ad-hoc algorithm is faster than propagating the (many) constraints in the decomposition.
  - Thanks to incremental computation!

Incremental Computation

- A propagation algorithm is often called multiple times.
  - We don't want to re-compute everything each time.
- Incremental computation can improve efficiency.
  - At the first call, some partial results are cached.
  - At the next invoke, we exploit the cached data.
- This requires access to more details about propagation:
  - which variable has been pruned?
  - which values have been pruned?

Dedicated BC Algorithm for Sum

- C: $\sum_i X_i = N$ where X_i and N are integer variables.
  - $\min(N) \geq \sum_i \min(X_i)$
  - $\max(N) \leq \sum_i \max(X_i)$
  - $\min(X_i) \geq \min(N) - \sum_{j \neq i} \max(X_j)$ for $1 \leq i \leq n$
  - $\max(X_i) \leq \max(N) - \sum_{j \neq i} \min(X_j)$ for $1 \leq i \leq n$

BC Decomposition for Sum

- C: $\sum_i X_i = N$ where X_i and N are integer variables.
  - $X_1 + X_2 = Y_1$
  - $Y_1 + X_3 = Y_2$
  - ...
  - $Y_{(n-1)} + X_n = N$

Filtering min(N)

- C: $\sum_i X_i = N$ where X_i and N are integer variables.
  - $\min(X_1) + \min(X_2) \leq \min(Y_1)$
  - $\min(Y_1) + \min(X_3) \leq \min(Y_2)$
  - ...
  - $\min(Y_{(n-1)}) + \min(X_n) \leq \min(N)$

  which is equivalent to

  $$\sum_i \min(X_i) \leq \min(N)$$

Number of Operations

- C: $\sum_i X_i = N$ where X_i and N are integer variables.
  - $\min(X_1) + \min(X_2) \leq \min(Y_1)$
  - $\min(Y_1) + \min(X_3) \leq \min(Y_2)$
  - ...
  - $\min(Y_{(n-1)}) + \min(X_n) \leq \min(N)$

Read access: 2(n-1)
Write access: n-1
Sum: n-1

$$\sum_i \min(X_i) \leq \min(N)$$

Read access: n
Write access: 1
Sum: n-1

Number of Operations

- C: $\sum_i X_i = N$ where X_i and N are integer variables.

  - $\max(X_1) + \max(X_2) \geq \max(Y_1)$
  - $\max(Y_1) + \max(X_3) \geq \max(Y_2)$
  - ...
  - $\max(Y_{(n-1)}) + \max(X_n) \geq \max(N)$

Read access: 2(n-1)
Write access: n-1
Sum: n-1

$$\sum_i \max(X_i) \geq \max(N)$$

Read access: n
Write access: 1
Sum: n-1

Incremental Computation

- C: $\sum_i X_i = N$ where X_i and N are integer variables.

  - max(N) ≤ $\sum_i \max(X_i)$
    - Cache max(N) as max$(N)
    - Whenever the bounds of a variable X_i is pruned:
      - max(N) ≤ max$(N) – (old(max($X_i$)) - max($X_i$))   O(1)

Incremental Computation

- C: $\sum_i X_i = N$ where X_i and N are integer variables.
  - Complexity reduces to O(1) from O(n)

Classical Sum	Incremental Sum
Read access: n	Read access: 3
Write access: 1	Write access: 1
Sum: n-1	Sum: 2

Dedicated Propagation Algorithms

- Dedicated ad-hoc algorithms provide effective and efficient propagation.
- Often:
  - GAC is maintained in polynomial time;
  - many more inconsistent values are detected compared to the decompositions;
  - computation is done incrementally.

Dedicated Propagation Algorithms

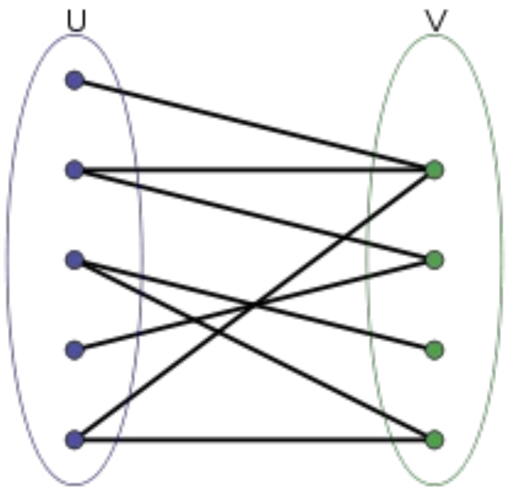
- Dedicated ad-hoc algorithms provide effective and efficient propagation.
- Often:
  - GAC is maintained in polynomial time;
  - many more inconsistent values are detected compared to the decompositions;
  - computation is done incrementally.

A GAC Propagation Algorithm

- Maintains GAC on alldifferent($[X_1, X_2, \ldots, X_k]$) and runs in polynomial time.
  - Jean-Charles Régin, "A Filtering Algorithm for Constraints of Difference in CSPs", in the Proc. of AAAI'1994
- Establishes a relation between the solutions of the constraint and the properties of a graph.
  - Maximal matching in a bipartite graph.
- A similar algorithm can be obtained with the use of flow theory.

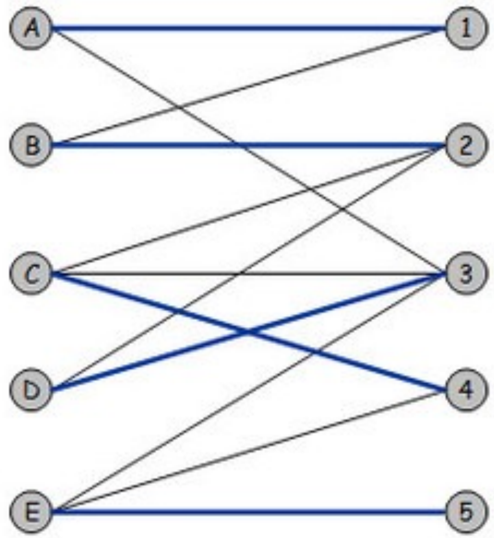
A GAC Algorithm for alldifferent

- A bipartite graph is a graph whose vertices are divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V.



A GAC Algorithm for alldifferent

- A matching in a graph is a subset of its edges such that no two edges have a node in common.
  - Maximal matching is the largest possible matching.
- In a bipartite graph, maximal matching covers one set of nodes.

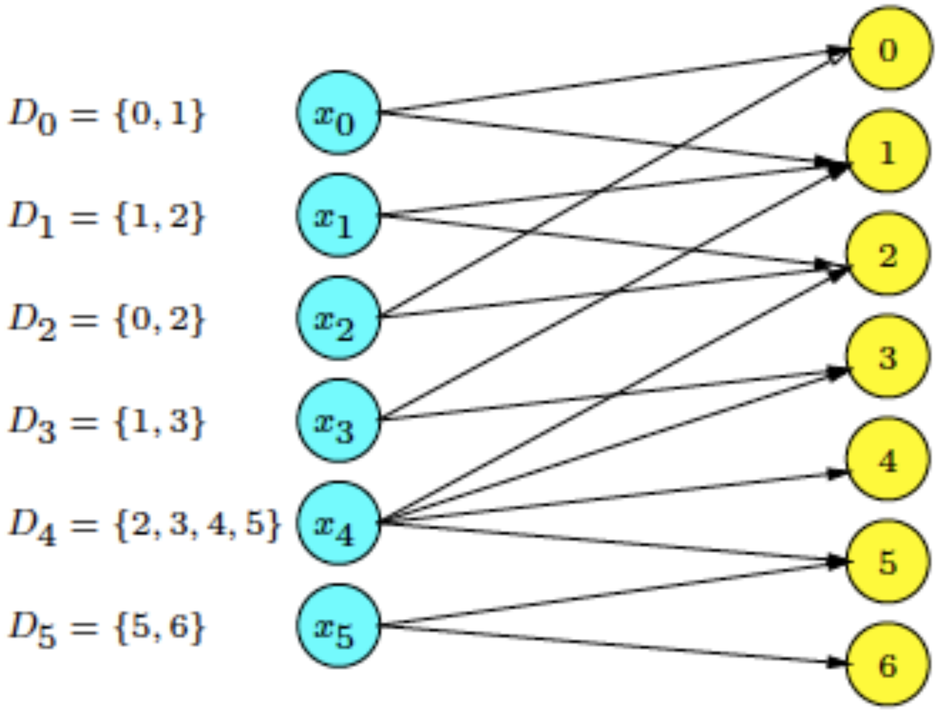


A GAC Algorithm for alldifferent

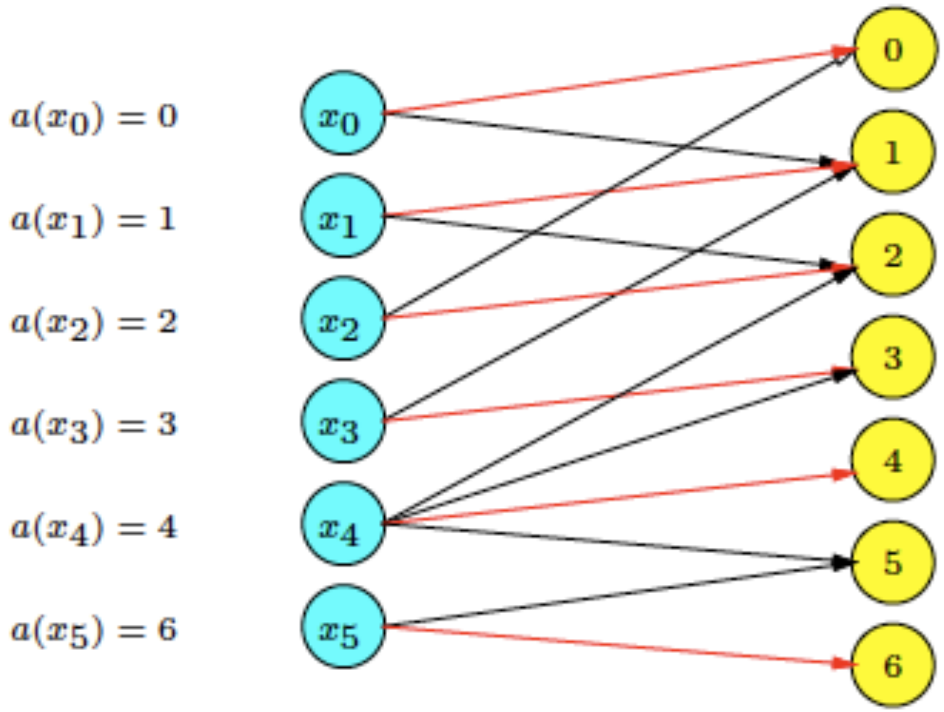
- Observation
  - Given a bipartite graph G constructed between the variables [X_1, X_2, …, X_k] and their possible values (variable-value graph),
  - an assignment of values to the variables is a solution iff it corresponds to a maximal matching in G.
    - A maximal matching covers all the variables.
  - By computing all maximal matchings, we can find all the consistent partial assignments.

Example

Variable-value graph



A Maximal Matching



Another Maximal Matching

$a(x_0) = 1$

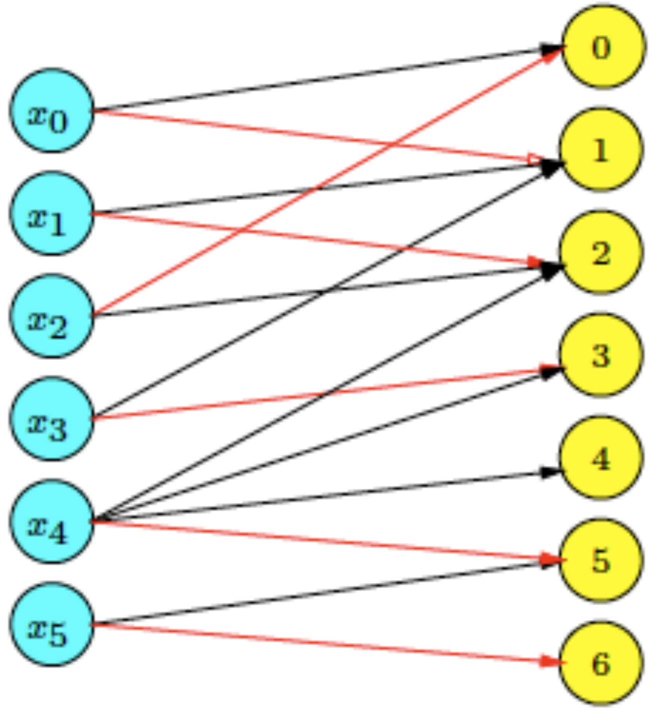
$a(x_1) = 2$

$a(x_2) = 0$

$a(x_3) = 3$

$a(x_4) = 5$

$a(x_5) = 6$



Matching Notations

- Edge
  - matching if takes part in a matching;
  - free otherwise.
- Node
  - matched if incident to a matching edge;
  - free otherwise.
- Vital edge
  - belongs to every maximal matching.

Free, Matched, Matching

$a(x_0) = 1$

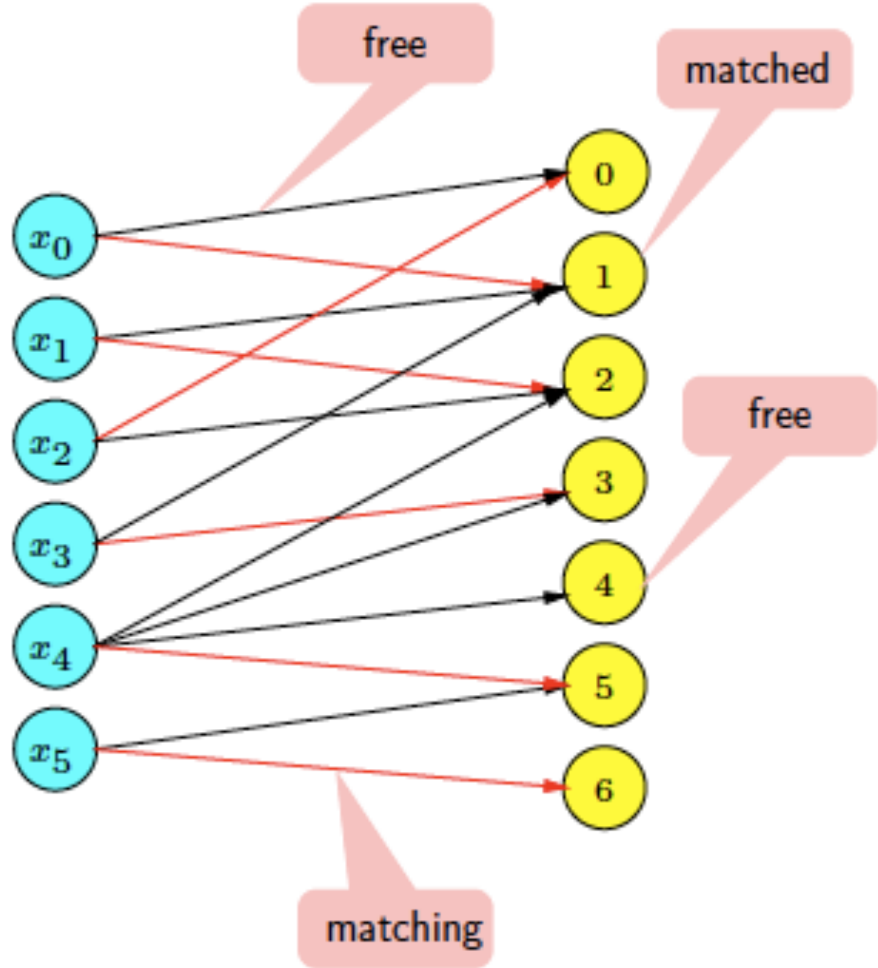
$a(x_1) = 2$

$a(x_2) = 0$

$a(x_3) = 3$

$a(x_4) = 5$

$a(x_5) = 6$



Algorithm

- Compute all maximal matchings.
- No maximal matching exists → failure.
- An edge free in all maximal matchings →
  - Remove the edge.
  - Amounts to removing the corresponding value from the domain of the corresponding variable.
- A vital edge →
  - Keep the edge.
  - Amounts to assigning the corresponding value to the corresponding variable.
- Edges matching in some but not all maximal matchings →
  - Keep the edge.

All Maximal Matchings

- Inefficient to compute them naïvely.
- Use matching theory to compute them efficiently.
  - One maximal matching can describe all maximal matchings!

Alternating Path and Cycle

- Alternating path
  - Simple path with edges alternating free and matching.
- Alternating cycle
  - Cycle with edges alternating free and matching.
- Length of path/cycle
  - Number of edges in the path/cycle.
- Even path/cycle
  - Path/cycle of even length.

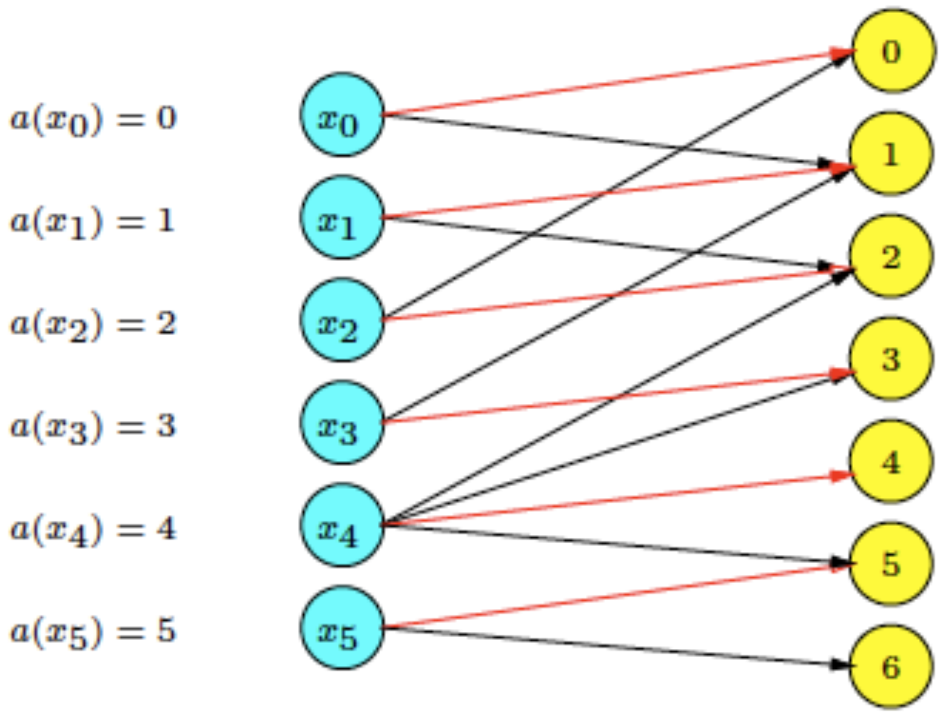
Matching Theory

- A result due to Claude Berge in 1970.
- An edge e belongs to a maximal matching iff for some arbitrary maximal matching M:
  - either e belongs to M;
  - or e belongs to even alternating path starting at a free node;
  - or e belongs to an even alternating cycle.

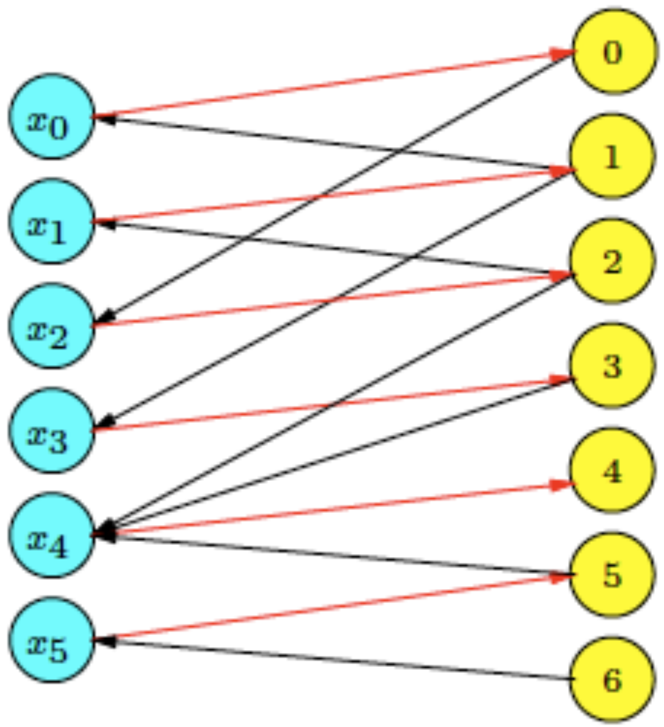
Oriented Graph

- To compute alternating path/cycles, we will orient edges of an arbitrary maximal matching:
  - matching edges → from variable to value;
  - free edges → from value to variable.

An Arbitrary Maximal Matching



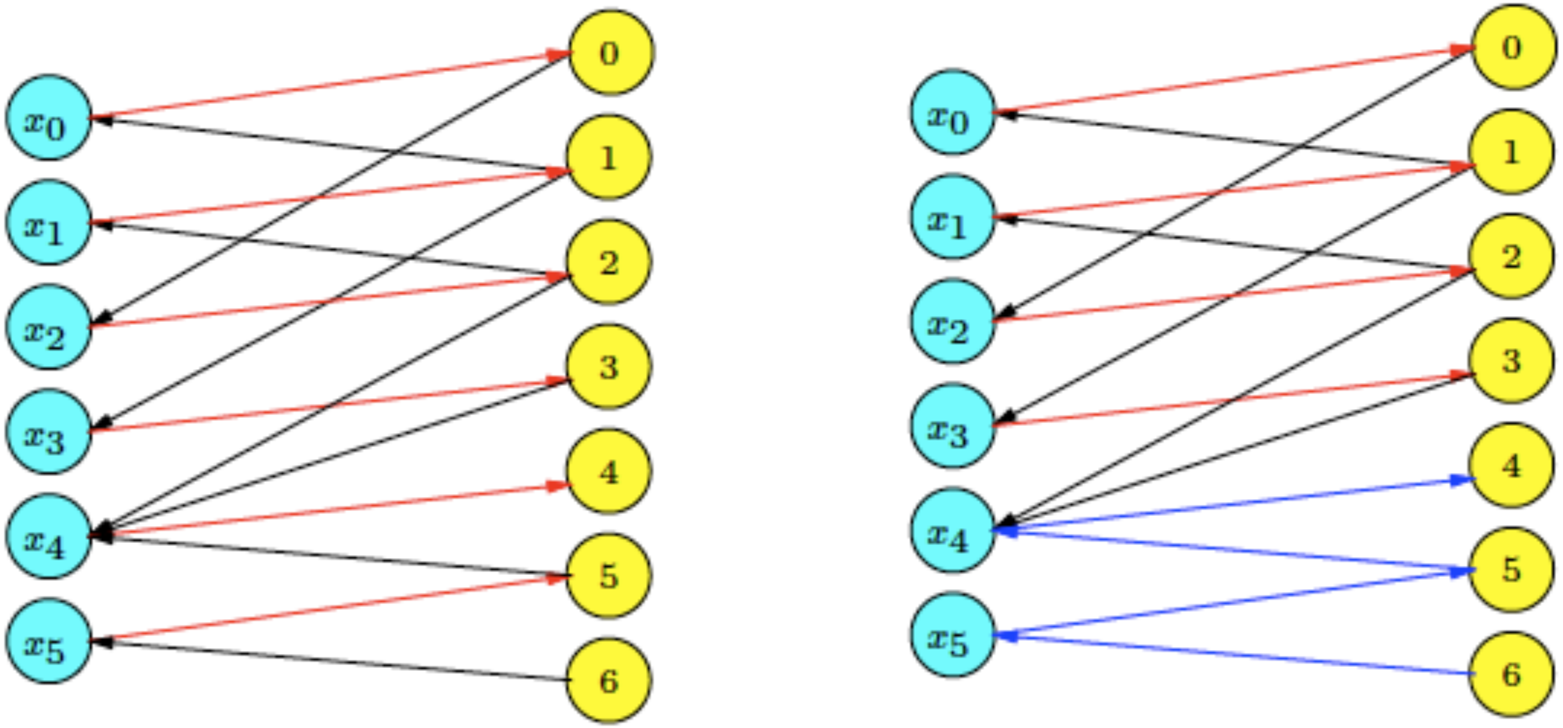
Oriented Graph



Even Alternating Paths

- Start from a free node and search for all nodes on directed simple path.
  - Mark all edges on path.
  - Alternation built-in.
- Start from a value node.
  - Variable nodes are all matched.
- Finish at a value node for even length.

Even Alternating Paths

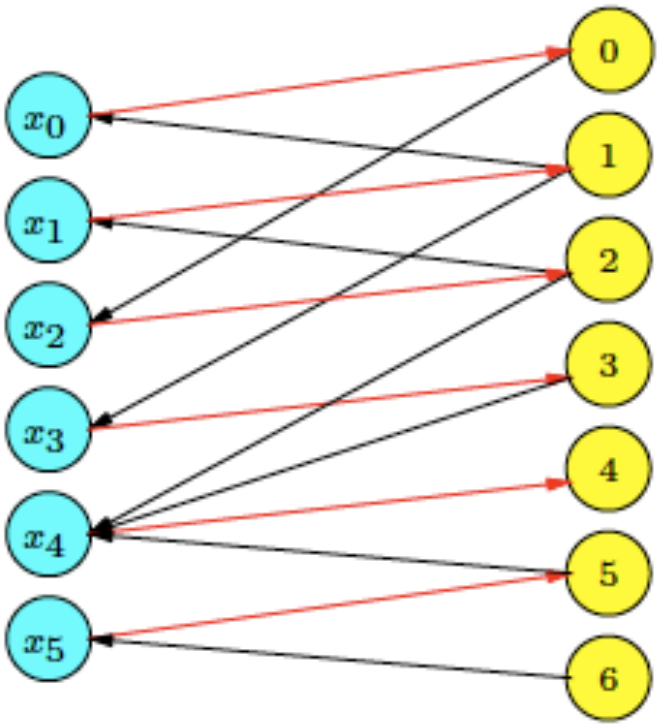


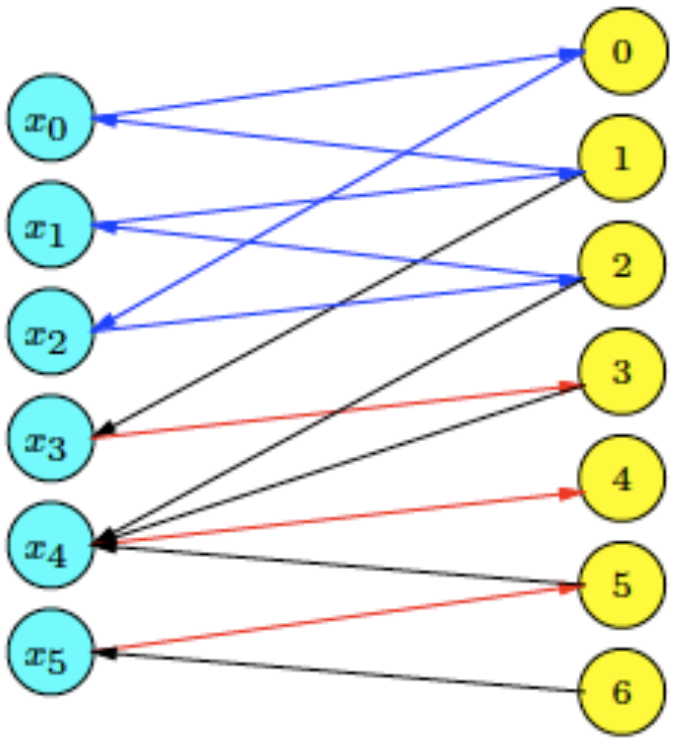
- Intuition: edges can be permuted.

Even Alternating Cycles

- Compute strongly connected components (SCCs).
  - Two nodes a and b are strongly connected iff there is a path from a to b and a path from b to a.
  - Strongly connected component: any two nodes are strongly connected.
  - Alternation and even length built-in.
- Mark all edges in all strongly connected components.

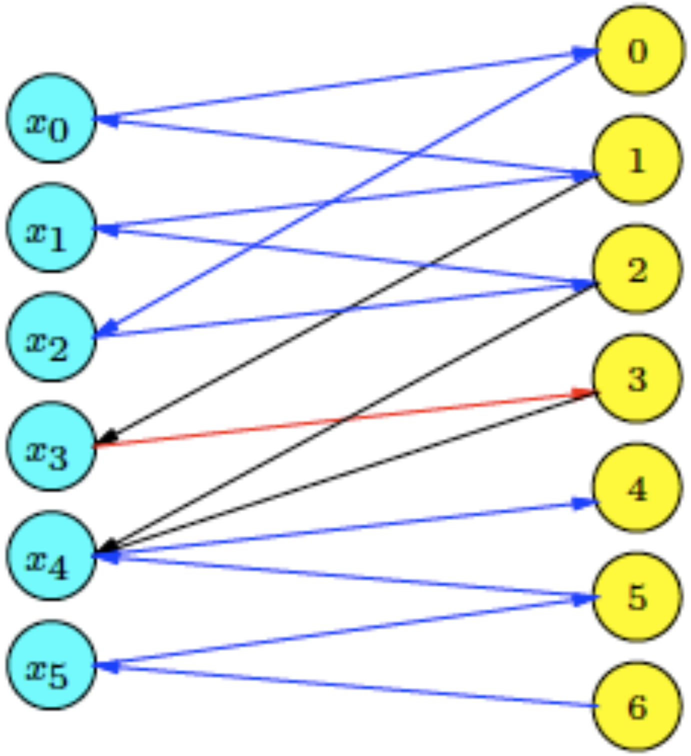
Even Alternating Cycles





- Intuition: variables consume all the values.

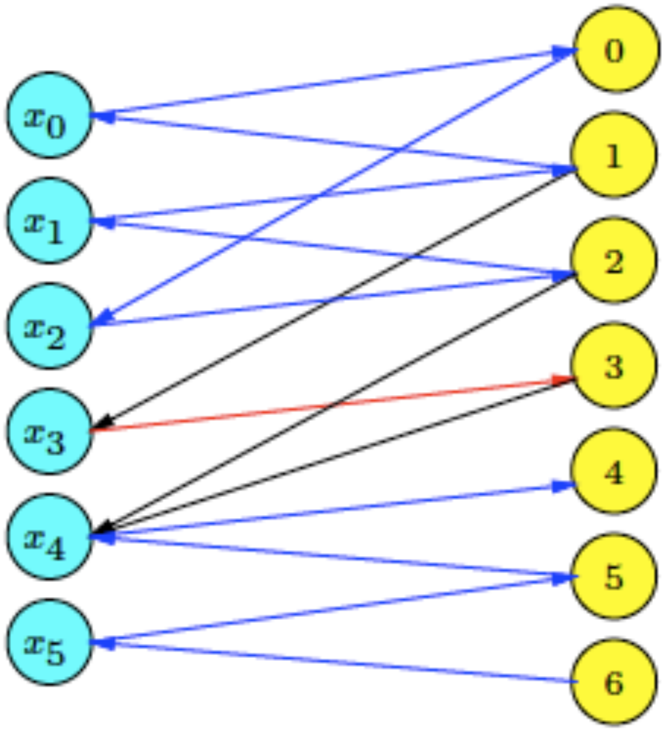
All Marked Edges



Removing Edges

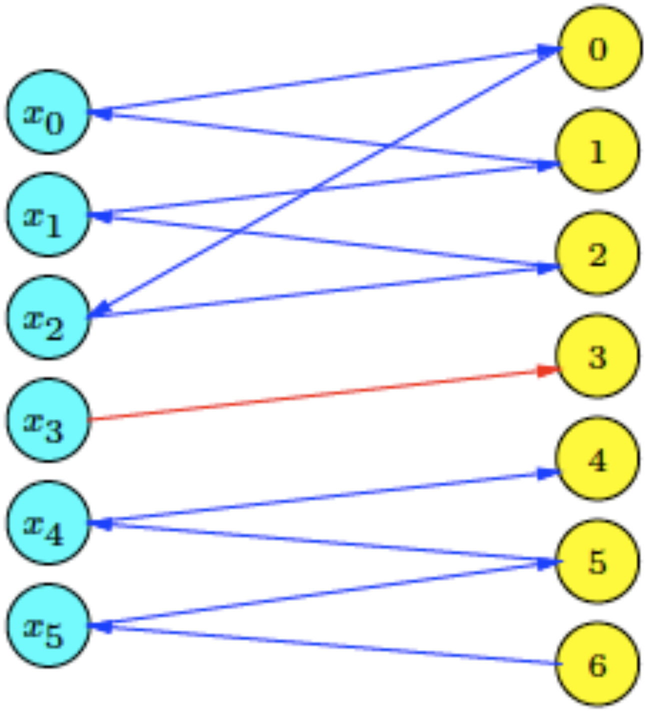
- Remove the edges which are:
  - free (does not occur in our arbitrary maximal matching) and not marked (does not occur in any maximal matching);
  - marked as black in our example.
- Keep the edge matched and not marked.
  - Marked as red in our example.
  - Vital edge!

Removing Edges



$D(X_0) = \{0,1\}$, $D(X_1) = \{1,2\}$, $D(X_2) = \{0,2\}$, $D(X_3) = \{1,3\}$
$D(X_4) = \{2,3,4,5\}$, $D(X_5) = \{5,6\}$

Edges Removed



$D(X_0) = \{0,1\}$, $D(X_1) = \{1,2\}$, $D(X_2) = \{0,2\}$, $D(X_3) = \{\cancel{1},3\}$
$D(X_4) = \{\cancel{2},\cancel{3},4,5\}$, $D(X_5) = \{5,6\}$

Summary of the Algorithm

- Construct the variable-value graph.
- Find a maximal matching M; otherwise fail.
- Orient graph (done while computing M).
- Mark edges starting from free value nodes using graph search.
- Compute SCCs and mark joining edges.
- Remove not marked and free edges.

Incremental Properties

- Keep the variable and value graph between different invocations.
- When re-executed:
  - remove marks on edges;
  - remove edges not in the domains of the respective variables;
  - if a matching edge is removed, compute a new maximal matching;
  - otherwise just repeat marking and removal.

Runtime Complexity

- alldifferent([X_1, X_2, …, X_k]) with m = $\sum_{i \in \{1,..k\}} |D(X_i)|$
- First call
  - Consistency check in O($\sqrt{k}$m) time.
    - Matching → O($\sqrt{k}$ m)
    - Alternating path → O(m)
    - SCCs → O(k+m)
  - Establishing GAC in O(m) time.
- After q variable domains have been modified
  - Matching in O(min{qm, $\sqrt{k}$ m}) time.
  - Establishing GAC in O(m) time.

Dedicated Ad-hoc Algorithms

- Is it always easy to develop a dedicated algorithm for a given constraint?
- A nice semantics often gives us a clue!
  - Graph theory
  - Flow theory
  - Combinatorics
  - Automata theory
  - Dynamic programming
  - Complexity theory, …

Dedicated Ad-hoc Algorithms

- GAC may as well be NP-hard!
  - E.g., nvalue, sequence+gcc, gcc using variables for occurrences.
  - Algorithms which maintain weaker consistencies are of interest.
    - BC
    - Between GAC and BC
    - GAC on some variables, BC on others
    - …

Dedicated Ad-hoc Algorithms

- What if it is difficult to:
  - decompose a constraint;
  - build an efficient and effective dedicated algorithm?

Outline

- Local Consistency
  - Generalized Arc Consistency (GAC)
  - Bounds Consistency (BC)
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation
  - Global Constraints
    - Decompositions
    - Ad-hoc Algorithms
- Global Constraints for Generic Purposes

Global Constraints for Generic Purposes

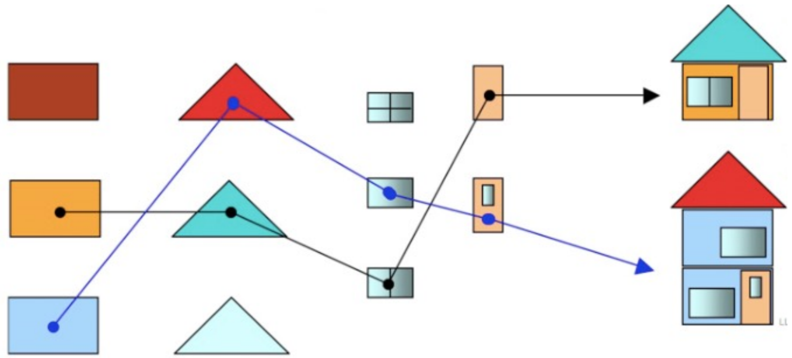
- Help propagate a wide range of constraints.
  - Table constraint.
  - Formal language-based constraints.

Table (Extensional) Constraint

- $C(X_1, X_2) = \{(0,0), (0,2), (1,3), (2,1)\}$
- Several algorithms exist to maintain GAC.
  - More efficient than $O(|D(X_1)|*|D(X_2)|*\ldots*|D(X_k)|)$.
  - More effective than the decomposition.
    - E.g., (X_1= 0 AND X_2 = 2 AND X_3 = 2) OR (X_1= 1 AND X_2 = 1 AND X_3 = 2) OR (X_1= 1 AND X_2 = 2 AND X_3 = 3)

Product Configuration Problems

- Compatibility constraints on product components.
  - Often only certain combination of components work together.
- Compatibility may not be a simple pairwise relationship.



A Configuration Problem

- Valid hw products are defined in a table of compatible components (Products):

Products	Motherboard	CPU	Freq	RAM	Hard drive
$Product_1$	TypeA	Intel	2GHz	5GB	100GB
$Product_2$	TypeB	Intel	3GHz	8GB	200GB
$Product_3$	TypeB	Amd	2GHz	5GB	200GB
…					

- Assume we have products P_i to configure each with 5 components for motherboard, CPU, Freq, RAM and h. drive $[X_{i1},X_{i2},X_{i3}, X_{i4}, X_{i5}]$.
- For each product P_i, we post table($[X_{i1},X_{i2},X_{i3}, X_{i4}, X_{i5}]$, Products).

Crossword Puzzles

- Valid words are defined in a table of compatible letters (i.e. dictionary).
  - table([X_1,X_2,X_3], dictionary)
  - table([X_1,X_{13},X_{16}], dictionary)
  - table([X_4,X_5,X_6,X_7], dictionary)
  - …
- No simple way to decide acceptable words other than to put them in a table.

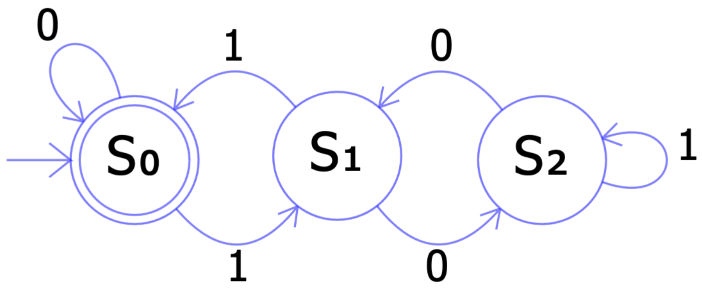
1 C	2 A	3 T		4 T	5 S	6 N	7 I			8 P	9 E	10 R	11 C	12 H
13 E	C	A		14 H	T	O	G		15 T	U	R	T	L	E
16 S	H	I	17 B	A	I	N	U		18 O	R	R		19 O	R
		20 L	A	I	C		21 A	22 B	E	R		23 F	W	D
24 B	25 O	W	L		26 K	27 A	N	E		28 S	29 H	E	D	I
30 S	W	A	L	31 C		32 R	A	S	33 P		34 O	W	E	N
35 E	N	G		36 H	37 A	M	S	T	E	38 R	S		39 R	G
		40 S	41 S	I	M				42 T	A	E	43 M		
44 S	45 F		46 P	A	R	47 A	48 K	49 E	E	T		50 U	51 S	52 A
53 C	E	54 I	C		55 E	Y	E	S		56 S	57 K	I	N	S
58 R	E	T	A	59 W		60 A	N	E	61 W		62 E	R	E	H
63 A	D	S		64 H	65 A	H	N		66 O	67 K	R	A		
68 T	E		69 A	E	S		70 E	71 U	K	A	N	U	72 B	73 A
74 C	R	75 A	T	E	S		76 L	A	E	R		77 Q	U	O
78 H	S	A	E	L			79 S	E	N	T		80 A	T	L

Formal Language-based Constraints

- The table constraint requires precomputing all the solutions of a constraint.
  - May not always be possible or practical.
- We can use a deterministic finite-state automaton to define the solutions.
  - Useful especially when valid assignments need to obey certain patterns.

Deterministic Finite State Automaton

- A dfsa is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string.
  - Recognizes a regular language.
- E.g., a dfa that accepts binary numbers that are multiples of 3.



  - Some accepted strings: 0, 11, 110, 1100, 1001, 10111101, …
  - Not accepted strings: 10, 100, 101, 10100, …

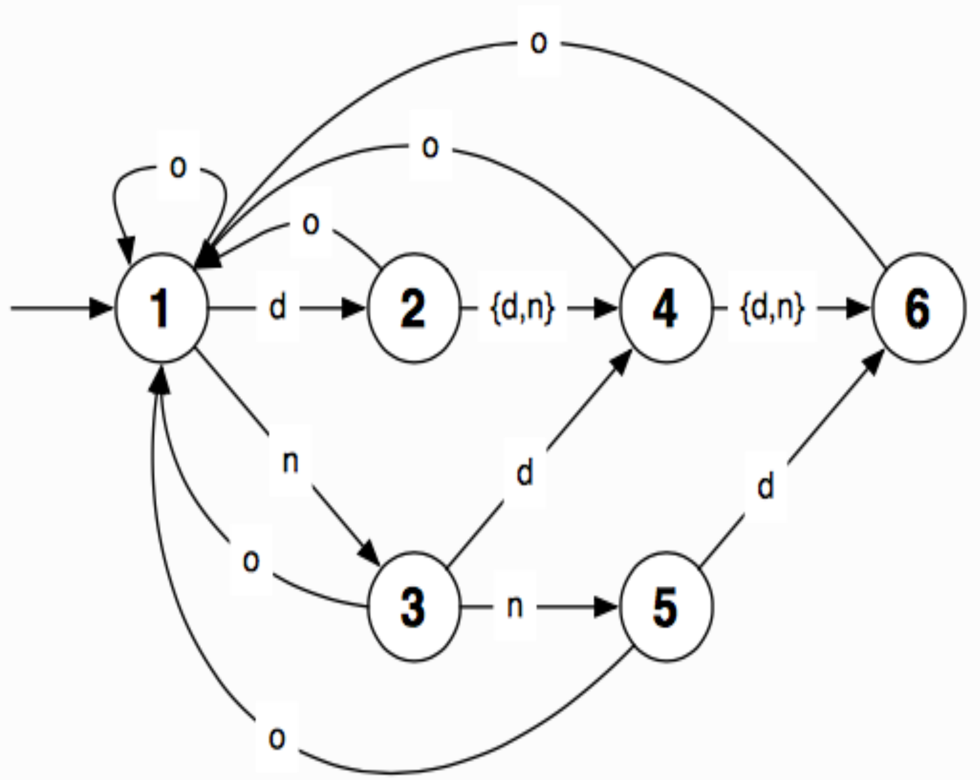
Regular Constraint

- A dfsa A is defined by a 5-tuple (q, sigma, t, q_0, f) where:
  - q : a finite set of states
  - sigma: a set of symbols (i.e. alphabet)
  - t: a partial transition function q x sigma → q
  - q_0 : initial state
  - f ⊆ q: accepting (final) states
- regular([X_1, X_2, …, X_k], A) holds iff <X_1, X_2, …, X_k> forms a string accepted by a dfsa A.

Rostering Problems

- Shifts are subject to regulations.
  - E.g., successive night shifts must be limited.
- In a nurse rostering problem, suppose:
  - each nurse is scheduled for each day either: (d) on day shift, (n) on night shift, or (o) off;
  - in each four day period, a nurse must have at least one day off;
  - no nurse can be scheduled for 3 night shifts in a row.

A Nurse Rostering Problem



- $q = \{q_1, \ldots, q_6\}$
- sigma = {d, n, o}
- t:

	d	n	o
1	2	3	1
2	4	4	1
3	4	5	1
4	6	6	1
5	6	0	1
6	0	0	1

- $q_0 : q_1$
- $f = q = \{q_1, \ldots, q_6\}$

- Assume nurses N_i to be scheduled for 30 days $[D_{i1}, \ldots, D_{i30}]$.
- For each nurse N_i, we post regular($[D_{i1}, \ldots, D_{i30}]$, A)

Regular Constraint

- Useful in sequencing and rostering problems.
- Many constraints are instances of regular:
  - among, lex, precedence, stretch, …
- Efficient GAC propagation with a dedicated algorithm and a decomposition into a sequence of ternary constraints.
  - Another example of the power of decompositions!