Global Constraints

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell (based on materials by Javier Larrosa)

March 15, 2021

Global Constraints

- Global constraints are
  classes of constraints defined by a Boolean formula of arbitrary arity

Global Constraints

- Global constraints are
  classes of constraints defined by a Boolean formula of arbitrary arity

- E.g., the `alldiff`$(x_1, \ldots, x_n)$ constraint forces that
  all the values of integer variables $x_1, \ldots, x_n$ must be different

- E.g., the `alo`$(x_1, \ldots, x_n)$ constraint forces that
  at least one of the Boolean variables $x_1, \ldots, x_n$ is set to true.

- E.g., the `amo`$(x_1, \ldots, x_n)$ constraint forces that
  at most one of the Boolean variables $x_1, \ldots, x_n$ is set to true.

Global Constraints

- Global constraints are
  classes of constraints defined by a Boolean formula of arbitrary arity

- E.g., the `alldiff`$(x_1, \ldots, x_n)$ constraint forces that
  all the values of integer variables $x_1, \ldots, x_n$ must be different

- E.g., the `alo`$(x_1, \ldots, x_n)$ constraint forces that
  at least one of the Boolean variables $x_1, \ldots, x_n$ is set to true.

- E.g., the `amo`$(x_1, \ldots, x_n)$ constraint forces that
  at most one of the Boolean variables $x_1, \ldots, x_n$ is set to true.

- The dual graph translation does not work well in practice.

AC for Non-binary Problems

- Can be naturally extended from the binary case
- Value $a \in d_i$ is AC wrt. (non-binary) constraint $c \in C$ iff
  there exists an assignment τ (the support of a) such that:
  - τ assigns a value to exactly the variables in $\mathrm{scope}(c)$
  - $\tau[x_i] = a$
  - $c(\tau)$ holds
- Constraint $c \in C$ is AC iff
  every $a \in d_i$ of every $x_i \in \mathrm{scope}(c)$ has a support in c
- A CSP is AC if all its constraints are AC
- For non-binary constraints, arc consistency is also called
  hyperarc consistency, generalized arc consistency or domain consistency

Example

- Consider the constraint $3x + 2y + z > 3$ over $x, y, z \in \{0, 1\}$
- Value 1 for x is AC: $\tau = (x \mapsto 1, y \mapsto 1, z \mapsto 1)$ is a support
- Value 0 for x is not AC: it does not have any support.
- Hence, the constraint is not AC

Example

- Note that AC depends on the syntax
- Consider $x_1 \in \{1, 2\}$, $x_2 \in \{1, 2\}$, $x_3 \in \{1, 3, 4\}$
- Case 1: constraints are $x_i \neq x_j$ for all $i < j$
  - All constraints are arc-consistent
- Case 2: there is only one constraint $\texttt{alldiff}(x_1, x_2, x_3)$
  - Value 1 for x_1 is AC
    because $\tau = (x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3)$ is a support for it.
  - Value 1 for x_3 is not AC: does not have any support
  - Hence, the constraint is not AC

Enforcing AC: $\texttt{Revise}(i, c)$

- Natural extension of binary case
- Removes values from the domain of x_i without a support in c

// Let $(x_1, \ldots, x_{i-1}, x_i, x_{i+1} \ldots, x_k)$ be the scope of c
function $\texttt{Revise}(i, c)$
    change $:=$ false
    for each $a \in d_i$ **do**
    if $\forall_{a_1 \in d_1, \ldots, a_{i-1} \in d_{i-1}, a_{i+1} \in d_{i+1}, \ldots, a_k \in d_k} \neg c(x_1 \leftarrow a_1, \ldots, x_i \leftarrow a, \ldots, x_k \leftarrow a_k)$
        remove a **from** d_i
        change $:=$ true
    return change

- The time complexity of $\texttt{Revise}(i, c)$ is $O(k \cdot |d_1| \cdots |d_k|)$
  (assuming that evaluating a constraint takes linear time in the arity)

AC-3

- The natural extension of binary AC-3
- $(i, c) \in Q$ means that
  "we cannot guarantee that all domain values of x_i have a support in c"

procedure $\texttt{AC3}(X, D, C)$
  $Q := \{(i, c) \mid c \in C, x_i \in \mathrm{scope}(C)\}$
  while $Q \neq \emptyset$ **do**
    $(i, c) := \texttt{Fetch}(Q)$ // selects and removes
    if $\texttt{Revise}(i, c)$ **then**
      $Q := Q \cup \{(j, c') \mid \; c' \in C, c' \neq c, j \neq i, \{x_i, x_j\} \subseteq \mathrm{scope}(c')\}$

- Let $m = \max_i\{|d_i|\}$, $e = |C|$ and $k = \max_c\{|\mathrm{scope}(c)|\}$
- Time complexity: $O(e \cdot k^3 \cdot m^{k+1})$
- Space complexity: $O(e \cdot k)$

AC for non-binary constraints

- Enforcing AC with generic algorithms is exponentially expensive in the maximum arity of the CSP
- Only practical with constraints of very small arity
- Is it possible to develop constraint-specific algorithms?

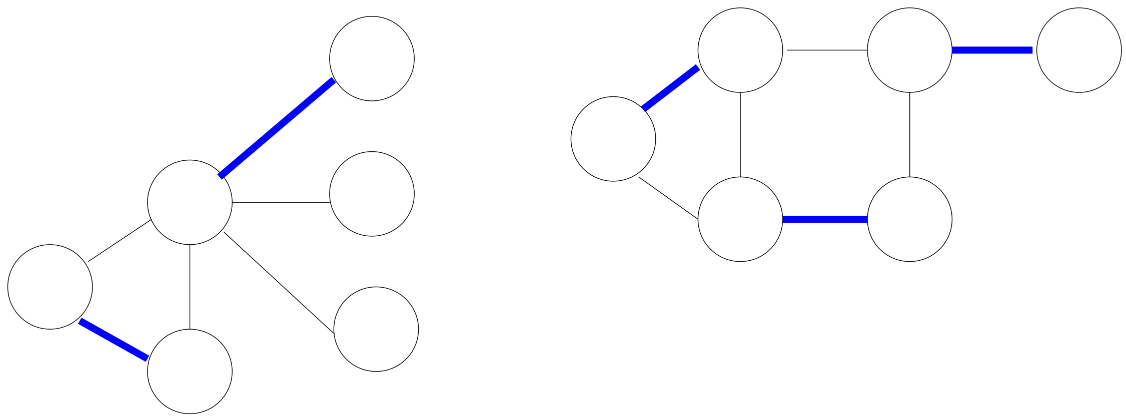
procedure `Revise`(c)
// removes every arc-inconsistent value $a \in d_i$ for all $x_i \in \text{scope}(c)$
endprocedure

- Next: `alldiff` constraint
- ... but first a diversion to matching theory

Begin Matching Theory

Definitions

- Given a graph $G = (V, E)$,
  a matching M is a set of pairwise non-incident edges
- A vertex is matched or covered if it is an endpoint of some $e \in M$,
  and it is free otherwise
- A maximum matching is a matching
  that contains the largest possible number of edges

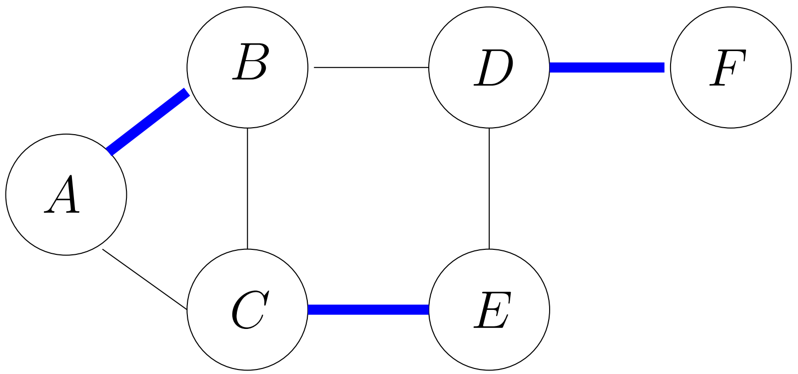


(edges in the matching, in blue)

- In particular, a perfect matching matches all vertices of the graph

Example

- We have to organize one round of a football league.
  Compatibility relation between teams is given by a graph



Perfect matchings $\leftrightarrow$ feasible arrangements of matches

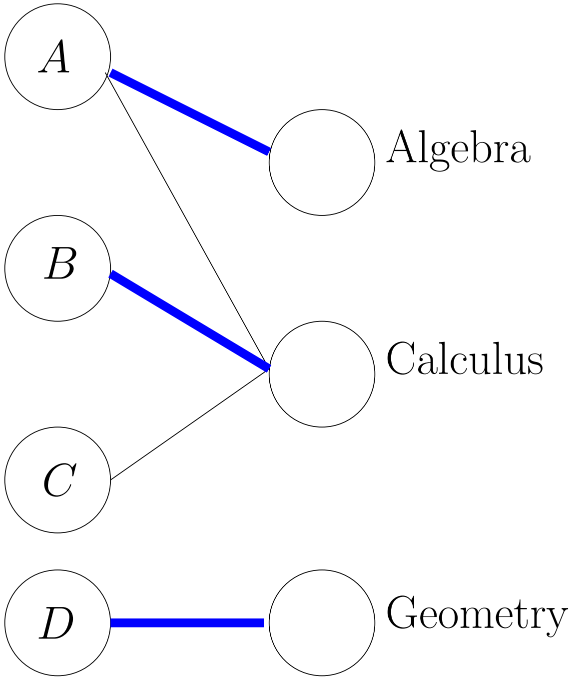
Bipartite Matching

- Graph $G = (V, E)$ is bipartite
  if there is a partition (L, R) of V (i.e., $L \cup R = V, L \cap R = \emptyset$)
  such that each $e \in E$ connects a vertex in L to one in R
- Now focus on maximum bipartite matching problem:
  given a bipartite graph, find a matching of maximum size
- From now on, assume $|V| \leq 2|E|$
  (isolated vertices can be removed)

Example (I)

- Assignment problem:
  - n workers, m tasks
  - list of pairs (w, t) meaning: "worker w can do task t"

Maximum matchings tell how to assign tasks to workers so that the maximum number of tasks are carried out



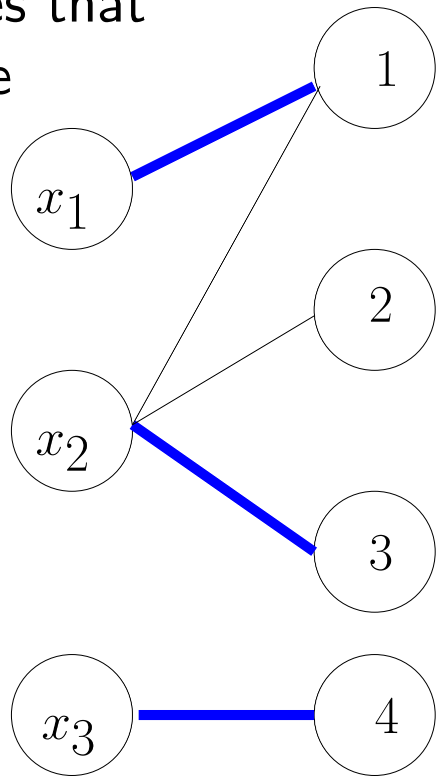
Example (II)

- We have n variables $x_1, ..., x_n$

  Variable x_i can take values in $D_i \subseteq \mathbb{Z}$ finite $(1 \leq i \leq n)$

  Constraint $\texttt{alldifferent}(x_1, ..., x_n)$ imposes that variables should take different values pairwise

$$
\begin{array}{rcl}
D_1 & = & \{1\} \\
D_2 & = & \{1, 2, 3\} \\
D_3 & = & \{4\}
\end{array}
$$



Matchings covering $x_1, \ldots, x_n$ correspond to solutions to $\texttt{alldifferent}(x_1, ..., x_n)$

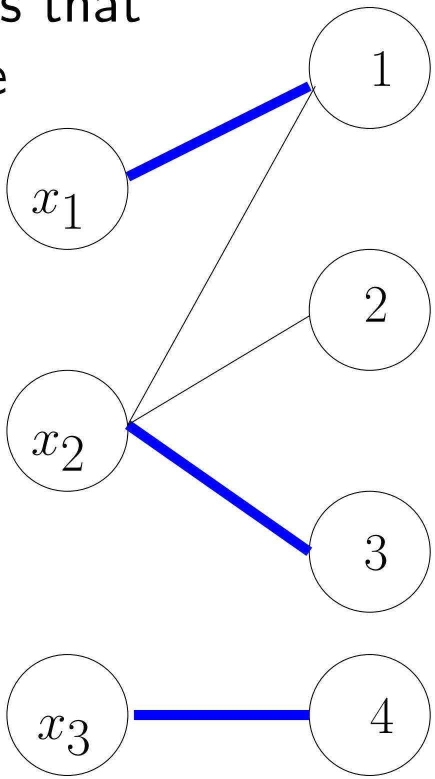
Example (II)

- We have n variables $x_1, ..., x_n$

  Variable x_i can take values in $D_i \subseteq \mathbb{Z}$ finite $(1 \leq i \leq n)$

  Constraint $\texttt{alldifferent}(x_1, ..., x_n)$ imposes that variables should take different values pairwise

$$\begin{array}{rcl} D_1 & = & \{1\} \\ D_2 & = & \{1, 2, 3\} \\ D_3 & = & \{4\} \end{array}$$



  Matchings covering $x_1, \ldots, x_n$ correspond to solutions to $\texttt{alldifferent}(x_1, ..., x_n)$

- Note that matchings covering $x_1, \ldots, x_n$ are maximum. However, a maximum matching may not cover $x_1, \ldots, x_n$

End Matching Theory

Arc Consistency for `alldiff`

[reminder]

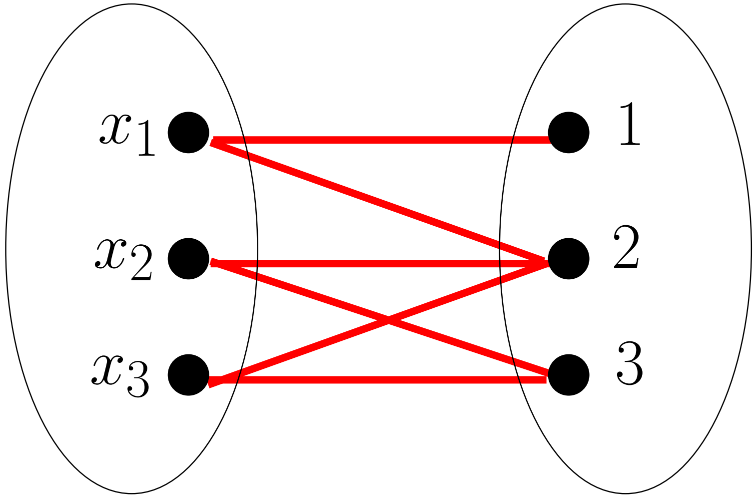
- Consider $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3\}$ and the constraint `alldiff`(x_1, x_2, x_3)
  - Value 1 for x_1 is AC since $\tau = (x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3)$ is a support for it.
  - Value 2 for x_1 is not AC: it does not have any support (no room left for x_2, x_3)
  - After enforcing AC: $x_1 \in \{1\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3\}$

Value Graph of `alldiff`

- Given variables $X = \{x_1, \ldots, x_n\}$ with domains $D_1, \ldots, D_n$,
  the value graph of $\texttt{alldiff}(x_1, \ldots, x_n)$ is
  the bipartite graph $G = (X \cup \bigcup_{i=1}^{n} D_i, E)$ where $(x_i, v) \in E$ iff $v \in D_i$

$\texttt{alldiff}(x_1, x_2, x_3)$

$$
\begin{array}{rcl}
D_1 & = & \{1, 2\} \\
D_2 & = & \{2, 3\} \\
D_3 & = & \{2, 3\}
\end{array}
$$

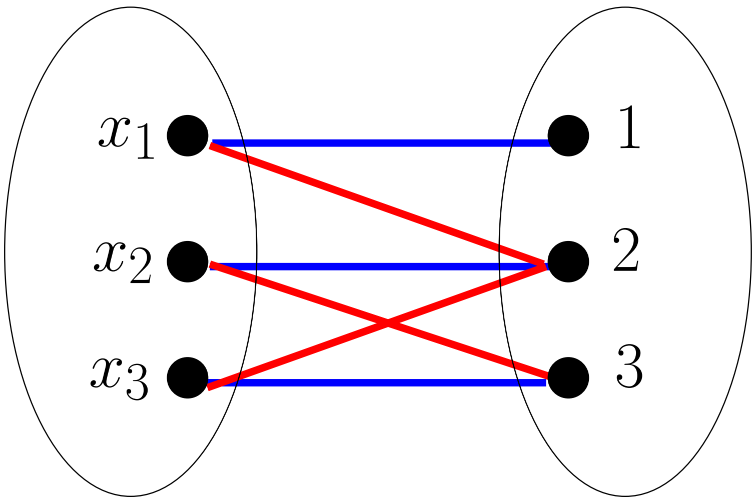


Solutions and Matchings

- We say a matching M covers a set S iff
  every vertex in S is covered (i.e, is an endpoint of an edge in M)
- Solutions to $\texttt{alldiff}(X) =$ matchings covering X

$$\texttt{alldiff}(x_1, x_2, x_3)$$

$$\begin{array}{lcl} D_1 & = & \{1,2\} \\ D_2 & = & \{2,3\} \\ D_3 & = & \{2,3\} \end{array} \qquad \begin{array}{l} x_1 = 1 \\ x_2 = 2 \\ x_3 = 3 \end{array}$$

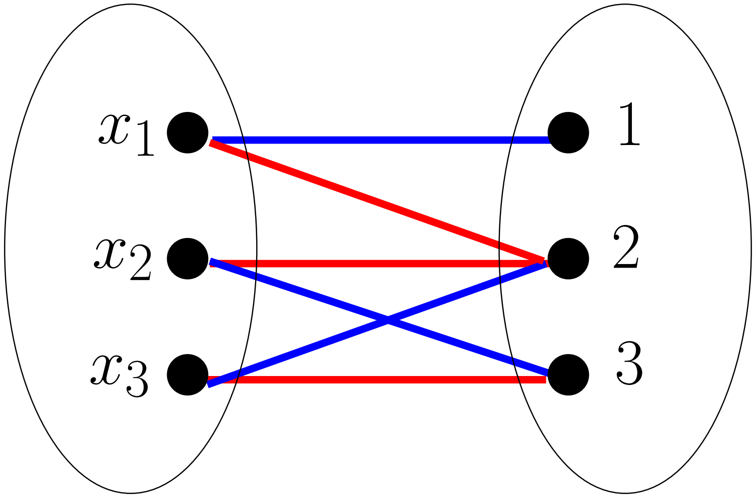


Solutions and Matchings

- We say a matching M covers a set S iff
  every vertex in S is covered (i.e, is an endpoint of an edge in M)
- Solutions to $\texttt{alldiff}(X) =$ matchings covering X

$$\texttt{alldiff}(x_1, x_2, x_3)$$

$$\begin{array}{lcl} D_1 & = & \{1,2\} \\ D_2 & = & \{2,3\} \\ D_3 & = & \{2,3\} \end{array} \qquad \begin{array}{l} x_1 = 1 \\ x_2 = 3 \\ x_3 = 2 \end{array}$$

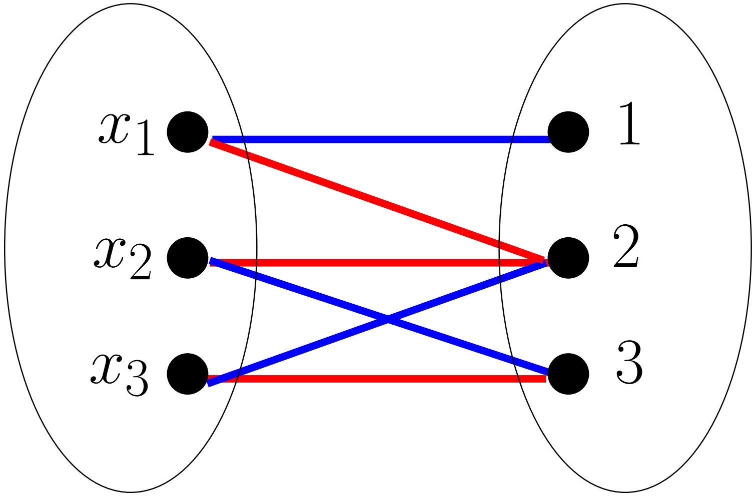


Solutions and Matchings

- We say a matching M covers a set S iff
  every vertex in S is covered (i.e, is an endpoint of an edge in M)
- Solutions to $\texttt{alldiff}(X) =$ matchings covering X

$$\texttt{alldiff}(x_1, x_2, x_3)$$

$$\begin{array}{lcl} D_1 & = & \{1,2\} \\ D_2 & = & \{2,3\} \\ D_3 & = & \{2,3\} \end{array} \qquad \begin{array}{l} x_1 = 1 \\ x_2 = 3 \\ x_3 = 2 \end{array}$$



- A matching covering X is a maximum matching
- There are solutions to $\texttt{alldiff}(X)$ iff size of maximum matchings is $|X|$

Solutions and Matchings

- Algorithm for checking feasibility of $\texttt{alldiff}(X)$:
  (with Hopcroft-Karp, in time $O(dn\sqrt{n})$, where $n = |X|$, $d = \max_i\{|D_i|\}$)

```
// Returns true iff there is a solution to alldiff(X)
// G is the value graph of alldiff(X)
M = COMPUTE_MAXIMUM_MATCHING(G)
if ( |M| < |X| ) return false

return true
```

Solutions and Matchings

- Algorithm for checking feasibility of $\texttt{alldiff}(X)$:
  (with Hopcroft-Karp, in time $O(dn\sqrt{n})$, where $n = |X|$, $d = \max_i\{|D_i|\}$)

```
// Returns true iff there is a solution to alldiff(X)
// G is the value graph of alldiff(X)
M = COMPUTE_MAXIMUM_MATCHING(G)
if ( |M| < |X| ) return false
else REMOVE_EDGES_FROM_GRAPH(G, M) // Remove non-AC values
return true
```

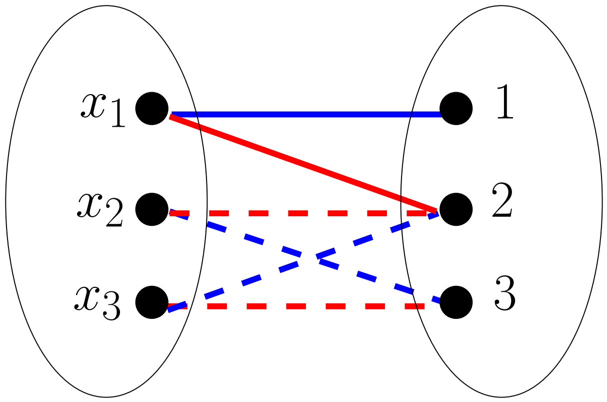
- But in addition to check feasibility we want to find arc-inconsistent values
- Assume $\texttt{alldiff}(X)$ has a solution corresponding to matching M. Then:
  value v from the domain of variable x is arc-inconsistent iff
  there is no solution to $\texttt{alldiff}(X)$ that assigns value v to x iff
  there is no matching covering X that contains edge (x, v) iff
  there is no maximum matching that contains edge (x, v)
- So we have to remove the edges not contained in any maximum matching
- Next we'll extend the algorithm to do so using (maximum) matching M

Filtering

- We want to remove the edges not contained in any maximum matching
- We will identify the complementary set:
  the edges contained in some maximum matching
- We say an edge is vital if it belongs to all maximum matchings
- Given a matching M, an alternating path is
  a simple path in which the edges belong alternatively to M and not to M.
- Given a matching M, an alternating cycle is
  a cycle in which the edges belong alternatively to M and not to M.

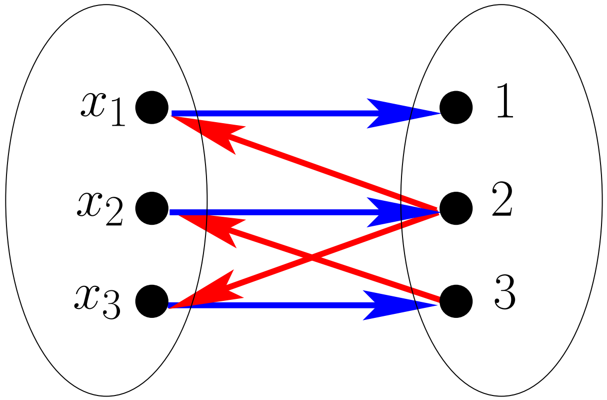
Filtering

- We want to remove the edges not contained in any maximum matching
- We will identify the complementary set:
  the edges contained in some maximum matching
- **Theorem.** Let M be an arbitrary maximum matching.
  An edge belongs to some maximum matching iff
  - it is vital; or
  - it belongs to an alternating cycle wrt. M; or
  - it belongs to an even-length simple alternating path starting at a free vertex wrt. M



Orienting Edges

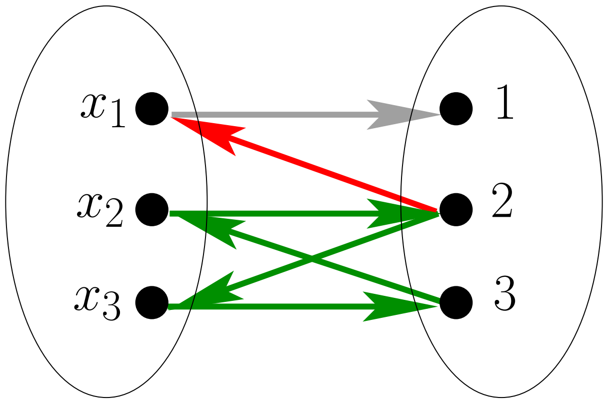
- It simplifies things to orient edges:
  - Edges $e \in M$ are oriented from left to right
  - Edges $e \notin M$ are oriented from right to left



Orienting Edges

- **Corollary.** Let M be an arbitrary maximum matching.
  An edge belongs to some maximum matching iff
  - it belongs to a cycle, or
  - it belongs to a simple path starting at a free vertex wrt. M, or
  - it is vital

  in the oriented graph.



Removing Arc-Inconsistent Edges

- We will actually identify AC edges, and the remaining ones will be non-AC
- An edge (u, v) belongs to a cycle in a digraph G iff u, v belong to the same strongly connected component (SCC) of G

REMOVE_EDGES_FROM_GRAPH(G, M)

0) Mark all edges in G as UNUSED

1) Compute SCC's, and
   mark as USED edges with vertices in same SCC

2) Do a depth-first search from free vertices, and
   mark as USED edges in simple paths starting at free vertices

3) Mark UNUSED edges of M as VITAL

4) Remove remaining UNUSED edges

- Time complexity: linear in the size of the value graph

Computing SCC's

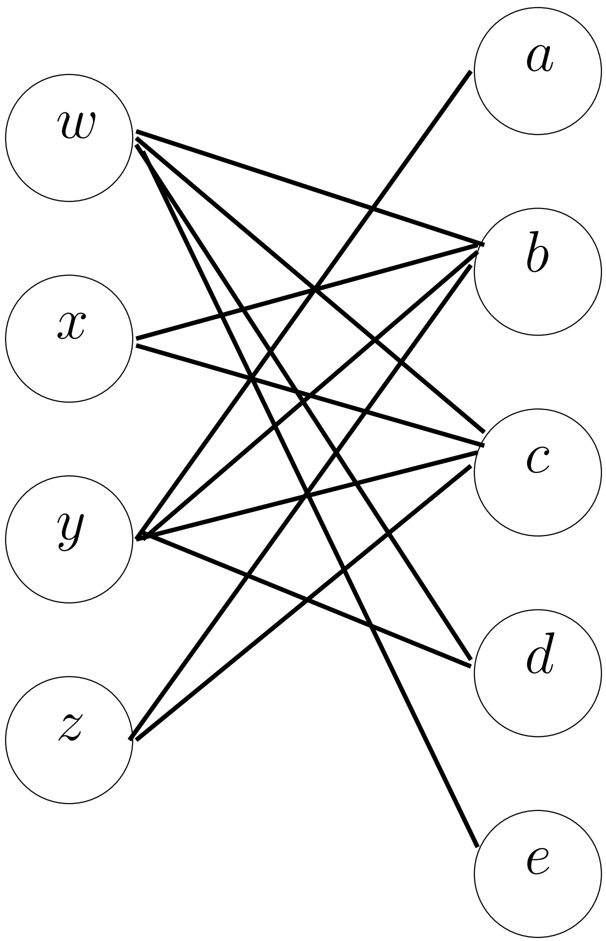
- Given a directed graph $G = (V, E)$,
  SCC's can be computed in time $O(|V| + |E|)$,
  e.g. with Kosaraju's algorithm:

  1. Do DFS
  2. Reverse the direction of the edges
  3. Do DFS in reverse chronological order of finish times wrt. step 1.
  4. Each tree in the previous DFS forest is a SCC

Example

- Variables $\{w, x, y, z\}$
- Domains

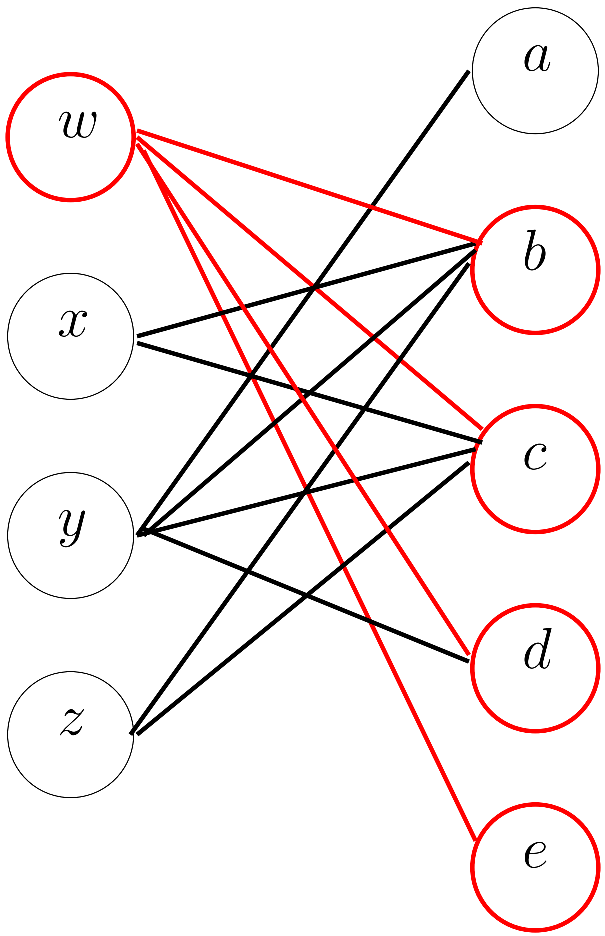
  $d(w) = \{b, c, d, e\}$,
  $d(x) = \{b, c\}$,
  $d(y) = \{a, b, c, d\}$,
  $d(z) = \{b, c\}$



Example

- Variables $\{w, x, y, z\}$
- Domains

  $d(w) = \{b, c, d, e\}$,
  $d(x) = \{b, c\}$,
  $d(y) = \{a, b, c, d\}$,
  $d(z) = \{b, c\}$



Example

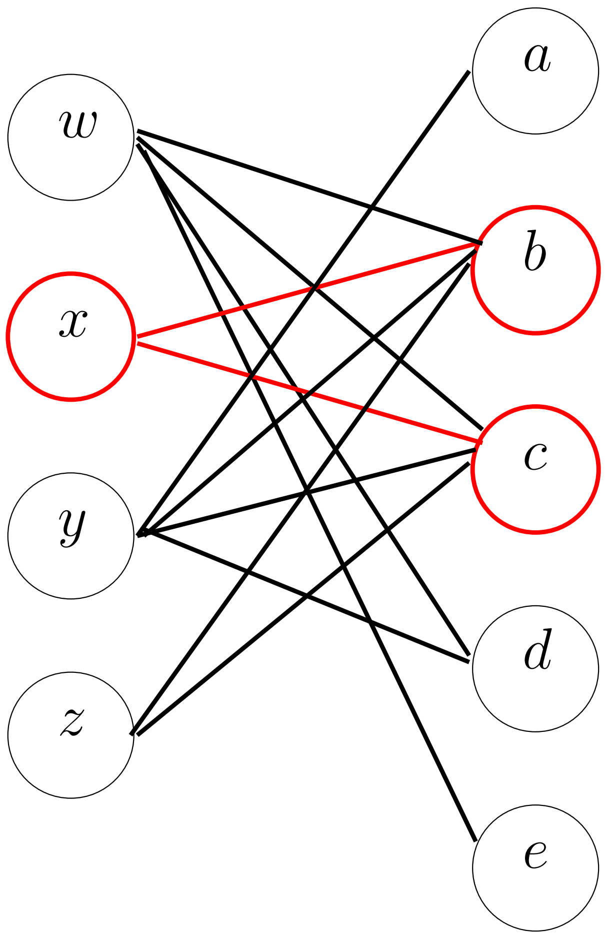
- Variables $\{w, x, y, z\}$
- Domains

  $d(w) = \{b, c, d, e\}$,

  $d(x) = \{b, c\}$,

  $d(y) = \{a, b, c, d\}$,

  $d(z) = \{b, c\}$



Example

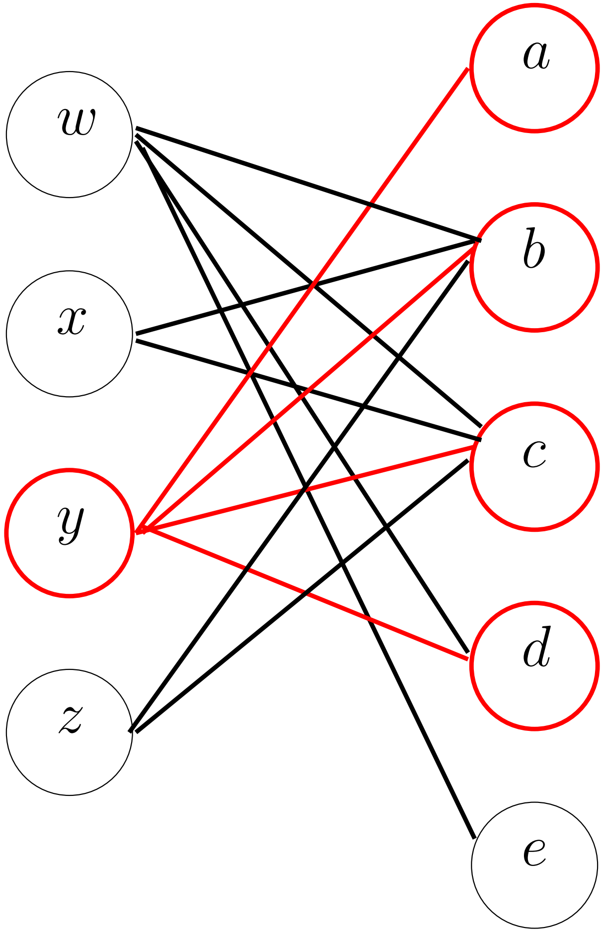
- Variables $\{w, x, y, z\}$
- Domains

  $d(w) = \{b, c, d, e\}$,

  $d(x) = \{b, c\}$,

  $d(y) = \{a, b, c, d\}$,

  $d(z) = \{b, c\}$

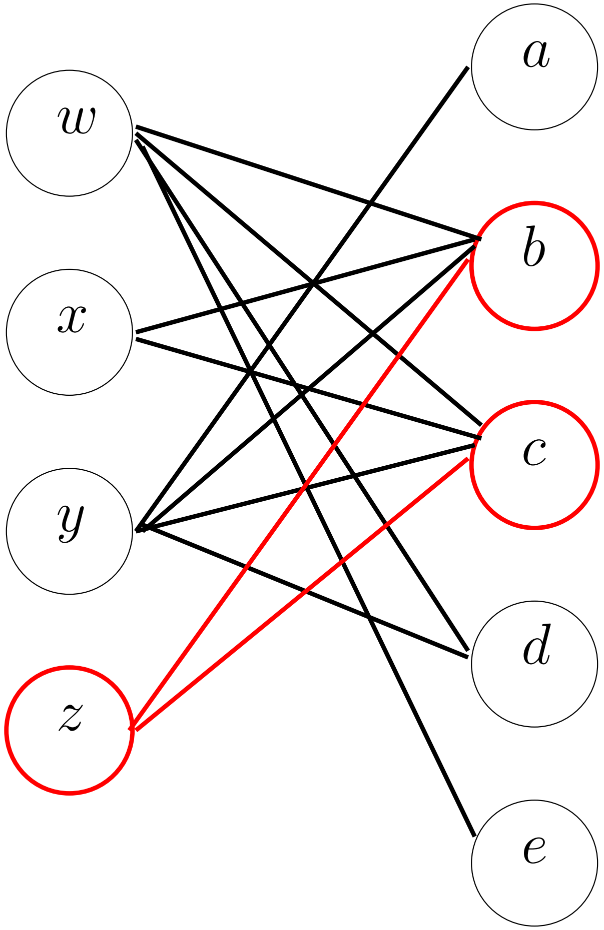


Example

- Variables $\{w, x, y, z\}$

- Domains

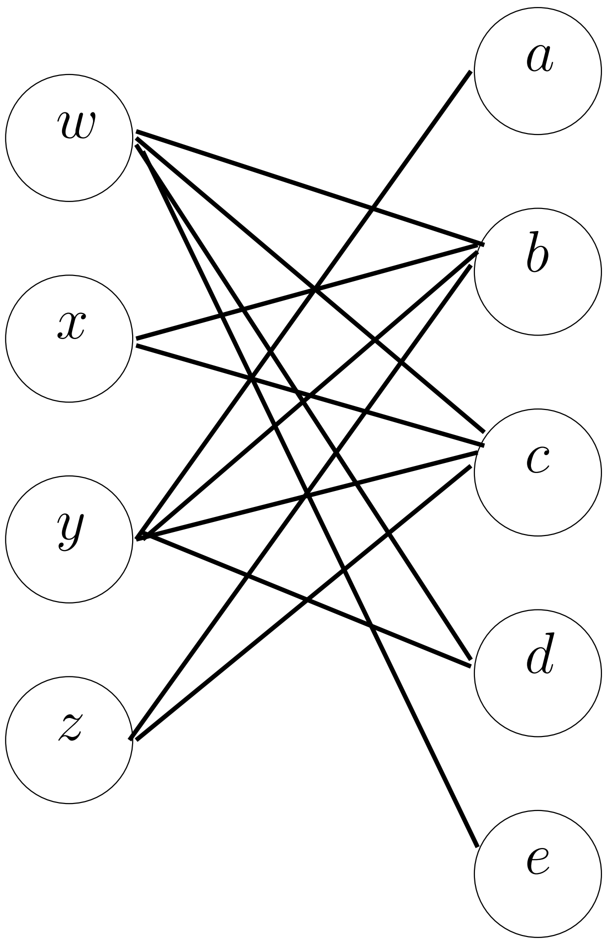
  $d(w) = \{b, c, d, e\}$,
  $d(x) = \{b, c\}$,
  $d(y) = \{a, b, c, d\}$,
  $d(z) = \{b, c\}$



Example

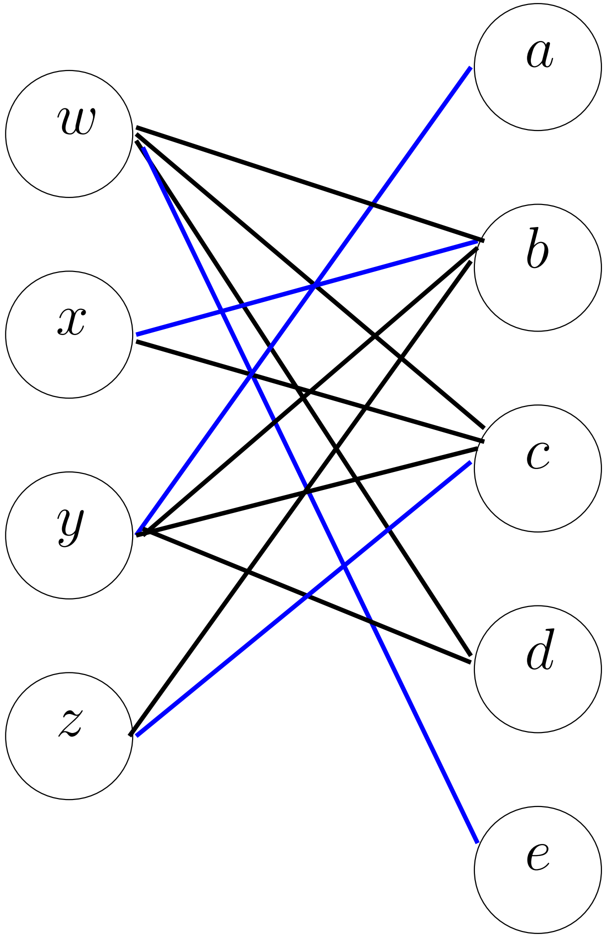
- Variables $\{w, x, y, z\}$
- Domains

  $d(w) = \{b, c, d, e\}$,
  $d(x) = \{b, c\}$,
  $d(y) = \{a, b, c, d\}$,
  $d(z) = \{b, c\}$



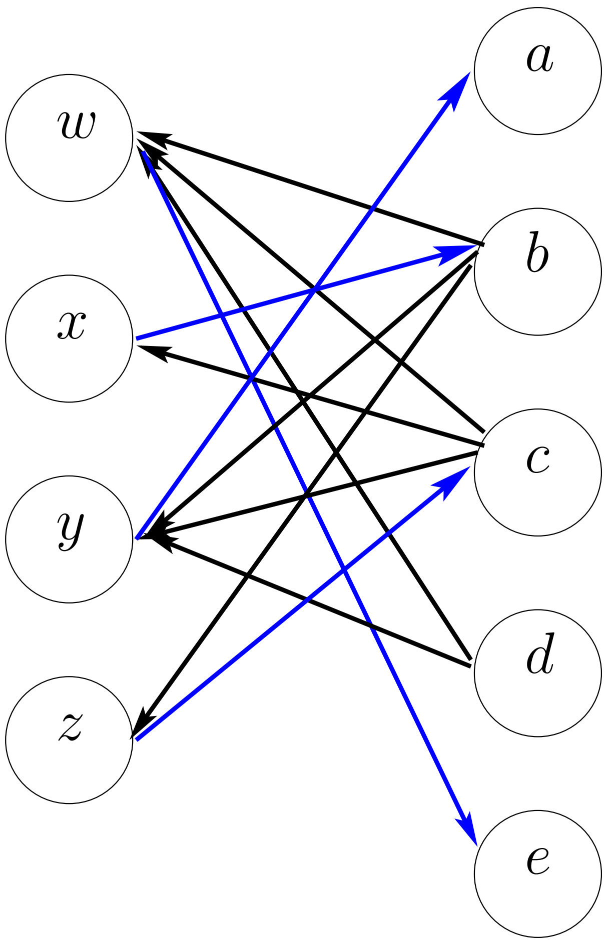
Example

- We assume we already have a maximum matching
- All variables are covered



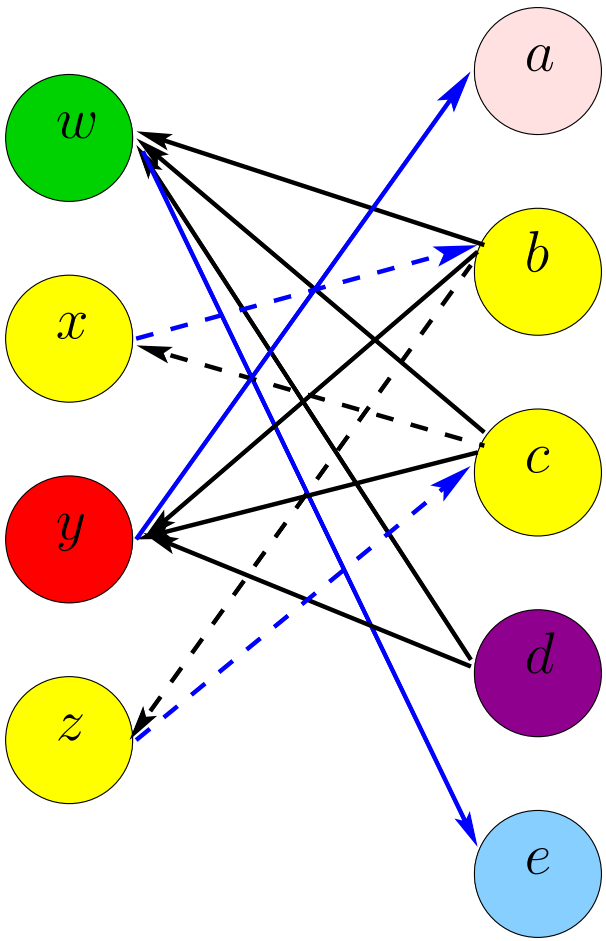
Example

- Direct the edges



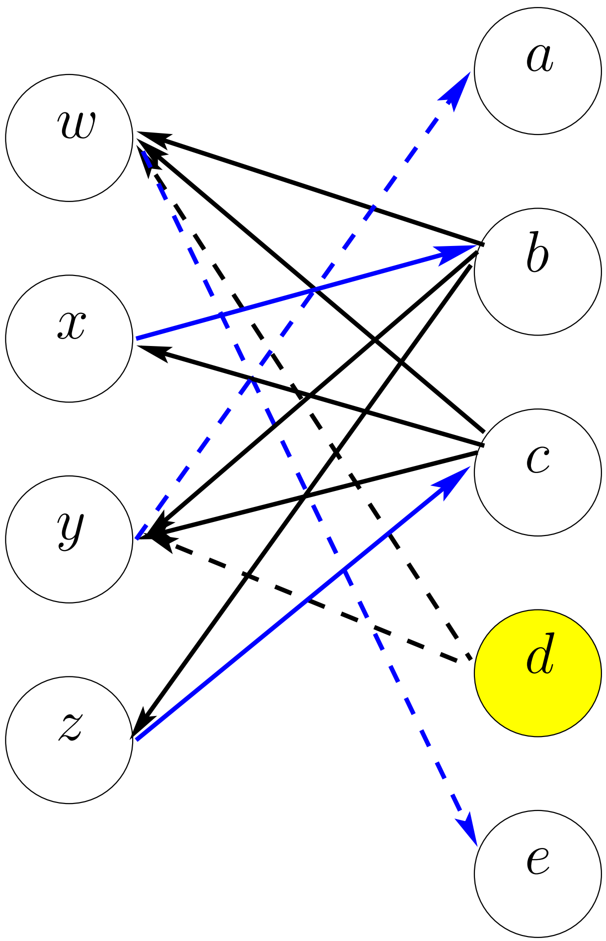
Example

- Compute SCC's



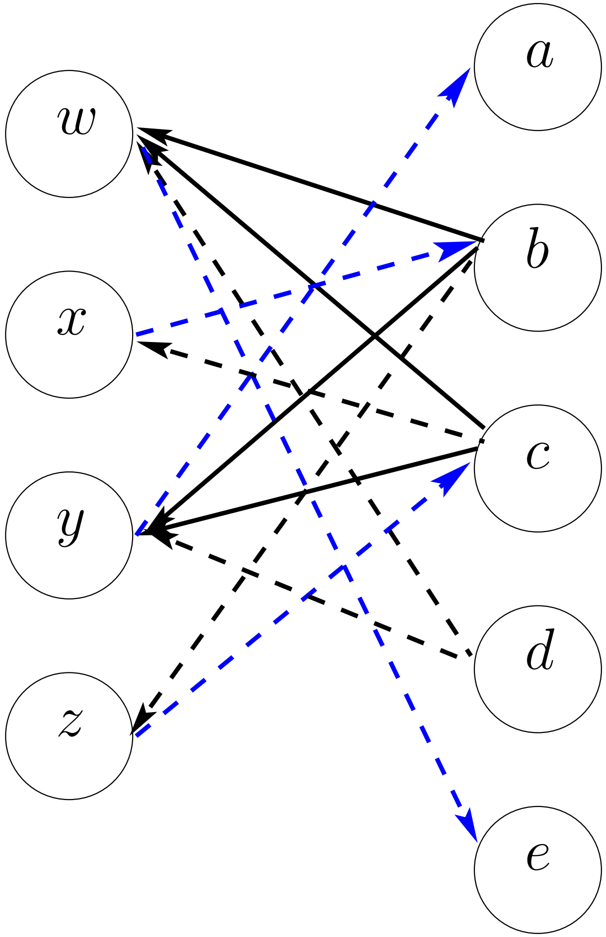
Example

- Compute all simple paths starting at a free vertex



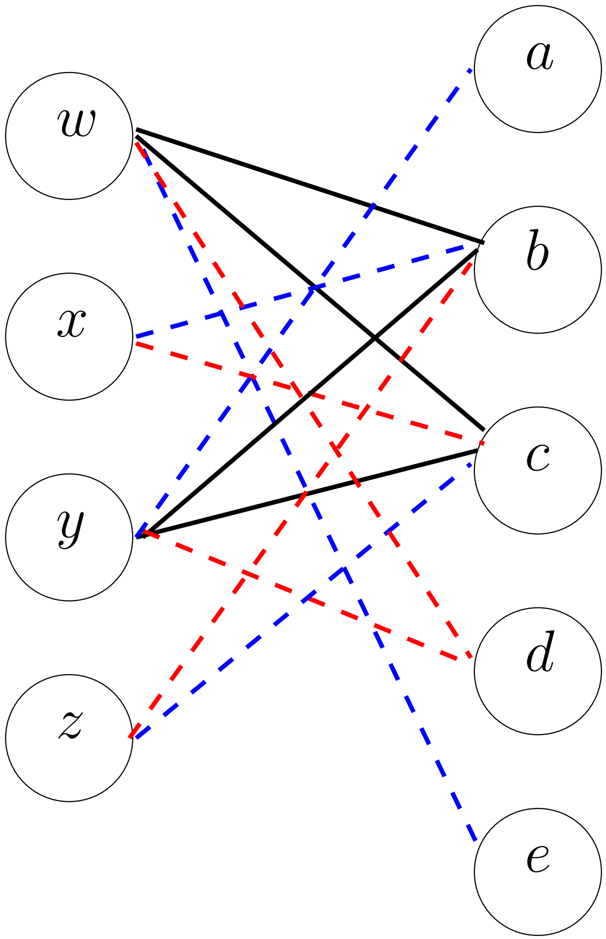
Example

- Identify vital edges (none in this case)



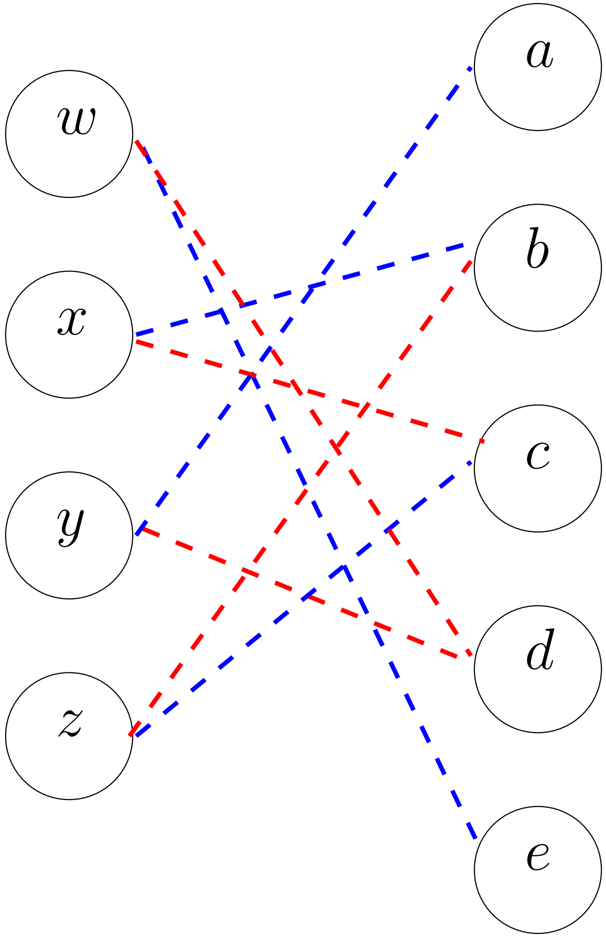
Example

- Remove unused edges that are not vital



Example

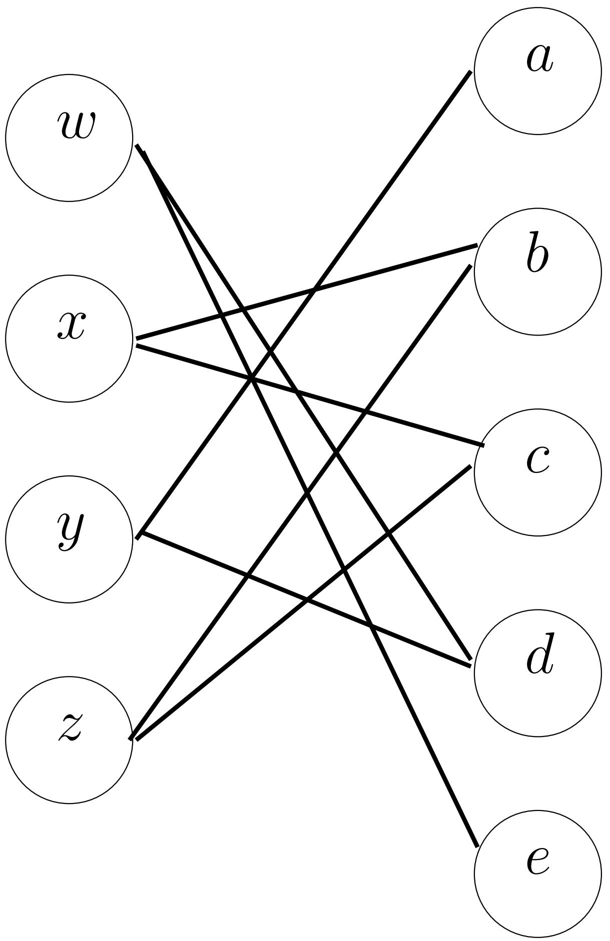
- Remove unused edges that are not vital



Example

- After enforcing arc consistency:

  $d(w) = \{d, e\}$,
  $d(x) = \{b, c\}$,
  $d(y) = \{a, d\}$,
  $d(z) = \{b, c\}$



Complexity

- Consider CSP with a single constraint $\texttt{alldiff}(x_1, \ldots, x_k)$ where $m = \max_i\{|D_i|\}$
- Cost of enforcing AC with AC-3: $O(k^3 m^{k+1})$
- Cost of enforcing AC with bipartite matching: $O(km\sqrt{k})$
  - Cost of constructing maximum matching: $O(km\sqrt{k})$
  - Cost of removing edges: $O(km)$