

# Foundations of Artificial Intelligence

## 25. Constraint Satisfaction Problems: Arc Consistency

Malte Helmert

University of Basel

April 8, 2019

# 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

Inference  
●○○○○

Forward Checking  
○○○

Arc Consistency  
○○○○○○○○○○○○○○

Summary  
○○○

# 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  $v_2$ )
- $R_{v_1v_2} = \{\langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 3 \rangle\}$  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  $\text{dom}(v) = \{d\}$   
~~ more inferences possible
  - during backtracking, derived constraints have to be **retracted** because they were based on the given assignment  
~~ powerful, but possibly expensive

# Backtracking with Inference

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

```
  if  $\alpha$  is inconsistent with  $\mathcal{C}$ :  
    return inconsistent
```

```
  if  $\alpha$  is a total assignment:  
    return  $\alpha$ 
```

```
   $\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle := \text{copy of } \mathcal{C}$   
  apply inference to  $\mathcal{C}'$ 
```

```
  if  $\text{dom}'(v) \neq \emptyset$  for all variables  $v$ :  
    select some variable  $v$  for which  $\alpha$  is not defined
```

```
    for each  $d \in \text{copy of } \text{dom}'(v)$  in some order:
```

```
       $\alpha' := \alpha \cup \{v \mapsto d\}$ 
```

```
       $\text{dom}'(v) := \{d\}$ 
```

```
       $\alpha'' := \text{BacktrackingWithInference}(\mathcal{C}', \alpha')$ 
```

```
      if  $\alpha'' \neq \text{inconsistent}$ :  
        return  $\alpha''$ 
```

```
  return inconsistent
```

# Backtracking with Inference

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

```
  if  $\alpha$  is inconsistent with  $\mathcal{C}$ :  
    return inconsistent
```

```
  if  $\alpha$  is a total assignment:  
    return  $\alpha$ 
```

```
   $\mathcal{C}' := \langle V, \text{dom}', (R'_{uv}) \rangle := \text{copy of } \mathcal{C}$   
  apply inference to  $\mathcal{C}'$ 
```

```
  if  $\text{dom}'(v) \neq \emptyset$  for all variables  $v$ :  
    select some variable  $v$  for which  $\alpha$  is not defined
```

```
    for each  $d \in \text{copy of } \text{dom}'(v)$  in some order:
```

```
       $\alpha' := \alpha \cup \{v \mapsto d\}$ 
```

```
       $\text{dom}'(v) := \{d\}$ 
```

```
       $\alpha'' := \text{BacktrackingWithInference}(\mathcal{C}', \alpha')$ 
```

```
      if  $\alpha'' \neq \text{inconsistent}$ :  
        return  $\alpha''$ 
```

```
  return inconsistent
```

# Backtracking with Inference: Discussion

- Inference is a placeholder:  
different inference methods can be applied.
- Inference methods can recognize unsolvability (given  $\alpha$ )  
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.

Inference  
oooooo

Forward Checking  
●oo

Arc Consistency  
oooooooooooo

Summary  
ooo

# Forward Checking

# Forward Checking

We start with a simple inference method:

## Forward Checking

Let  $\alpha$  be a partial assignment.

**Inference:** For all unassigned variables  $v$  in  $\alpha$ ,  
remove all values from the domain of  $v$  that are in conflict  
with already assigned variable/value pairs in  $\alpha$ .

~ 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.

Inference  
oooooo

Forward Checking  
ooo

Arc Consistency  
●oooooooooooo

Summary  
ooo

# Arc Consistency

# Arc Consistency: Definition

## Definition (Arc Consistent)

Let  $\mathcal{C} = \langle V, \text{dom}, (R_{uv}) \rangle$  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  $d \in \text{dom}(v)$  there exists a value  $d' \in \text{dom}(v')$  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  $v' \in V$ .

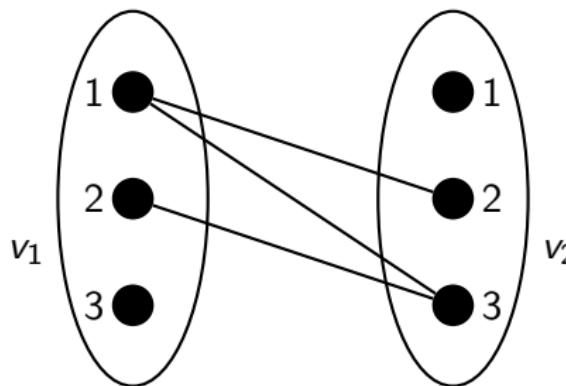
German: kantenkonsistent

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  $\text{dom}(v_1) = \text{dom}(v_2) = \{1, 2, 3\}$  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  $\text{dom}(v)$  that violate the arc consistency of  $v$  with respect to  $v'$ , is a correct inference method. ([Why?](#))
- **more powerful** than forward checking ([Why?](#))

# Enforcing Arc Consistency

- **Enforcing arc consistency**, i.e., removing values from  $\text{dom}(v)$  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  $\text{dom}(v)$

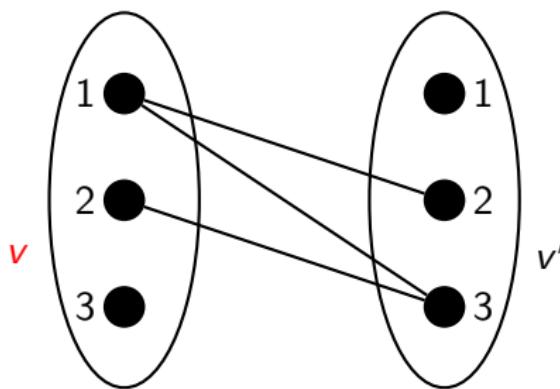
**input:** constraint network  $\mathcal{C}$  and two variables  $v, v'$  of  $\mathcal{C}$

**effect:**  $v$  arc consistent with respect to  $v'$ .

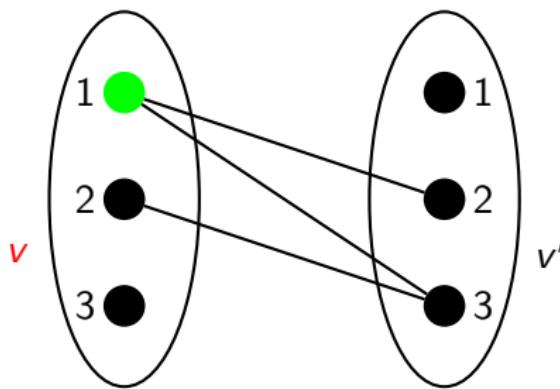
All violating values in  $\text{dom}(v)$  are removed.

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

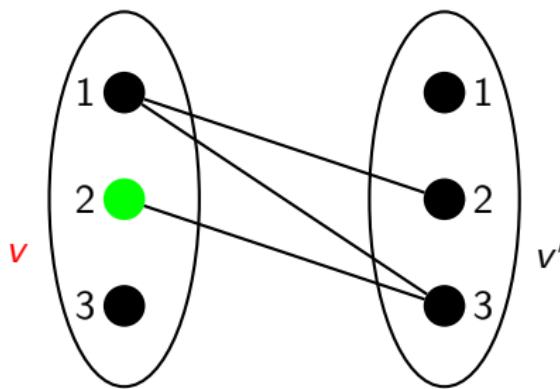
## Example: revise



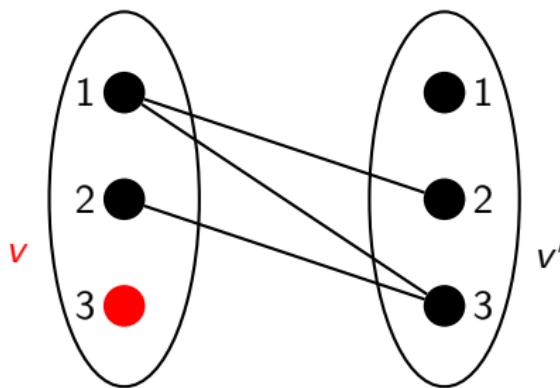
## Example: revise



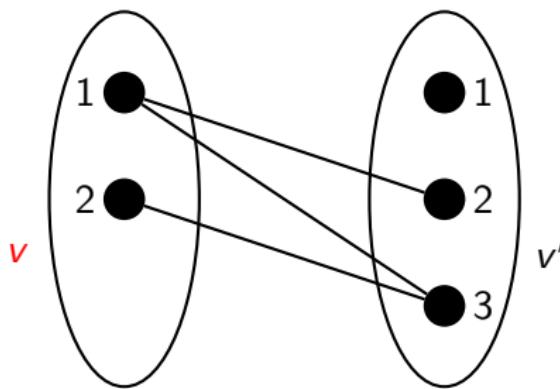
## Example: revise



## Example: revise



## 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: ?

# 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.

↝ 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**  $\text{dom}(u)$  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)
  - ↝ 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  $\langle u, v \rangle \notin \text{queue}$ , 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)

### Proof.

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.)

## AC-3: Time Complexity (Proof)

### Proof.

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.

## AC-3: Time Complexity (Proof)

### Proof.

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.

## AC-3: Time Complexity (Proof)

### Proof.

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.

## AC-3: Time Complexity (Proof)

### Proof.

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  $O(ek) \cdot O(k^2) = O(ek^3)$ . □

Inference  
oooooo

Forward Checking  
ooo

Arc Consistency  
oooooooooooo

Summary  
●oo

# Summary

## Summary: Inference

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