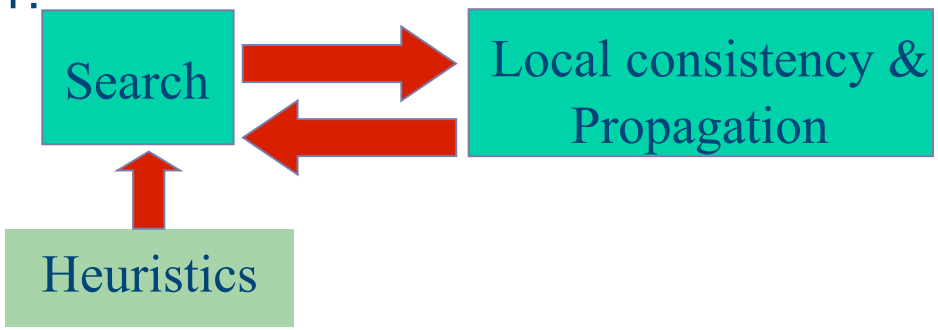
AIMMS

- Modeling language with an interface to CP and MIP solvers (http://www.aimms.com/cp)
- Student license
- GUI support available only in the Windows version
  - Create a virtual machine with Windows OS in your computer
  - Virtual Box (https://www.virtualbox.org/)
- Extensive documentation
  - One-hour tutorial (general introduction)
  - Chapter 21 – “Constraint Programming” of Language Reference

PART II: Local Consistency & Constraint Propagation

Solving CSPs

- Search algorithm.
  - Usually backtracking search performing a depth-first traversal of a search tree.
- Local consistency and constraint propagation.
- Heuristics for branching on the search tree.
  - which variable to branch on?
  - which value to branch on?



Solving CSPs

- Search algorithm.
  - Usually backtracking search performing a depth-first traversal of a search tree.
- Local consistency and constraint propagation.
- Heuristics for branching on the search tree.
  - which variable to branch on?
  - which value to branch on?

Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalized Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation Algorithms
  - Global Constraints
- Generalized Propagation Algorithms
  - AC algorithms

Local Consistency

- Backtrack tree search aims to extend a partial assignment of variables to a complete and consistent one.
- Some inconsistent partial assignments obviously cannot be completed.
- Local consistency is a form of inference which detects inconsistent partial assignments.
  - Consequently, the backtrack search commits into less inconsistent assignments.
- Local, because we examine individual constraints.
  - Global consistency is NP-hard!

Local Consistency: An example

- $D(X_1) = \{1,2\}$, $D(X_2) = \{3,4\}$, C_1: $X_1 = X_2$, C_2: $X_1 + X_2 \geq 1$
- $X_1 = 1$
- $X_1 = 2$
- $X_2 = 3$
- $X_2 = 4$

all inconsistent partial assignments wrt the constraint C_1

  - No need to check the other constraint.
  - Unsatisfiability of the CSP can be inferred without having to search!

Several Local Consistencies

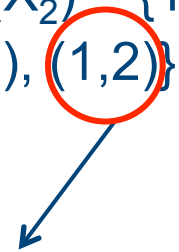
- Most popular local consistencies are domain based:
  - Arc Consistency (AC);
  - Generalised Arc Consistency (GAC);
  - Bounds Consistency (BC).
- They detect inconsistent partial assignments of the form $X_i = j$, hence:
  - j can be removed from $D(X_i)$ via propagation;
  - propagation can be implemented easily.

Arc Consistency (AC)

- Defined for binary constraints.
- A binary constraint **C** is a relation on two variables $\mathbf{X_i}$ and $\mathbf{X_j}$, giving the set of allowed combinations of values (i.e. tuples):
  - $\mathbf{C \subseteq D(X_i) \times D(X_j)}$
  - E.g., $D(X_1) = \{0,1\}$, $D(X_2) = \{1,2\}$, C: $X_1 < X_2$
    $C(X_1,X_2) = \{(0,1), (0,2), (1,2)\}$

Arc Consistency (AC)

- Defined for binary constraints.
- A binary constraint **C** is a relation on two variables $\mathbf{X_i}$ and $\mathbf{X_j}$, giving the set of allowed combinations of values (i.e. tuples):
  - $\mathbf{C \subseteq D(X_i) \times D(X_j)}$
  - E.g., $D(X_1) = \{0,1\}$, $D(X_2) = \{1,2\}$, C: $X_1 < X_2$
    $C(X_1,X_2) = \{(0,1), (0,2), (1,2)\}$



Each tuple $\mathbf{(d_1,d_2)}$ is called a support for C

Arc Consistency (AC)

- C is AC iff:
  - forall $v \in D(X_i)$, exists $w \in D(X_j)$ s.t. $(v,w) \in C$.
    - $v \in D(X_i)$ is said to have/belong to a support wrt the constraint C.
  - forall $w \in D(X_j)$, exists $v \in D(X_i)$ s.t. $(v,w) \in C$.
    - $w \in D(X_j)$ is said to have/belong to a support wrt the constraint C.
- A CSP is AC iff all its binary constraints are AC.

AC: An example

- $D(X_1) = \{1,2,3\}$, $D(X_2) = \{2,3,4\}$, C: $X_1 = X_2$
- AC(C)?
  - $1 \in D(X_1)$ does not have a support.
  - $2 \in D(X_1)$ has $2 \in D(X_2)$ as support.
  - $3 \in D(X_1)$ has $3 \in D(X_2)$ as support.
  - $2 \in D(X_2)$ has $2 \in D(X_1)$ as support.
  - $3 \in D(X_2)$ has $3 \in D(X_1)$ as support.
  - $4 \in D(X_2)$ does not have a support.
- $X_1 = 1$ and $X_2 = 4$ are inconsistent partial assignments.
- $1 \in D(X_1)$ and $4 \in D(X_2)$ must be removed to achieve AC.
- $D(X_1) = \{2,3\}$, $D(X_2) = \{2,3\}$, C: $X_1 = X_2$
  - AC(C)

Generalised Arc Consistency

- Generalisation of AC to k-ary constraints.
- A constraint **C** is a relation on k variables **X_1,…, X_k:**
  - **$C \subseteq D(X_1) \times \ldots \times D(X_k)$**
  - E.g., $D(X_1) = \{0,1\}$, $D(X_2) = \{1,2\}$, $D(X_2) = \{2,3\}$ C: $X_1 + X_2 = X_3$
    $C(X_1,X_2,X_3) = \{(0,2,2), (1,1,2), (1,2,3)\}$
- A support is a tuple $(d_1,\ldots,d_k) \in C$ where $d_i \in D(X_i)$.

Generalised Arc Consistency

- C is GAC iff:
  - forall X_i in $\{\mathbf{X_1},\ldots,\mathbf{X_k}\}$, forall $v \in D(X_i)$, v belongs to a support.
- AC is a special case of GAC.
- A CSP is GAC iff all its constraints are GAC.

GAC: An example

- $D(X_1) = \{1,2,3\}$, $D(X_2) = \{1,2\}$, $D(X_3) = \{1,2\}$
  C: alldifferent($[X_1, X_2, X_3]$)

- GAC(C)?
  - $X_1 = 1$ and $X_1 = 2$ are not supported!
- $D(X_1) = \{3\}$, $D(X_2) = \{1,2\}$, $D(X_3) = \{1,2\}$
  C: alldifferent($[X_1, X_2, X_3]$)
  - GAC(C)

Bounds Consistency (BC)

- Defined for totally ordered (e.g. integer) domains.
- Relaxes the domain of X_i from $D(X_i)$ to $[\min(X_i)..\max(X_i)]$.
  - E.g., $D(X_i) = \{1,3,5\} \rightarrow [1..5]$

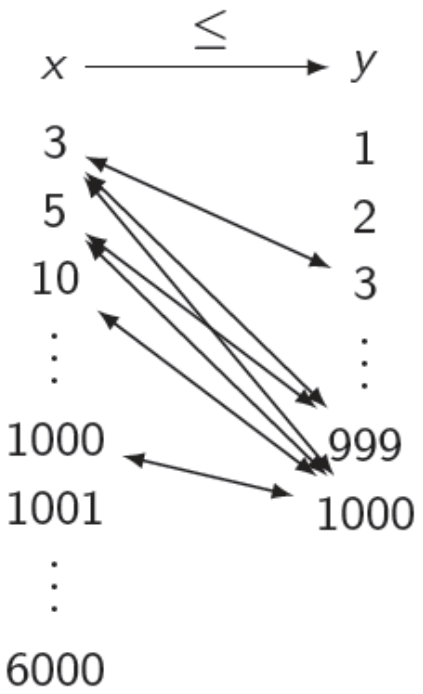
Bounds Consistency (BC)

- A constraint **C** is a relation on k variables $\mathbf{X_1,\ldots, X_k}$:
  - $\mathbf{C \subseteq D(X_1) \times \ldots \times D(X_k)}$
- A bound support is a tuple $(d_1,\ldots,d_k) \in C$ where $d_i \in [\min(X_i)..\max(Xi)]$.
- C is BC iff:
  - forall X_i in $\{\mathbf{X_1,\ldots, X_k}\}$, $\min(X_i)$ and $\max(X_i)$ belong to a bound support.

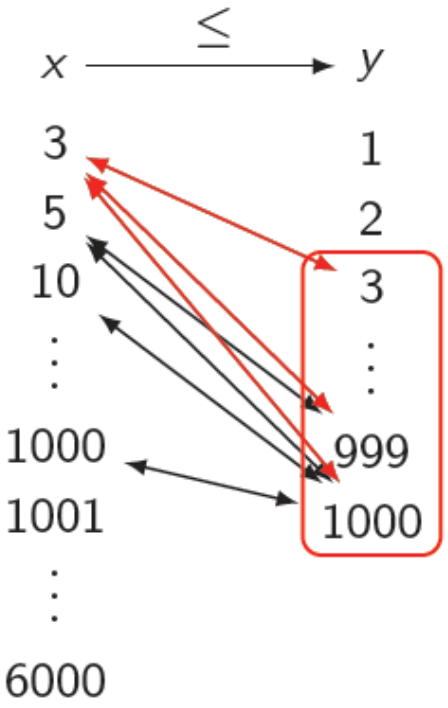
Bounds Consistency (BC)

- Disadvantage
  - BC might not detect all GAC inconsistencies in general.
- Advantages
  - Might be easier to look for a support in a range than in a domain.
  - Achieving BC is often cheaper than achieving GAC.
  - Achieving BC is enough to achieve GAC for monotonic constraints.

GAC = BC: An example



GAC = BC: An example



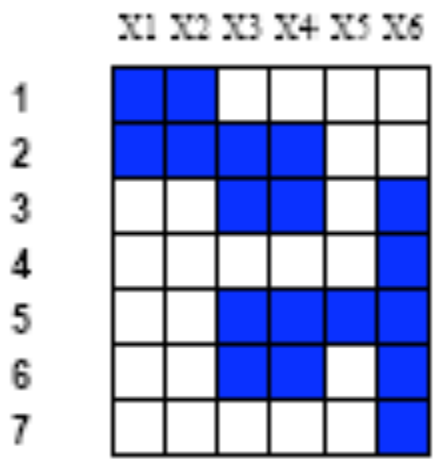
- If (v,w) is a support of X ≤ Y, then:
  - for all u ∈ D(Y) s.t. u ≥ w:
    - (v,u) is also a support;
  - for all u ∈ D(X) s.t. u ≤ v:
    - (u,w) is also a support.
- All values of D(X) smaller than or equal to max(Y) are GAC.
- All values of D(Y) greater than or equal to min(X) are GAC.
- Enough to adjust max(X) and min(Y).

GAC > BC: An example

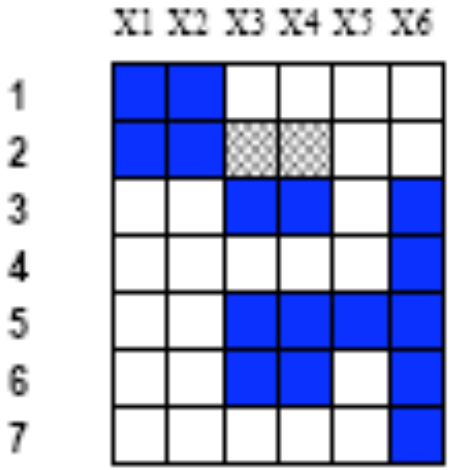
- $D(X_1) = D(X_2) = \{1,2\}$, $D(X_3) = D(X_4) = \{2,3,5,6\}$, $D(X_5) = \{5\}$, $D(X_6) = \{3,4,5,6,7\}$

  C: alldifferent($[X_1, X_2, X_3, X_4, X_5, X_6]$)

- BC(C): $2 \in D(X_3)$ and $2 \in D(X_4)$ have no support.



Original



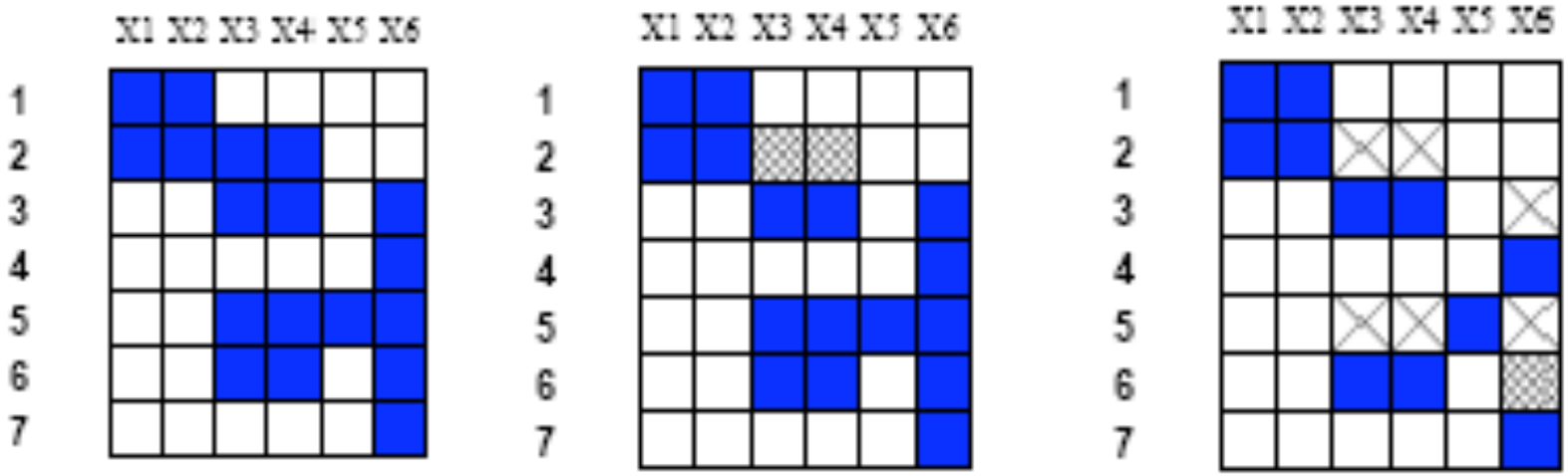
BC

GAC > BC: An example

- $D(X_1) = D(X_2) = \{1,2\}$, $D(X_3) = D(X_4) = \{2,3,5,6\}$, $D(X_5) = \{5\}$, $D(X_6) = \{3,4,5,6,7\}$

  C: alldifferent($[X_1, X_2, X_3, X_4, X_5, X_6]$)

- GAC(C): $\{2,5\} \in D(X_3)$, $\{2,5\} \in D(X_4)$, $\{3,5,6\} \in D(X_6)$ have no support.



Sudoku

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

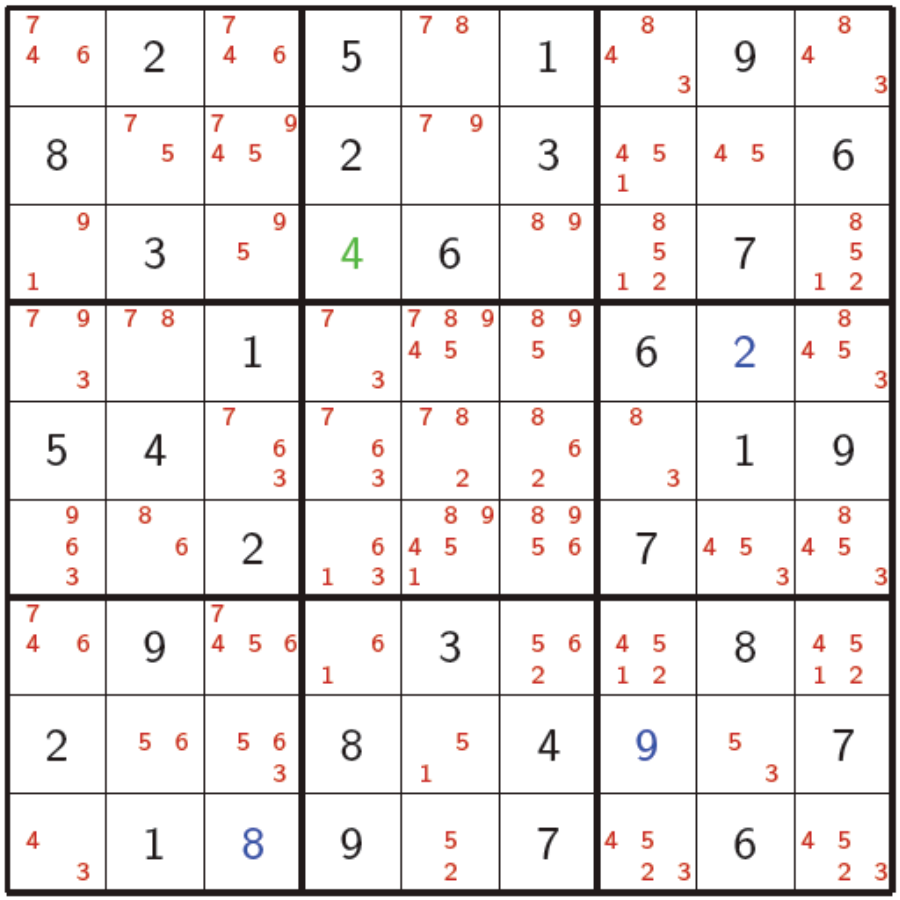
Sudoku

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

Sudoku with AC(≠)

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

Sudoku with BC(alldifferent)

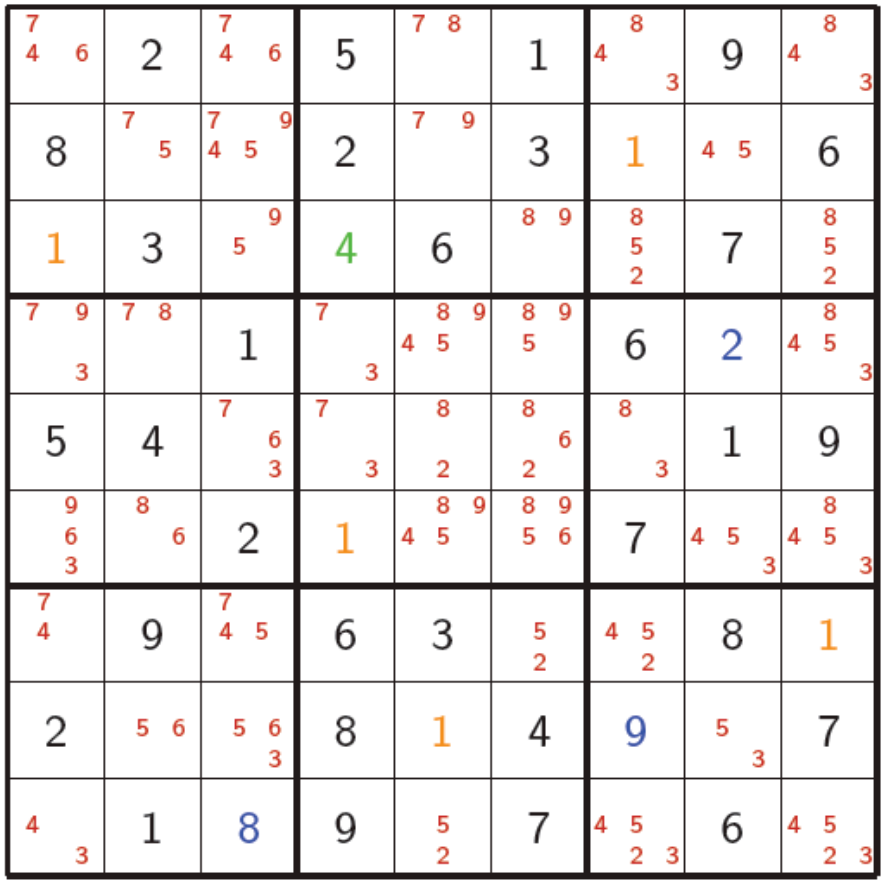


Check the column

Check the 3x3 box

Check the 3x3 box

Sudoku with GAC(alldifferent)



Higher Levels of Consistencies

- Path consistency, k-consistencies, (i,j) consistencies, …
- Not much used in practice:
  - detect inconsistent partial assignments with more than one <variable,value> pair.
  - cannot be enforced by removing single values from domains.
- Domain based consistencies stronger than (G)AC.
  - Singleton consistencies, triangle-based consistencies, …
  - Becoming popular.
    - Shaving in scheduling.

Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency
- Constraint Propagation
  - Constraint Propagation Algorithms
- Specialised Propagation Algorithms
  - Global Constraints
- Generalized Propagation Algorithms
  - AC Algorithms

Constraint Propagation

- Can appear under different names:
  - constraint relaxation;
  - filtering algorithm;
  - local consistency enforcing, …
- Similar concepts in other fields.
  - E.g., unit propagation in SAT.
- Local consistencies define properties that a CSP must satisfy after constraint propagation:
  - the operational behavior is completely left open;
  - the only requirement is to achieve the required property on the CSP.

Constraint Propagation: A simple example

Input CSP:$D(X_1) = \{1,2\}$, $D(X_2) = \{1,2\}$, C: $X_1 < X_2$

A constraint propagation algorithm for enforcing AC

Output CSP:$D(X_1) = \{1\}$, $D(X_2) = \{2\}$, C: $X_1 < X_2$

We can write different algorithms with different complexities to achieve the same effect.

Constraint Propagation Algorithms

- A constraint propagation algorithm propagates a constraint C.
  - It removes the inconsistent values from the domains of the variables of C.
  - It makes C locally consistent.
  - The level of consistency depends on C:
    - GAC might be NP-complete, BC might not be possible, …

Constraint 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.

Constraint Propagation: An 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;
  - each algorithm maintains (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 Constraint Propagation Algorithms

- It is not enough to remove inconsistent values from domains once.
- A constraint propagation algorithm must wake up again when necessary, otherwise may not achieve the desired local consistency property.
- Events that trigger a constraint propagation:
  - when the domain of a variable changes;
  - when a variable is assigned a value;
  - when the minimum or the maximum values of a domain changes.

Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation Algorithms
  - Global Constraints
    - Decompositions
    - Ad-hoc algorithms
- Generalized Propagation Algorithms
  - AC Algorithms

Specialized Propagation Algorithms

- A constraint propagation algorithm can be general or specialized:
  - general, if it is applicable to any constraint;
  - specialized, if it is specific to a constraint.
- Specialized algorithms
  - Disadvantages
    - Limited use.
    - Not always easy to develop one.
  - Advantages
    - Exploits the constraint semantics.
    - Potentially much more efficient than a general algorithm.
- Worth developing for recurring constraints.

Specialized Propagation Algorithms

- C: $X_1 \leq X_2$
- Observation
  - A support of $\min(X_2)$ supports all the values in $D(X_2)$.
  - A support of $\max(X_1)$ supports all the values in $D(X_1)$.
- Propagation algorithm
  - Filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$.
  - Filter $D(X_2)$ s.t. $\min(X_1) \leq \min(X_2)$.
- The result is GAC (and thus BC).

Example

- $D(X_1) = \{3, 4, 7, 8\}$, $D(X_2) = \{1, 2, 3, 5\}$, C: $X_1 \leq X_2$

Example

- $D(X_1) = \{3, 4, 7, 8\}$, $D(X_2) = \{1, 2, 3, 5\}$, C: $X_1 \leq X_2$
- Propagation
  - Filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$.

Example

- $D(X_1) = \{3, 4, \cancel{7}, \cancel{8}\}$, $D(X_2) = \{1, 2, 3, 5\}$, C: $X_1 \leq X_2$
- Propagation
  - Filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$.

Example

- $D(X_1) = \{3, 4, \cancel{7}, \cancel{8}\}$, $D(X_2) = \{1, 2, 3, 5\}$, C: $X_1 \leq X_2$
- Propagation
  - Filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$.
  - Filter $D(X_2)$ s.t. $\min(X_1) \leq \min(X_2)$.

Example

- $D(X_1) = \{3, 4, \cancel{7}, \cancel{8}\}$, $D(X_2) = \{\cancel{1}, \cancel{2}, 3, 5\}$, C: $X_1 \leq X_2$
- Propagation
  - Filter $D(X_1)$ s.t. $\max(X_1) \leq \max(X_2)$.
  - Filter $D(X_2)$ s.t. $\min(X_1) \leq \min(X_2)$.

Global Constraints

- Many real-life constraints are complex and not binary.
  - Specialised algorithms are often developed for such constraints!
- A complex and n-ary constraint which encapsulates a specialised propagation algorithm is called a global constraint.

Examples of Global Constraints

- Alldifferent constraint
  - alldifferent($[X_1, X_2, \ldots, X_n]$) holds iff
    $X_i \neq X_j$ for $i < j \in \{1,\ldots,n\}$
  - Useful in a variety of context:
    - timetabling (e.g. exams with common students must occur at different times)
    - tournament scheduling (e.g. a team can play at most once in a week)
    - configuration (e.g. a particular product cannot have repeating components)
    - …

Beyond Alldifferent

- NValue constraint
  - One generalization of alldifferent.
  - nvalue($[X_1, X_2, \ldots, X_n]$, N) holds iff
    $$N = |\{X_i \mid 1 \leq i \leq n\}|$$
  - nvalue([1, 2, 2, 1, 3], 3).
  - alldifferent when N = n.
  - Useful when values represent resources and we want to limit the usage of resources. E.g.,
    - minimize the total number of resources used;
    - the total number of resources used must be between a specific interval;
    - …

Beyond Alldifferent

- Global cardinality constraint
  - Another generalisation of alldifferent.
  - gcc($[X_1, X_2, \ldots, X_n]$, $[v_1, \ldots, v_m]$, $[O_1, \ldots, O_m]$) iff
    forall $j \in \{1,\ldots, m\}$ $O_j = |\{X_i \mid X_i = v_j, 1 \leq i \leq n\}|$
  - gcc([1, 1, 3, 2, 3], [1, 2, 3, 4], [2, 1, 2, 0])
  - Useful again when values represent resources.
  - We can now limit the usage of each resource individually. E.g.,
    - resource 1 can be used at most three times;
    - resource 2 can be used min 2 max 5 times;
    - …

Symmetry Breaking Constraints

- Consider the following scenario.
  - $[X_1, X_2, \ldots, X_n]$ and $[Y_1, Y_2, \ldots, Y_n]$ represent the 2 day event assignments of a conference.
  - Each day has n slots and the days are indistinguishable.
  - Need to avoid symmetric assignments.
- Global constraints developed for this purpose are called symmetry breaking constraints.
- Lexicographic ordering constraint
  - lex($[X_1, X_2, \ldots, X_n]$, $[Y_1, Y_2, \ldots, Y_n]$) holds iff:
    $X_1 < Y_1$ OR $(X_1 = Y_1$ AND $X_2 < Y_2)$ OR …
    $(X_1 = Y_1$ AND $X_2 = Y_2$ AND …. AND $X_n \leq Y_n)$
  - lex ([1, 2, 4],[1, 3, 3])

Sequencing Constraints

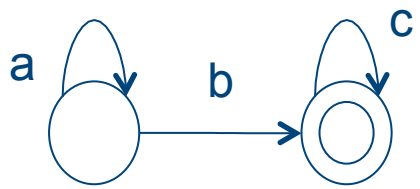
- We might want a sequence of variables obey certain patterns.
- Global constraints developed for this purpose are called sequencing constraints.

Sequencing Constraints

- Sequence constraint
  - Constrains the number of values taken from a given set in any sequence of k variables.
  - sequence(l, u, k, $[X_1, X_2, \ldots, X_n]$, v) iff
    among($[X_i, X_{i+1}, \ldots, X_{i+k-1}]$, v, l, u) for $1 \leq i \leq n-k+1$
  - among($[X_1, X_2, \ldots, X_n]$, v, l, u) iff
    $l \leq |\{i \mid X_i \in v, 1 \leq i \leq n\}| \leq u$
  - E.g.,
    - every employee has 2 days off in any 7 day of period;
    - at most 1 in 3 cars along the production line can have a sun-roof fitted.

Sequencing Constraints

- Sometimes patterns can be best described by grammars or automata that accept some language.
- Regular constraint
  - regular([X_1, X_2, …, X_n], A) holds iff <X_1, X_2, …, X_n> forms a string accepted by the DFA A (which accepts a regular language).
  - regular([a, a, b], A), regular([b], A), regular([b, c, c, c, c, c], A) with A:



- Many global constraints are instances of regular. E.g., among, lex, strech, …

Scheduling Constraints

- We might want to schedule tasks with respective release times, duration, and deadlines, using limited resources in a time D.
- Global constraints developed for this purpose are called scheduling constraints.

Scheduling Constraints

- Cumulative constraint
  - Useful for scheduling non-preemptive tasks who share a single resource with limited capacity.
  - Given tasks t_1,.., t_n, with each t_i associated with 3 variables S_i, D_i, and C_i, and a resource of capacity C:

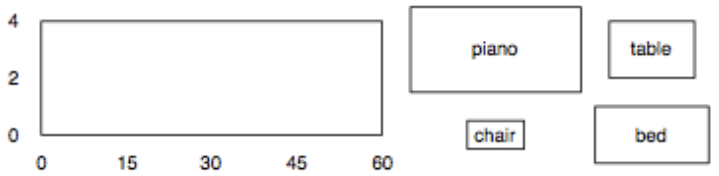
    cumulative([S_1, S_2, …, S_n], [D_1, D_2, …, D_n], [C_1, C_2, …, C_n], C)

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

Scheduling Example

- Tom is moving house. He has 4 people to do the move and must move in 1 hour. 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

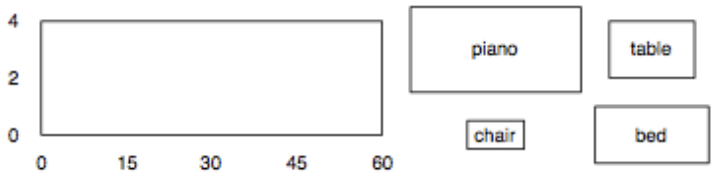


How can we model this?

Scheduling Example

- Tom is moving house. He has 4 people to do the move and must move in 1 hour. Piano must be moved before bed.

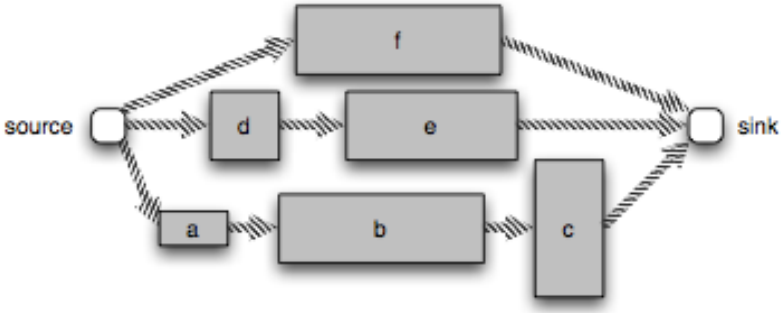
Item	Time	No. of people
piano	30 min	3
chair	10 min	1
bed	20 min	2
table	15 min	2

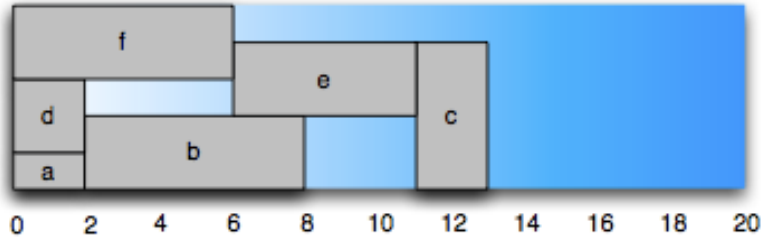


How can we model this?

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

Scheduling Example





- 6 tasks [a,b,c,d,e,f,g] to place in an 5x20 box with the durations [2,6,2,2,5,6] requiring [1,2,4,2,2,2] units of resource with precedences as in the figure.
- Model
  - D(Sa)=D(Sb)=D(Sc)=D(Sd)=D(Se), D(Sf) = [0..20]
  - Sa + 2 ≤ Sb, Sb + 6 ≤ Sc, Sd + 2 ≤ Se
  - Sa + 2 ≤ 20, Sb + 6 ≤ 20, Sc + 2 ≤ 20, Sd + 2 ≤ 20, Se + 5 ≤ 20, Sf + 6 ≤ 20
  - cumulative([Sa,Sb,Sc,Sd, Se, Sf], [2,6,2,2,5,6],[1,2,4,2,2,2],5)

Specialised Algorithms for Global Constraints

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

Constraint Decomposition

- A global constraint is decomposed into smaller and simpler constraints each 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.

Decomposition of Among

- Among constraint
  - among($[X_1, X_2, \ldots, X_n]$, $[d_1, d_2, \ldots, d_m]$, N) iff
    $N = |\{i \mid X_i \in \{d_1, d_2, \ldots, d_m\}, 1 \leq i \leq n\}|$

Decomposition of Among

- Among constraint
  - among($[X_1, X_2, \ldots, X_n]$, $[d_1, d_2, \ldots, d_m]$, N) iff
    $N = |\{i \mid X_i \in \{d_1, d_2, \ldots, d_m\}, 1 \le i \le n\}|$
- Decomposition
  - B_i with $D(B_i) = \{0, 1\}$ for $1 \le i \le n$
  - C_i: $B_i = 1 \leftrightarrow X_i \in \{d_1, d_2, \ldots, d_m\}$ for $1 \le i \le n$
  - $\sum_i B_i = N$
- AC(C_i) for $1 \le i \le n$ and BC($\sum_i B_i = N$) ensures GAC on among.

Decomposition of Lex

- C: lex($[X_1, X_2, \ldots, X_n]$, $[Y_1, Y_2, \ldots, Y_n]$)
- Decomposition
  - B_i with $D(B_i) = \{0, 1\}$ for $1 \leq i \leq n+1$ to indicate the vectors have been ordered by position i-1
  - $B_1 = 0$
  - C_i: $(B_i = B_{i+1} = 0$ AND $X_i = Y_i)$ OR $(B_i = 0$ AND $B_{i+1} = 1$ AND $X_i < Y_i)$ OR $(B_i = B_{i+1} = 1)$ for $1 \leq i \leq n$
- GAC(C_i) ensures GAC on C.

Decomposition of Regular

- C: regular($[X_1, X_2, \ldots, X_n]$, A)
- Decomposition
  - Q_i for $1 \leq i \leq n+1$ to indicate the states of the DFA.
  - Q_1= starting state.
  - Q_{n+1}= accepting state.
  - C_i (X_i, Q_i, Q_{i+1}) for $1 \leq i \leq n$.
  - C_i $(X_i, Q_i\, Q_{i+1})$ iff DFA goes from Q_i to Q_{i+1} on symbol X_i.
    - GAC(C_i) ensures GAC on C.

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.
- E.g., C: alldifferent($[X_1, X_2, \ldots, X_n]$)
- Decomposition
  - C_{ij}: $X_i \neq X_j$ for $i < j \in \{1,\ldots,n\}$
  - AC on the decomposition is weaker than GAC on alldifferent.
  - E.g., $D(X_1) = D(X_2) = D(X_3) = \{1,2\}$, C: alldifferent($[X_1, X_2, X_3]$)
  - C_{12}, C_{13}, C_{23} are all AC, but C is not GAC.

Constraint Decompositions

- C: lex($[X_1, X_2, \ldots, X_n]$, $[Y_1, Y_2, \ldots, Y_n]$)
- OR decomposition
  - $X_1 < Y_1$ OR ($X_1 = Y_1$ AND $X_2 < Y_2$) OR …

    ($X_1 = Y_1$ AND $X_2 = Y_2$ AND …. AND $X_n \leq Y_n$)
  - AC on the decomposition is weaker than GAC on C.
  - E.g., $D(X_1) = \{0,1,2\}$, $D(X_2) = \{0,1\}$, $D(Y_1) = \{0,1\}$, $D(Y_2) = \{0,1\}$

    C: Lex($[X_1, X_2]$, $[Y_1, Y_2]$)
  - C is not GAC but the decomposition does not prune anything.

Constraint Decompositions

- AND decomposition
  - $X_1 \leq Y_1$ AND $(X_1 = Y_1 \rightarrow X_2 \leq Y_2)$ AND …

    $(X_1 = Y_1$ AND $X_2 = Y_2$ AND …. $X_{n-1} = Y_{n-1} \rightarrow X_n \leq Y_n)$
  - AC on the decomposition is weaker than GAC on C.
  - E.g., $D(X_1) = \{0, 1\}$, $D(X_2) = \{1\}$, $D(Y_1) = \{0,1\}$, $D(Y_2) = \{0\}$
    C: Lex($[X_1, X_2]$, $[Y_1, Y_2]$)
  - C is not GAC but the decomposition does not prune anything.

Constraint Decompositions

- Different decompositions of a constraint may be incomparable.
  - Difficult to know which one gives a better propagation for a given instance of a constraint.
- C: Lex($[X_1, X_2]$, $[Y_1, Y_2]$)
  - $D(X_1) = \{0, 1\}$, $D(X_2) = \{1\}$, $D(Y_1) = \{0,1\}$, $D(Y_2) = \{0\}$
    - AND decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC.
  - $D(X_1) = \{0, 1, 2\}$, $D(X_2) = \{0, 1\}$, $D(Y_1) = \{0, 1\}$, $D(Y_2) = \{0, 1\}$
    - OR decomposition is weaker than GAC on lex, whereas OR decomposition maintains GAC.

Constraint Decompositions

- Even if effective, may not always provide an efficient propagation.
- Often GAC on a constraint via a specialised algorithm is maintained faster than (G)AC on the constraints in the decomposition.

Constraint Decompositions

- C: Lex($[X_1, X_2]$, $[Y_1, Y_2]$)
  - $D(X_1) = \{0, 1\}$, $D(X_2) = \{0, 1\}$, $D(Y_1) = \{1\}$, $D(Y_2) = \{0\}$
    - AND decomposition is weaker than GAC on C, whereas OR decomposition maintains GAC on C.
  - $D(X_1) = \{0, 1, 2\}$, $D(X_2) = \{0, 1\}$, $D(Y_1) = \{0, 1\}$, $D(Y_2) = \{0, 1\}$
    - OR decomposition is weaker than GAC on C, whereas OR decomposition maintains GAC on C.
- AND or OR decompositions have complementary strengths.
  - Combining them gives us a decomposition which maintains GAC on C.
- Too many constraints to post and propagate!
- A dedicated algorithm runs amortized in O(1).

Dedicated 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.

Benefits of Global Constraints

- Modeling benefits
  - Reduce the gap between the problem statement and the model.
  - Capture recurring modeling patterns.
  - May allow the expression of constraints that are otherwise not possible to state using primitive constraints (semantic).
- Solving benefits
  - More inference in propagation (operational).
  - More efficient propagation (algorithmic).

Dedicated GAC Algorithm for Sum

- C: $\sum_i X_i = N$ where $D(X_i)= \{0,1\}$ and N is an integer variable.

Dedicated GAC Algorithm for Sum

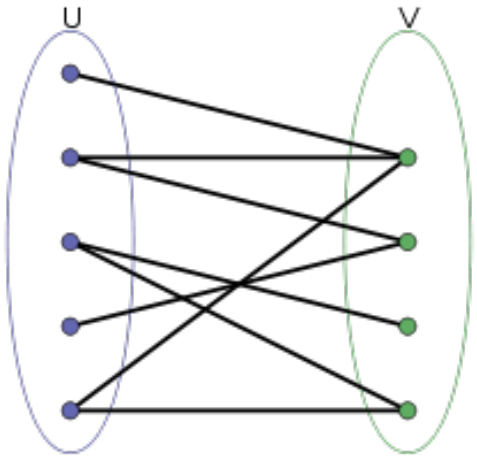
- C: $\sum_i X_i = N$ where D(X_i)= {0,1} and N is an integer variable.
  - min(N) ≥ $\sum_i \min(X_i)$
  - max(N) ≤ $\sum_i \max(X_i)$
  - min(X_i) ≥ min(N) - $\sum_{j \neq i} \max(X_j)$ for 1 ≤ i ≤ n
  - max(X_i) ≤ max(N) - $\sum_{j \neq i} \min(X_j)$ for 1 ≤ i ≤ n

Dedicated GAC Algorithm for Alldifferent

- alldifferent([X_1, X_2, …, X_n])
- Runs in time $O(d^2n^{2.5})$.
- Establishes a relation between the solutions of the constraint and the properties of a graph.
  - Maximal matching in a bipartite graph.

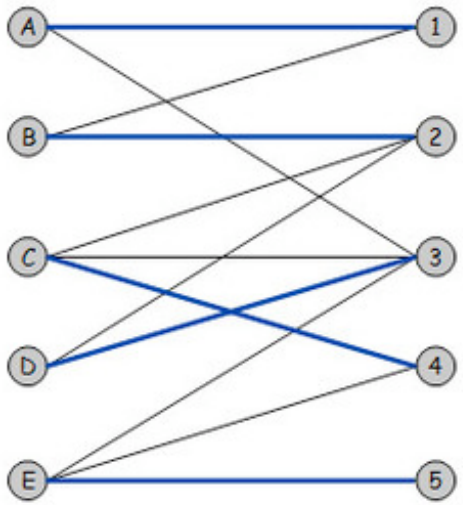
Dedicated GAC Algorithm for Alldifferent

- 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.



Dedicated 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.

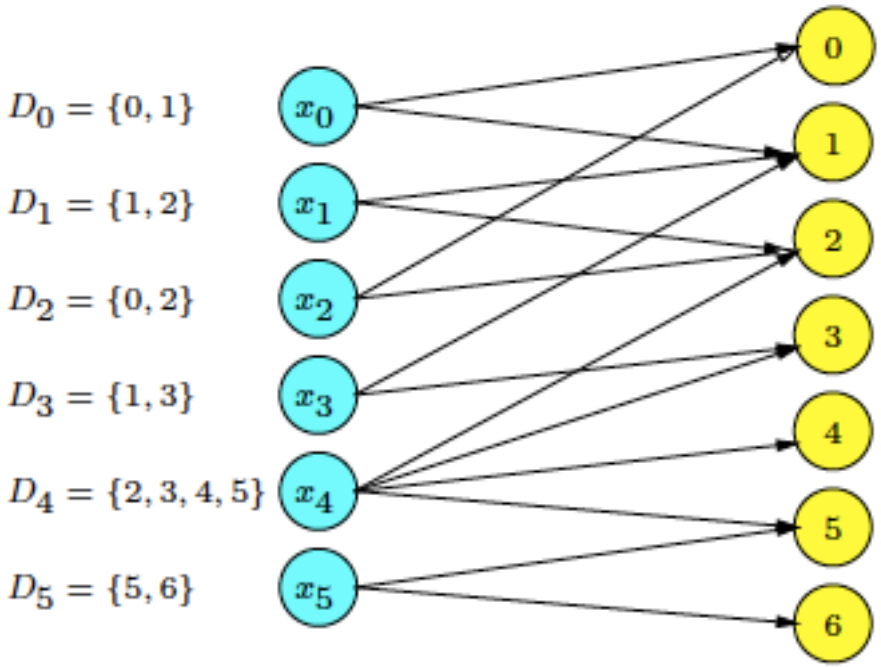


Dedicated GAC Algorithm for Alldifferent

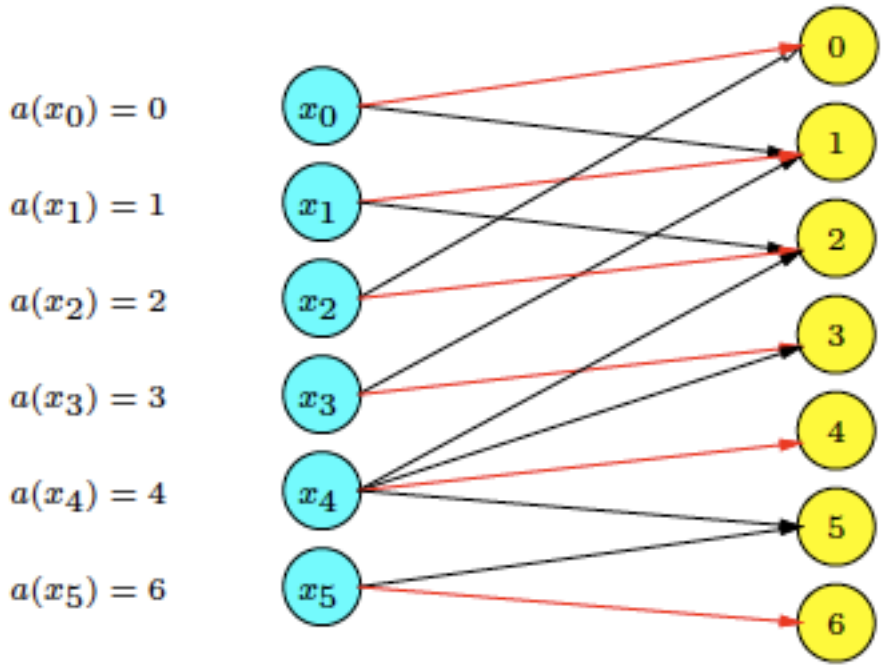
- Observation
  - Construct a bipartite graph G between the variables $[X_1, X_2, \ldots, X_n]$ and their possible values $\{1, \ldots, d\}$ (variable-value graph).
  - An assignment of values to the variables is a solution iff it corresponds to a maximal matching in G.
    - A maximal matching covers all the variables.
  - Use matching theory to compute all maximal matchings efficiently.
    - One maximal matching can describe all maximal matchings!

Example

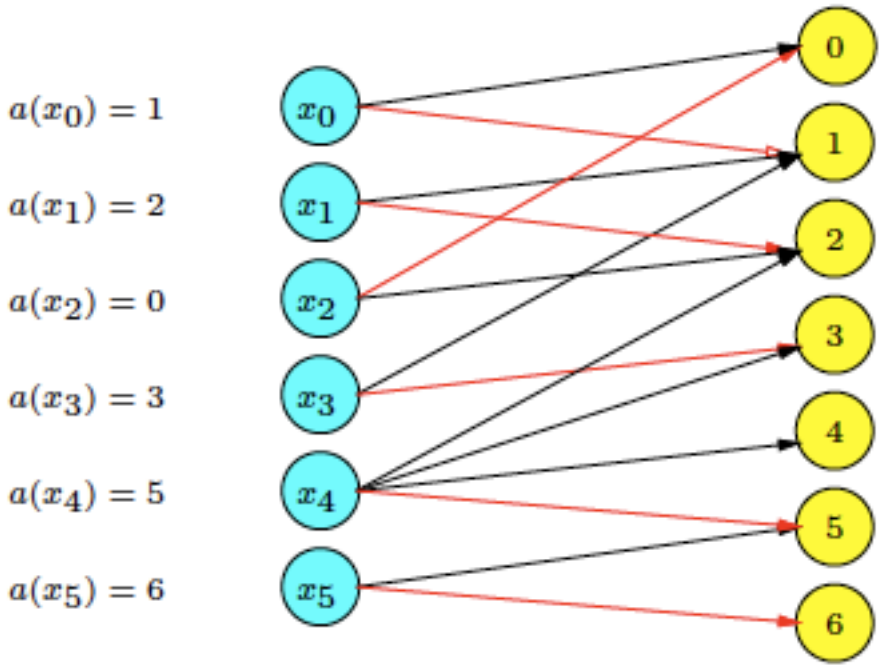
Variable-value graph



A First Maximal Matching



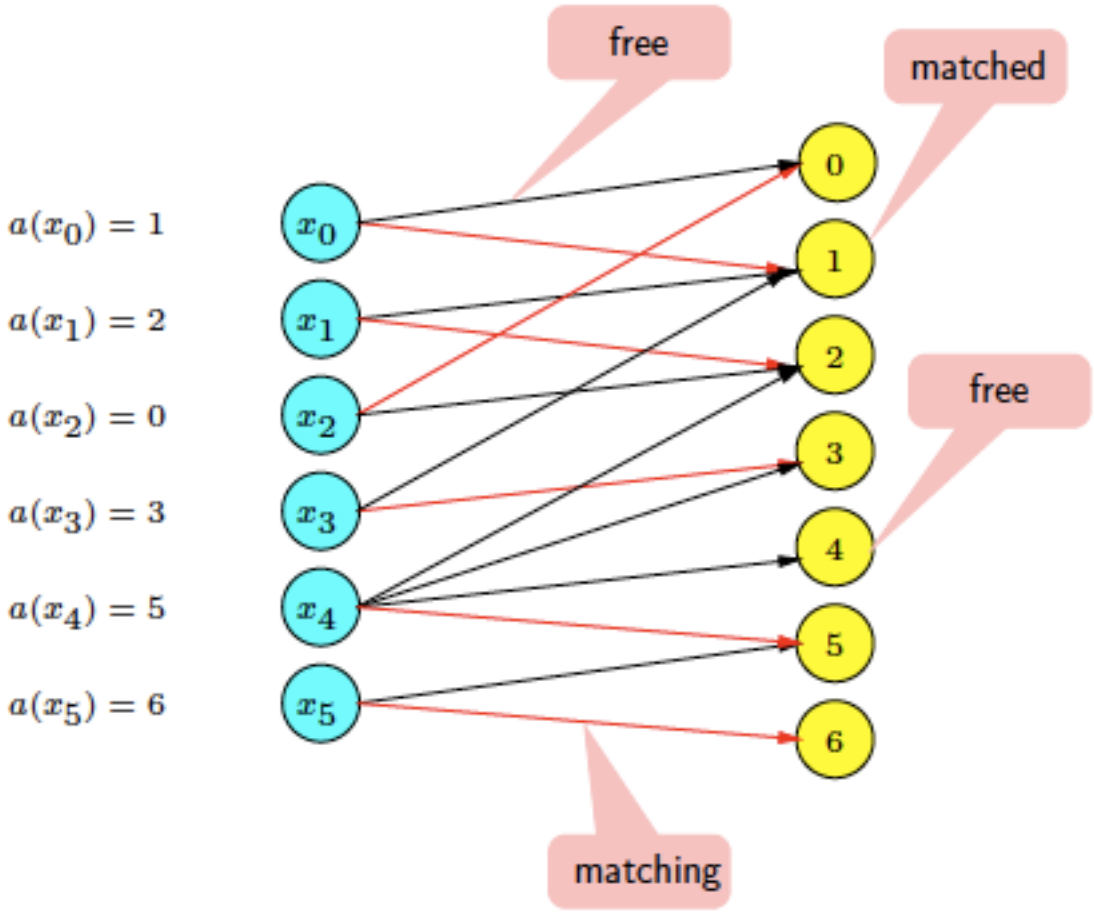
Another Maximal Matching



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



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

- Use matching theory to compute all maximal matchings efficiently.
  - One maximal matching can describe all maximal matchings!
  - Connection between the edges of one matching and edges in the other 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.

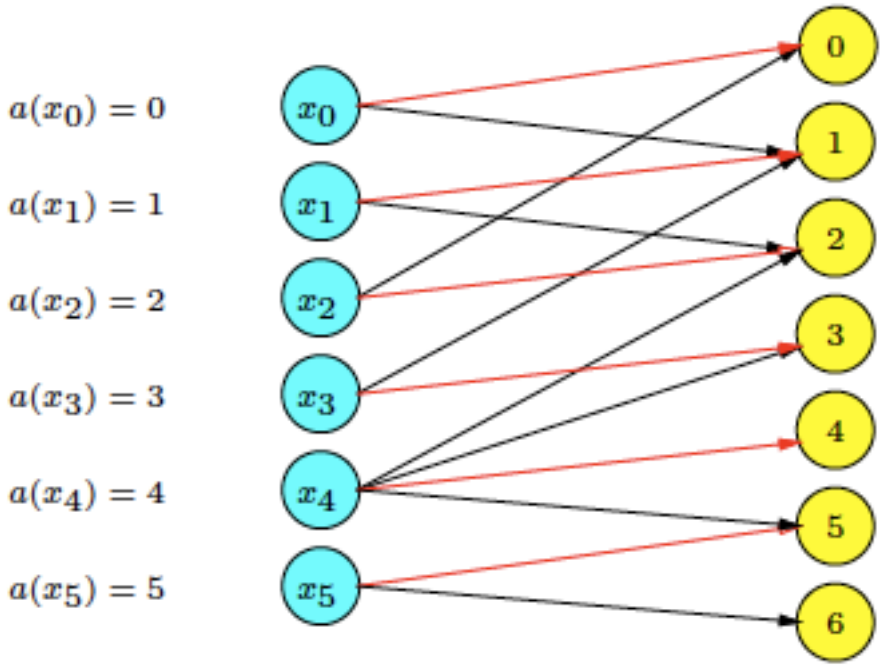
Matching Theory

- An edge e belongs to a maximal matching iff:
  - For some arbitrary maximal matching M:
    - Either e belongs to even alternating path starting at a free node;
    - Or e belongs to an even alternating cycle.
- The result is due to Claude Berge in 1970.

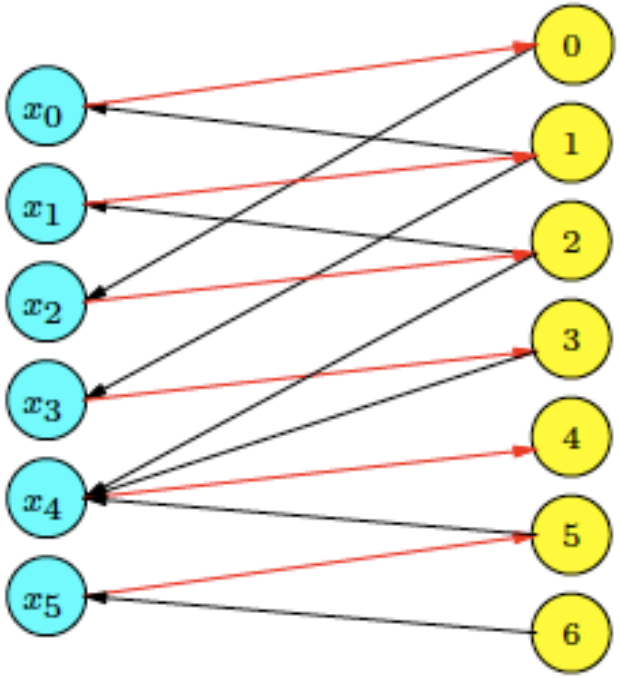
Oriented Graph

- As we are interested in even alternating path/cycle, we orient edges of an arbitrary maximal matching for simplicity:
  - Matching edges → from variable to value;
  - Free edges → from value to variable.

An Arbitrary Maximal Matching



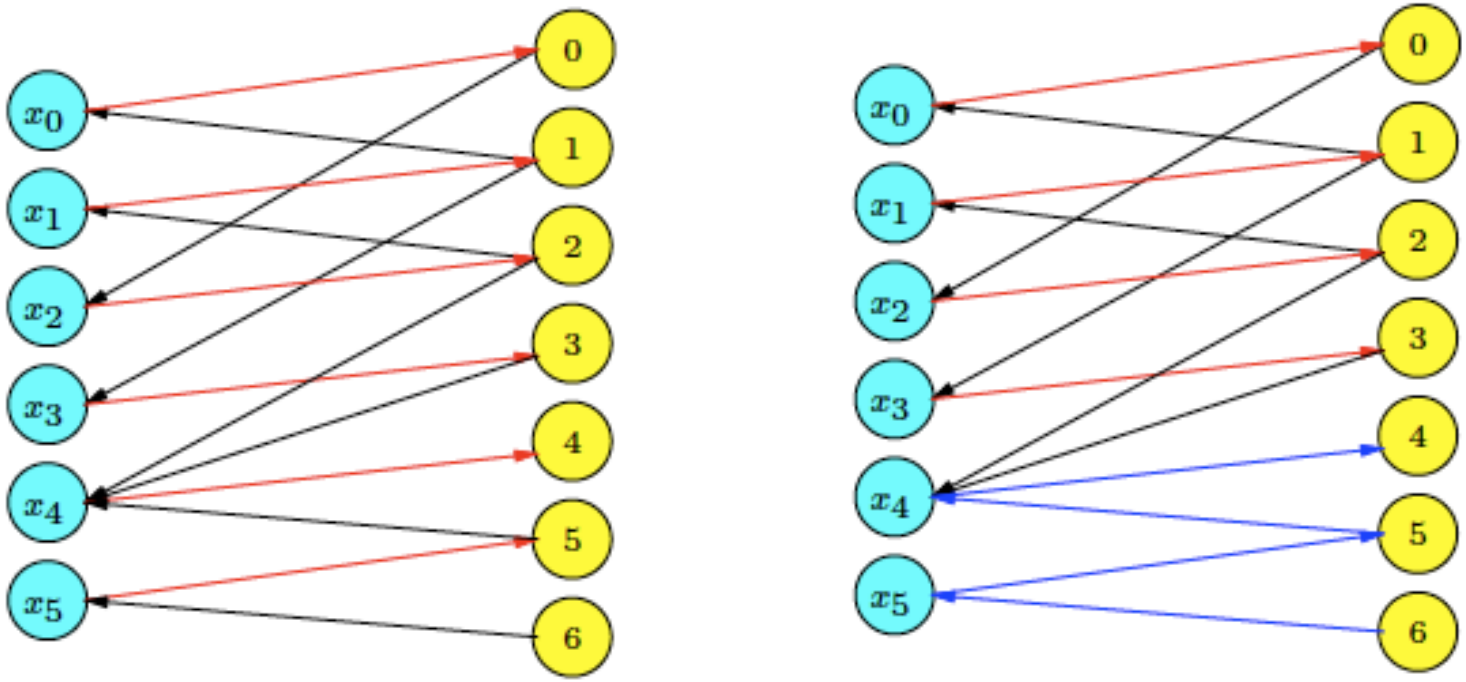
Oriented Graph



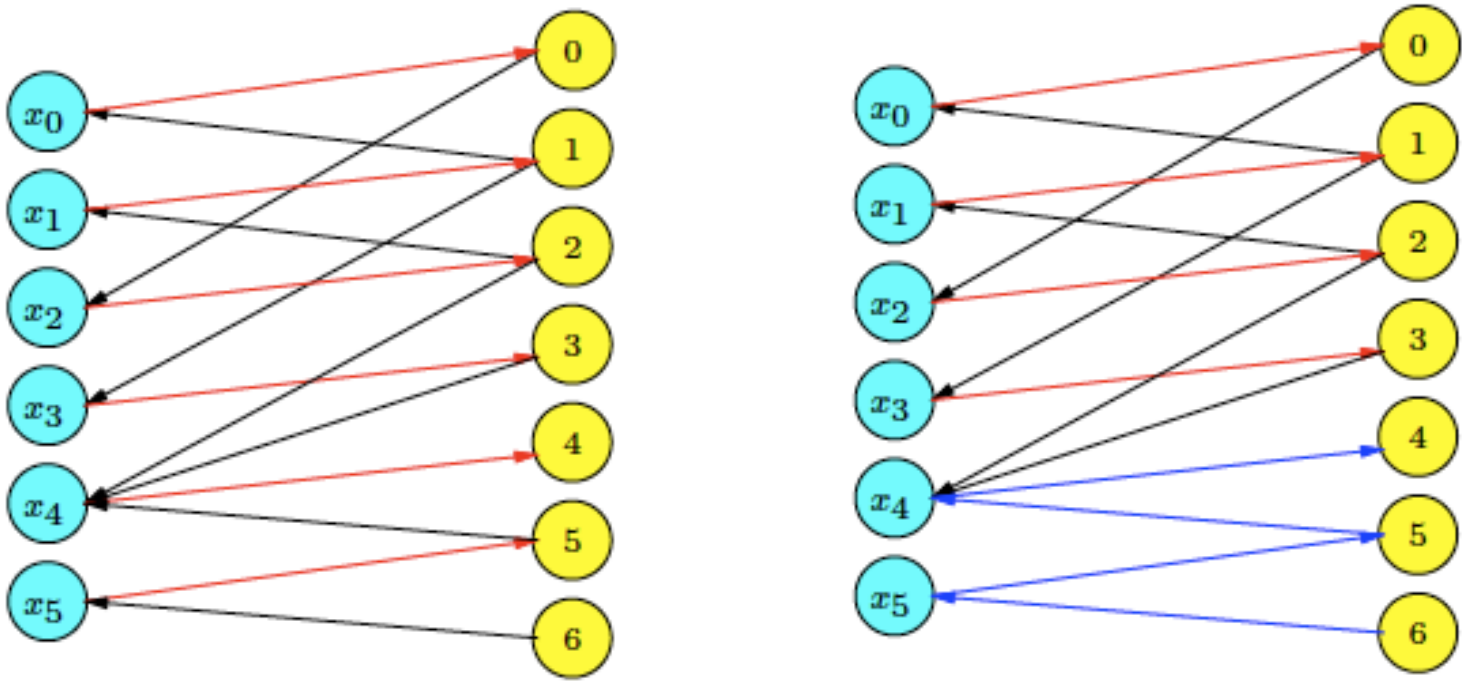
Even Alternating Paths

- 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



Even Alternating Paths

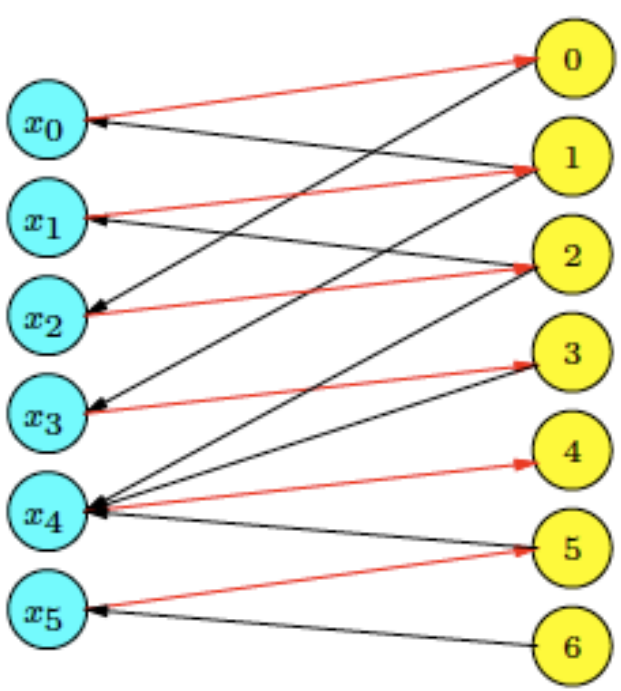


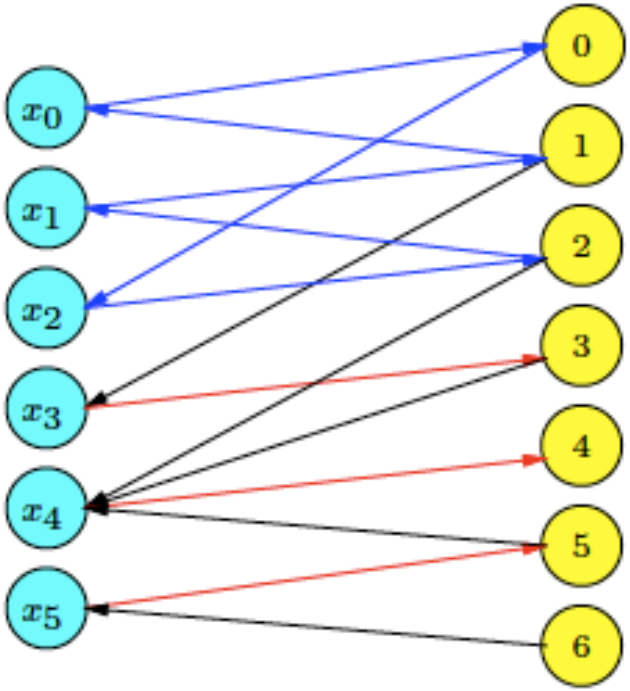
- Intuition: edges can be permuted.

Even Alternating Cycles

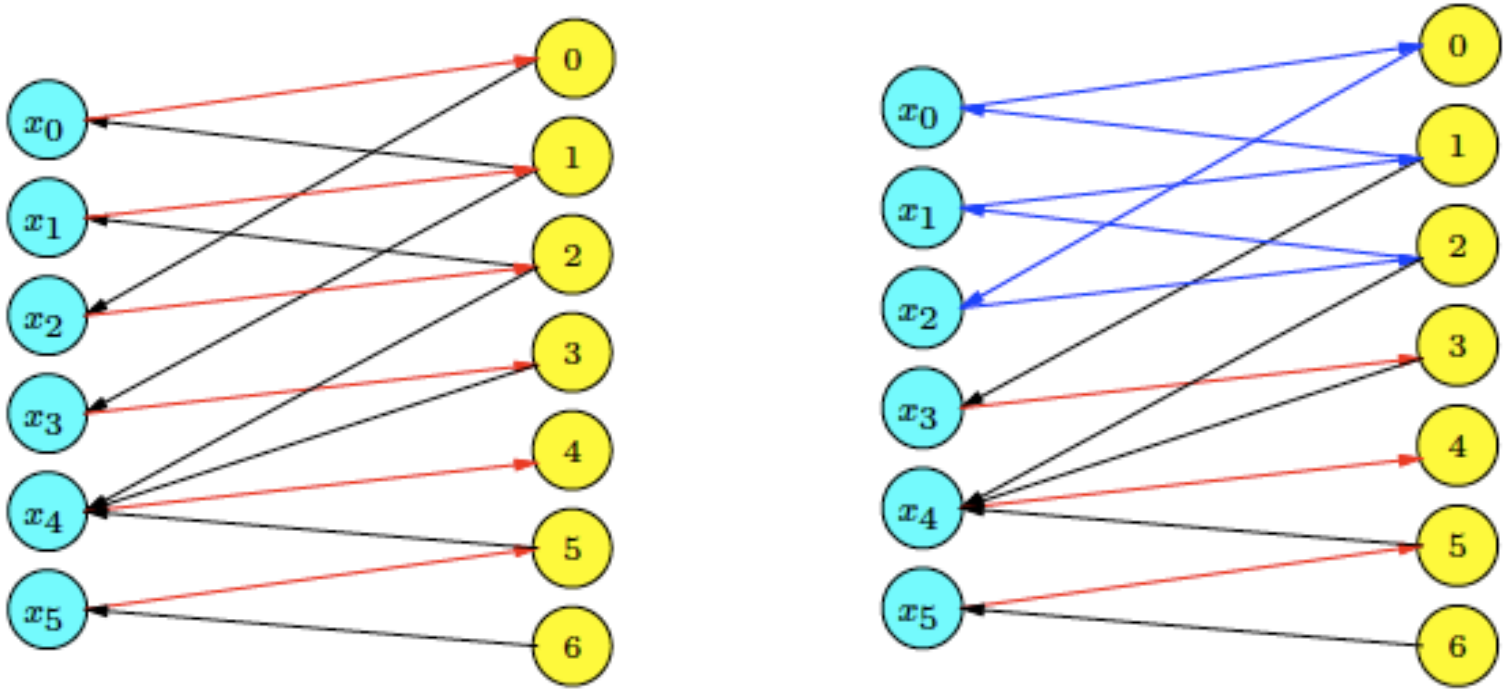
- Compute strongly connected components (SCCs).
  - Two nodes a and b are strongly connected iff there is a path from a to b and a path from b to a.
  - Strongly connected component: any two nodes are strongly connected.
  - Alternation and even length built-in.
- Mark all edges in all strongly connected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

Even Alternating Cycles



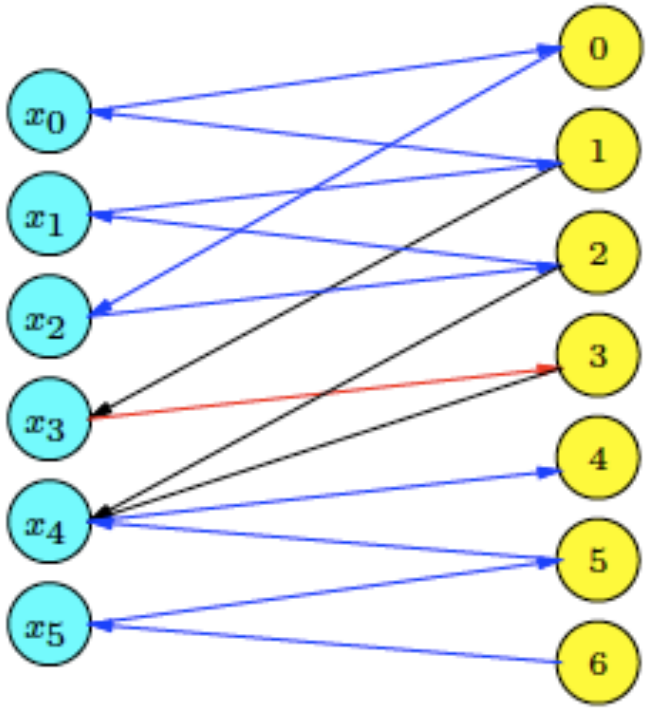


Even Alternating Cycles



- Intuition: variables consume all the values.

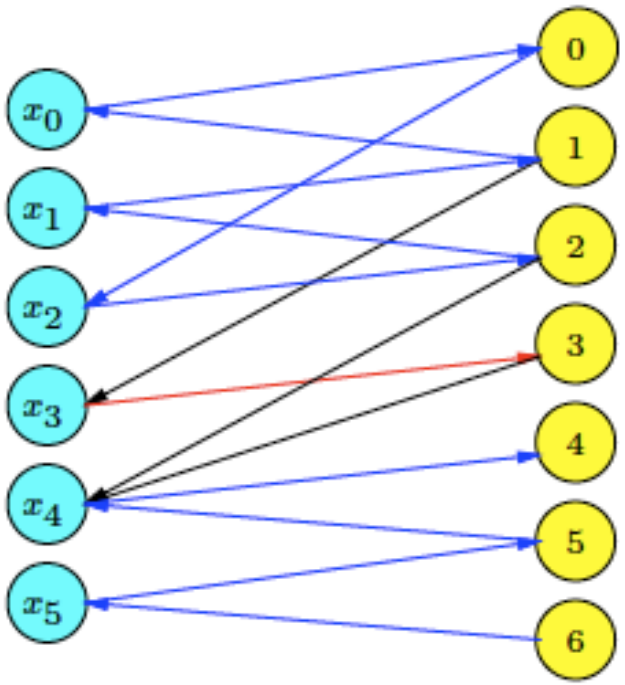
All Marked Edges



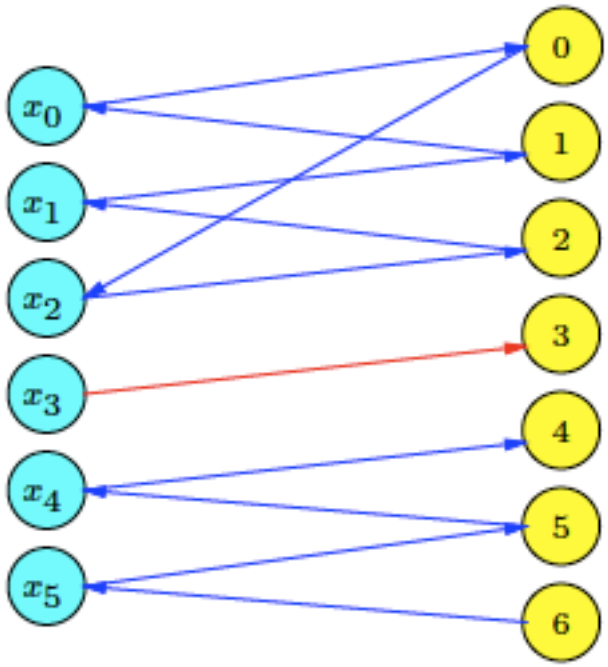
Removing Edges

- 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



Edges Removed



$D(X_1) = \{0,1\}$, $D(X_2) = \{1,2\}$, $D(X_3) = \{0,2\}$, $D(X_4) = \{3\}$, $D(X_5) = \{4,5\}$, $D(X_6) = \{5,6\}$

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.
- Complexity: $O(d^2n^{2.5})$
  - $O(nd)$ for SCCs
  - $O(n^{1.5}d)$ for computing a maximal matching

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.

Dedicated Algorithms

- Is it always easy to develop a dedicated algorithm for a given constraint?
- There's no single recipe!
- A nice semantics often gives us a clue!
  - Graph Theory
  - Flow Theory
  - Combinatorics
  - Complexity Theory, …
- GAC may as well be NP-hard!
  - In that case, algorithms which maintain weaker consistencies (like BC) are of interest.

GAC for Nvalue Constraint

- nvalue($[X_1, X_2, \ldots, X_n]$, N) holds iff $N = |\{X_i \mid 1 \leq i \leq n\}|$
- Reduction from 3 SAT.
  - Given a Boolean fomula in k variables (labelled from 1 to k) and m clauses, we construct an instance of nvalue($[X_1, X_2, \ldots, X_{k+m}]$, N):
    - $D(X_i) = \{i, i'\}$ for $i \in \{1,\ldots, k\}$ where X_i represents the truth assignment of the SAT variables;
    - X_i where $i > k$ represents a SAT clause (disjunction of literals);
    - for a given clause like x V y' V z, $D(X_i) = \{x, y', z\}$.
  - By construction, $[X_1, \ldots, X_k]$ will consume all the k distinct values.
  - When N = k, nvalue has a solution iff the original SAT problem has a satisfying assignment.
    - Otherwise we will have more than k distinct values.
    - Hence, testing a value for support is NP-complete, and enforcing GAC is NP-hard!

GAC for Nvalue Constraint

- E.g., C_1: (a OR b' OR c) AND
  C_2: (a' OR b OR d) AND
  C_3: (b' OR c' OR d)
- The formula has 4 variables (a, b, c, d) and 3 clauses (C_1, C_2, C_3).
- We construct nvalue([X_1, X_2, …, X_7], 4) where:
  - $D(X_1)$ = {a, a'}, $D(X_2)$ = {b, b'}, $D(X_3)$ = {c, c'}, $D(X_4)$ = {d, d'}, $D(X_5)$ = {a, b', c}, $D(X_6)$ = {a', b, d}, $D(X_7)$ = {b', c', d}
- An assignment to X_1, …, X_4 will consume 4 distinct values.
- Not to exceed 4 distinct values, the rest of the variables must have intersecting values with X_1, …, X_4.
- Such assignments will make the SAT formula TRUE.

Outline

- Local Consistency
  - Arc Consistency (AC)
  - Generalised Arc Consistency (GAC)
  - Bounds Consistency (BC)
  - Higher Levels of Consistency
- Constraint Propagation
  - Propagation Algorithms
- Specialized Propagation Algorithms
  - Global Constraints
    - Decompositions
    - Ad-hoc algorithms
- Generalized Propagation Algorithms
  - AC Algorithms

Generalized Propagation Algorithms

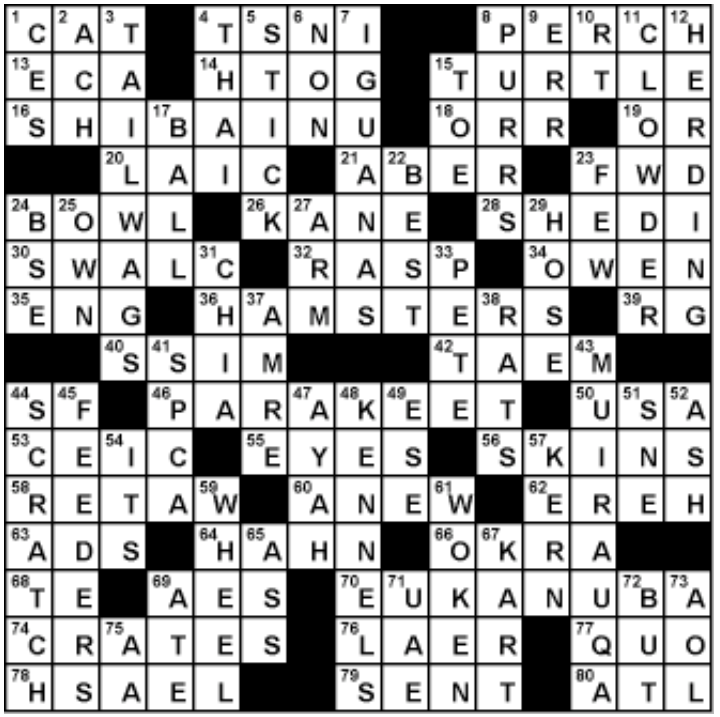
- Not all constraints have nice semantics we can exploit to devise an efficient specialized propagation algorithm.
- Consider a product configuration problem.
  - Compatibility constraints on hardware components:
    - only certain combinations of components work together.
  - Compatibility may not be a simple pairwise relationship:
    - video cards supported function of motherboard, CPU, clock speed, O/S, ...

Production Configuration Problem

- 5-ary constraint:
  - Compatible (motherboard345, intelCPU, 2GHz, 1GBRam, 80GBdrive).
  - Compatible (motherboard346, intelCPU, 3GHz, 2GBRam, 100GBdrive).
  - Compatible (motherboard346, amdCPU, 2GHz, 2GBRam, 100GBdrive).
  - …

Crossword Puzzle

- Constraints with different arity:
  - Word_1 ($[X_1, X_2, X_3]$)
  - Word_2 ($[X_1, X_{13}, X_{16}]$)
  - …
- No simple way to decide acceptable words other than to put them in a table.



GAC Schema

- A generic propagation algorithm.
  - Enforces GAC on an n-ary constraint given by:
    - a set of allowed tuples;
    - a set of disallowed tuples;
    - a predicate answering if a constraint is satisfied or not.
  - Sometimes called the “table” constraint:
    - user supplies table of acceptable values.
- Complexity: $O(ed^n)$ time
- Hence, n cannot be too large!
  - Many solvers limits it to 3 or so.

Arc Consistency Algorithms

- Generic AC algorithms with different complexities and advantages:
  - AC3
  - AC4
  - AC6
  - AC2001
  - …

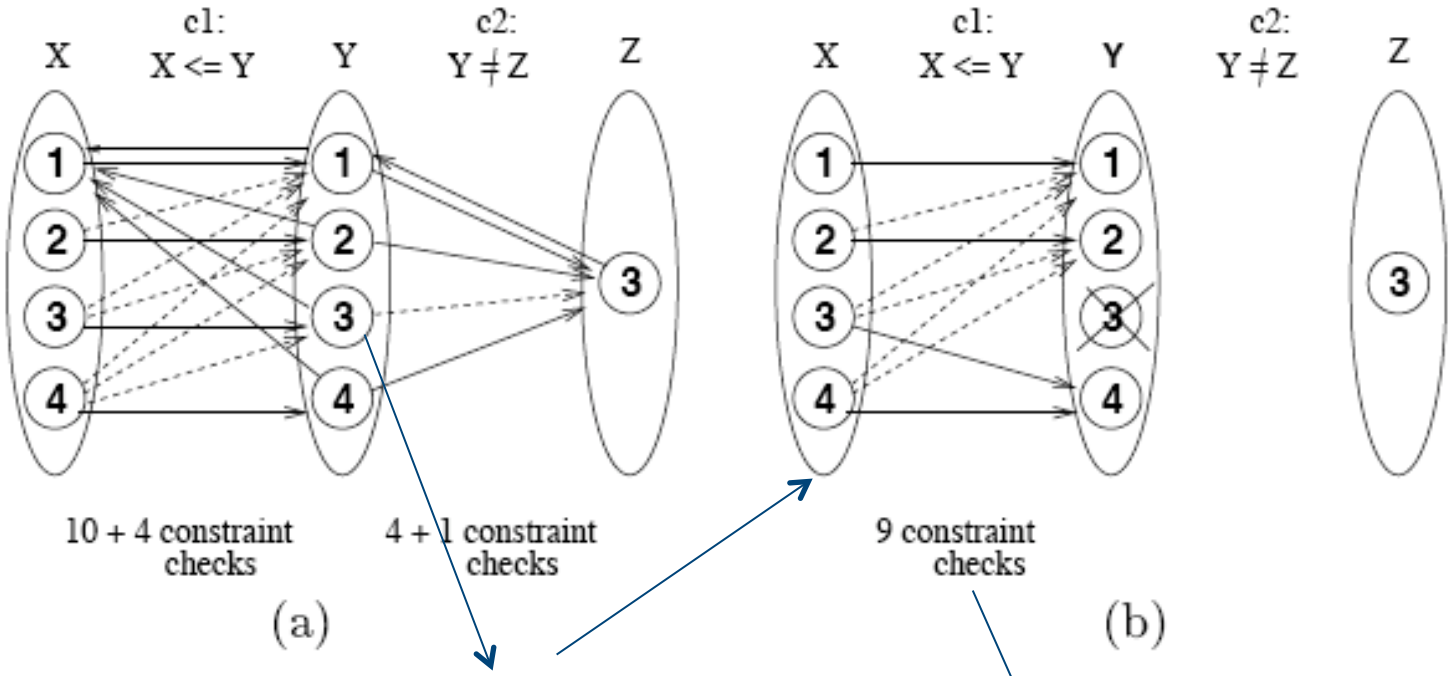
AC-3

- Idea
  - Revise (X_i, C): removes unsupported values of X_i and returns TRUE.
  - Place each (X_i, C) where X_i participates to C and its domain is potentially not AC, in a queue Q;
  - while Q is not empty:
    - select and remove (X_i, C) from Q;
    - if revise(X_i, C) then
      - If $D(X_i) = \{\}$ then return FALSE;
      - else place $\{(X_j, C') \mid X_i, X_j$ participate in some $C'\}$ into Q.

AC-3

- AC-3 achieves AC on binary CSPs in $O(ed^3)$ time and $O(e)$ space.
  - Time complexity is not optimal ☹
  - Revise does not remember anything about past computations and re-does unnecessary work.

AC-3



(X, C_1) is put in Q

only check of X ← 3 was necessary!

AC-4

- Stores max. amount of info in a preprocessing step so as to avoid redoing the same constraints checks.
- Idea:
  - Start with an empty queue Q.
  - Maintain counter[X_i, v_j, X_k] where X_i, X_k participate in a constraint C_{ik} and $v_j \in D(X_i)$
    - Stores the number of supports for $X_i \leftarrow v_j$ on C_{ik}.
  - Place all supports of $X_i \leftarrow v_j$ (in all constraints) in a list S[X_i, v_j].

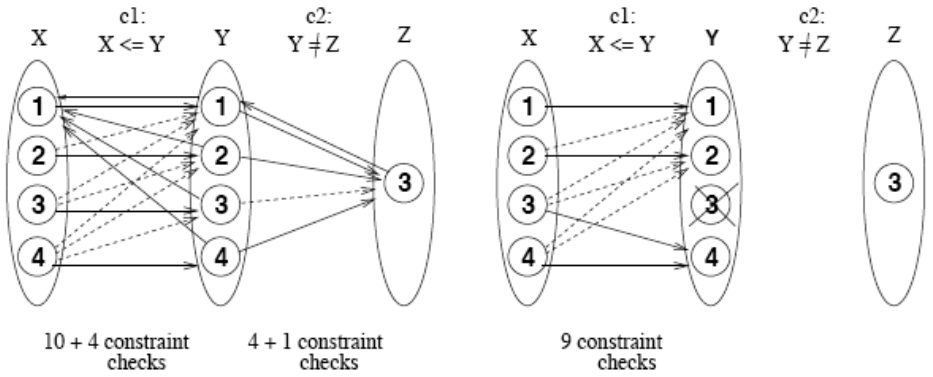
AC-4

- Initialisation:
  - All possible constraint checks are performed.
  - Each time a support for $X_i \leftarrow v_j$ is found, the corresponding counters and lists are updated.
  - Each time a support for $X_i \leftarrow v_j$ is not found, remove v_j from $D(X_i)$ and place (X_i, v_j) in Q for future propagation.
  - If $D(X_i) = \{\}$ then return FALSE.

AC-4

- Propagation:
  - While Q is not empty:
    - Select and remove (X_i, v_j) from Q;
    - For each (X_k, v_t) in $S[X_i, v_j]$
      - If $v_t \in D(X_k)$ then
        - decrement counter$[X_k, v_t, X_i]$
        - If counter$[X_k, v_t, X_i] = 0$ then
          - Remove v_t from $D(X_k)$; add (X_k, v_t) to Q
          - If $D(X_k) = \{\}$ then return FALSE.

AC-4



No additional constraint check!

(y,3) is put in Q

$$\texttt{counter}[x,1,y]=4 \quad \texttt{counter}[y,1,x]=1 \quad \texttt{counter}[y,1,z]=1$$
$$\texttt{counter}[x,2,y]=3 \quad \texttt{counter}[y,2,x]=2 \quad \texttt{counter}[y,2,z]=1$$
$$\texttt{counter}[x,3,y]=2 \quad \texttt{counter}[y,3,x]=3 \quad \texttt{counter}[y,3,z]=0$$
$$\texttt{counter}[x,4,y]=1 \quad \texttt{counter}[y,4,x]=4 \quad \texttt{counter}[y,4,z]=1$$
$$\texttt{counter}[z,3,y]=3$$

$$S[x,1]=\{(y,1),(y,2),(y,3),(y,4)\} \quad S[y,1]=\{(x,1),(z,3)\}$$
$$S[x,2]=\{(y,2),(y,3),(y,4)\} \quad S[y,2]=\{(x,1),(x,2),(z,3)\}$$
$$S[x,3]=\{(y,3),(y,4)\} \quad S[y,3]=\{(x,1),(x,2),(x,3)\}$$
$$S[x,4]=\{(y,4)\} \quad S[y,4]=\{(x,1),(x,2),(x,3),(x,4),(z,3)\}$$
$$S[z,3]=\{(y,1),(y,2),(y,4)\}$$

AC-4

- AC-3 achieves AC on binary CSPs in $O(ed^2)$ time and $O(ed^2)$ space.
  - Time complexity is optimal ☺
  - Space complexity is not optimal ☹
- AC-6 and AC-2001 achieve AC on binary CSPs in $O(ed^2)$ time and $O(ed)$ space.
  - Time complexity is optimal ☺
  - Space complexity is optimal ☺

PART IV: Search Algorithms