Artificial Intelligence

8. CSP, Part II: Inference and Decomposition Methods

How to *Efficiently* Satisfy All These Constraints

Prof Sara Bernardini
bernardini@diag.uniroma1.it
www.sara-bernardini.com

SAPIENZA UNIVERSITÀ DI ROMA

Autumn Term

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 5 and 6.

- **Chapter 7:** 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).

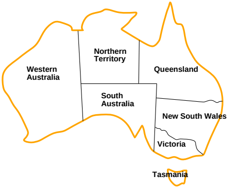
- **This Chapter:** Inference and decomposition methods.

  → Inference reduces the search space of backtracking. Decomposition methods break the problem into smaller pieces. Both are crucial for efficiency in practice.

Illustration: Inference

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

Constraint network γ:



→ An additional constraint we can add without losing any solutions?

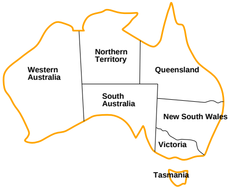
For example, $C_{WA\,Q} :=$ "$=$". If WA and Q are assigned different colors, then NT must be assigned the 3rd color, leaving no color for SA.

Illustration: Decomposition

→ 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 below.

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.

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 a state of the art 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.

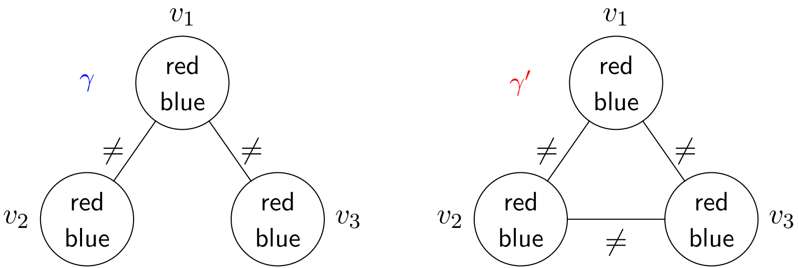
It's what you do all the time when playing SuDoKu:

	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

$\rightarrow$ 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 every solution of γ is a solution of γ', and every solution of γ' is a solution of γ.*

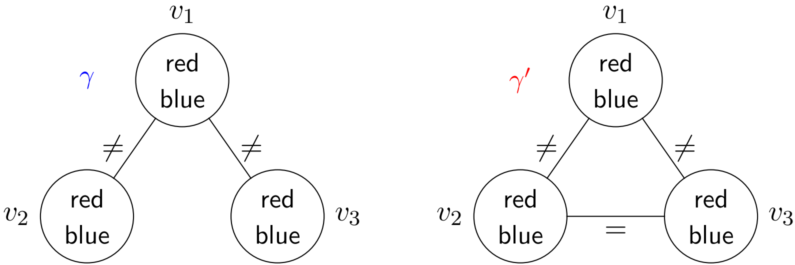


Are these constraint networks equivalent? No.

$\rightarrow$ 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 every solution of* γ *is a solution of* γ', *and every solution of* γ' *is a solution of* γ.



Are these constraint networks equivalent? Yes.

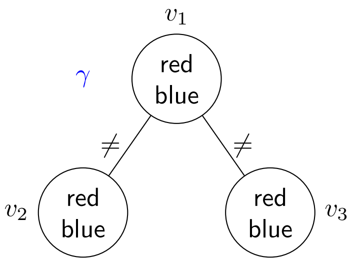
→ Equivalence: "γ' has the same solutions as γ".

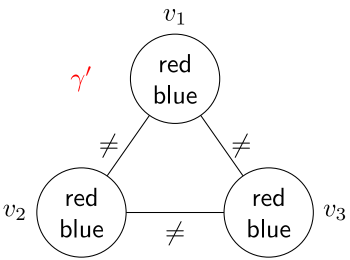
Tightness

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:*

1. *For all $v \in V$: $D'_v \subseteq D_v$.*
2. *For all $u \neq v \in V$: either $C_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.*

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





Here, we do have $\gamma' \sqsubseteq \gamma$.

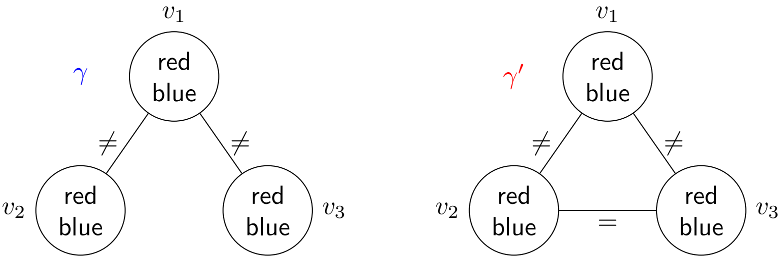
→ Tightness: "γ' has the same constraints as γ, plus some others."

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:*

1. *For all $v \in V$: $D'_v \subseteq D_v$.*
2. *For all $u \neq v \in V$: either $C_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.*

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



Here, we do have $\gamma' \sqsubseteq \gamma$.

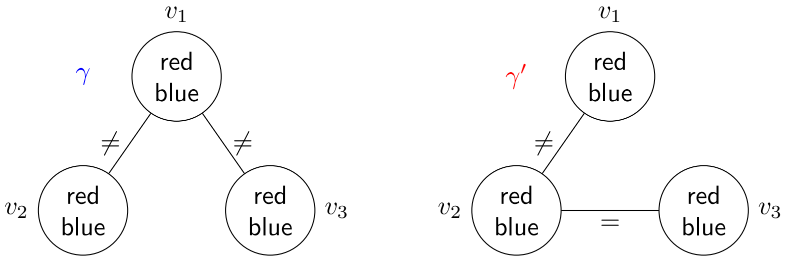
→ Tightness: "γ' has the same constraints as γ, plus some others."

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:*

1. *For all $v \in V$: $D'_v \subseteq D_v$.*
2. *For all $u \neq v \in V$: either $C_{uv} \notin C$ or $C'_{uv} \subseteq C_{uv}$.*

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



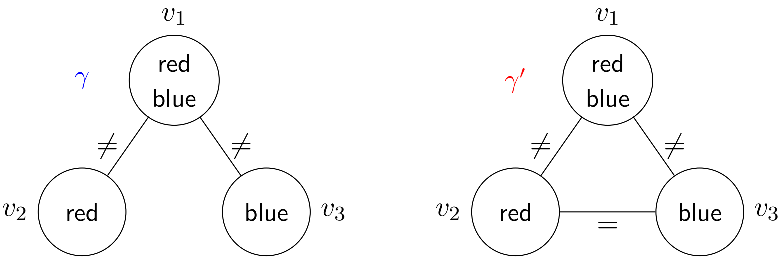
Here, we do not have $\gamma' \sqsubseteq \gamma$.

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

Equivalence + Tightness = Inference

Theorem. *Let γ and γ' be constraint networks s.t. $\gamma' \equiv \gamma$ and $\gamma' \sqsubseteq \gamma$. Then, γ' has the same solutions as γ but fewer consistent partial assignments than γ.*

$\rightarrow$ γ' is a better encoding of the underlying problem.



$\rightarrow$ 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.

- Encode partial assignment as unary constraints (i.e., for $a(v) = d$, set the unary constraint $D_v := \{d\}$), so that inference reasons about *the network restricted to the commitments already made*.

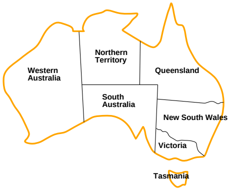
Backtracking With Inference

```
function BacktrackingWithInference(γ, a) returns a solution, or "inconsistent"
  if a is inconsistent then return "inconsistent"
  if a is a total assignment then return a
  γ' := a copy of γ      /* γ' = (V, D', C') */
  γ' := Inference(γ')
  if exists v with D'_v = ∅ then return "inconsistent"
  select some variable v for which a is not defined
  for each d ∈ copy of D'_v in some order do
     a' := a ∪ {v = d}; D'_v := {d}      /* makes a explicit as a constraint */
     a'' := BacktrackingWithInference(γ', a')
     if a'' ≠ "inconsistent" then return a''
  return "inconsistent"
```

- Inference(): Any procedure delivering a (tighter) equivalent network.
- Inference typically prunes domains; indicate unsolvability by $D'_v = \emptyset$.
- 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): $C_{WA\,Q} :=$ "$\neq$" (B): $C_{WA\,Q} :=$ "$=$"

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

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

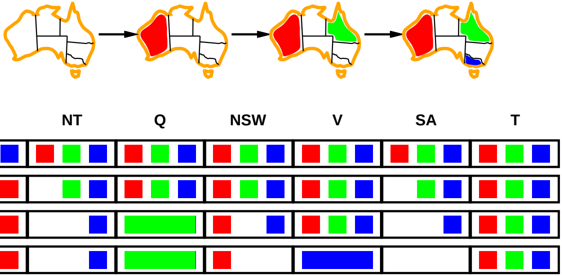
$\rightarrow$ (A): No. There are solutions in which WA and Q have the same value.

$\rightarrow$ (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(γ, a) **returns** modified γ
>   **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** γ



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.*
  $\rightarrow$ Recall here that the partial assignment a is represented as unary constraints inside γ.
- Incremental computation: Instead of the first for-loop, use only the 2nd one every time a new assignment $a(v) = d'$ is added.

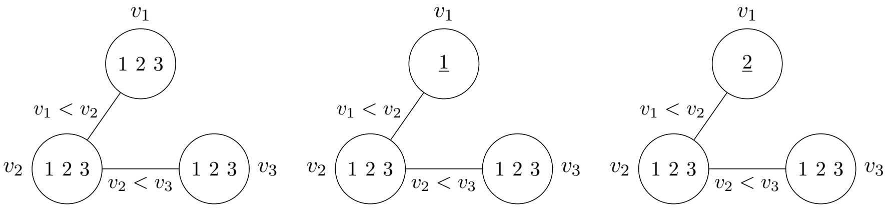
Practice:

- Cheap but useful inference method.
- Rarely a good idea to not use forward checking (or a stronger inference method subsuming it).

$\rightarrow$ 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.

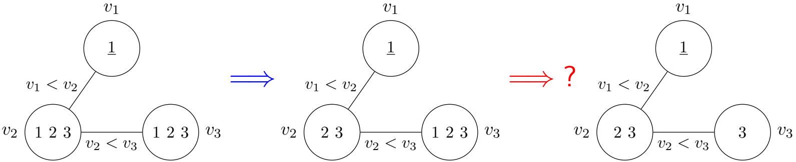


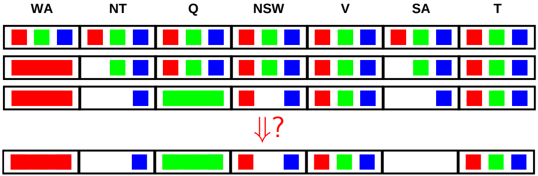
Question!

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

$\rightarrow$ Left: None, as there are no assignments. Middle: $D_{v_2} := \{2, 3\}$ then stop. Right: $D_{v_2} := \{3\}$ 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) forward checking is the most canonical method and already is enough for many purposes, see slides 35 and 40.)

When Forward Checking is Not Good Enough





$\rightarrow$ Forward checking propagates information only "from assigned to unassigned" variables. No propagation between unassigned variables.

Arc Consistency: Definition

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

(i) *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 $(d, d') \in C_{uv}$.*

(ii) *The network γ is arc consistent if every variable $u \in V$ is arc consistent relative to every other variable $v \in V$.*

$\rightarrow$ Arc consistency = for every domain value and constraint, at least one value on the other side of the constraint "works".

$\rightarrow$ Note the asymmetry between u and v: arc consistency is "directed".

Examples: (previous slide)

- On top, middle, is v_3 arc consistent relative to v_2? No. For values 1 and 2, D_{v_2} does not have a value that works.
- And on the right? Yes. (But v_2 is not arc consistent relative to v_3.)
- SA is not arc consistent relative to NT in the Australia example, 3rd row.

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(γ))

- *AC(γ) is sound: guarantees to deliver an equivalent network.*

  $\rightarrow$ 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(γ) subsumes forward checking: AC(γ) $\sqsubseteq$ ForwardChecking(γ).* (Recall from slide 11 that $\gamma' \sqsubseteq \gamma$ means γ' is tighter than γ.)

  $\rightarrow$ 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.

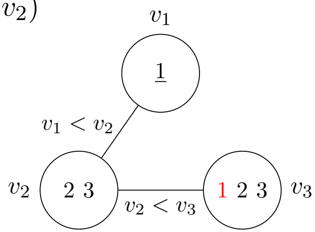
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v:

function Revise(γ, u, v) **returns** modified γ
  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 γ

→ 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_3, v_2)



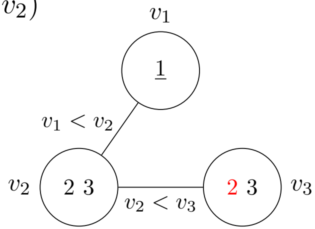
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v:

> **function** Revise(γ, u, v) **returns** modified γ
>   **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** γ

$\rightarrow$ 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_3, v_2)



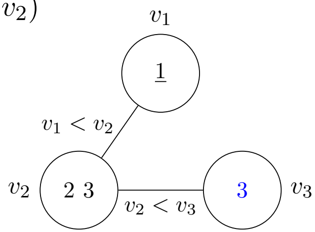
Enforcing Arc Consistency for *One* Pair of Variables

Algorithm enforcing consistency of u relative to v:

function Revise(γ, u, v) **returns** modified γ
  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 γ

$\rightarrow$ 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_3, v_2)



AC-1

Idea: Apply pairwise revisions up to a fixed point.

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

- Obviously, this does indeed enforce 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

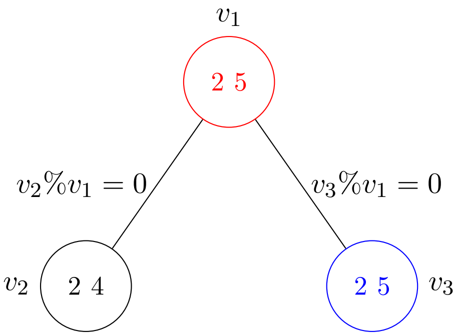
Idea: Remember the potentially inconsistent variable pairs.

function AC-3(γ) **returns** modified γ
  $M := \emptyset$
  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(γ, u, v)
    if D_u has changed in the call to Revise **then**
      for each constraint $C_{\{w,u\}} \in C$ where $w \neq v$ **do**
        $M := M \cup \{(w, u)\}$
  return γ

- 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 "where $w \neq v$"? v is the *reason* why D_u just changed. Thus, if v was arc consistent relative to u before, that continues to hold: the values just removed from D_u did not match any values from D_v anyway.

AC-3: Example

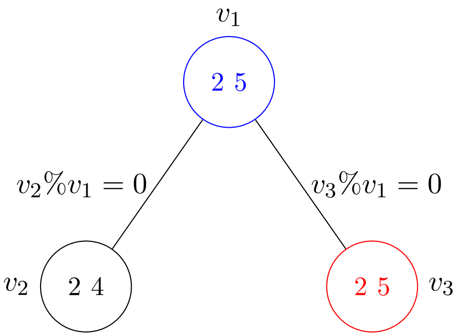
Example: ($y\%x = 0$: y modulo x is 0, i.e., y can be divided by x)



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

AC-3: Example

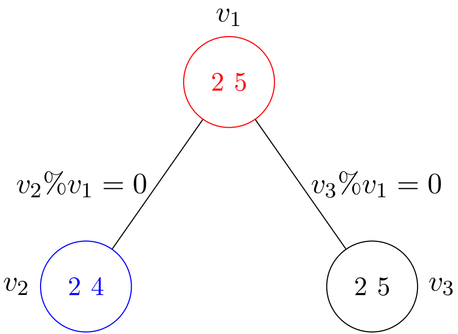
Example: ($y\%x = 0$: y modulo x is 0, i.e., y can be divided by x)



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

AC-3: Example

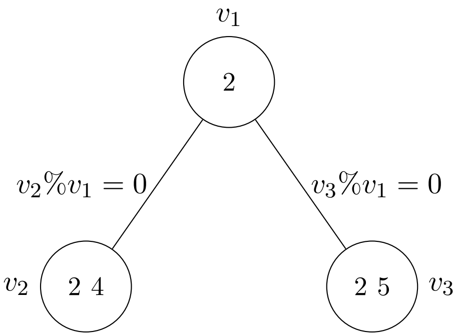
Example: ($y\%x = 0$: y modulo x is 0, i.e., y can be divided by x)



$$\begin{array}{c} M \\ \hline (v_2, v_1) \\ (v_1, v_2) \end{array}$$

AC-3: Example

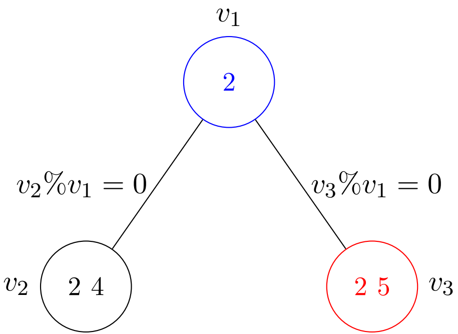
Example: ($y\%x = 0$: y modulo x is 0, i.e., y can be divided by x)



M
(v_2, v_1)
(v_3, v_1)

AC-3: Example

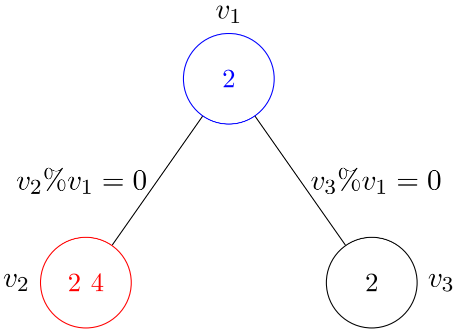
Example: ($y\%x = 0$: y modulo x is 0, i.e., y can be divided by x)



M
(v_2, v_1)
(v_3, v_1)

AC-3: Example

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

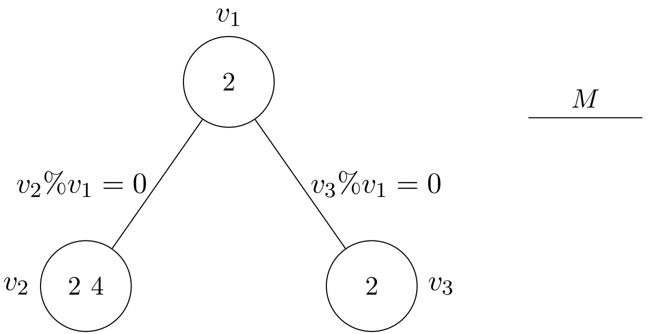


M

(v_2, v_1)

AC-3: Example

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



AC-3: Runtime

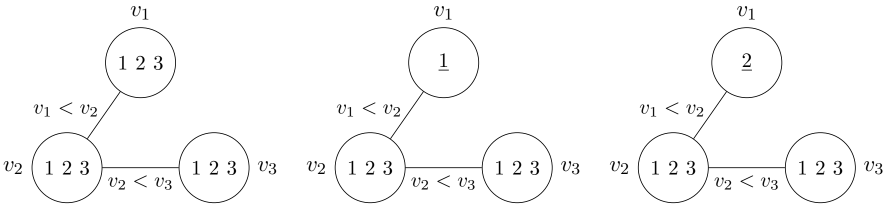
Theorem (Runtime of AC-3). *Let $\gamma = (V, D, C)$ be a constraint network with m constraints, and maximal domain size k. Then AC-3(γ) runs in time $O(mk^3)$.*

Proof. Each call to Revise(γ, 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(γ, 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.

Questionnaire



Question!

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

→ Left: Revise$(2,3)$ reduces D_{v_2} to $\{1,2\}$, Revise$(2,1)$ then reduces it to $\{2\}$. From here, Revise$(1,2)$ and Revise$(3,2)$ reduce each domain to a singleton. Thus enforcing arc consistency solves this network.

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

→ Right: Revise$(2,3)$, Revise$(2,1)$ reduces D_{v_2} 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.

  $\rightarrow$ *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.

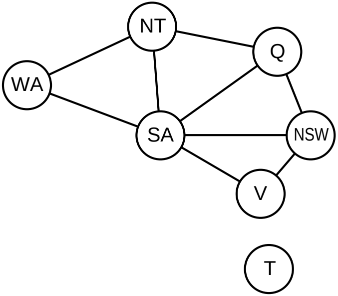
  $\rightarrow$ *Often, we can exploit the structure of a network to decompose it into smaller parts that are easier to solve.*

$\rightarrow$ 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":



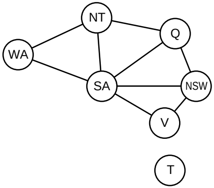
"Decomposition" 1.0: Disconnected Constraint Graphs

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}.

$\rightarrow$ If two parts of γ are not connected, then they are independent.

Examples:



$\rightarrow$ Color Tasmania separately.

- γ with $n = 40$ variables, each domain size $k = 2$. Four separate connected components each of size 10.
- Reduction of worst-case when using decomposition:

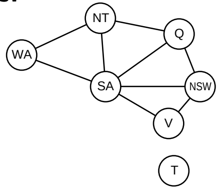
  $\rightarrow$ No decomposition: 2^{40}. With decomposition: $4 * 2^{10}$. Gain: 2^{28}.

"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 $O(nk^2)$.* (Proof: See next slide.)

$\rightarrow$ Constraint networks with acyclic constraint graphs can be solved in (low-order) polynomial time.

Examples:



$\rightarrow$ Not acyclic. But: see next section.

- γ with $n = 40$ variables, each domain size $k = 2$. Acyclic constraint graph.
- Reduction of worst-case when using decomposition:

  $\rightarrow$ No decomposition: 2^{40}. With decomposition: $40 * 2^2$. Gain: $> 2^{32}$.

Acyclic Constraint Graphs: How To

Algorithm: AcyclicCG(γ)

1. Obtain a directed tree from γ's constraint graph, picking an arbitrary variable v as the root, and directing arcs outwards.[1]
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. Revise($\gamma, v_{parent(i)}, v_i$).
   2. **if** $D_{v_{parent(i)}} = \emptyset$ **then return** "inconsistent"

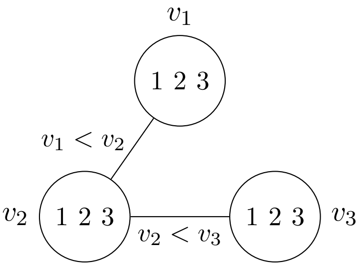
   $\rightarrow$ Now, every variable is arc consistent relative to its children.
4. Run BacktrackingWithInference with forward checking, using the variable order $v_1, \dots, v_n$.

   $\rightarrow$ This algorithm will find a solution without ever having to backtrack!

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

AcyclicCG(γ): Example

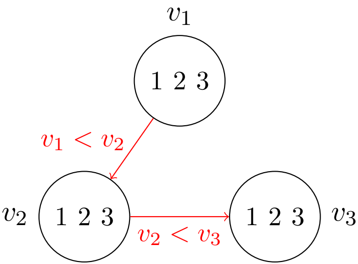
Example AcyclicCG() execution:



Input network γ.

AcyclicCG(γ): Example

Example AcyclicCG() execution:

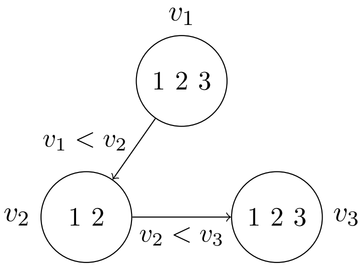


Step 1: Directed tree for root v_1.

Step 2: Order v_1, v_2, v_3.

AcyclicCG(γ): Example

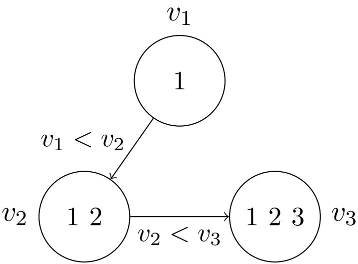
Example AcyclicCG() execution:



Step 3: After Revise(γ, v_2, v_3).

AcyclicCG(γ): Example

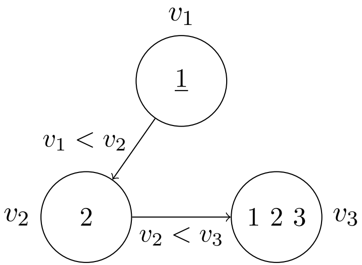
Example AcyclicCG() execution:



Step 3: After Revise(γ, v_1, v_2).

AcyclicCG(γ): Example

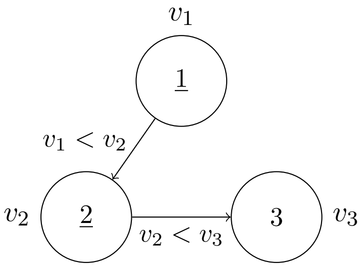
Example AcyclicCG() execution:



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

AcyclicCG(γ): Example

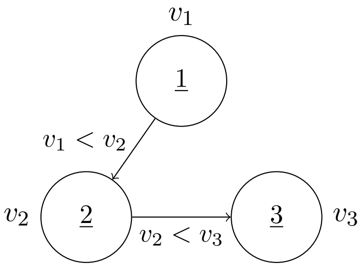
Example AcyclicCG() execution:



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

AcyclicCG(γ): Example

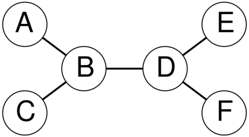
Example AcyclicCG() execution:



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

Questionnaire

Constraint graph of γ:



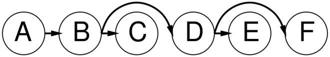
Question!

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

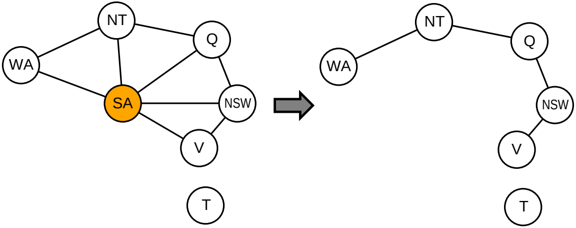
$\rightarrow$ (A) is correct. Any vertex can be picked as the root, and once the root is picked the directed tree is unique. The number of calls to Revise() is always the number of arcs in the original constraint graph. Example:



$\rightarrow$ 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.
   $\rightarrow$ 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:
   $\rightarrow$ We can solve the remaining sub-problem with AcyclicCG().

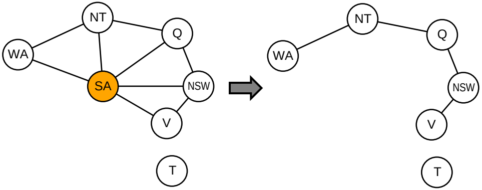
"Decomposition" 3.0: Cutset Conditioning

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.*

```
V0 := a cutset; return CutsetConditioning(γ, V0, ∅)
function CutsetConditioning(γ, V0, a) returns a solution, or "inconsistent"
  γ' := a copy of γ; γ' := ForwardChecking(γ', a)
  if exists v with D'_v = ∅ then return "inconsistent"
  if exists v ∈ V0 s.t. a(v) is undefined then select such v
     else a' := AcyclicCG(γ'); if a' ≠ "inconsistent" then return a ∪ a'
                                                       else return "inconsistent"
  for each d ∈ copy of D'_v in some order do
     a' := a ∪ {v = d}; D'_v := {d};
     a'' := CutsetConditioning(γ', V0, a')
     if a'' ≠ "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

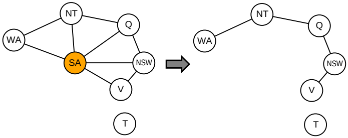
(B): 2 / 4

(C): 3 / 12

(D): 4 / 12

$\rightarrow$ (B) is correct. The first call to CutsetConditioning() is with empty a. The second call with some color for SA; no matter which color it is, 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() because that is the number of arcs (cf. slide 37).

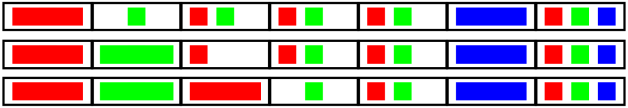
And Now in Detail . . .



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.
- Enforcing arc consistency 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 cuset, 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 ($\rightarrow$ E.g., 8-Queens).
- 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" (Chapters 7 and 8), 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.