Enforcing consistency

- The more explicit and tight constraint networks are, the more restricted is the search space of partial solutions.
- **Idea:** infer new constraints without “removing” (by methods called local consistency enforcing, bounded consistency inference, constraint propagation).
- Consistency-enforcing algorithms aim at assisting search: How can we extend a given partial solution of a small subnetwork to a partial solution of a larger subnetwork?
Some useful conventions

- In what follows we will always assume that the variables of a constraint network appear in some order.
- Further, we assume that $C$ does not contain unary constraints, i.e., constraints in $C$ are always relations with arity $n > 1$, but we allow that the domains $D_i$ are possibly empty. This is no restriction, since we can rewrite $D_i$:

$$D_i \leftarrow D_i \cap R_{v_i}$$

and then remove $R_{v_i}$ from the network.

$D_i$ will be referred to as domains, unary constraint, or domain constraint.

- We write constraints with scheme $(v_i, \ldots, v_j, \ldots, v_k)$ in the form $R_{i \ldots j \ldots k}$.

Arc consistency

Let $N = (V, D, C)$ be a constraint network.

Definition

(a) A variable $v_i$ is arc-consistent relative to variable $v_j$ if for each value $a_i \in D_i$, there exists an $a_j \in D_j$ with $(a_i, a_j) \in R_{ij}$ (in case that $R_{ij}$ exists in $C$).

(b) An “arc constraint” $R_{ij}$ is arc-consistent if $v_i$ is arc-consistent relative to $v_j$ and $v_j$ is arc-consistent rel. to $v_i$.

(c) A network $N$ is arc-consistent if all its arc constraints are arc-consistent.

Lemma

Checking whether a network $N = (V, D, C)$ is arc-consistent requires at most $e \cdot k^2$ operations (where $e$ is the number of its binary constraints and $k$ is an upper bound of its domain sizes).

Example

Consider a constraint network with two variables $v_1$ and $v_2$, domains $D_1 = D_2 = \{1, 2, 3\}$, and the binary constraint expressed by $v_1 < v_2$.

Revising a single domain

Revise $(v_i, v_j)$:

Input: a network with two variables $v_i, v_j$, domains $D_i$ and $D_j$, and constraint $R_{ij}$

Result: a network with refined $D_i$ such that $v_i$ is arc-consistent relative to $v_j$

for each $a_i \in D_i$

if there is no $a_j \in D_j$ with $(a_i, a_j) \in R_{ij}$

then remove $a_i$ from $D_i$

endif

endfor

This is equivalent to applying:

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$$
Revising a single domain

Lemma
The complexity of Revise is $O(k^2)$, where $k$ is an upper bound of the domain sizes.

Note: With a simple modification of the Revise algorithm one could improve to $O(t)$, where $t$ is the maximal number of tuples occurring in one of the binary constraints in the network.

Enforcing arc consistency: AC1

**AC1(N):**

*Input:* a constraint network $N = (V, D, C)$

*Result:* $N$ arc-consistent, but equivalent to input network

repeat
  for each arc $\{v_i, v_j\}$ with $R_{ij} \in C$
    Revise($v_i, v_j$)
    Revise($v_j, v_i$)
  endfor
until no domain is changed

Example: AC1

Consider a constraint network with three variables $v_1$, $v_2$, and $v_3$, domains $D_1 = D_2 = \{1, 2, 3\}$, and the binary constraints expressed by $v_1 < v_2$ and $v_2 < v_3$.

Note: Enforcing arc consistency may already be sufficient to show that a constraint network is inconsistent. For example, add the constraint $v_3 < v_1$ to the network just considered.
Enforcing arc consistency: AC3

Idea: no need to process all constraints if only a few domains have changed. Operate on a queue of constraints to be processed.

\[ \text{AC3}(N) \]

\textbf{Input:} a constraint network \( N = (V, D, C) \)
\textbf{Result:} an equivalent, but arc-consistent network

\textbf{for} each pair \( v_i, v_j \) that occurs in a constraint \( R_{ij} \) 
\textbf{queue} \( \leftarrow \) \textbf{queue} \( \cup \{ (v_i, v_j), (v_j, v_i) \} \)
\textbf{endfor}

\textbf{while} \textbf{queue} is not empty
\textbf{select and remove} \( (v_i, v_j) \) from \textbf{queue}
\text{Revise} (\( v_i, v_j \))
\textbf{if} \text{Revise} (\( v_i, v_j \)) changes \( D_i \)
\textbf{then} \textbf{queue} \( \leftarrow \) \textbf{queue} \( \cup \{ (v_k, v_i) : k \neq i, k \neq j \} \)
\textbf{endif}
\textbf{endwhile}

Lemma

Let \( N \) be a constraint network with \( n \) variables, each with a domain of size \( \leq k \), and \( e \) binary constraints.
Applying AC3 on the network runs in time \( O(e \cdot k^3) \).

Proof.

Consider a single constraint. Each time, when it is reintroduced into the queue, the domain of one of its variables must have been changed. Since there are at most \( 2 \cdot k \) values, AC3 processes each constraint at most \( 2 \cdot k \) times. Because we have \( e \) constraints and processing of each is in time \( O(k^2) \), we obtain \( O(e \cdot k^3) \).

Note: If the input network is arc-consistent, then AC3 runs in time \( O(e \cdot k^2) \).

Example: Consider a constraint network with 3 variables \( v_1, v_2, v_3 \) with domains \( D_1 = \{2, 4\} \) and \( D_2 = D_3 = \{2, 5\} \), and two constraints expressed by \( v_3 | v_1 \) and \( v_3 | v_2 \) (“divides”).

\[ \text{Queue} \]

\( (v_1, v_3) \)
\( (v_3, v_1) \)
\( (v_2, v_3) \)
\( (v_3, v_2) \)

\[ v_1 2,4 \]
\[ v_2 2,5 \]
\[ v_3 2,5 \]
Example: AC4

The initialization steps yield:

\[
\begin{align*}
S[v_3, 2] &= \{(v_1, 2), (v_1, 4), (v_2, 2)\} & S[v_3, 5] &= \{(v_2, 5)\} \\
S[v_2, 2] &= \{(v_3, 2)\} & S[v_2, 5] &= \{(v_3, 5)\} \\
S[v_1, 2] &= \{(v_3, 2)\} & S[v_1, 4] &= \{(v_3, 2)\}
\end{align*}
\]

Furthermore:

\[
counter(v_3, 2, v_1) = 2 \quad \text{and} \quad counter(v_3, 5, v_1) = 0.
\]

All other counters are 1 (note: we only need consider counters between connected variables).

\[
Q = \{(v_3, 5)\} \quad \text{and} \quad D_3 = \{2\}.
\]

When \((v_3, 5)\) is selected (and removed) from \(Q\), we obtain \(counter(v_2, 5, v_3) = 0\). \((v_2, 5)\) is added to \(Q\) and 5 deleted from \(D_2\). Then \((v_2, 5)\) is selected from \(Q\). \((v_2, 5)\) has only support for \((v_3, 5)\), but 5 has already been removed from \(D_3\). . .

Example: AC4

Consider the same network as for AC3.
Constraints: \(v_3|v_1\) and \(v_3|v_2\).

\[\begin{tikzpicture}
  \node (v1) at (0,0) {$v_1$};
  \node (v2) at (1,0) {$v_2$};
  \node (v3) at (0.5,1) {$v_3$};
  \draw (v1) -- (v2) node [midway, above] {2,4};
  \draw (v1) -- (v3) node [midway, below] {2,5};
  \draw (v3) -- (v2) node [midway, above] {2,5};
\end{tikzpicture}\]

The initialization steps yield:

\[
\begin{align*}
S[v_3, 2] &= \{(v_1, 2), (v_1, 4), (v_2, 2)\} & S[v_3, 5] &= \{(v_2, 5)\} \\
S[v_2, 2] &= \{(v_3, 2)\} & S[v_2, 5] &= \{(v_3, 5)\} \\
S[v_1, 2] &= \{(v_3, 2)\} & S[v_1, 4] &= \{(v_3, 2)\}
\end{align*}
\]

\[\text{Enforcing arc consistency: AC2001}\]

- **Fine-grained** algorithms (like AC4) directly propagate the removal of a value \((v_i, a_i)\) to values \((v_j, a_j)\) which were supported by \((v_i, a_i)\)
- \text{... while coarse-grained} algorithms (like AC3) propagate changes on the level of the domains only
- Nevertheless coarse-grained algorithms have advantages: no need for additional data structures \(S[v_j, a_j]\) (costs for initialization and maintenance)
- **AC2001** is a coarse-grained method: works like AC3, but with a different revise function: achieves optimal run time \(O(e \cdot k^2)\).
Revise2001 in AC2001

- Assume orderings on each of the domains (use dummy value nil smaller than all domain values)
- AC2001 first initializes and maintains pointers $Last(v_i, a_i, v_j) \leftarrow nil$

```
Revise2001(v_i, v_j):
Input: a network with two variables $v_i, v_j$, domains $D_i$ and $D_j$, and constraint $R_{ij}$
Result: a network with a refined domain $D_i$ for each $a_i \in D_i$ with $Last(v_i, a_i, v_j) \notin D_j$
for each $a_i \in D_i$ with $Last(v_i, a_i, v_j) \notin D_j$
    $a_i \leftarrow$ the smallest value $a$ in $D_i$ with $a > Last(v_i, a_i, v_j)$ and $(a, a) \in R_{ij}$
    if $a_j$ exists then
        $Last(v_i, a_i, v_j) \leftarrow a_j$
    else
        remove $a_i$ from $D_i$
```
**Path Consistency**

**An example**

![Diagram](image)

Figure: This network is arc-consistent, but not path-consistent.

---

**Revising a path**

- **Revise-3**$(\{v_i, v_j\}, v_k)$:
  - **Input:** a binary network $\langle V, D, C \rangle$ with variables $v_i, v_j, v_k$
  - **Result:** a revised constraint $R_{ij}$ path-consistent with $v_k$
  - **Procedure:***
    - for each pair $(a_i, a_j) \in R_{ij}$
      - if there is no $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_j, a_k) \in R_{jk}$
        - then remove $(a_i, a_j)$ from $R_{ij}$
    - endfor

This is equivalent to applying:

$$R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{kj})$$

---

**Revising a path: Properties**

**Lemma**
When applied to a constraint network $N$, procedure Revise-3$(\{v_i, v_j\}, v_k)$:
- does not do anything if the pair $v_i, v_j$ is path-consistent relative to $v_k$, and otherwise
- transforms the network into an equivalent form where the pair $v_i, v_j$ is path-consistent relative to $v_k$.

**Proof.**
From the definition of path consistency.

---

**Revising a path: Complexity**

**Lemma**
Let $t$ be the maximal number of tuples in one of the binary constraints, and let $k$ be an upper bound for the domain sizes.

- The worst-case runtime of Revise-3 is $O(t \cdot k)$.
- The best-case runtime of Revise-3 is $\Omega(t)$.

With respect to $k$, the complexity of Revise-3 can also be expressed as $O(k^3)$ in the worst and $\Omega(k^2)$ in the best case.
Enforcing path consistency: PC1

**PC1(N):**

*Input:* a constraint network \( N = (V, D, C) \)

*Result:* an equivalent, path-consistent network

repeat
  for each (ordered) triple of variables \( v_i, v_j, v_k \):
    Revise-3(\{v_i, v_j\}, v_k)
  endfor
until no constraint is changed

Enforcing path consistency: Soundness of PC1

**Lemma**

When applied to a constraint network \( N \), the PC1 algorithm computes a path-consistent constraint network which is equivalent to \( N \).

**Proof.**

Follows directly from the properties of Revise-3.

Enforcing path consistency: Complexity of PC1

**Lemma**

Let \( N \) be a constraint network with \( n \) variables, each with a domain of size \( \leq k \). Let \( t \) be an upper bound of the number of tuples in one of the binary constraints in \( C \).

The worst-case runtime of PC1 on this network is \( O(n^5 \cdot t^2 \cdot k) \).

The best-case runtime of PC1 on this network is \( \Omega(n^3 \cdot t) \).

The runtime bounds can also be stated as \( O(n^5 \cdot k^3) \) and \( \Omega(n^3 \cdot k^2) \), respectively.

Enforcing path consistency: Complexity of PC1

**Proof (worst case).**

In each iteration of the outer loop in PC1, only one value pair might be removed from one of the constraints. Hence the number of iterations may be as large as \( O(n^2 \cdot t) \).

Processing a specific triple of constraints (there are \( O(n^3) \) many such triples) costs \( O(t \cdot k) \).

Hence each iteration costs \( O(n^3 \cdot t \cdot k) \).

**Proof (best case).**

In the best case, the network is already path-consistent and only one iteration through the outer loop is needed. There are \( \Omega(n^3) \) calls to Revise-3, each requiring time \( \Omega(t) \) in the best case.
Enforcing path consistency: PC2

**PC2(N):**

*Input:* a constraint network \( N = (V, D, C) \)

*Result:* an equivalent, path-consistent network \( N' \)

queue ← \{ (i, k, j) : 1 ≤ i < j ≤ n, 1 ≤ k ≤ n, k ≠ i, k ≠ j \}

while queue is not empty

select and remove a triple \((i, k, j)\) from queue

Revise-3\({(v_i, v_j, v_k)}\)

if \( R_{ij} \) has changed then

queue ← queue ∪ \{ (l, i, j), (l, j, i) : 1 ≤ l ≤ n, l ≠ i, j \}

endif

endwhile

Enforcing path consistency: Soundness of PC2

**Lemma**

When applied to a constraint network \( N \), the PC2 algorithm computes a path-consistent constraint network which is equivalent to \( N \).

**Proof.**

Equivalence follows directly from the properties of Revise-3.

To see that the remaining constraint network is path-consistent, verify the following invariant:

*Before and after each iteration of the while-loop, for each pair \( v_i, v_j \) which is not path-consistent relative to \( v_k \), one of the triples \((i, k, j)\) and \((j, k, i)\) is contained in the queue.*

Enforcing path consistency: Complexity of PC2

**Lemma**

Let \( N \) be a constraint network with \( n \) variables, each with a domain of size \( ≤ k \). Let \( t \) be an upper bound of the number of tuples in one of the binary constraints in \( N \).

The worst-case runtime of PC2 on this network is \( O(n^3 \cdot t^2 \cdot k) \).

The best-case runtime of PC2 on this network is \( \Omega(n^3 \cdot t) \).

Because of \( t ≤ k^2 \), the runtime bounds can also be stated as \( O(n^3 \cdot k^3) \) and \( \Omega(n^3 \cdot k^2) \), respectively.

**Proof (worst case).**

There are initially \( O(n^3) \) elements in the queue. Whenever some constraint \( R_{ij} \) is reduced, which can happen at most \( O(n^2 \cdot t) \) many times, \( O(n) \) elements are added to the queue. Thus, the total number of elements added to the queue is bounded by \( O(n^3 \cdot t) \).

Each iteration of the while loop removes an element from the queue, so there are at most \( O(n^3 \cdot t) \) iterations and hence at most \( O(n^3 \cdot t) \) calls to Revise-3, each requiring time \( O(t \cdot k) \), for a total runtime bound of \( O(n^3 \cdot t^2 \cdot k) \).

**Proof (best case).**

Similar to PC1.
Arc and path consistency: Overview

<table>
<thead>
<tr>
<th></th>
<th>Worst Case</th>
<th>Best Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC1</td>
<td>$O(n \cdot k \cdot e \cdot t)$</td>
<td>$\Omega(e \cdot k)$</td>
</tr>
<tr>
<td>AC3</td>
<td>$O(e \cdot k \cdot t)$</td>
<td>$\Omega(e \cdot k)$</td>
</tr>
<tr>
<td>AC4</td>
<td>$O(e \cdot k^2)$</td>
<td>$\Omega(e \cdot k^2)$</td>
</tr>
<tr>
<td>PC1</td>
<td>$O(n^5 \cdot t^2 \cdot k)$</td>
<td>$\Omega(n^3 \cdot t^3)$</td>
</tr>
<tr>
<td>PC2</td>
<td>$O(n^3 \cdot t^2 \cdot k)$</td>
<td>$\Omega(n^3 \cdot t)$</td>
</tr>
<tr>
<td>PC4*</td>
<td>$O(n^3 \cdot t \cdot k)$</td>
<td>$\Omega(n^3 \cdot t \cdot k)$</td>
</tr>
</tbody>
</table>

Remark: $O(n^3 \cdot t \cdot k)$ is the optimal (worst-case) runtime for enforcing path consistency, i.e., there are (arbitrarily large) constraint networks for which no better algorithm exists.

Higher levels of $i$-consistency

The local consistency notions presented so far can be roughly summarized as follows:

- **Arc consistency**: Every consistent assignment to a single variable can be consistently extended to any second variable.
- **Path consistency**: Every consistent assignment to two variables can be consistently extended to any third variable.

(Side remark: This is a bit of an oversimplification because we ignored $k$-ary constraints with $k \geq 3$ so far.)

It is easy to see that the general idea of local consistency can be readily extended to larger variable sets.

3 Higher Levels of $i$-Consistency

**$i$-Consistency**

Let $N = \langle V, D, C \rangle$ be a constraint network.

**Definition**

(a) A relation $R_S \in C$ with scope $S$ of size $i - 1$ is $i$-consistent relative to variable $v_i \notin S$ if for every tuple $t \in R_S$, there exists an $a \in D_i$ such that $(t, a)$ is consistent.

(b) A constraint network is $i$-consistent if any consistent instantiation of $i - 1$ (distinct) variables $v_1, \ldots, v_{i-1}$ of the network can be extended to a consistent instantiation of the variables $v_1, \ldots, v_i$, where $v_i$ is any variable in $V$ distinct from $v_1, \ldots, v_{i-1}$.
Global consistency

Definition

- A network $N$ is strongly $i$-consistent if it is $j$-consistent for each $j \leq i$.
- A network $N$ with $n$ variables is globally consistent if it is strongly $n$-consistent.

Note: Solutions to globally consistent networks can be found without search. (How?)

Arc/path consistency vs. 2/3-consistency

Note:

- 2-consistency coincides with arc consistency.
- For networks containing binary constraints only, 3-consistency coincides with path consistency.
- Each 3-consistent network is path-consistent.
- The converse is not true: For networks with constraints of arity $\geq 3$, 3-consistency is stricter than path consistency.

3-Consistency: Examples

Example

$V = \{v_1, v_2, v_3\}$
$D_1 = D_2 = D_3 = \{0, 1\}$
$R_{123} = \{(0, 0, 0)\}$

Example

$V = \{v_1, v_2, v_3\}$
$D_1 = D_2 = D_3 = \{0, 1\}$
$R_{123} = \{(0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0)\}$
$R_{12} = R_{13} = R_{23} = \{(0, 1), (1, 0), (1, 1)\}$

Revise-$i$:

$\text{Revise-}i(\{v_1, \ldots, v_{i-1}\}, v_i)$:

Input: a network $(V, D, C)$ and a constraint $R_S$ with scope $S = \{v_1, \ldots, v_{i-1}\}$

Result: a constraint $R_S$ which is $i$-consistent rel. to $v_i$

for each instantiation $\overline{a}_{i-1} \in R_S$

if there is no $a_i \in D_i$ such that $(\overline{a}_{i-1}, a_i)$ is consistent

then remove $\overline{a}_{i-1}$ from $R_S$

endif

endfor

- $R_S$ can be the universal relation wrt. $S$.
- If the input network is binary, then Revise-$i$ runs in time $O(k^i)$.
- In general, Revise-$i$ runs in time $O((2 \cdot k)^i)$, since $O(2^i)$ constraints must be processed for each tuple.
i-Consistency: Algorithm

\textbf{Enforce }i\text{-Consistency} (\(N\)): \\
\textbf{Input:} a constraint network \(N = \langle V, D, C \rangle\). \\
\textbf{Result:} an \(i\)-consistent network equivalent to \(N\).

repeat \\
\hspace{1em} for each subset of \(S \subseteq V\) of size \(i - 1\) and each \(v_i \notin S\) \\
\hspace{2em} Revise-\(i\)\(\{v_1, \ldots, v_{i-1}\}, v_i\) \\
endfor \\
until no constraint is changed

The Revise-\(i\) call can equivalently be stated as follows:
Let \(S\) be the set of all subsets of \(\{v_1, \ldots, v_i\}\) that contain \(v_i\) and occur as scopes of some constraint in the network. Then apply \\
\[R_S \leftarrow R_S \cap \pi_S(\bigwedge_{S' \in S} R_{S'}).\]

Remark: \(O(n^i \cdot k^i)\) is the optimal (worst-case) runtime for enforcing \(i\)-consistency, i.e., there are (arbitrarily large) constraint networks for which no better algorithm exists.

\[\text{Worst Case} \quad \begin{array}{ll}
\text{AC1} & O(n \cdot k \cdot e \cdot t) = O(n^3 \cdot k^3) \\
\text{AC3} & O(e \cdot k \cdot t) = O(n^2 \cdot k^3) \\
\text{AC4} & O(n^2 \cdot k^2) \\
\text{improved }i\text{-consistency}^*, i = 2 & O(n^2 \cdot k^2) \\
\text{PC1} & O(n^5 \cdot t^2 \cdot k) = O(n^5 \cdot k^5) \\
\text{PC2} & O(n^3 \cdot t^2 \cdot k) = O(n^3 \cdot k^5) \\
\text{PC4}^* & O(n^3 \cdot k^3) \\
\text{improved }i\text{-consistency}^*, i = 3 & O(n^3 \cdot k^3) \\
\end{array}\]

\(^*\text{not discussed in this lecture}\]
Extensions of Arc consistency

▶ General i-consistency is powerful, but expensive to enforce.
▶ Usually, arc consistency and path consistency offer a good compromise between pruning power and computational overhead.
▶ However, they are of limited usefulness for constraints on more than two variables.

Example
Consider a constraint network with three integer variables $v_1, v_2, v_3 \geq 0$ and the constraints $v_3 \geq 13$ and $v_1 + v_2 + v_3 \leq 15$. We should be able to infer $v_1 \leq 2$ and $v_2 \leq 2$, but regular arc consistency is not enough!

Generalized arc consistency: Update rule

To enforce generalized arc consistency, repeatedly apply

$$D_i \leftarrow D_i \cap \pi_i(R \triangledown S \{v_i\})$$

Note how this generalizes the usual arc consistency update rule:

$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \triangledown D_j)$$

Alternatives to generalized arc consistency

▶ Like arc consistency, generalized arc consistency propagates constraints by considering a single constraint at a time.
▶ In particular, it considers how assignments to each individual variable are restricted by the values allowed for the other variables participating in the constraint.
▶ Alternatively, we can consider how each individual variable restricts the values allowed for the other variables participating in the constraint:

$$R_{S \{v_i\}} \leftarrow R_{S \{v_i\}} \cap \pi_{S \{v_i\}}(R \triangledown D_i)$$

(relational arc consistency)

Note that in the case of binary constraints, these two cases are the same, so both approaches are natural generalizations of (binary) arc consistency.
Generalizations of arc consistency: Comparison

\[
\begin{align*}
\text{AC: } & D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j) \\
\text{generalized AC: } & D_i \leftarrow D_i \cap \pi_i(R_S \bowtie D_{S \setminus \{v_i\}}) \\
\text{relational AC: } & R_{S \setminus \{v_i\}} \leftarrow R_{S \setminus \{v_i\}} \cap \pi_{S \setminus \{v_i\}}(R_S \bowtie D_i)
\end{align*}
\]

Example
Consider a constraint network with three integer variables \(v_1, v_2, v_3 \geq 0\) and the constraints \(v_3 \geq 13\) and \(v_1 + v_2 + v_3 \leq 15\).

- Generalized AC infers \(v_1 \leq 2, v_2 \leq 2\).
- Relational AC infers \(v_1 + v_2 \leq 2\).