Constraint Satisfaction Problems

Enforcing Consistency

Bernhard Nebel, **Julien Hué**, and **Stefan Wölfl**

Albert-Ludwigs-Universität Freiburg

May 9/21/23, 2012

Constraint Satisfaction Problems

May 9/21/23, 2012 — Enforcing Consistency

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?

1 Arc Consistency

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

Definition

(a) A variable v_i is arc-consistent relative to variable v_j if for 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 $N = \langle V, D, C\rangle$ 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 $v_1 < v_2$.

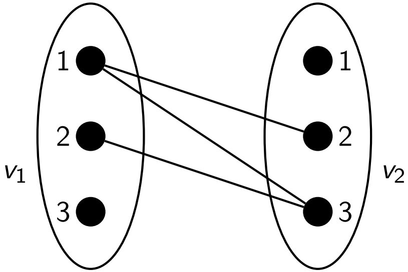


Figure: A network that is not arc-consistent

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:

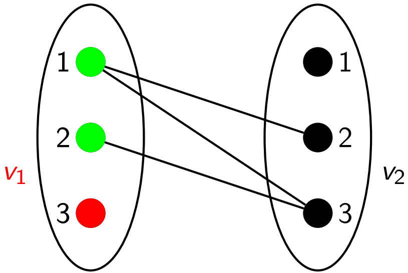
$$D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$$

Revising a single domain

Lemma

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

Note: With a simple modification of the Revise algorithm one could improve to $\mathcal{O}(t)$, 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 $N = \langle V, D, C \rangle$
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

Enforcing arc consistency: AC1

Lemma

Let N be a constraint network with n variables, each with a domain of size $\leq k$, and e binary constraints.
Applying AC1 on the network runs in time $\mathcal{O}(e \cdot n \cdot k^3)$.

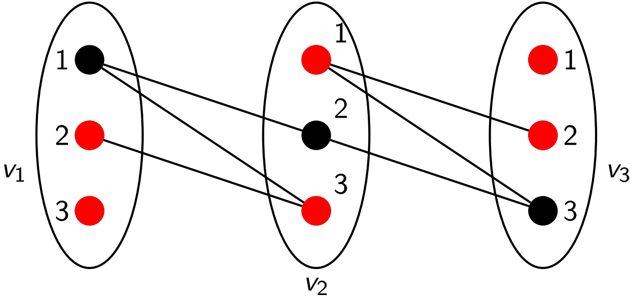
Proof.

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

Note: If the input network is already arc-consistent, then AC1 runs in time $\mathcal{O}(e \cdot k^2)$.

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.

AC3(N):

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

for each pair v_i, v_j that occurs in a constraint R_{ij}
    $queue \leftarrow queue \cup \{(v_i, v_j), (v_j, v_i)\}$
endfor
while *queue* is not empty
    select and remove (v_i, v_j) from *queue*
    Revise(v_i, v_j)
    if Revise(v_i, v_j) changes D_i
        then $queue \leftarrow queue \cup \{(v_k, v_i) \, : \, k \neq i, k \neq j\}$
    endif
endwhile

Enforcing arc consistency: AC3

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 $\mathcal{O}(e \cdot k^3)$.

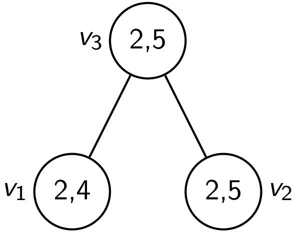
Proof.

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 $\mathcal{O}(k^2)$, we obtain $\mathcal{O}(e \cdot k^3)$. □

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

Enforcing arc consistency: AC3

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").



Queue

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

Enforcing arc consistency: AC4

- To verify that a network is arc-consistent needs $e \cdot k^2$ operations.
- The following algorithm AC4 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);
- remove a value a_i if it looses support from any other variable

Details:

- Q: queue of unsupported variable-value pairs;
- $counter(v_i, a_i, v_j)$: amount of support for a_i from v_j;
- $S[v_j, a_j]$: set containing variable-value pairs (v_i, a_i) (with $i \neq j$) supported by (v_j, a_j).

Enforcing arc consistency: AC4

AC4(N):

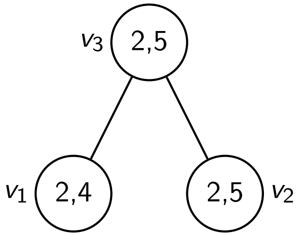
Input: a constraint network $N = \langle V, D, C \rangle$
Result: an equivalent, but arc-consistent network

$Q \leftarrow \emptyset$;
$S[v_j, a_j] \leftarrow \emptyset$, $counter(v_i, a_i, v_j) \leftarrow 0$ for all $R_{ij} \in C$, $a_i \in D_i, a_j \in D_j$
for each $R_{ij} \in C$, $a_i \in D_i$
  for each $a_j \in D_j$
    if $(a_i, a_j) \in R_{ij}$ **then**
      increment $counter(v_i, a_i, v_j)$ and add (v_i, a_i) to $S[v_j, a_j]$
  if $counter(v_i, a_i, v_j) = 0$ **then**
    add (v_i, a_i) to Q and remove a_i from D_i

while Q is not empty
  select and remove (v_j, a_j) from Q
  for each (v_i, a_i) in $S[v_j, a_j]$
    if $a_i \in D_i$ **then**
      decrement $counter(v_i, a_i, v_j)$
      if $counter(v_i, a_i, v_j) = 0$ **then**
        add (v_i, a_i) to Q and remove a_i from D_i

Example: AC4

Consider the same network as for AC3.
Constraints: $v_3 | v_1$ and $v_3 | v_2$.



The initialization steps yield:

$$S[v_3, 2] = \{(v_1, 2), (v_1, 4), (v_2, 2)\} \qquad S[v_3, 5] = \{(v_2, 5)\}$$
$$S[v_2, 2] = \{(v_3, 2)\} \qquad S[v_2, 5] = \{(v_3, 5)\}$$
$$S[v_1, 2] = \{(v_3, 2)\} \qquad S[v_1, 4] = \{(v_3, 2)\}$$

Example: AC4

The initialization steps yield:

$$S[v_3, 2] = \{(v_1, 2), (v_1, 4), (v_2, 2)\} \qquad S[v_3, 5] = \{(v_2, 5)\}$$
$$S[v_2, 2] = \{(v_3, 2)\} \qquad S[v_2, 5] = \{(v_3, 5)\}$$
$$S[v_1, 2] = \{(v_3, 2)\} \qquad S[v_1, 4] = \{(v_3, 2)\}$$

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, ...

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)
- ... 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 $\mathcal{O}(e \cdot k^2)$.

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 $Last(v_i, a_i, v_j) \notin D_j$
    $a_j \leftarrow$ the smallest value a in D_j with
        $a > Last(v_i, a_i, v_j)$ and $(a_i, a) \in R_{ij}$
    if a_j exists **then**
        $Last(v_i, a_i, v_j) \leftarrow a_j$
    else
        remove a_i from D_i

2 Path Consistency

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

Path consistency

Definition

(a) A binary constraint R_{ij} for variables v_i, v_j is path-consistent relative to a third variable v_k if for every pair $(a_i, a_j) \in R_{ij}$, there exists an $a_k \in D_k$ such that $(a_i, a_k) \in R_{ik}$ and $(a_k, a_j) \in R_{kj}$.

(b) A pair of distinct variables v_i, v_j is path-consistent relative to variable v_k if any instantiation a of $\{v_i, v_j\}$ with $(a(v_i), a(v_j)) \in R_{ij}$ can be extended to an instantiation a' of $\{v_i, v_j, v_k\}$ such that $(a'(v_i), a'(v_k)) \in R_{ik}$ and $(a'(v_k), a'(v_j)) \in R_{kj}$ ("extended" means: $a = a'|_{\{v_i, v_j\}}$).

(c) A set of distinct variables $\{v_i, v_j, v_k\}$ is path-consistent if any pair of these variables is path-consistent relative to the omitted third variable.

(d) A constraint network is path-consistent if all its three-element subsets of variables are path-consistent.

An example

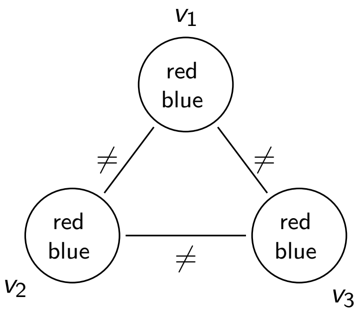


Figure: This network is arc-consistent, but not path-consistent.

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

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}
    endif
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 $\mathcal{O}(t \cdot k)$.
The best-case runtime of Revise-3 is $\Omega(t)$.

With respect to k, the complexity of Revise-3 can also be expressed as $\mathcal{O}(k^3)$ in the worst and $\Omega(k^2)$ in the best case.

Enforcing path consistency: PC1

PC1(N):

Input: a constraint network $N = \langle V, D, C \rangle$
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 $\mathcal{C}$.

The worst-case runtime of PC1 on this network is $\mathcal{O}(n^5 \cdot t^2 \cdot k)$.
The best-case runtime of PC1 on this network is $\Omega(n^3 \cdot t)$.

The runtime bounds can also be stated as $\mathcal{O}(n^5 \cdot k^5)$ 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 $\mathcal{O}(n^2 \cdot t)$.
Processing a specific triple of constraints (there are $\mathcal{O}(n^3)$ many such triples) costs $\mathcal{O}(t \cdot k)$.
Hence each iteration costs $\mathcal{O}(n^3 \cdot t \cdot k)$. □

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 $N = \langle V, D, C \rangle$
Result: an equivalent, path-consistent network N'

$queue \leftarrow \{(i,k,j) \,:\, 1 \le i < j \le n, 1 \le k \le n, k \ne i, k \ne j\}$
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**
        $queue \leftarrow queue \cup \{(l,i,j),\ (l,j,i) : 1 \le l \le n, l \ne i,j\}$
    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 $\leq k$. 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 $\mathcal{O}(n^3 \cdot t^2 \cdot k)$.
The best-case runtime of PC2 on this network is $\Omega(n^3 \cdot t)$.

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

Enforcing path consistency: Complexity of PC2

Proof (worst case).

There are initially $\mathcal{O}(n^3)$ elements in the queue. Whenever some constraint R_{ij} is reduced, which can happen at most $\mathcal{O}(n^2 \cdot t)$ many times, $\mathcal{O}(n)$ elements are added to the queue. Thus, the total number of elements added to the queue is bounded by $\mathcal{O}(n^3 \cdot t)$.
Each iteration of the **while** loop removes an element from the queue, so there are at most $\mathcal{O}(n^3 \cdot t)$ iterations and hence at most $\mathcal{O}(n^3 \cdot t)$ calls to Revise-3, each requiring time $\mathcal{O}(t \cdot k)$, for a total runtime bound of $\mathcal{O}(n^3 \cdot t^2 \cdot k)$. □

Proof (best case).

Similar to PC1. □

Arc and path consistency: Overview

	Worst Case	Best Case
AC1	$\mathcal{O}(n \cdot k \cdot e \cdot t)$	$\Omega(e \cdot k)$
AC3	$\mathcal{O}(e \cdot k \cdot t)$	$\Omega(e \cdot k)$
AC4	$\mathcal{O}(e \cdot k^2)$	$\Omega(e \cdot k^2)$
PC1	$\mathcal{O}(n^5 \cdot t^2 \cdot k)$	$\Omega(n^3 \cdot t)$
PC2	$\mathcal{O}(n^3 \cdot t^2 \cdot k)$	$\Omega(n^3 \cdot t)$
PC4*	$\mathcal{O}(n^3 \cdot t \cdot k)$	$\Omega(n^3 \cdot t \cdot k)$

*not discussed in this lecture

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

3 Higher Levels of *i*-Consistency

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.

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 ≥ 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

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

Input: a network $\langle V, D, C\rangle$ 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 $\mathcal{O}(k^i)$.
- In general, Revise-i runs in time $\mathcal{O}((2 \cdot k)^i)$, since $\mathcal{O}(2^i)$ constraints must be processed for each tuple.

i-Consistency: Algorithm

Enforce *i*-Consistency(N):

Input: a constraint network $N = \langle V, D, C \rangle$.
Result: an i-consistent network equivalent to N.

repeat
    for each subset of $S \subseteq V$ of size $i - 1$ and each $v_i \notin S$
        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 $\mathcal{S}$ 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

$$R_S \leftarrow R_S \cap \pi_S(\bowtie_{S' \in \mathcal{S}} R_{S'}).$$

i-Consistency: Complexity

Lemma

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

Proof.

Each call to Revise-*i* requires time $\mathcal{O}((2 \cdot k)^i)$. In each iteration of the outer loop, $\mathcal{O}(n^i)$ 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 $\mathcal{O}(n^{i-1} \cdot k^{i-1})$ times.
This leads to an overall runtime of $\mathcal{O}(2^i \cdot (n \cdot k)^{2i-1})$. □

Note: Improvements similar to AC4 and PC4 exist and achieve a worst-case runtime of $\mathcal{O}(n^i \cdot k^i)$.

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

	Worst Case
i-consistency, $i = 2$	$\mathcal{O}(n^3 \cdot k^3)$
AC1	$\mathcal{O}(n \cdot k \cdot e \cdot t) = \mathcal{O}(n^3 \cdot k^3)$
AC3	$\mathcal{O}(e \cdot k \cdot t) = \mathcal{O}(n^2 \cdot k^3)$
AC4	$\mathcal{O}(n^2 \cdot k^2)$
improved i-consistency*, $i = 2$	$\mathcal{O}(n^2 \cdot k^2)$
i-consistency, $i = 3$	$\mathcal{O}(n^5 \cdot k^5)$
PC1	$\mathcal{O}(n^5 \cdot t^2 \cdot k) = \mathcal{O}(n^5 \cdot k^5)$
PC2	$\mathcal{O}(n^3 \cdot t^2 \cdot k) = \mathcal{O}(n^3 \cdot k^5)$
PC4*	$\mathcal{O}(n^3 \cdot k^3)$
improved i-consistency*, $i = 3$	$\mathcal{O}(n^3 \cdot k^3)$

*not discussed in this lecture

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

4 Extensions of Arc Consistency

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!

$\rightsquigarrow$ Consider generalizations of arc consistency to non-binary constraints.

Generalized arc consistency

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

Definition

(a) A variable v_i is (generalized) arc-consistent relative to a constraint $(S, R) \in C$ with v_i in $S = (v_1, \dots, v_n)$ if for every value $a_i \in D_i$ there exists a tuple $\overline{a} \in R \cap (D_1 \times \dots \times D_n)$ with $\overline{a}[i] = a_i$, i.e.,

$$D_i \subseteq \pi_i(R \cap (D_1 \times \dots \times D_n)).$$

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

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

Generalized arc consistency: Update rule

To enforce generalized arc consistency, repeatedly apply

$$D_i \leftarrow D_i \cap \pi_i(R_S \bowtie D_{S \setminus \{v_i\}})$$

Note how this generalizes the usual arc consistency update rule:

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

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:

  $$R_{S\setminus\{v_i\}} \leftarrow R_{S\setminus\{v_i\}} \cap \pi_{S\setminus\{v_i\}}(R_S \bowtie D_i)$$

  (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

$$\text{AC: } D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$$
$$\text{generalized AC: } D_i \leftarrow D_i \cap \pi_i(R_S \bowtie D_{S\setminus\{v_i\}})$$
$$\text{relational AC: } R_{S\setminus\{v_i\}} \leftarrow R_{S\setminus\{v_i\}} \cap \pi_{S\setminus\{v_i\}}(R_S \bowtie D_i)$$

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

Christian Bessiere.
Constraint propagataion,
Chapter 3 of Handbook of Constraint Programming, 2006

Rina Dechter.
Constraint Processing,
Chapter 3, Morgan Kaufmann, 2003

Alan K. Mackworth.
Constraint satisfaction.
In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. Wiley, Chichester, England, 1987.

Alan K. Mackworth.
Consistency in networks of relations.
Artificial Intelligence, 8:99–118, 1977.

Ugo Montanari.
Networks of constraints: fundamental properties and applications to picture processing.
Information Science, 7:95–132, 1974.