Foundations of Artificial Intelligence

25. Constraint Satisfaction Problems: Arc Consistency

Martin Wehrle

Universität Basel

April 15, 2016

Foundations of Artificial Intelligence

April 15, 2016 — 25. Constraint Satisfaction Problems: Arc Consistency

Constraint Satisfaction Problems: Overview

Chapter overview: constraint satisfaction problems:

- 22.–23. Introduction
- 24.–26. Basic Algorithms
  - 24. Backtracking
  - 25. Arc Consistency
  - 26. Path Consistency
- 27.–28. Problem Structure

25.1 Inference

Inference

Inference

Derive additional constraints (here: unary or binary)
that are implied by the given constraints,
i.e., that are satisfied in all solutions.

example: constraint network with variables v_1, v_2, v_3
with domain $\{1, 2, 3\}$ and constraints $v_1 < v_2$ and $v_2 < v_3$.

it follows:

- v_2 cannot be equal to 3
  (new unary constraint = tighter domain of v_2)
- $R_{v_1 v_2} = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,3\rangle\}$ can be tightened to $\{\langle 1,2\rangle\}$
  (tighter binary constraint)
- $v_1 < v_3$
  ("new" binary constraint = trivial constraint tightened)

Trade-Off Search vs. Inference

Inference formally

For a given constraint network $\mathcal{C}$, replace $\mathcal{C}$
with an equivalent, but tighter constraint network.

Trade-off:

- the more complex the inference, and
- the more often inference is applied,
- the smaller the resulting state space, but
- the higher the complexity per search node.

When to Apply Inference?

different possibilities to apply inference:

- once as preprocessing before search
- combined with search: before recursive calls
  during backtracking procedure
  - already assigned variable $v \mapsto d$ corresponds to $\text{dom}(v) = \{d\}$
    $\rightsquigarrow$ more inferences possible
  - during backtracking, derived constraints have to be retracted
    because they were based on the given assignment

  $\rightsquigarrow$ powerful, but possibly expensive

Backtracking with Inference

function BacktrackingWithInference($\mathcal{C}, \alpha$):

if α is inconsistent with $\mathcal{C}$:
    return inconsistent
if α is a total assignment:
    return α
$\mathcal{C}' := \langle V, \text{dom}', (R'_{uv})\rangle :=$ copy of $\mathcal{C}$
apply inference to $\mathcal{C}'$
if $\text{dom}'(v) \neq \emptyset$ for all variables v:
    select some variable v for which α is not defined
    for each $d \in$ copy of $\text{dom}'(v)$ in some order:
        $\alpha' := \alpha \cup \{v \mapsto d\}$
        $\text{dom}'(v) := \{d\}$
        $\alpha'' :=$ BacktrackingWithInference($\mathcal{C}', \alpha'$)
        if $\alpha'' \neq$ **inconsistent**:
            return α''
return inconsistent

Backtracking with Inference: Discussion

- Inference is a placeholder:
  different inference methods can be applied.
- Inference methods can recognize unsolvability (given α)
  and indicate this by clearing the domain of a variable.
- Efficient implementations of inference are often incremental:
  the previously assigned variable/value pair $v \mapsto d$ is taken
  into account to speed up the inference computation.

25.2 Forward Checking

Forward Checking

We start with a simple inference method:

Forward Checking

Let α be a partial assignment.
Inference: For all unassigned variables v in α,
remove all values from the domain of v that are in conflict
with already assigned variable/value pairs in α.

⇝ definition of conflict as in the previous chapter

Incremental computation:

- When adding $v \mapsto d$ to the assignment,
  delete all pairs that conflict with $v \mapsto d$.

Forward Checking: Discussion

properties of forward checking:

- correct inference method (retains equivalence)
- affects domains (= unary constraints),
  but not binary constraints
- consistency check at the beginning of the backtracking
  procedure no longer needed (Why?)
- cheap, but often still useful inference method

⇝ apply at least forward checking in the backtracking procedure

In the following, we will consider more powerful inference methods.

25.3 Arc Consistency

Arc Consistency: Definition

Definition (Arc Consistent)

Let $\mathcal{C} = \langle V, \mathrm{dom}, (R_{uv}) \rangle$ be a constraint network.

(a) The variable $v \in V$ is arc consistent with respect to another variable $v' \in V$, if for every value $d \in \mathrm{dom}(v)$ there exists a value $d' \in \mathrm{dom}(v')$ with $\langle d, d' \rangle \in R_{vv'}$.

(b) The constraint network $\mathcal{C}$ is arc consistent, if every variable $v \in V$ is arc consistent with respect to every other variable $v' \in V$.

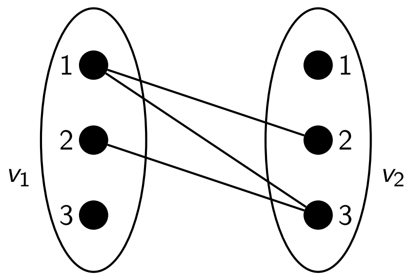
German: kantenkonsistent

remarks:

- definition for variable pair is not symmetrical
- v always arc consistent with respect to v' if the constraint between v and v' is trivial

Arc Consistency: Example

Consider a constraint network with variables v_1 and v_2, domains $\mathrm{dom}(v_1) = \mathrm{dom}(v_2) = \{1, 2, 3\}$ and the constraint expressed by $v_1 < v_2$.



Arc consistency of v_1 with respect to v_2 and of v_2 with respect to v_1 are violated.

Enforcing Arc Consistency

- Enforcing arc consistency, i.e., removing values from $\mathrm{dom}(v)$ that violate the arc consistency of v with respect to v', is a correct inference method. (Why?)
- more powerful than forward checking (Why?)
  - ⇝ Forward checking is a special case: enforcing arc consistency of all variables with respect to the just assigned variable corresponds to forward checking.

We will next consider algorithms that enforce arc consistency.

Processing Variable Pairs: revise

function revise($\mathcal{C}, v, v'$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

for each $d \in \text{dom}(v)$:

  if there is no $d' \in \text{dom}(v')$ with $\langle d, d' \rangle \in R_{vv'}$:

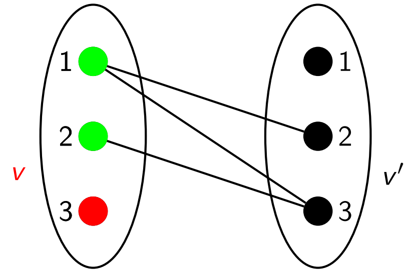
    remove d from dom(v)

input: constraint network $\mathcal{C}$ and two variables v, v' of $\mathcal{C}$

effect: v arc consistent with respect to v'.
All violating values in dom(v) are removed.

time complexity: $O(k^2)$, where k is maximal domain size

Example: revise



Enforcing Arc Consistency: AC-1

function AC-1($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv}) \rangle := \mathcal{C}$

repeat

  for each nontrivial constraint R_{uv}:

    revise($\mathcal{C}, u, v$)

    revise($\mathcal{C}, v, u$)

until no domain has changed in this iteration

input: constraint network $\mathcal{C}$

effect: transforms $\mathcal{C}$ into equivalent arc consistent network

time complexity: $O(n \cdot e \cdot k^3)$, with n variables,
e nontrivial constraints and maximal domain size k

AC-1: Discussion

- AC-1 does the job, but is rather inefficient.
- Drawback: Variable pairs are often checked again and again although their domains have remained unchanged.
- These (redundant) checks can be saved.

$\rightsquigarrow$ more efficient algorithm: AC-3

Enforcing Arc Consistency: AC-3

idea: store potentially inconsistent variable pairs in a queue

function AC-3($\mathcal{C}$):

$\langle V, \text{dom}, (R_{uv})\rangle := \mathcal{C}$

queue $:= \emptyset$

for each nontrivial constraint R_{uv}:
    insert $\langle u, v\rangle$ into *queue*
    insert $\langle v, u\rangle$ into *queue*

while *queue* $\neq \emptyset$:
    remove an arbitrary element $\langle u, v\rangle$ from *queue*
    revise($\mathcal{C}, u, v$)
    if dom(u) changed in the call to revise:
        for each $w \in V \setminus \{u, v\}$ where R_{wu} is nontrivial:
            insert $\langle w, u\rangle$ into *queue*

AC-3: Discussion

- *queue* can be an arbitrary data structure that supports insert and remove operations (the order of removal does not affect the result)

$\rightsquigarrow$ use data structure with fast insertion and removal, e.g., stack

- AC-3 has the same effect as AC-1: it enforces arc consistency
- proof idea: invariant of the **while** loop: If $\langle u, v\rangle \notin$ *queue*, then u is arc consistent with respect to v

AC-3: Time Complexity

Proposition (Time Complexity of AC-3)

Let $\mathcal{C}$ be a constraint network with e nontrivial constraints and maximal domain size k.

The time complexity of AC-3 *is $O(e \cdot k^3)$.*

AC-3: Time Complexity (Proof)

Beweis.

Consider a pair $\langle u, v\rangle$ such that there exists a nontrivial constraint R_{uv} or R_{vu}. (There are at most $2e$ of such pairs.)

Every time this pair is inserted to the queue (except for the first time) the domain of the second variable has just been reduced.

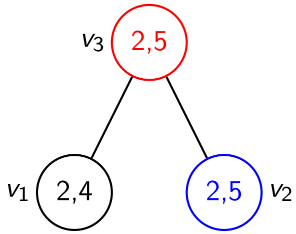
This can happen at most k times.

Hence every pair $\langle u, v\rangle$ is inserted into the queue at most $k + 1$ times $\rightsquigarrow$ at most $O(ek)$ insert operations in total.

This bounds the number of **while** iterations by $O(ek)$, giving an overall time complexity of $O(ek) \cdot O(k^2) = O(ek^3)$. □

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3 | v_1$ and $v_3 | v_2$ ("divides exactly").

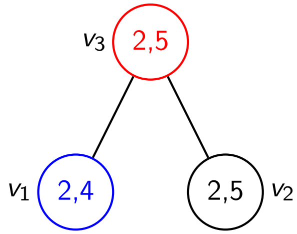


Queue

$\langle v_1, v_3 \rangle$
$\langle v_3, v_1 \rangle$
$\langle v_2, v_3 \rangle$
$\langle v_3, v_2 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3 | v_1$ and $v_3 | v_2$ ("divides exactly").

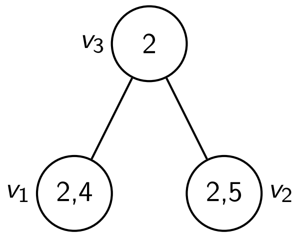


Queue

$\langle v_1, v_3 \rangle$
$\langle v_3, v_1 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3 | v_1$ and $v_3 | v_2$ ("divides exactly").

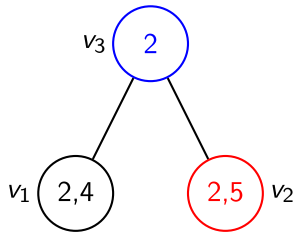


Queue

$\langle v_1, v_3 \rangle$
$\langle v_2, v_3 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3 | v_1$ and $v_3 | v_2$ ("divides exactly").

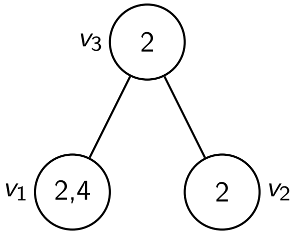


Queue

$\langle v_1, v_3 \rangle$
$\langle v_2, v_3 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3|v_1$ and $v_3|v_2$ ("divides exactly").

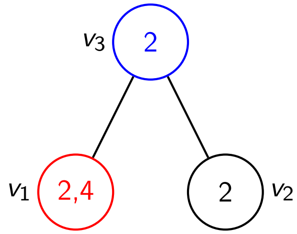


Queue

$\langle v_1, v_3 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3|v_1$ and $v_3|v_2$ ("divides exactly").

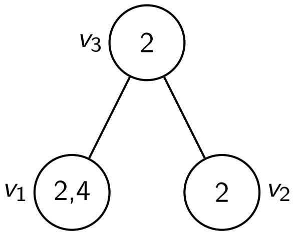


Queue

$\langle v_1, v_3 \rangle$

AC-3: Example

Consider a constraint network with three variables v_1, v_2, v_3 with $\text{dom}(v_1) = \{2, 4\}$ and $\text{dom}(v_2) = \text{dom}(v_3) = \{2, 5\}$ and the constraints $v_3|v_1$ and $v_3|v_2$ ("divides exactly").



Queue

25.4 Summary

Summary: Inference

- inference: derivation of additional constraints that are implied by the known constraints

$\rightsquigarrow$ tighter equivalent constraint network

- trade-off search vs. inference
- inference as preprocessing or integrated into backtracking

Summary: Forward Checking, Arc Consistency

- cheap and easy inference: forward checking
  - remove values that conflict with already assigned values
- more expensive and more powerful: arc consistency
  - iteratively remove values without a suitable "partner value" for another variable until fixed-point reached
  - efficient implementation of AC-3: $O(ek^3)$ with e: #nontrivial constraints, k: size of domain