Enforcing Consistency

- The more explicit and tight constraint networks are, the more restricted is the search space of partial solutions.
- **Idea:** infer at least a limited number of new constraints (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?

Arc Consistency

Convention

In what follows we will always assume that the variables of a constraint network appear in some order. Then we can write constraint networks in the form:

$$C = \langle V, D, C \rangle,$$

where $D_i$ is the (possibly empty) domain of variable $v_i$, and constraints in the form $R_{ijk}$, where $\{v_i, v_j, v_k\}$ is the scope of the relation.

Further, we assume that $C$ does not contain unary constraints, i.e., constraints in $C$ are always relations with arity $n > 1$. This is possible, since we can define:

$$D_i := \text{dom}(v_i) \cap R_{v_i}$$

and then delete $R_{v_i}$ from the original network. $D_i$ will be referred to as domains, unary constraint, or domain constraint.
Arc Consistency

Let \( C = \langle V, D, C \rangle \) be a constraint network.

Definition

(a) A variable \( v_i \) is arc-consistent relative to variable \( v_j \) if for every 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 relative to \( v_i \).

(c) A network \( C \) is arc-consistent if all its arc constraints are arc-consistent.

Lemma

Checking whether a network \( C = \langle V, D, C \rangle \) is arc-consistent requires \( e \cdot k^2 \) 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

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: AC-1

AC-1(C):

Input: a constraint network \( C = (V, D, C) \)
Output: an equivalent, but arc-consistent network \( C' \)
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

Lemma
Let \( C \) be a constraint network with \( n \) variables, each with a domain of size \( \leq k \), and \( e \) binary constraints.
Applying AC-1 on the network runs in time \( O(e \cdot n \cdot k^3) \).

Proof.
One cycle through all binary constraints takes \( O(e \cdot k^2) \). In the worst case, one cycle just removes one value from one domain. Moreover, there are at most \( n \cdot k \) values. This result in an upper bound of \( O(e \cdot n \cdot k^3) \).

Note: If the input network is already arc-consistent, then AC-1 runs in time \( O(e \cdot k^2) \).

Example: AC-1
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 \( v_2 < v_3 \).

AC-3(C):

Input: a constraint network \( C = (V, D, C) \)
Output: an equivalent, but arc-consistent network \( C' \)
for each pair \( v_i, v_j \) that occurs in a constraint \( R_{ij} \)
  queue ← queue \( \cup \{ (v_i, v_j), (v_j, v_i) \} \)
endfor
while queue is not empty
  select and delete \( (v_i, v_j) \) from queue
  Revise\( (v_i, v_j) \)
  if Revise\( (v_i, v_j) \) changes \( D_i \)
    then queue ← queue \( \cup \{ (v_k, v_i) : k \neq i, k \neq j \} \)
  endif
endwhile

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: AC-3

Lemma
Let $C$ be a constraint network with $n$ variables, each with a domain of size $\leq k$, and $e$ binary constraints.
Applying AC-3 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, AC-3 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 AC-3 runs in time $O(e \cdot k^2)$.

Enforcing Arc Consistency: AC-4

To verify that a network is arc-consistent needs $e \cdot k^2$ operations.
The following algorithm AC-4 achieves optimal performance, ... at the cost of "best case performance", which is $\Omega(e \cdot k^2)$.

Idea:
- Associate to each value $a_i$ in the domain of variable $v_i$ the amount of support from variable $v_j$ (i.e., the number of values in $D_j$ that are consistent with $a_i$);
- Delete a value $a_i$ if it has no support from any other variable

Details:
- $List$: currently unsupported variable-value pairs;
- $counter(x_i, a_i, x_j)$: support for $a_i$ from $x_j$;
- $S_{x_i, a_i}$: array pointing to all values in other variables supported by $(x_j, a_i)$;
- $M$: list of removed values.

AC-4($C$):

Input: a constraint network $C = \langle V, D, C \rangle$
Output: an equivalent, but arc-consistent network $C'$

$M \leftarrow \emptyset$
initialize $S_{x_i, a_i}$ and $counter(x_i, a_i, x_j)$ for all $R_{ij}$
for each counter
    if $counter(x_i, a_i, x_j) = 0$
        then add $(x_i, a_i)$ to $List$
    endif
endfor
while $List$ is not empty
    choose and remove $(x_i, a_i)$ from $List$, and add it to $M$
    for each $(x_j, a_j)$ in $S_{x_i, a_i}$
        decrement $counter(x_j, a_j, x_i)$
        if $counter(x_j, a_j, x_i) = 0$
            then add $(x_j, a_j)$ to $List$
        endif
    endfor
endwhile
Example: AC-4

Consider the same network as for AC-3.

Constraints: \( v_3 \mid v_1 \) and \( v_3 \mid v_2 \).

The initialization steps yield:

\[
\begin{align*}
S_{v_1,2} &= \{(v_1, 2) \} & S_{v_5,5} &= \{(v_2, 5)\} \\
S_{v_2,2} &= \{(v_3, 2)\} & S_{v_5,2} &= \{(v_3, 5)\} \\
S_{v_1,2} &= \{(v_3, 2)\} & S_{v_1,4} &= \{(v_3, 2)\}
\end{align*}
\]

Furthermore:

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

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

\( List = \{(v_3, 5)\} \) and \( M = \emptyset \).

When \( (v_3, 5) \) is removed from \( List \) and added to \( M \), we obtain 
\( \text{counter}(v_2, 5, v_3) = 0 \) and add \( (v_2, 5) \) to \( List \). Then \( (v_2, 5) \) is removed from \( List \) and added to \( M \). \( (v_2, 5) \) is only supported by \( (v_3, 5) \), but that pair is already in \( M \), and we are done.

Beyond Arc Consistency

- Sometimes “enforcing arc consistency” is sufficient for detecting inconsistent (unsolvable) networks; but . . .
- enforcing arc consistency is not complete for deciding consistency of networks; because . . .
- inferences rely only on domain constraints and single binary constraints defined on the domains.

\( \Rightarrow \) We consider further concepts of local consistency
An Example

\[ \begin{align*}
\text{red} & \quad \text{blue} \\
\begin{tikzpicture}[node distance=1.5cm, thick, main/.style = {draw}]
  \node (v1) [main] {v_1};
  \node (v2) [main] at (1,0) {v_2};
  \node (v3) [main] at (2,0) {v_3};

  \path
    (v1) edge (v2)
    (v1) edge (v3)
    (v2) edge (v3);
\end{tikzpicture}
\end{align*} \]

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

Revising a Path

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

\textbf{Input:} a binary network \((V, D, C)\) with variables \(v_i, v_j, v_k\)

\textbf{Output:} a revised constraint \(R_{ij}\) path-consistent with \(v_k\)

\textbf{for} each pair \((a_i, a_j) \in R_{ij}\)

\textbf{if} there is no \(a_k \in D_k\) such that \((a_i, a_k) \in R_{ik}\)

\textbf{and} \((a_j, a_k) \in R_{jk}\)

\textbf{then} delete \((a_i, a_j)\) from \(R_{ij}\)

\textbf{endif}

\textbf{endfor}

This is equivalent to applying:

\[ R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{ik} \bowtie D_k \bowtie R_{jk}) \]

Revising a Path: Properties

Lemma

When applied to a constraint network \(C\), 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 \(\Omega(t)\).

Note that \(t \leq k^2\), so the complexity of Revise-3 can also be expressed as \(O(k^3)\) in the worst and \(\Omega(k^2)\) in the best case.
Path Consistency

Enforcing Path Consistency: PC-1

**PC-1(C):**

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

*Output:* an equivalent, path-consistent network \( C' \)

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 PC-1

**Lemma**

When applied to a constraint network \( C \), the PC-1 algorithm computes a path-consistent constraint network which is equivalent to \( C \).

**Proof.**

Follows directly from the properties of Revise-3.

Enforcing Path Consistency: Complexity of PC-1

**Lemma**

Let \( C \) 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 PC-1 on this network is \( O(n^5 \cdot t^2 \cdot k) \).

The best-case runtime of PC-1 on this network is \( \Omega(n^3 \cdot t) \).

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

Enforcing Path Consistency: Complexity of PC-1

**Proof (worst case).**

In each iteration of the outer loop in PC-1, only one value pair might be deleted 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: PC-2

**PC-2(\(C\)):**

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

*Output:* an equivalent, path-consistent network \(C'\)

\[
\text{queue} \leftarrow \{(i, k, j) : 1 \leq i < j \leq n, 1 \leq k \leq n, k \neq i, k \neq j\}
\]

**while** queue is not empty

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

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

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

**queue** \leftarrow queue \(\cup\) \{(l, i, j) : 1 \leq l \leq n, l \neq i, l \neq j\}

**endif**

**endwhile**

---

Enforcing Path Consistency: Soundness of PC-2

**Lemma**

When applied to a constraint network \(C\), the PC-2 algorithm computes a path-consistent constraint network which is equivalent to \(C\).

**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 PC-2

**Lemma**

Let \(C\) 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 PC-2 on this network is \(O(n^3 \cdot t^2 \cdot k)\).

The best-case runtime of PC-2 on this network is \(\Omega(n^3 \cdot t)\).

Because \(t \leq k^2\), the runtime bounds can also be stated as \(O(n^3 \cdot k^5)\) 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 PC-1.
### 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. More on this later.)

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

### Global Consistency

**Definition**

- A network $C$ is **strongly $i$-consistent** if it is $j$-consistent for each $j \leq i$.
- A network $C$ 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-\(i\)

\[ \text{Revise-}i(\{v_1, \ldots, v_{i-1}\}, v_i): \]
\[ \text{Input:} \text{ a network } \langle V, D, C \rangle \text{ and a constraint } R_S \]
\[ \text{with scope } S = \{v_1, \ldots, v_{i-1}\} \]
\[ \text{Output:} \text{ a constraint } R_S \text{ which is } i\text{-consistent rel. to } v_i \]
\[
\text{for each instantiation } \bar{a}_{i-1} \in R_S \\
\text{if there is no } a_i \in D_i \text{ such that } (\bar{a}_{i-1}, a_i) \text{ is consistent} \\
\text{then delete } \bar{a}_{i-1} \text{ from } R_S 
\]

\[ R_S \text{ can be the universal relation wrt. } S. \]
- 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.

3-Consistency: Algorithm

\[ \text{Enforce } i\text{-Consistency}(C): \]
\[ \text{Input:} \text{ A constraint network } C = \langle V, D, C \rangle. \]
\[ \text{Output:} \text{ An } i\text{-consistent network equivalent to } C. \]
\[
\text{repeat} \\
\text{for each subset of } S \subseteq V \text{ of size } i - 1 \text{ and each } v_i \notin S \\
\text{Revise-}i(\{v_1, \ldots, v_{i-1}\}, v_i) \\
\text{endfor} \\
\text{until no constraint is changed} 
\]

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
\[ R_S \leftarrow R_S \cap \pi_S(\exists x \in S R_S'). \]
**i-Consistency: Complexity**

**Lemma**

Let $C$ be a constraint network with $n$ variables, each with a domain of size $\leq k$. When applied to $C$, the “Enforce i-Consistency” algorithm runs in time $O(2^i \cdot (n \cdot k)^{2^{i-1}})$.

**Proof.**

Each call to Revise-$i$ requires time $O((2 \cdot k)^i)$. In each iteration of the outer loop, $O(n^i)$ combinations of $S$ and $v_i$ need to be processed. If only one tuple is removed from one constraint in each iteration up to the final one, the outer loop may need to iterate $O(n^{i-1} \cdot k^{i-1})$ times. This leads to an overall runtime of $O(2^i \cdot (n \cdot k)^{2^{i-1}})$. □

Note: Improvements similar to AC-4 and PC-4 exist and achieve a worst-case runtime of $O(n^i \cdot k^i)$.

**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!

Consider generalizations of arc consistency to non-binary constraints.

**i-Consistency: Comparison to AC-x and PC-x**

<table>
<thead>
<tr>
<th>$i$-consistency, $i = 2$</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC-1</td>
<td>$O(n \cdot k \cdot e \cdot t) = O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>AC-3</td>
<td>$O(e \cdot k \cdot t) = O(n^2 \cdot k^3)$</td>
</tr>
<tr>
<td>AC-4</td>
<td>$O(e \cdot t) = O(n^2 \cdot k^2)$</td>
</tr>
<tr>
<td>improved $i$-consistency*, $i = 2$</td>
<td>$O(n^2 \cdot k^2)$</td>
</tr>
<tr>
<td>$i$-consistency, $i = 3$</td>
<td>$O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>PC-1</td>
<td>$O(n^5 \cdot t^2 \cdot k) = O(n^5 \cdot k^3)$</td>
</tr>
<tr>
<td>PC-2</td>
<td>$O(n^3 \cdot t^2 \cdot k) = O(n^3 \cdot k^5)$</td>
</tr>
<tr>
<td>PC-4*</td>
<td>$O(n^3 \cdot t \cdot k) = O(n^3 \cdot k^3)$</td>
</tr>
<tr>
<td>improved $i$-consistency*, $i = 3$</td>
<td>$O(n^3 \cdot k^3)$</td>
</tr>
</tbody>
</table>

*not discussed in this lecture

**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.

**Generalized Arc Consistency**

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

**Definition**

(a) A variable $v_i$ is (generalized) arc-consistent relative to a constraint $R \in C$ whose scope contains $v_i$ if for every value $a_i \in D_i$ there exists a tuple $\bar{a} \in R$ with $\bar{a}_i = a_i$.

(b) A constraint $R \in C$ is (generalized) arc-consistent iff all variables in its scope are generalized arc-consistent relative to $R$.

(c) A network $C$ is (generalized) arc-consistent if all its constraints are generalized arc-consistent.
Generalized Arc Consistency: Update Rule

To enforce generalized arc consistency, repeatedly apply

\[ D_i \leftarrow D_i \cap \pi_i(R_S \triangleright D_{S \setminus \{v_i\}}) \]

Note how this generalizes the usual arc consistency update rule:

\[ D_i \leftarrow D_i \cap \pi_i(R_{ij} \triangleright D_j) \]

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:

\[ R_S\setminus\{v_i\} \leftarrow R_S\setminus\{v_i\} \cap \pi_S\setminus\{v_i\}(R_S \triangleright D_i) \]

(relation 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

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

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 \(v_1 + v_2 \leq 2\).

Literature