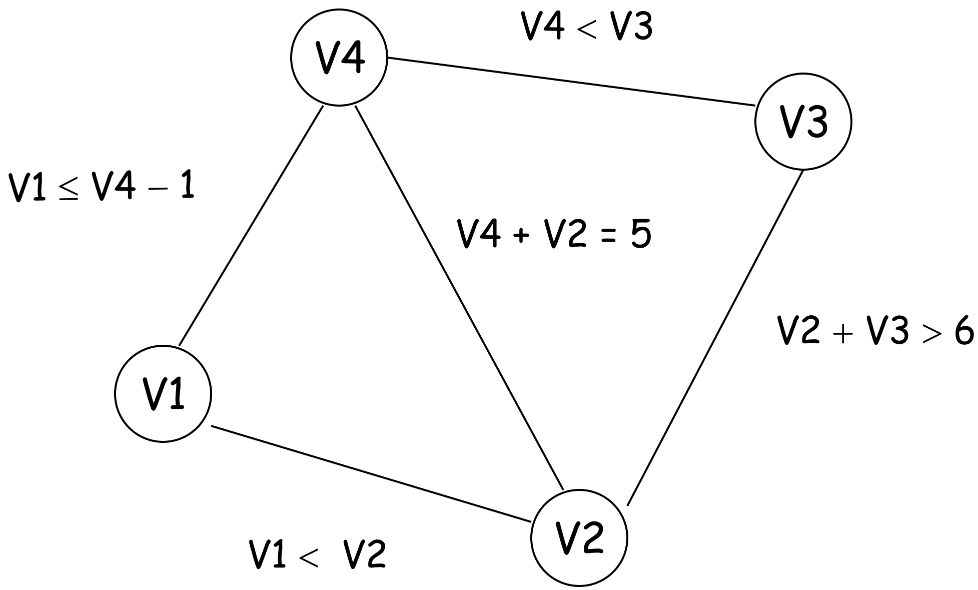
Arc consistency (ac)

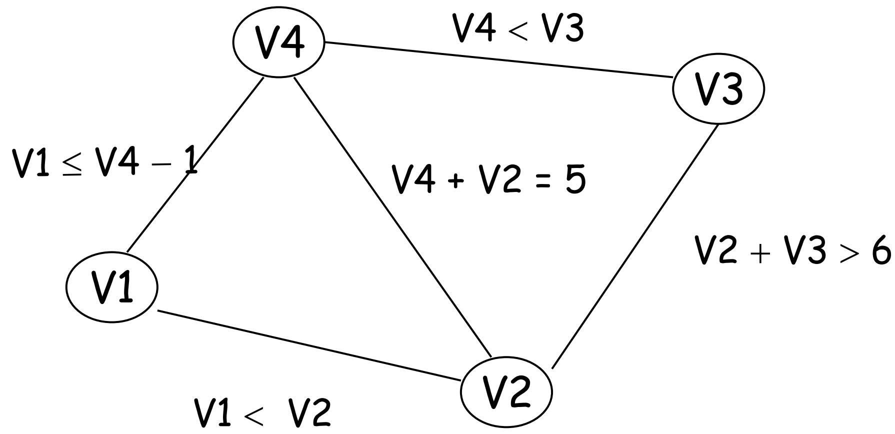
Simple algorithm: ac3 (1977)



$D_i = \{1,2,3,4,5\}$

What can you infer?

AR33 figure 18, page 35



Di = {1,2,3,4,5}

D1 = {1,2}
D2 = {2,3}
D3 = {4,5}
D4 = {2,3}

Was that easy?

Do you agree?

Here's the reasoning

V1 < V2

V4 < V3

V2 + V3 > 6

V1 ≤ V4 – 1

V4 + V2 = 5

Di = {1,2,3,4,5}

V1 < V2	D1 = {1,2,3,4,5}	D2 = {1,2,3,4,5}	D1 := {1,2,3,4}
V2 > V1	D1 = {1,2,3,4}	D2 = {1,2,3,4,5}	D2 := {2,3,4,5}
V4 ≥ V1 + 1	D4 = {1,2,3,4,5}	D1 = {1,2,3,4}	D4 := {2,3,4,5}
V1 ≤ V4 - 1	D1 = {1,2,3,4}	D4 = {2,3,4,5}	no change
V2 + V3 > 6	D2 = {2,3,4,5}	D3 = {1,2,3,4,5}	no change
V3 + V2 > 6	D3 = {1,2,3,4,5}	D2 = {2,3,4,5}	D3 := {2,3,4,5}
V2 + V4 = 5	D2 = {2,3,4,5}	V4 = {2,3,4,5}	D2 = {2,3}
V1 < V2	D1 = {1,2,3,4}	D2 = {2,3}	D1 = {1,2}
V2 > V1	D2 = {2,3}	D1 = {1,2}	no change
V4 ≥ V1 + 1	D4 = {2,3,4,5}	D1 = {1,2}	no change
V3 + V2 > 6	D3 = {2,3,4,5}	D2 = {2,3}	D3 = {4,5}
V2 + V3 > 6	D2 = {2,3}	D3 = {4,5}	no change
V4 + V2 = 5	D4 = {2,3,4,5}	D2 = {2,3}	D4 = {2,3}
V1 ≤ V4 - 1	D1 = {1,2}	D4 = {2,3}	no change
V2 + V4 = 5	D2 = {2,3}	D4 = {2,3}	no change
V4 < V3	D4 = {2,3}	D3 = {4,5}	no change
V3 > V4	D3 = {4,5}	D4 = {2,3}	no change

Arc consistency: so what's that then?

A constraint Cij is arc consistent if

- for every value x in Di there exists a value y in Dj that supports x
  - i.e. if v[i] = x and v[j] = y then Cij holds
  - note: we are assuming Cij is a binary constraint

A csp (V,D.C) is arc consistent if

- every constraint is arc consistent

This is also called 2-consistency

If (V,D,C) is arc consistent then
- I can choose any variable v[i]
- assign it a value x from its domain Di
- I can now choose any other variable v[j]
- I can find a consistent instantiation for v[j] from Dj

NOTE: this is in isolation, where I have only 2 variables that I instantiate

A constraint Cij is arc consistent if

- for every value x in Di there exists a value y in Dj that supports x
  - i.e. if v[i] = x and v[j] = y then Cij holds
  - note: we are assuming Cij is a binary constraint

$$AC(C_{i,j}) = \forall x \in D_i \exists y \in D_j [C_{i,j}(x,y)]$$

A csp (V,D.C) is arc consistent if

- every constraint is arc consistent

$$AC(V,D,C) = \forall C_{i,j} \in C[AC(C_{i,j})]$$

Just because a problem (V,D,C) is arc consistent does not mean that it has a solution!

$$D_i \in \{1,2\}$$
$$V_1 \neq V_2$$
$$V_1 \neq V_3$$
$$V_2 \neq V_3$$

Arc-consistency processes a problem and removes from the domains of variables values that CANNOT occur in any solution

Arguably, it makes the resultant problem easier. Why?

The arc-consistent problem has the same set of solutions as the original problem

Note: if constraint graph is a tree, AC is a decision procedure

AC is not a decision procedure

So, is there 1-consistency?

Yip

- when we have unary constraints
- example odd(V[i])
- 1-consistency, we weed out all odd values from Di
- also called ***node-consistency (NC)***

3-consistency?

Given constraints Cij and Cjk, disallow all pairs (x,z) in the constraint Cik where there is no value y in Dj such that Cij(x,y) and Cjk(y,z)

This adds nogood tuples to an existing constraints, or creates a new constraint!

sometimes called ***path-consistency (PC)***

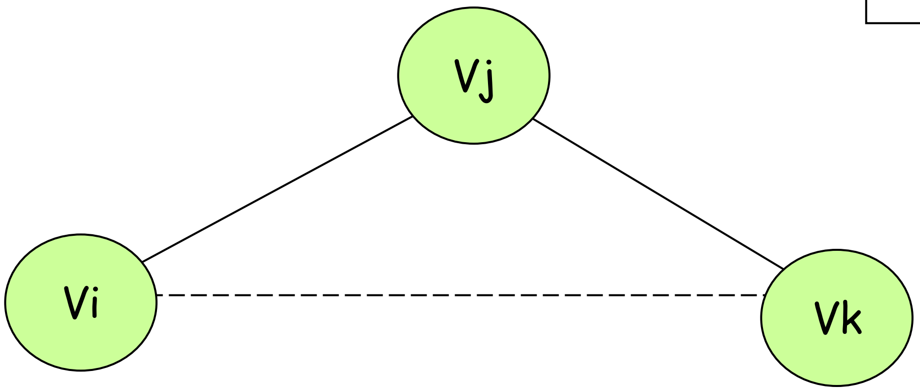
(Given 2 variables in isolation, we can instantiate those consistently, and pick any third variable and …)

Path-consistency (aka 3-consistency)

$$3Con(i,j,k): \forall x \in D_i \forall z \in D_k \exists y \in D_j [C_{i,j}(x,y) \wedge C_{i,k}(x,z) \wedge C_{j,k}(y,z)]$$

$$O(n^3 d^2)$$

M. Singh, TAI-95



It may create nogood tuples {(i/x,k/z),...}
Therefore increases size of model/problem.
May result in more constraints to check!

There might be no constraint Cik
Therefore 3-consistency may create it!

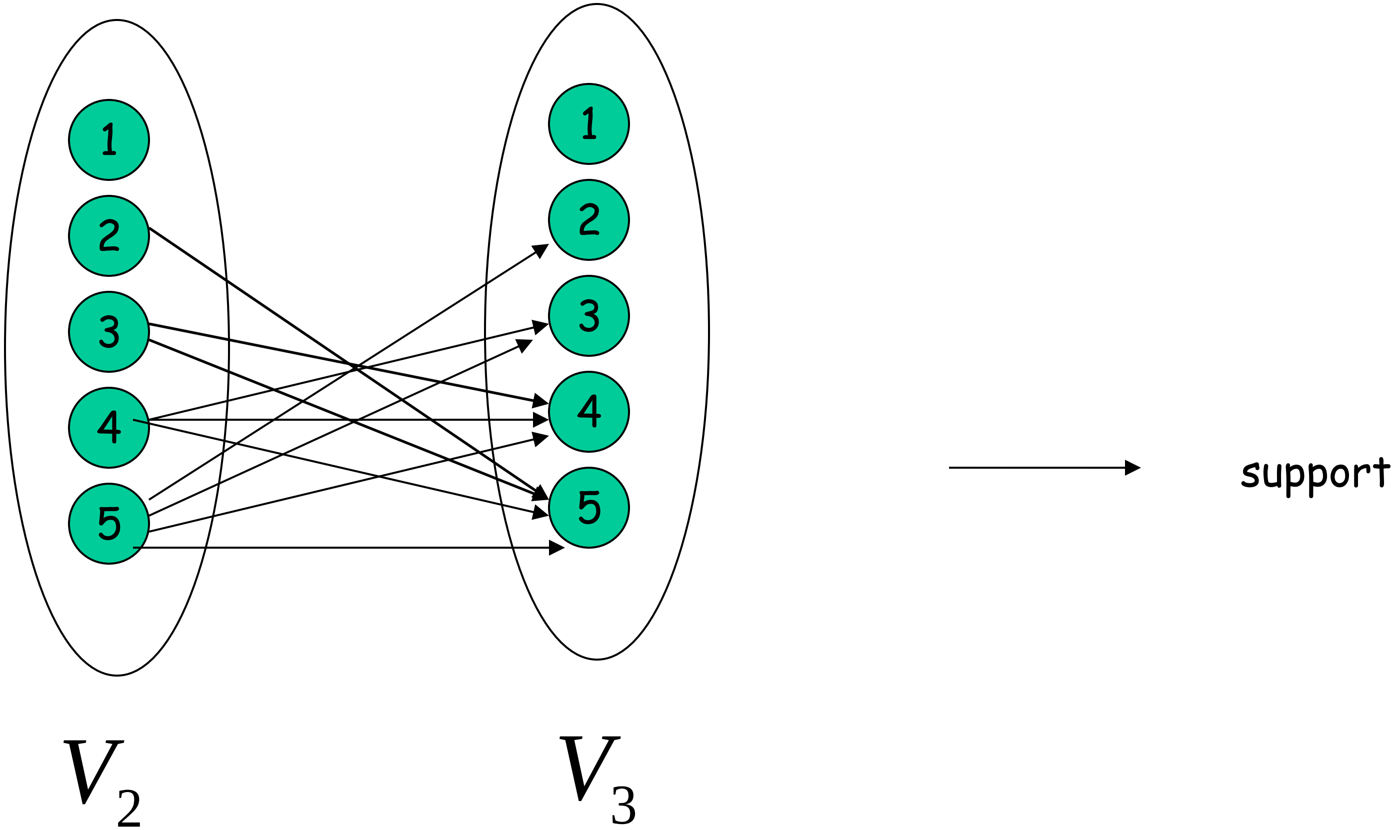
ac3: Mackworth 1977

Alan Mackworth presented ac1, ac2, and ac3 in 1977. ac1 and ac2 were "straw men"

ac3: revise a constraint (pseudo code)

Given constraint C_ij remove from the domain d_i all values that have no support in d_j

```
revise(i,j)
    revised := false
    for x in d[i]                                  // iterate over all values in d[i]
    do supported := false
        for y in d[j] while ¬supported   // find first support in d[j] for x
        do supported := check(i,x,j,y)  // is v[i]=x && v[j]=y consistent?
        if ¬supported                           // if no support, delete x from d[i]
        then d[i] := d[i] \ {x}
                revised := true                // and set revised to true
    return revised                               // delivers true or false
```



$$C_{2,3} = V_2 + V_3 > 6$$

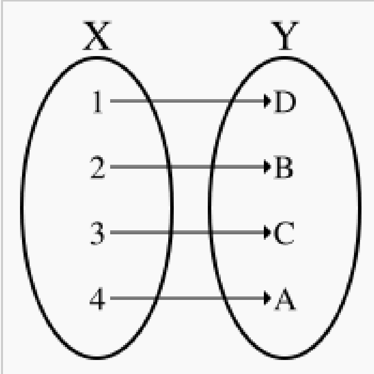
revise searches for 1st support for a value

Is it a bijection? What are implications of this?

Bijection

From Wikipedia, the free encyclopedia

In mathematics, a **bijection**, or a **bijective function** is a function *f* from a set *X* to a set *Y* with the property that, for every *y* in *Y*, there is exactly one *x* in *X* such that
$f(x) = y$.



A bijective function.

Alternatively, *f* is bijective if it is a **one-to-one correspondence** between those sets; i.e., both **one-to-one** (injective) and **onto** (surjective).[1] (See also Bijection, injection and surjection.)

For example, consider the function succ, defined from the set of integers $\mathbb{Z}$ to $\mathbb{Z}$, that to each integer x associates the integer $\mathrm{succ}(x) = x + 1$. For another example, consider the function sumdif that to each pair (x,y) of real numbers associates the pair $\mathrm{sumdif}(x,y) = (x + y, x - y)$.

A bijective function is also called a **permutation**. This is more commonly used when $X = Y$. It should be noted that *one-to-one function* means *one-to-one correspondence* (i.e., *bijection*) to some authors, but *injection* to others. The set of all bijections from X to Y is denoted as $X \leftrightarrow Y$.

Bijective functions play a fundamental role in many areas of mathematics, for instance in the definition of isomorphism (and related concepts such as homeomorphism and diffeomorphism), permutation group, projective map, and many others.

Contents [hide]

Composition and inverses

[edit]

Ac3 (pseudo code)

```
ac3(v,d,c)
 consistent := true;
 q := c                                        // enqueue all constraints
 while q ≠ {} & consistent
 do (i,j) := dequeue(q)                        // get a constraint
     if revise(i,j)                            // if d_i has values removed
     then q := q U {(k,i) | C_ik in C}         // need to revise all constraints C_ki
           consistent := d[i] ≠ {}             // stop if domain wipe out
  return consistent
```

ac3

```
if revise(i,j)
then q := q U {(k,i) | (k,i) in c}
```

What?

Note: ac3 has a queue of constraints that need revision
because some values in the domains of the variables may
be unsupported.

Remember: ac3 processes a queue of constraints

But forgive me, the queue might be treated as just a set ac1 and ac2 (the straw men) essentially revised constraints over and over again, until no change … until reaching a **fixed point**

$$O(e.d^3)$$

e is number of constraints
d is domain size

Complexity of ac3 proved in 1985 by Mackworth & Freuder (AIJ 25)

$$O(e.d^3)$$

e is number of constraints
d is domain size

Prove it!
Also look at paper by Zhang & Yap

The complexity of ac3

A beautiful proof

- A constraint C_i,j is revised iff it enters the Q
- C_i,j enters the Q iff some value in d[j] is deleted
- C_i,j can enter Q at most d times (the size of domain d[j])
- A constraint can be revised at most d times
- There are e constraints in C (the set of constraints)
- revise is therefore executed at most e.d times
- the complexity of revise is $O(d^2)$
- the complexity of ac3 is then $O(e.d^3)$

The order that we revise the constraints make no difference to the outcome
It reaches the same fixed point, the same set of arc-consistent domains

The order that we revise the constraints may make a difference to run time.

… constraint ordering heuristic, anyone?

Revise ignored any semantics of the constraint

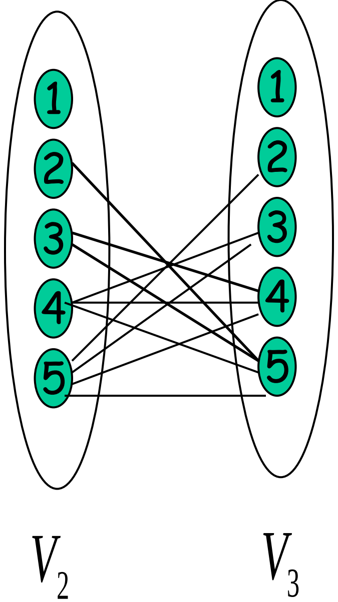
Is that dumb, or what?

Could we get round this?
- Use OOP?
- A class of constraint?
- revise as a specialised method?

AC4, AC6, AC7, ….

“Optimal” support counting algorithms

AC4, AC6, AC7, ...



$$C_{2,3} = V_2 + V_3 > 6$$

Associate with each value in Di
- a counter supportCount[x,i,j]
  - the number of values in Dj that support x
- a boolean supports[x,y,j]
  - true if x supports y in Dj

1st stage of the algorithm builds up the supportCount and support flags

2nd stage
- if supportCount[x,i,j] = 0 (x has no support in Dj over constraint Cij)
  - delete(Di,x)
  - decrement supportCount[y,k,i] (where supports[x,y,k] is true)
- continue this till no change
  - i.e. propagate

If x supports y in Dk and x is deleted from Di
Then support count for y in Dk over constraint Cki is decremented

Best case ***and*** worst case performance of ac4 is the same

$$O(e.d^2)$$

Ac6, 7, and 8 exploit symmetries, and lazy evaluation
- if x supports y over constraint Cij then y supports x over Cji
- find the 1st support for x, and only look for more when support is lost

Ac3 worst case performance rarely occurs
(experimental evidence due to Rick Wallace)

Why is best case and worst case performance of ac4

$$O(e.d^2)$$

?

History Lesson

- ac1/2/3 due to Alan Mackworth 1977
- ac4 Mohr & Henderson AIJ28 1986
- ac6 , 7, 8 due to Freuder, Bessiere, Regin, and others
  - in AIJ, IJCAI, etc

Downside of ac4, ac6, ac7, and ac8 algorithms is “hard to code”

ac3 is easy!

AC5

A generic arc-consistency algorithm and its specializations
AIJ 57 (2-3) October 1992
P. Van Hentenryck, Y. Deville, and C.M. Teng

ac5

Ac5 is a "generic" ac algorithm and can be specialised for special constraints (i.e. made more efficient when we know something about the constraints)

Ac5 is at the heart of constraint programming

Constraint is an object with its own propagator

ac5, the intuition

- take an OOP approach
- constraint is a class that can then be specialised
- have a method to revise a constraint object
- allow specialisation
- have basic methods such as
    - revise when lwb increases
    - revise when upb decreases
    - revise when a value is lost
    - revise when variable instantiated
    - revise initially
- methods take as arguments
    - the variable in the constraint that has changed
    - possibly, what values have been lost

- arc-consistency is at the heart of constraint programming
- it is the inferencing step used inside search
- it has to be efficient
- data structures and algorithms are crucial to success
    - ac is established possibly millions of times when solving
    - it has to be efficient
- we have had an optimal algorithm many times
    - ac4, ac6, ac7, ac2001
- ease of implementation is an issue
    - we like simple things
- but we might still resort to empirical study!
- modern approach is constraint as object with specialised propagator

Arguably, arc consistency is an out-of-date concept

We have specialised constraints with specialized propagators

Arc-consistency used on explicit representation of binary constraints

Table constraints

But still a useful concept

Levels of consistency

generalized arc consistency (GAC)

domain consistency versus ***bound*** consistency

Singleton consistency (SAC)

MAC

What's that then

Maintain arc-consistency

- Instantiate a variable v[i] := x
    - impose unary constraint d[i] = {x}
    - make future problem ac
    - if domain wipe out
        - backtrack and impose constraint d[i] ≠ x
        - make future ac
- and so on

Maintain arc-consistency

- why use instantiation?
  - Domain splitting?
  - resolve disjunctions first
    - for example ($V1 < V2$ OR $V2 < V1$)