

Artificial Intelligence

7. CSP, Part II: Inference, and Decomposition Methods

How to *Efficiently* Satisfy All These Constraints

Jörg Hoffmann, Daniel Fiser, Daniel Höller, Sophia Saller



Summer Term 2022

Agenda

- 1 Introduction
- 2 Inference
- 3 Forward Checking
- 4 Arc Consistency
- 5 Decomposition: Constraint Graphs, and Two Simple Cases
- 6 Cutset Conditioning
- 7 Conclusion

Reminder: Our Agenda for This Topic

→ Our treatment of the topic “Constraint Satisfaction Problems” consists of Chapters 6 and 7.

- **Chapter 6:** Basic definitions and concepts; naïve backtracking search.
 - Sets up the framework. Backtracking underlies many successful algorithms for solving constraint satisfaction problems (and, naturally, we start with the simplest version thereof).
- **This Chapter:** Inference and decomposition methods.
 - Inference reduces the search space of backtracking. Decomposition methods break the problem into smaller pieces. Both are crucial for efficiency in practice.

Illustration: Inference

Constraint network γ :



→ An additional constraint we can add without losing any solutions? For example, $C_{WAQ} := "="$. If WA and Q are assigned different colors, then NT must be assigned the 3rd color, leaving no color for SA .

→ Adding constraints without losing solutions = obtaining an equivalent network with a “tighter description” and hence with a smaller number of consistent partial assignments.

Illustration: Decomposition

Constraint network γ :



→ We can separate this into two independent constraint networks.
Namely? Tasmania is not adjacent to any other state. Thus we can color Australia first, and assign an arbitrary color to Tasmania afterwards.

→ Decomposition methods exploit the structure of the constraint network. They identify separate parts (sub-networks) whose inter-dependencies are “simple” and can be handled efficiently.

→ Extreme case: No inter-dependencies at all, as in our example here.

Our Agenda for This Chapter

- **Inference:** How does inference work in principle? What are relevant practical aspects?
→ Fundamental concepts underlying inference, basic facts about its use.
- **Forward Checking:** What is the simplest instance of inference?
→ Gets us started on this subject.
- **Arc Consistency:** How to make inferences between variables whose value is not fixed yet?
→ Details the canonical advanced inference method.
- **Decomposition: Constraint Graphs, and Two Simple Cases:** How to capture dependencies in a constraint network? What are “simple cases”?
→ Basic results on this subject.
- **Cutset Conditioning:** What if we're not in a simple case?
→ Outlines the most easily understandable technique for decomposition in the general case.

Inference: Basic Facts

Inference

Deducing additional constraints (unary or binary), that follow from the already known constraints, i.e., that are satisfied in all solutions.

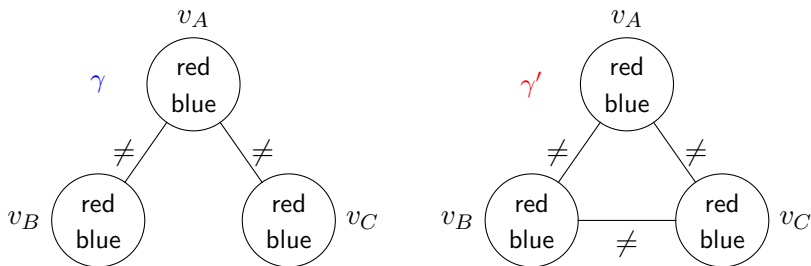
It's what you do all the time when playing SuDoKu:

	5	8	7		6	9	4	1
		9	8		4	3	5	7
4		7	9		5	2	6	8
3	9	5	2	7	1	4	8	6
7	6	2	4	9	8	1	3	5
8	4	1	6	5	3	7	2	9
1	8	4	3	6	9	5	7	2
5	7	6	1	4	2	8	9	3
9	2	3	5	8	7	6	1	4

→ Formally: Replace γ by an equivalent and strictly tighter constraint network γ' . Up next.

Equivalent Constraint Networks

Definition (Equivalence). Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ be constraint networks sharing the same set of variables. We say that γ and γ' are *equivalent*, written $\gamma' \equiv \gamma$, if they have the same set of solutions.

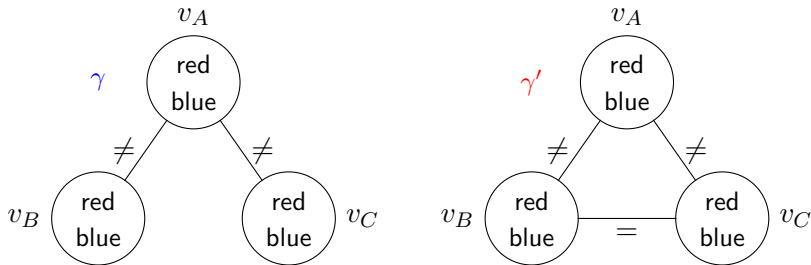


Are these constraint networks equivalent? No.

→ Equivalence: " γ' has the same solutions as γ ".

Equivalent Constraint Networks

Definition (Equivalence). Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ be constraint networks sharing the same set of variables. We say that γ and γ' are *equivalent*, written $\gamma' \equiv \gamma$, if they have the same set of solutions.



Are these constraint networks equivalent? Yes.

→ Equivalence: " γ' has the same solutions as γ ".

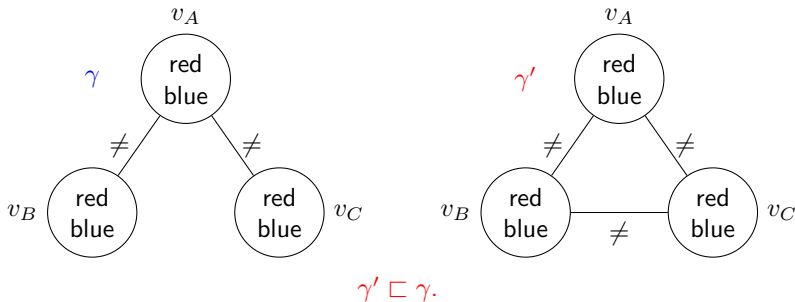
Tightness

Definition (Tightness). Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ be constraint networks sharing the same set of variables. We say that γ' is **tighter** than γ , written $\gamma' \sqsubseteq \gamma$, if:

❶ For all $v \in V$: $D'_v \subseteq D_v$.

❷ For all $u \neq v \in V$: $C'_{uv} \subseteq C_{uv}$ (where $C_{uv} \notin C$ is $C_{uv} = D_u \times D_v$).

γ' is **strictly tighter** than γ , $\gamma' \sqsubset \gamma$, if at least one of these inclusions is strict.



→ Tightness: " γ' has the same constraints as γ , plus some".

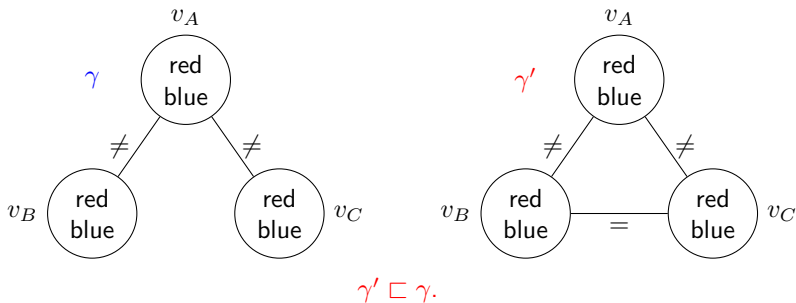
Tightness

Definition (Tightness). Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ be constraint networks sharing the same set of variables. We say that γ' is **tighter** than γ , written $\gamma' \sqsubseteq \gamma$, if:

❶ For all $v \in V$: $D'_v \subseteq D_v$.

❷ For all $u \neq v \in V$: $C'_{uv} \subseteq C_{uv}$ (where $C_{uv} \notin C$ is $C_{uv} = D_u \times D_v$).

γ' is **strictly tighter** than γ , $\gamma' \sqsubset \gamma$, if at least one of these inclusions is strict.



→ Tightness: “ γ' has the same constraints as γ , plus some”.

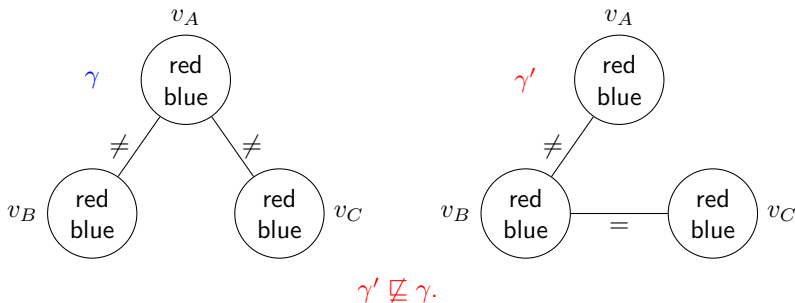
Tightness

Definition (Tightness). Let $\gamma = (V, D, C)$ and $\gamma' = (V, D', C')$ be constraint networks sharing the same set of variables. We say that γ' is **tighter** than γ , written $\gamma' \sqsubseteq \gamma$, if:

❶ For all $v \in V$: $D'_v \subseteq D_v$.

❷ For all $u \neq v \in V$: $C'_{uv} \subseteq C_{uv}$ (where $C_{uv} \notin C$ is $C_{uv} = D_u \times D_v$).

γ' is **strictly tighter** than γ , $\gamma' \sqsubset \gamma$, if at least one of these inclusions is strict.

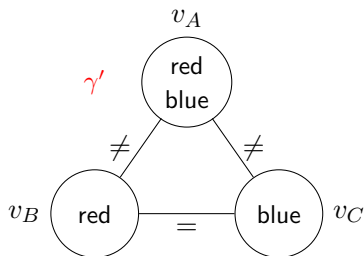
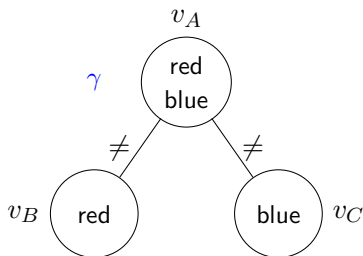


→ Tightness: " γ' has the same constraints as γ , plus some".

Equivalence + Tightness = Inference

Proposition. Let γ and γ' be constraint networks s.t. $\gamma' \equiv \gamma$ and $\gamma' \sqsubset \gamma$. Then γ' has the same solutions as, but less consistent partial assignments than, γ .

→ γ' is a better encoding of the underlying problem.



→ a cannot be extended to a solution (neither in γ nor in γ' because they're equivalent). a is consistent with γ , but not with γ' .

How to Use Inference?

Inference as a pre-process:

- Just once before search starts.
- Little runtime overhead, little pruning power. Not considered here.

Inference during search:

- At every recursive call of backtracking.
- Strong pruning power, may have large runtime overhead.

Search vs. Inference

The more complex the inference, the *smaller* the number of search nodes, but the *larger* the runtime needed at each node.

Backtracking With Inference

```
function BacktrackingWithInference( $\gamma, a$ ) returns a solution, or “inconsistent”  
  if  $a$  is inconsistent then return “inconsistent”  
  if  $a$  is a total assignment then return  $a$   
   $\gamma' :=$  a copy of  $\gamma$     /*  $\gamma' = (V, D', C')$  */  
   $\gamma' :=$  Inference( $\gamma'$ )  
  if exists  $v$  with  $D'_v = \emptyset$  then return “inconsistent”  
  select some variable  $v$  for which  $a$  is not defined  
  for each  $d \in$  copy of  $D'_v$  in some order do  
     $a' := a \cup \{v = d\}; D'_v := \{d\}$     /* makes  $a$  explicit as a constraint */  
     $a'' :=$  BacktrackingWithInference( $\gamma', a'$ )  
    if  $a'' \neq$  “inconsistent” then return  $a''$   
  return “inconsistent”
```

- **Inference()**: Any procedure delivering a (tighter) equivalent network.
- Inference typically prunes domains; indicate unsolvability by $D'_v = \emptyset$.
- Partial assignment \rightarrow unary constraints in γ' for call of inference in next recursion level.
- When backtracking out of a search branch, retract the inferred constraints: these were dependent on a , the search commitments so far.

Questionnaire

Constraint network γ :



Question!

Which modifications yield an equivalent and strictly tighter γ' ?

(A): $C_{WAQ} := "\neq"$

(B): $C_{WAQ} := "="$

(C): $D_{WA} := \{red, blue\}$

(D): $D_Q := \{green\}$

→ (C) and (D): No. Colors can be permuted in solutions, so fixing them is not equivalence-preserving.

→ (A): No. There are solutions in which WA and Q have the same value.

→ (B): Yes (cf. slide 5). If WA and Q are assigned different values, then NT must be assigned the 3rd value, and all 3 values are ruled out for SA . Thus every solution assigns WA and Q the same value, and γ' is equivalent to γ .

Forward Checking

Inference(), version 1: Forward Checking

```

function ForwardChecking( $\gamma, a$ ) returns modified  $\gamma$ 
  for each  $v$  where  $a(v) = d'$  is defined do
    for each  $u$  where  $a(u)$  is undefined and  $C_{uv} \in C$  do
       $D_u := \{d \mid d \in D_u, (d, d') \in C_{uv}\}$ 
  return  $\gamma$ 

```



WA	NT	Q	NSW	V	SA	T
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>
<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>	<div><div>■</div><div>■</div><div>■</div></div>

Forward Checking: Discussion

Properties:

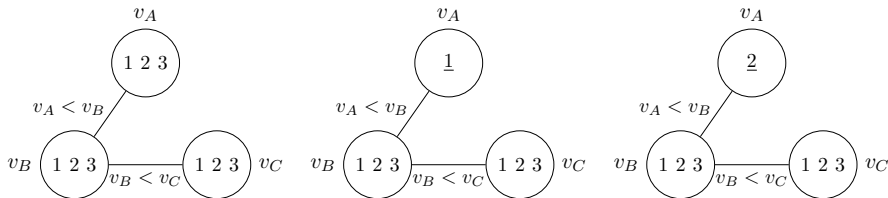
- Forward checking is **sound**: *Its tightening of constraints does not rule out any solutions. In other words: it guarantees to deliver an equivalent network.*
 - Recall here that the partial assignment a is represented as unary constraints in the network γ to which forward checking is applied. (And please excuse the slight arguments-mismatch with the call of “Inference(γ')” on slide 14.)
- Incremental computation: Instead of the first for-loop, use only the 2nd one every time a new assignment $a(v) = d'$ is added.

Practice:

- Cheap but useful inference method.
 - Rarely a good idea to not use forward checking (or a stronger inference method **subsuming** it).
- Up next: A stronger inference method (subsuming Forward Checking).

Questionnaire

Here and in what follows: **Underlined values = values set in a , i.e., chosen by backtracking.**

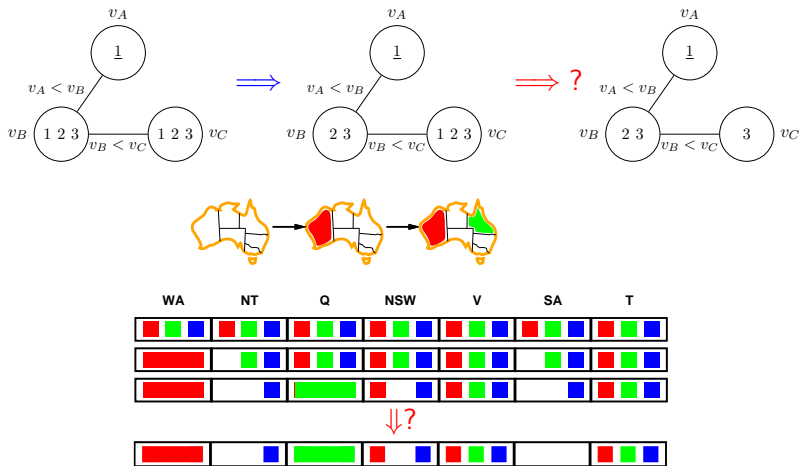


Question!

Which inferences does forward checking make, for each of these partial assignments?

→ Left: None, as there are no assignments. Middle: $D_{v_B} := \{2, 3\}$ then stop. Right: $D_{v_B} := \{3\}$ then stop! Forward Checking makes inferences only for *assigned* variables, *not* for ones whose domain has become singleton. (One could of course do that, but (a) this takes more runtime; and (b) while forward checking is an extremely simple method, it already is enough for many purposes, see slides 36 and 41.)

When Forward Checking is Not Good Enough



→ Forward checking makes inferences only “from assigned to unassigned” variables.

Arc Consistency: Definition

Definition (Arc Consistency). Let $\gamma = (V, D, C)$ be a constraint network.

- ❶ A variable $u \in V$ is **arc consistent** relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exists a value $d' \in D_v$ such that $(d, d') \in C_{uv}$.
- ❷ The network γ is **arc consistent** if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.

→ Arc consistency = for every domain value and constraint, at least one value on the other side of the constraint “works”.

→ Note the asymmetry between u and v : arc consistency is “directed”.

Examples: (previous slide)

- On top, middle, is v_C arc consistent relative to v_B ? No. For values 1 and 2, D_{v_B} does not have a value that works.
- And on the right? Yes. But v_B is not arc consistent relative to v_C .
- In the Australia example 3rd row, SA is not arc consistent relative to NT, and NT is not arc consistent relative to SA.

Enforcing Arc Consistency: General Remarks

Inference(), version 2: "Enforcing Arc Consistency" = removing variable domain values until γ is arc consistent. (Up next)

Note: (Assuming such an inference method $AC(\gamma)$)

- $AC(\gamma)$ is **sound**: *guarantees to deliver an equivalent network.*
→ If, for $d \in D_u$, there does not exist a value $d' \in D_v$ such that $(d, d') \in C_{uv}$, then $u = d$ cannot be part of any solution.
- $AC(\gamma)$ **subsumes** forward checking: $AC(\gamma) \sqsubseteq ForwardChecking(\gamma)$.
(Recall from slide 11 that $\gamma' \sqsubseteq \gamma$ means γ' is tighter than γ .)
→ Forward checking (cf. slide 17) removes d from D_u only if there is a constraint C_{uv} such that $D_v = \{d'\}$ (when v was assigned the value d'), and $(d, d') \notin C_{uv}$. Clearly, enforcing arc consistency of u relative to v removes d from D_u as well.

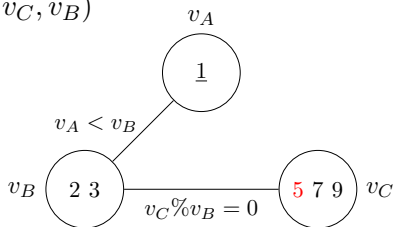
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v :

```
function Revise( $\gamma, u, v$ ) returns modified  $\gamma$ 
  for each  $d \in D_u$  do
    if there is no  $d' \in D_v$  with  $(d, d') \in C_{uv}$  then  $D_u := D_u \setminus \{d\}$ 
  return  $\gamma$ 
```

→ Runtime, if k is maximal domain size? $O(k^2)$, based on implementation where the test “ $(d, d') \in C_{uv}$?” is constant time.

Example: Revise(γ, v_C, v_B)



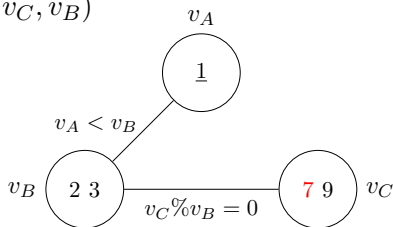
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v :

```
function Revise( $\gamma, u, v$ ) returns modified  $\gamma$ 
  for each  $d \in D_u$  do
    if there is no  $d' \in D_v$  with  $(d, d') \in C_{uv}$  then  $D_u := D_u \setminus \{d\}$ 
  return  $\gamma$ 
```

→ Runtime, if k is maximal domain size? $O(k^2)$, based on implementation where the test “ $(d, d') \in C_{uv}$?” is constant time.

Example: Revise(γ, v_C, v_B)



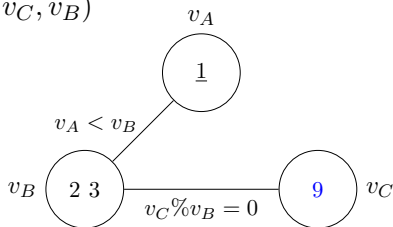
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v :

```
function Revise( $\gamma, u, v$ ) returns modified  $\gamma$ 
  for each  $d \in D_u$  do
    if there is no  $d' \in D_v$  with  $(d, d') \in C_{uv}$  then  $D_u := D_u \setminus \{d\}$ 
  return  $\gamma$ 
```

→ Runtime, if k is maximal domain size? $O(k^2)$, based on implementation where the test “ $(d, d') \in C_{uv}$?” is constant time.

Example: Revise(γ, v_C, v_B)



AC-1

Idea: Apply pairwise revisions up to a fixed point.

```
function AC-1( $\gamma$ ) returns modified  $\gamma$ 
  repeat
    changesMade := False
    for each constraint  $C_{uv}$  do
      Revise( $\gamma, u, v$ )    /* if  $D_u$  reduces, set changesMade := True */
      Revise( $\gamma, v, u$ )    /* if  $D_v$  reduces, set changesMade := True */
    until changesMade = False
  return  $\gamma$ 
```

- Obviously, this enforces arc consistency.
- Runtime, if n variables, m constraints, k maximal domain size?
 $O(mk^2 * nk)$: mk^2 for each inner loop, fixed point reached at the latest once all nk variable values have been removed.
- Redundant computations? u and v are revised even if their domains haven't changed since the last time.

AC-3

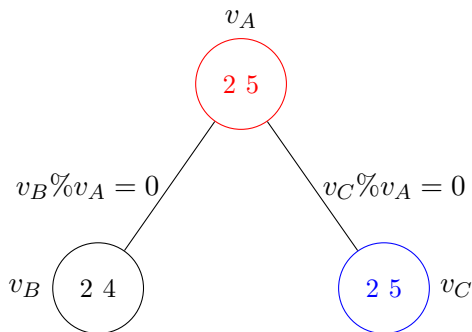
Idea: Remember the potentially inconsistent variable pairs.

```
function AC-3( $\gamma$ ) returns modified  $\gamma$ 
   $M := \emptyset$ 
  for each constraint  $C_{uv} \in C$  do
     $M := M \cup \{(u, v), (v, u)\}$ 
  while  $M \neq \emptyset$  do
    remove any element  $(u, v)$  from  $M$ 
    Revise( $\gamma, u, v$ )
    if  $D_u$  has changed in the call to Revise then
      for each constraint  $C_{xy} \in C$  where
         $\{x, y\} = \{u, w\}$  with  $w \neq v$  do
         $M := M \cup \{(w, u)\}$ 
  return  $\gamma$ 
```

- **AC-3(γ) enforces arc consistency because?** At any time during the while-loop, if $(u, v) \notin M$ then u is arc consistent relative to v .
- **Why only " $w \neq v$ "?** v is the *reason* why D_u just changed. Thus, if v was arc consistent relative to u before, then that still is so: the values just removed from D_u did not match any values from D_v anyway.

AC-3: Example

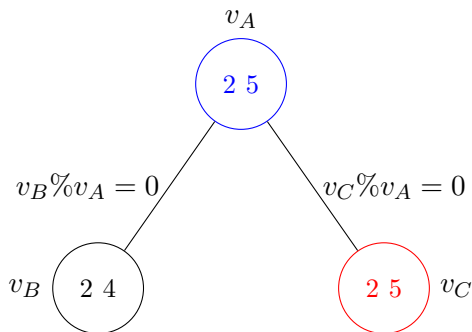
Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



M
(v_B, v_A)
(v_A, v_B)
(v_C, v_A)
(v_A, v_C)

AC-3: Example

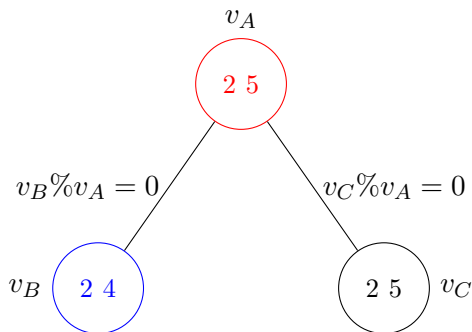
Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



$$\frac{M}{\begin{array}{l} (v_B, v_A) \\ (v_A, v_B) \\ (v_C, v_A) \end{array}}$$

AC-3: Example

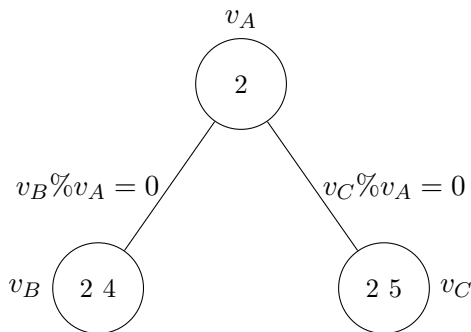
Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



$$\frac{M}{(v_B, v_A)} \\ (v_A, v_B)$$

AC-3: Example

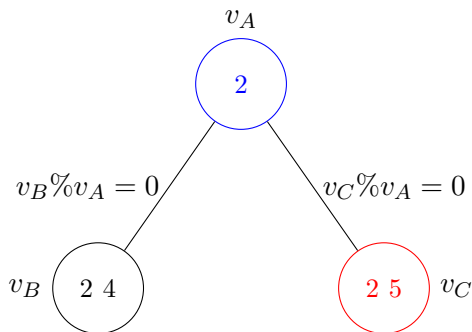
Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



$$\frac{M}{\begin{array}{l} (v_B, v_A) \\ (v_C, v_A) \end{array}}$$

AC-3: Example

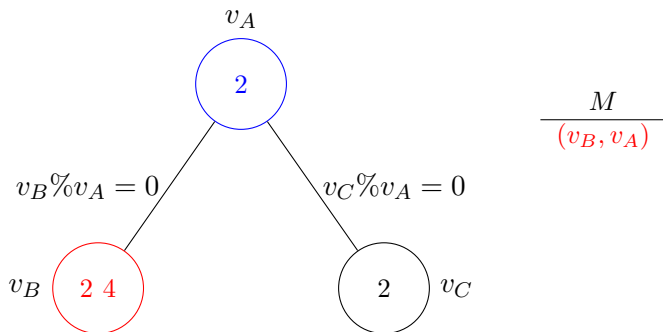
Example: $(y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



$$\frac{M}{\begin{matrix} (v_B, v_A) \\ (v_C, v_A) \end{matrix}}$$

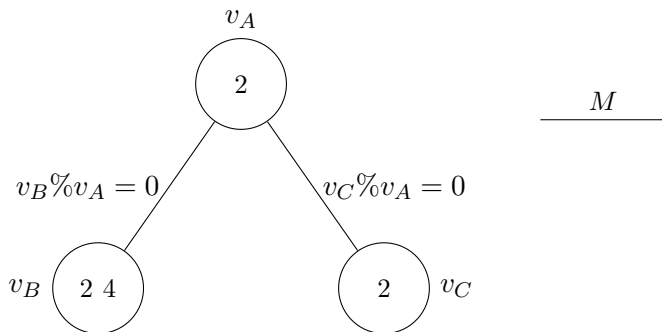
AC-3: Example

Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



AC-3: Example

Example: ($y \% x = 0$: y modulo x is 0, i.e., y can be divided by x)



AC-3: Runtime

Theorem (Runtime of AC-3). *Let $\gamma = (V, D, C)$ be a constraint network with m constraints, and maximal domain size k . Then $AC-3(\gamma)$ runs in time $O(mk^3)$.*

Proof. Each call to $Revise(\gamma, u, v)$ takes time $O(k^2)$ so it suffices to prove that at most $O(mk)$ of these calls are made.

The number of calls to $Revise(\gamma, u, v)$ is the number of iterations of the while-loop, which is at most the number of insertions into M . Consider any constraint C_{uv} .

Two variable pairs corresponding to C_{uv} are inserted in the for-loop. In the while loop, if a pair corresponding to C_{uv} is inserted into M , then beforehand the domain of either u or v was reduced, which happens at most $2k$ times. Thus we have $O(k)$ insertions per constraint, and $O(mk)$ insertions overall, as desired.

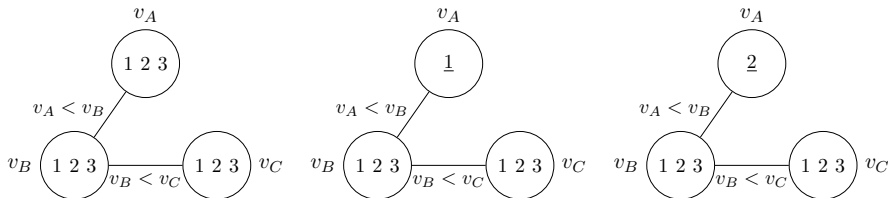
Arbitrary AC(γ): Unique Outcome

Theorem (Outcome of AC(γ)). *For a given constraint network γ , there is a unique γ' which is the outcome of every process (in particular, every possible execution of AC-1 or AC-3) that removes variable domain values until arc consistency is first achieved.*

Proof. First, observe that the outcome γ' of any such process is (1) a sub-network of γ , i.e., is identical with γ except for reduced variable domains, which in what follows we denote by $\gamma' \subseteq \gamma$; is (2) arc-consistent; and is (3) maximal with respect to (1) and (2), i.e., there exists no γ'' with (1) and (2) such that $\gamma' \subsetneq \gamma'' \subseteq \gamma$ (because, if that were the case, the process would have stopped at γ'' already).

Now the claim follows because there exists exactly one maximal arc-consistent sub-network of γ . Assume to the contrary that there are two such sub-networks, $\gamma' \neq \gamma''$. Consider the network $\gamma' \cup \gamma''$ constructed by setting each variable domain to the union of those in γ' and γ'' . Obviously, $\gamma' \subsetneq \gamma' \cup \gamma''$ and $\gamma'' \subsetneq \gamma' \cup \gamma''$. Furthermore, $\gamma' \cup \gamma''$ is arc-consistent: for each constraint C_{uv} , every $d \in D_u$ is compatible with a value $d' \in D_v$ coming from either of γ' or γ'' . So $\gamma' \cup \gamma''$ is a larger arc-consistent sub-network, in contradiction.

Questionnaire



Question!

Which inferences does enforcing arc consistency make, for each of these partial assignments?

→ Left: $\text{Revise}(v_B, v_C)$ reduces D_{v_B} to $\{1, 2\}$, $\text{Revise}(v_B, v_A)$ then reduces it to $\{2\}$. From here, $\text{Revise}(v_A, v_B)$ and $\text{Revise}(v_C, v_B)$ reduce each domain to a singleton. Thus **enforcing arc consistency solves this network**.

→ Middle: Same. (Special case of Left).

→ Right: $\text{Revise}(v_B, v_C)$, $\text{Revise}(v_B, v_A)$ reduces D_{v_B} to \emptyset . Thus **enforcing arc consistency determines that this partial assignment cannot be extended to a solution**. (In contrast to Forward Checking, cf. slide 19.)

Reminder: The Big Picture

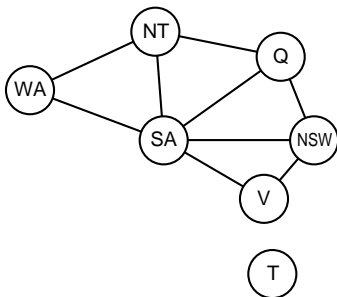
- Say γ is a constraint network with n variables and maximal domain size k . To solve γ , k^n total assignments must be tested in the worst case.
- **Inference:** One method to try to avoid, or at least ameliorate, this explosion in practice.
 - Often, from an assignment to some variables, we can easily make inferences regarding other variables.
- **Decomposition:** Another method to try to avoid, or at least ameliorate, this explosion in practice.
 - Often, we can exploit the *structure* of a network to *decompose* it into smaller parts that are easier to solve.

→ What is “structure”, and how to “decompose”?

“Structure”: Constraint Graphs

Definition (Constraint Graph). Let $\gamma = (V, D, C)$ be a constraint network. The *constraint graph* of γ is the undirected graph whose vertices are the variables V , and that has an arc $\{u, v\}$ if and only if $C_{uv} \in C$.

Example “Coloring Australia”:



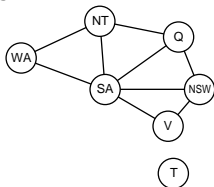
“Decomposition” 1.0: Disconnected Constraint Graphs

Theorem (Disconnected Constraint Graphs). Let $\gamma = (V, D, C)$ be a constraint network. Let a_i be a solution to each connected component V_i of the network's constraint graph. Then $a := \bigcup_i a_i$ is a solution to γ .

Proof. a satisfies all C_{uv} where u and v are inside the same connected component. The latter is the case for all C_{uv} .

→ If two parts of γ are not connected, then they are independent.

Examples:



→ Color Tasmania separately.

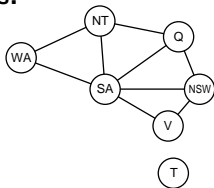
- γ with $n = 40$ variables, each domain size $k = 2$. Four separate connected components each of size 10.
- Worst-case without decomposition: 2^{40} . Worst-case with decomposition: $4 * 2^{10}$. Gain: 2^{28} .

“Decomposition” 2.0: Acyclic Constraint Graphs

Theorem (Acyclic Constraint Graphs). Let $\gamma = (V, D, C)$ be a constraint network with n variables and maximal domain size k , whose constraint graph is acyclic. Then we can find a solution for γ , or prove γ to be inconsistent, in time $O(nk^2 + n^2k)$. (Proof: Next slide.)

→ Constraint networks with acyclic constraint graphs can be solved in low-order polynomial time.

Examples:



→ Not acyclic. But: see next section.

- γ with $n = 40$ variables, each domain size $k = 2$. Acyclic constraint graph.
- Worst-case without decomposition: 2^{40} . Worst-case with decomposition (see next slide): $40^2 + 40 + 40 * 2^2 + 40^2 * 2 < 2^{13}$. Gain: $> 2^{27}$.

The AcyclicCG Algorithm

Algorithm: $\text{AcyclicCG}(\gamma)^1$

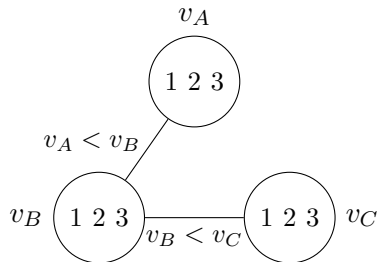
- ① Obtain a directed tree from γ 's constraint graph, picking an arbitrary variable v as the root, and directing arcs outwards.
- ② Order the variables topologically, i.e., such that each vertex is ordered before its children; denote that order by v_1, \dots, v_n .
- ③ **for** $i := n, n-1, \dots, 2$ **do**:
 - ① $\text{Revise}(\gamma, v_{\text{parent}(i)}, v_i)$.
 - ② **if** $D_{v_{\text{parent}(i)}} = \emptyset$ **then return** “inconsistent”
→ Now, every variable is arc consistent relative to its children.
- ④ Run $\text{BacktrackingWithInference}$ with forward checking, using the variable order v_1, \dots, v_n . → This will find a solution without having to backtrack.
(Proof: Possible exercise for you)

→ Runtime: 1. $O(n^2)$, 2. $O(n)$ given tree data structure, 3. $O(nk^2)$, 4. $O(n^2k)$ using incremental computation for forward checking (cf. slide 18).

¹We assume that γ 's constraint graph is connected. If it is not, do the following for each connected component separately.

AcyclicCG(γ): Example

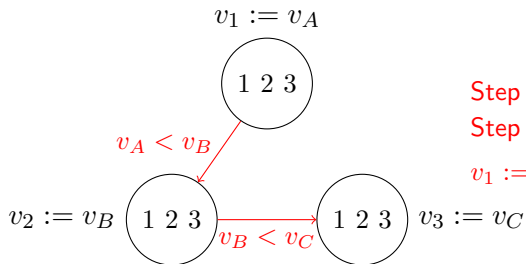
Example AcyclicCG() execution:



Input network γ

AcyclicCG(γ): Example

Example AcyclicCG() execution:



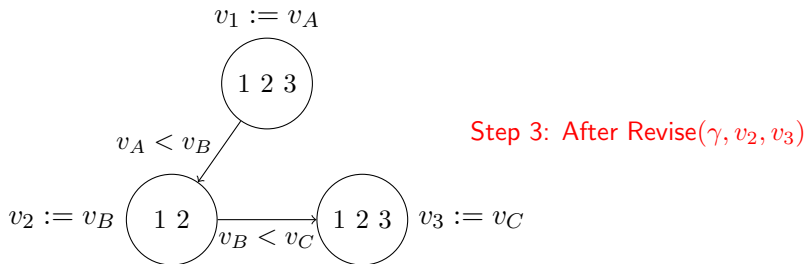
Step 1: Directed tree for root v_A .

Step 2: Order

$v_1 := v_A, v_2 := v_B, v_3 := v_C$

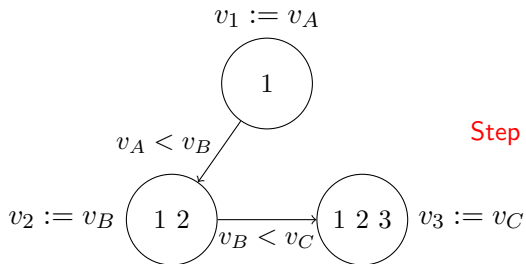
AcyclicCG(γ): Example

Example AcyclicCG() execution:



AcyclicCG(γ): Example

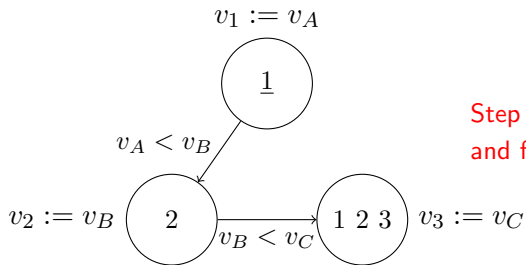
Example AcyclicCG() execution:



Step 3: After Revise(γ, v_1, v_2)

AcyclicCG(γ): Example

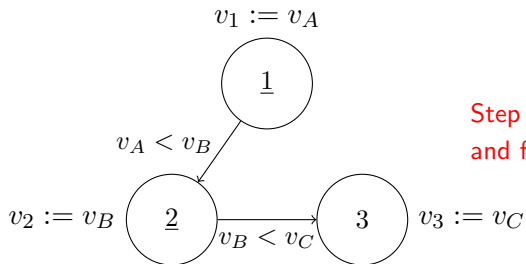
Example AcyclicCG() execution:



Step 4: After $a(v_1) := 1$
and forward checking

AcyclicCG(γ): Example

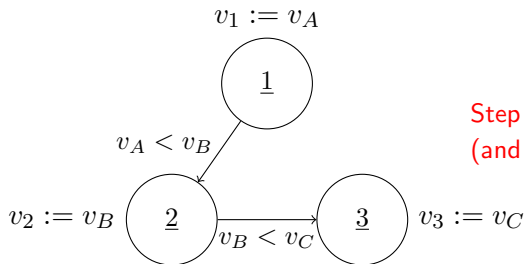
Example AcyclicCG() execution:



Step 4: After $a(v_2) := 2$
and forward checking

AcyclicCG(γ): Example

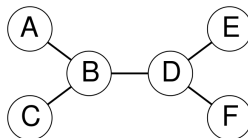
Example AcyclicCG() execution:



Step 4: After $a(v_3) := 3$
(and forward checking)

Questionnaire

Constraint graph of γ :



Question!

How many different directed trees can we obtain/how many calls to `Revise()` are done for each?

(A): 6 / 5

(B): 4 / 5

(C): 24 / 5

(D): 6 / Between 4 and 6

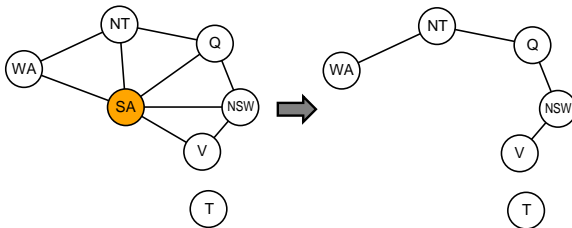
→ (A) is correct. Any vertex can be picked as the root, and once the root is picked the directed tree is unique. The number of calls to `Revise()` is the number of arcs in the tree and hence always is the number of arcs in the original constraint graph. Example:



→ `Revise(D, F)`, `Revise(D, E)`, `Revise(B, D)`,
`Revise(B, C)`, `Revise(A, B)`.

“Almost” Acyclic Constraint Graphs

Example “Coloring Australia”:



Cutset Conditioning: Idea

- ① Choose the variable order so that removing the first d variables renders the constraint graph acyclic.
→ Then we won't have to search deeper than d , because:
- ② Recursive call of backtracking on a s.t. the sub-graph of the constraint graph induced by $\{v \in V \mid a(v) \text{ is undefined}\}$ is acyclic:
→ We can solve the remaining sub-problem with `AcyclicCG()`.

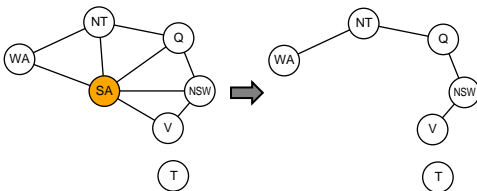
“Decomposition” 3.0: Cutset Conditioning

Definition (Cutset). Let $\gamma = (V, D, C)$ be a constraint network, and $V_0 \subseteq V$. V_0 is a **cutset** for γ if the sub-graph of γ 's constraint graph induced by $V \setminus V_0$ is acyclic. V_0 is **optimal** if its size is minimal among all cutsets for γ .

```
 $V_0 :=$  a cutset; return CutsetConditioning( $\gamma, V_0, \emptyset$ )  
function CutsetConditioning( $\gamma, V_0, a$ ) returns a solution, or “inconsistent”  
   $\gamma' :=$  a copy of  $\gamma$ ;  $\gamma' :=$  ForwardChecking( $\gamma', a$ )  
  if ex.  $v$  with  $D'_v = \emptyset$  then return “inconsistent”  
  if ex.  $v \in V_0$  s.t.  $a(v)$  is undefined then select such  $v$   
    else  $a' :=$  AcyclicCG( $\gamma'$ ); if  $a' \neq$  “inconsistent” then return  $a \cup a'$   
    else return “inconsistent”  
  
  for each  $d \in$  copy of  $D'_v$  in some order do  
     $a' := a \cup \{v = d\}$ ;  $D'_v := \{d\}$ ;  
     $a'' :=$  CutsetConditioning( $\gamma', V_0, a'$ )  
    if  $a'' \neq$  “inconsistent” then return  $a''$   
  return “inconsistent”
```

- Forward Checking required so that $a \cup a'$ is consistent in γ .
- Runtime is exponential only in $|V_0|$, not in $|V|$...!
- Finding optimal cutsets is **NP-hard**, but practical approximations exist.

Questionnaire



Question!

With $V_0 = \{SA\}$, how many recursive calls to `CutsetConditioning()` are made / how many calls of `Revise()` are made?

(A): 1 / 4

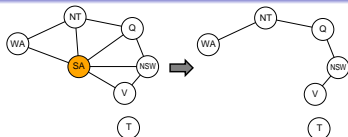
(B): 2 / 4

(C): 3 / 12

(D): 4 / 12

→ (B) is correct. The first call to `CutsetConditioning()` is with empty a . The second call with some color assigned to SA ; the remaining sub-problem is solvable so `AcyclicCG()` returns a solution and the algorithm stops. The single call to `AcyclicCG()` uses 4 calls to `Revise()`: the number of arcs, cf. slide 38.

The Example in Detail

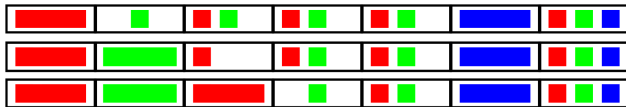


Algorithm trace: with $V_0 = \{SA\}$

- Say CutsetConditioning paints SA blue. After forward checking:



- Say WA is the root and our order is WA, NT, Q, NSW, V, T .
- Calls of `Revise()` from children to parents: No values are removed.
- Backtracking with forward checking, when choosing to paint WA red:



etc. ...

Summary

- γ and γ' are **equivalent** if they have the same solutions. γ' is **tighter than** γ if it is more constrained.
- **Inference** tightens γ without losing equivalence, during backtracking. This reduces the amount of search needed; that benefit must be traded off against the runtime overhead for making the inferences.
- **Forward checking** removes values conflicting with an assignment already made.
- **Arc consistency** removes values that do not comply with any value still available at the other end of a constraint. This subsumes forward checking.
- The **constraint graph** captures the dependencies between variables. Separate connected components can be solved independently. Networks with **acyclic constraint graphs** can be solved in low-order polynomial time.
- A **cutset** is a subset of variables removing which renders the constraint graph acyclic. **Cutset decomposition** backtracks only on such a cutset, and solves a sub-problem with acyclic constraint graph at each search leaf.

Topics We Didn't Cover Here

- **Path consistency:** Generalizes arc consistency to size- k subsets of variables.
- **Tree decomposition:** Instead of instantiating variables until the leaf nodes are trees, distribute the variables and constraints over sub-CSPs whose connections form a tree.
- **Backjumping:** Like backtracking, but with ability to back up *across several levels* (to a previous assignment identified to be responsible for failure).
- **No-Good Learning:** Inferring additional constraints based on information gathered during backtracking.
- **Local search:** In space of total (but not necessarily consistent) assignments.
- **Tractable CSP:** Classes of CSPs that can be solved in polynomial time.
- **Global Constraints:** Constraints over many/all variables, with associated specialized inference methods.
- **Constraint Optimization Problems (COP):** Utility function over solutions, need an optimal one.

Reading

- *Chapter 6: Constraint Satisfaction Problems*, Sections 6.2, 6.3.2, and 6.5 [Russell and Norvig (2010)].

Content: Compared to our treatment of the topic “Constraint Satisfaction Problems”, RN covers much more material, but less formally and in much less detail (in particular, my slides contain many additional in-depth examples). Nice background/additional reading, can’t replace the lecture.

Section 6.3.2: Somewhat comparable to my “Inference” (except that equivalence and tightness are not made explicit in RN) together with “Forward Checking”.

Section 6.2: Similar to my “Arc Consistency”, less/different examples, much less detail, additional discussion of path consistency and global constraints.

Section 6.5: Similar to my “Decomposition: Constraint Graphs, and Two Simple Cases” and “Cutset Conditioning”, less/different examples, much less detail, additional discussion of tree decomposition.

References I

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Third Edition)*. Prentice-Hall, Englewood Cliffs, NJ, 2010.