Global Constraints

Combinatorial Problem Solving (CPS)

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

March 8, 2019
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 $\tau$ (the support of $a$) such that:
  
  - $\tau$ assigns a value to exactly the variables in $\text{scope}(c)$
  - $\tau[x_i] = a$
  - $c(\tau)$ holds

- Constraint $c \in C$ is AC iff every $a \in d_i$ of every $x_i \in \text{scope}(c)$ 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 \( x_1 \in \{a, b\} \), \( x_2 \in \{a, b\} \), \( x_3 \in \{a, c, d\} \).

- **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 \( \text{alldiff}(x_1, x_2, x_3) \).
  - Value \( a \) for \( x_1 \) is AC because \( \tau = (x_1 \mapsto a, x_2 \mapsto b, x_3 \mapsto c) \) is a support for it.
  - Value \( a \) 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 domain values 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)
The natural extension of binary AC-3

(i, c) ∈ Q means that
"we cannot guarantee that all domain values of x_i have a support in c"

procedure AC3(X, D, C)

Q := {(i, c) | c ∈ C, x_i ∈ scope(C)}

while Q ≠ ∅ do
    (i, c) := Fetch(Q) // selects and removes
    if Revise(i, c) then
        Q := Q ∪ {(j, c′) | c′ ∈ C, c′ ≠ c, j ≠ i, {x_i, x_j} ⊆ scope(c′)}

Let m = max_i{|d_i|}, e = |C| and k = max_c{|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?

```plaintext
procedure Revise(c)
  // removes every arc-inconsistent value \( a \in d_i \) for all \( x_i \in X(c) \)
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.

![Graph Diagram]

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 \( x_1, \ldots, x_n \)
- Variable \( x_i \) can take values in \( D_i \subseteq \mathbb{Z} \) finite \((1 \leq i \leq n)\)
- Constraint \texttt{alldifferent}(x_1, \ldots, x_n)\)
  imposes that variables should take different values pairwise

\[
\begin{align*}
D_1 &= \{1\} \\
D_2 &= \{1, 2, 3\} \\
D_3 &= \{4\}
\end{align*}
\]

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

Let $M$ be a matching of $G = (V, E)$ (not necessarily bipartite).

We view paths as sequences of edges rather than vertices.

- An **alternating path** is a simple path in which the edges belong alternatively to $M$ and not to $M$.

- An **alternating cycle** is a cycle in which the edges belong alternatively to $M$ and not to $M$.

- An **augmenting path** is an alternating path that starts and ends at different free vertices.

- **Berge’s Lemma.** A matching is maximum if and only if it does not have any augmenting path.
Properties (I)

- An alternating cycle has as many edges in $M$ as not in $M$.
- An augmenting path has 1 more edge not in $M$ than in $M$.
- Given two sets $A, B \subseteq X$, their symmetric difference is $A \oplus B = (A - B) \cup (B - A)$.

If $P$ is an augmenting path wrt. $M$, then $M \oplus P$ is a matching and $|M \oplus P| = |M| + 1$.

I.e., if we paint edges $\in M$ in blue and edges $\not\in M$ in red, then flipping the colors of $P$ results in a valid matching.
Proof of Berge’s Lemma (I)

Let us prove the contrapositive:

\( G \) has a matching larger than \( M \) if and only if \( G \) has an augmenting path wrt. \( M \)

(\( \Leftarrow \)) Just proved in the last slide.
Proof of Berge’s Lemma (II)

(⇒) Let $M'$ be a matching in $G$ larger than $M$.

Each vertex of $M \oplus M'$ has degree at most two: incident with $\leq 1$ edge from $M$ and $\leq 1$ edge from $M'$

So $M \oplus M'$ is a vertex-disjoint union of simple paths and cycles.

Furthermore, paths and cycles in $M \oplus M'$ are alternating (wrt. $M$, and wrt. $M'$)

Edges $\in M$, $\notin M'$

Edges $\in M'$, $\notin M$
(⇒) (cont.) Since \(|M'| > |M|\), \(M \oplus M'\) must contain at least one connected component that has more edges from \(M'\) than from \(M\).

Such a component is a simple path in \(G\) that starts and ends at different vertices with edges \(\notin M\).

The extreme vertices are free.

So the path is augmenting.
Aug. Paths in Bipartite Graphs

- **Idea:** Starting from the empty matching, increase the size of the current matching by finding augmenting paths.
- Now assume the graph is **bipartite**.
- For finding augmenting paths, do the following:
  1. Mark vertices as matched or free.
  2. Start DFS (Depth First Search) or BFS (Breadth First Search) from each of the free vertices in \( L \).
  3. Traverse edges \( \notin M \) from \( L \) to \( R \).
  4. Traverse edges \( \in M \) from \( R \) to \( L \).
  5. Stop successfully if a free vertex from \( R \) is reached.
  6. Stop with failure if search terminates without finding a free vertex from \( R \).
- **Cost:** \( O(|E|) \)
Algorithm

```c
int MAX_BIPARTITE_MATCHING(bipartite_graph G) {
    M = ∅;
    P = AUG_PATH(G, M);
    while (P != NULL) {
        M = M ⊕ P;
        P = AUG_PATH(G, M);
    }
    return M.size();
}
```

- Cost: $O(|V||E|)$
  - Each iteration costs $O(|E|)$
  - At each iteration 2 new vertices are matched (one from $L$ and one from $R$)
    
    So at most $\min(|L|, |R|) = O(|V|)$ iterations suffice
Example

Bipartite graph \( G = (L \cup R, E) \)

Initially matching \( M \) is empty.

Blue edges: \( e \in M \)

Red edges: \( e \notin M \)

Let us look for an augmenting path using DFS.
Example

Mark vertices as matched \((m)\) or free \((f)\).

Start at a free vertex in \(L\).

Left \(\rightarrow\) right: red edges

Right \(\rightarrow\) left: blue edges
Example

Found a free vertex in $R$.

Found an augmenting path.
Example

Flip colors of augmenting path and a new $M$ is obtained
Example

Let us look for another augmenting path.

By symmetry.
Example

Let us look for another augmenting path.

Mark vertices as matched ($m$) or free ($f$).

Start at a free vertex in $L$.

Left $\rightarrow$ right: red edges

Right $\rightarrow$ left: blue edges
Example

Found a free vertex in $R$.

Found an augmenting path.
Example

Flip colors of augmenting path and a new $M$ is obtained.
Example

By symmetry.

No more augmenting paths, $M$ is a maximum matching
Hopcroft-Karp Algorithm

- If \( P_1, \ldots, P_k \) are vertex-disjoint augmenting paths wrt. \( M \), then \( M \oplus (P_1 \cup \cdots \cup P_k) \) is a matching of \( |M| + k \) edges

- **Idea:** instead of finding 1 augmenting path per iteration, let us find a maximal set of vertex-disjoint shortest augmenting paths. This reduces the number of iterations from \( O(|V|) \) to \( O(\sqrt{|V|}) \)

```plaintext
int HOPCROFT_KARP(bipartite_graph G) {
    M = \emptyset;
    S = MAXIMAL_SET_VD_SHORTEST_AUG_PATHS(G, M);
    while (S != \emptyset) {
        M = M \oplus \bigcup\{ P \mid P \in S \};
        S = MAXIMAL_SET_VD_SHORTEST_AUG_PATHS(G, M);
    }
    return M.size();
}
```
Max. Vertex-Disjoint Shortest AP

- Let us find a maximal set of vertex-disjoint shortest augmenting paths
- Let $l$ be the length of the shortest augmenting paths wrt. $M$

**Goal:** compute a maximal (*not necessarily maximum*) set of vertex-disjoint augmenting paths of length $l$

- **Phase 1:** compute length $l$ and augmenting paths of length $l$
  
  1. BFS but start simultaneously at all free vertices in $L$
  2. Traverse edges $\not\in M$ from $L$ to $R$
  3. Traverse edges $\in M$ from $R$ to $L$
  4. If a free vertex is found in $R$:
     current distance is $l$, the length of the shortest augmenting paths
  5. Complete BFS after finding all free vertices in $R$ at distance $l$
Max. Vertex-Disjoint Shortest AP

- We need augmenting paths to be vertex-disjoint
- Phase 2: ensure vertex-disjointness and maximality
  Let $X$ be the set of all free vertices in $R$ at distance $l$

1. Compute DFS from $u \in X$ to the free vertices in $L$, using the BFS distances to guide the search:
   - the DFS is only allowed to follow edges that lead to an unused vertex in the previous distance layer
   - the DFS must alternate between matched and unmatched edges.

2. Once an augmenting path is found, mark its vertices as used and continue the DFS from the next $u \in X$.

- Cost of Phase 1: $O(|E|)$ (1 single BFS!)
- Cost of Phase 2: $O(|E|)$ (1 single DFS!)
Theorem. Let:

- $l =$ length of a shortest augmenting path wrt. $M$
- $P_1, \ldots, P_k =$ a maximal set of vertex-disjoint shortest augmenting paths wrt. $M$
- $M' = M \oplus (P_1 \cup \ldots \cup P_k)$
- $P =$ a shortest augmenting path with respect to $M'$

Then $|P| > l$.

I.e., from one iteration to the next one, the length of the shortest augmenting path increases.
Proof. Let us consider two cases:

1. \( P \) is vertex-disjoint from \( P_1, \ldots, P_k \). By contradiction.

   Since \( P \) is an augmenting path wrt. \( M' \) and is vertex-disjoint from \( P_1, \ldots, P_k \), \( P \) is an augmenting path wrt. \( M \).

   Then \( |P| \geq l \).

   If \( |P| = l \), then \( P \) is a shortest augmenting path wrt. \( M \).

   But this contradicts the maximality of \( P_1, \ldots, P_k \).

   So \( |P| > l \).
Progress in Hopcroft-Karp

2. \( P \) is not vertex-disjoint from \( P_1, \ldots, P_k \).

By def., \( M' = M \oplus (P_1 \cup \ldots \cup P_k) \).

So \( M \oplus M' = (M \oplus M) \oplus (P_1 \cup \ldots \cup P_k) = P_1 \cup \ldots \cup P_k \).

So \( H := M \oplus M' \oplus P = (P_1 \cup \ldots \cup P_k) \oplus P \).

But \( H \) is a set of vertex-disjoint cycles and simple paths.

And \( |M' \oplus P| - |M| = |M' \oplus P| - |M'| + |M'| - |M| = k + 1 \)

So there are at least \( k + 1 \) simple paths in \( H \) that use more edges from \( M' \oplus P \) than from \( M \).

Each of these is an augmenting path wrt. \( M \).

So \( H \) contains \( \geq k + 1 \) vertex-disjoint augmenting paths with respect to \( M \), each of which of length \( \geq l \).
Progress in Hopcroft-Karp

(cont.) So \(|H| = |(P_1 \cup \ldots \cup P_k) \oplus P| \geq (k + 1)l.\)

Hence \(|(P_1 \cup \ldots \cup P_k) - P| + |P - (P_1 \cup \ldots \cup P_k)| \geq (k + 1)l\)

As \(P_1, \ldots, P_k\) are vertex-disjoint and have length \(l\), they contribute to \(|(P_1 \cup \ldots \cup P_k) - P|\) with at most \(kl\) distinct edges.

So \(P - (P_1 \cup \ldots \cup P_k)\) contributes with at least \(l\) edges to the inequality.

I.e., \(|P - (P_1 \cup \ldots \cup P_k)| \geq l.\) So \(|P| = |P \cap (P_1 \cup \ldots \cup P_k)| + |P - (P_1 \cup \ldots \cup P_k)| \geq l + |P \cap (P_1 \cup \ldots \cup P_k)|\)

Now let us see that \(|P \cap (P_1 \cup \ldots \cup P_k)| \geq 1\)

Let \(v\) be a vertex shared by \(P\) and some \(P_i\).

As \(P_i\) is an augmenting path wrt. \(M\), there is an edge \(e \in P_i - M\) with endpoint \(v\).

So \(e \in M'\) and \(e\) is the only edge of \(M'\) with endpoint \(v\).

As \(v\) is matched in \(M'\), \(v \in P\) and \(P\) is an augmenting path wrt. \(M'\), there is a unique edge in \(P \cap M'\) with endpoint \(v\), which must be \(e\).

So we have that \(e \in P \cap P_i\), that \(|P \cap (P_1 \cup \ldots \cup P_k)| \geq 1\), and \(|P| > l\)
Complexity of Hopcroft-Karp

- We already know that each iteration takes $O(|E|)$ time.

- **Theorem.** Hopcroft-Karp runs in $O(\sqrt{|V||E|})$ time. (actually, in $O(\sqrt{\min(|L|,|R|)|E|})$ time)

- **Lema.** Hopcroft-Karp takes at most $2\sqrt{\min(|L|,|R|)}$ iterations.

- **Proof.** Wlog. let us assume that $|L| \leq |R|$.

  After $\sqrt{|L|}$ iterations:

  1. either the algorithm terminated because a maximum matching was found, or

  2. a matching $M$ was obtained for which the shortest augmenting path (wrt. $M$) has length $\geq 2\sqrt{|L|} + 1$
Complexity of Hopcroft-Karp

**Proof.** (contd.)

Assume 2. Let $M'$ be a maximum matching of $G$.

$M' \oplus M$ contains at least $|M'| - |M|$ vertex-disjoint augmenting paths with respect to $M$.

Each of those paths has length $\geq 2\sqrt{|L|} + 1$.

Since each vertex of $M' \oplus M$ has degree $\leq 2$, $M' \oplus M$ is a vertex-disjoint union of simple paths and cycles. As the graph $G$ is bipartite:

1. In a simple path $P \subseteq M' \oplus M$ of odd length, the number of vertices from $L$ is $(|P| + 1)/2$, which is $\geq |P|/2$.
2. In a simple path $P \subseteq M' \oplus M$ of even length, the number of vertices from $L$ is $|P|/2$ or $1+|P|/2$, which is $\geq |P|/2$.
3. In a cycle $C \subseteq M' \oplus M$ (thus, of even length, as it is alternating), the number of vertices from $L$ is $|C|/2$.

By vertex-disjointness, there are $\geq \frac{|M' \oplus M|}{2}$ vertices from $L$ in $M' \oplus M$.

So in $L$ there are at least $\frac{|M' \oplus M|}{2}$ different vertices.
Proof. (contd.)

Thus

\[ 2|L| \geq |M' \oplus M| \geq (|M'| - |M|)(2\sqrt{|L|} + 1) \]

So

\[ |M'| - |M| \leq \frac{2|L|}{2\sqrt{|L|}+1} = \frac{2\sqrt{|L|}\sqrt{|L|}}{2\sqrt{|L|}+1} \leq \frac{(2\sqrt{|L|}+1)\sqrt{|L|}}{2\sqrt{|L|}+1} = \sqrt{|L|}. \]

Hence, after another at most \( \sqrt{|L|} \) iterations, the algorithm is guaranteed to find a maximum matching.
Example

Bipartite graph $G = (L \cup R, E)$

Initially matching $M$ is empty.

Blue edges: $e \in M$

Red edges: $e \notin M$

Let us look for a maximal set of shortest augmenting paths using BFS.
Example

Mark vertices as matched \((m)\) or free \((f)\).

Start at all free vertices in \(L\).

Left \(\rightarrow\) right: red edges

Right \(\rightarrow\) left: blue edges
Example

Shortest augmenting path has length 1.

Found all free vertices in $R$ at distance 1.

Found maximal set of shortest aug. paths. (note that it is not maximum)
Example

Flip colors of augmenting paths and new $M$ is obtained
Example

Another maximal set of shortest augmenting paths?

Mark vertices as matched ($m$) or free ($f$).

Start at all free vertices in $L$.

Left $\rightarrow$ right: red edges

Right $\rightarrow$ left: blue edges
Example

Shortest augmenting path has length 3.

Found all free vertices in $R$ at distance 3.

Found maximal set of shortest aug. paths
Example

Flip colors of augmenting path and a new $M$ is obtained

No more augmenting paths, $M$ is a maximum matching
End Matching Theory
Arc Consistency for \texttt{alldiff}

[reminder]

Consider $x_1 \in \{1, 2\}$, $x_2 \in \{2, 3\}$, $x_3 \in \{2, 3\}$ and the constraint \texttt{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 $\text{alldiff}(x_1, \ldots, x_n)$ is the bipartite graph $G = (X \cup \bigcup_{i=1}^{n} D_i, E)$ where $(x_i, v) \in E$ iff $v \in D_i$

\[
\begin{align*}
\text{alldiff}(x_1, x_2, x_3) \\
D_1 &= \{1, 2\} \\
D_2 &= \{2, 3\} \\
D_3 &= \{2, 3\}
\end{align*}
\]
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 $\text{alldiff}(X) = \text{matchings covering } X$

\[
\text{alldiff}(x_1, x_2, x_3) = \begin{cases} 
D_1 = \{1, 2\} & x_1 = 1 \\
D_2 = \{2, 3\} & x_2 = 2 \\
D_3 = \{2, 3\} & x_3 = 3
\end{cases}
\]
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 $\text{alldiff}(X) = \text{matchings covering } X$

\[\text{alldiff}(x_1, x_2, x_3)\]

\[\begin{align*}
D_1 &= \{1, 2\} \quad x_1 = 1 \\
D_2 &= \{2, 3\} \quad x_2 = 3 \\
D_3 &= \{2, 3\} \quad x_3 = 2
\end{align*}\]
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 $\text{alldiff}(X) = \text{matchings covering } X$.

  \[
  \text{alldiff}(x_1, x_2, x_3)
  \]

  \[
  D_1 = \{1, 2\} \quad x_1 = 1
  \]

  \[
  D_2 = \{2, 3\} \quad x_2 = 3
  \]

  \[
  D_3 = \{2, 3\} \quad x_3 = 2
  \]

- A matching covering $X$ is a maximum matching.

- There are solutions to $\text{alldiff}(X)$ iff size of maximum matchings is $|X|$. 
Solutions and Matchings

Algorithm for checking feasibility of $\text{alldiff}(X)$:
(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 $\text{alldiff}(X)$
// G is the value graph of $\text{alldiff}(X)$
M = COMPUTE_MAXIMUM_MATCHING(G)
if ( |M| < |X| ) return false

return true
Solutions and Matchings

Algorithm for checking feasibility of $\text{alldiff}(X)$:
(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 $\text{alldiff}(X)$
// G is the value graph of $\text{alldiff}(X)$
M = COMPUTE_MAXIMUM_MATCHING(G)
if ( |M| < |X| ) return false
else REMOVE_EDGES_FROM_GRAPH(G, M)
return true

But in addition to check feasibility we want to find arc-inconsistent values

Assume $\text{alldiff}(X)$ has a solution. Then:
value $v$ from the domain of variable $x$ is arc-inconsistent iff
there is no solution to $\text{alldiff}(X)$ 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 the 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.
- **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. $M$.

![Diagram](image-url)
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.
- **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. $M$.
Filtering

Proof: \(\Leftarrow\) Let us consider all cases:

- If edge \(e\) is vital, then by definition it belongs to a maximum matching.

- If \(e\) belongs to an alternating cycle \(P\) wrt. maximum matching \(M\), then \(M\) and \(M \oplus P\) are maximum matchings, one contains \(e\) and the other does not.

- Similarly if \(e\) belongs to an even-length path starting at a free vertex that is alternating wrt. maximum matching \(M\).
Proof: \( \Rightarrow \) Let \( e \) be an edge that belongs to a maximum matching. Let us assume that \( e \) is not vital.

Two cases:

- Suppose \( e \in M \). Since \( e \) is not vital, there exists a maximum matching \( M' \) such that \( e \notin M' \). Then \( e \in M \oplus M' \).

But \( M \oplus M' \) is a vertex-disjoint union of:
Filtering

Proof: ⇒) Let $e$ be an edge that belongs to a maximum matching. Let us assume that $e$ is not vital.

Two cases:

- Suppose $e \in M$. Since $e$ is not vital, there exists a maximum matching $M'$ such that $e \notin M'$. Then $e \in M \oplus M'$.

But $M \oplus M'$ is a vertex-disjoint union of:

Recall that $M, M'$ are maximum matchings.
Filtering

Proof: \( \Rightarrow \) Let \( e \) be an edge that belongs to a maximum matching.
Let us assume that \( e \) is not vital.

Two cases:

- Suppose \( e \notin M \). Let \( M' \) be a maximum matching such that \( e \in M' \) (which exists by hypothesis). Then the same argument as before applies.
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
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**
  
  \[
  \begin{align*}
  d(w) &= \{b, c, d, e\}, \\
  d(x) &= \{b, c\}, \\
  d(y) &= \{a, b, c, d\}, \\
  d(z) &= \{b, c\}
  \end{align*}
  \]
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

- Remove unused edges that are not vital
- 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: \( O(k m \sqrt{k}) \)
  - Cost of constructing maximum matching: \( O(k m \sqrt{k}) \)
  - Cost of removing edges: \( O(k m) \)