Search and Optimisation

- An overview

- Algorithms to enforce Node- and Arc-consistency
- Non-Binary Networks
- Consistency and Satisfiability
- Bounds-Consistency and Generalised Arc-Consistency

Enforcing Node-Consistency

Definition (**Node Consistency**):

A constraint satisfaction problem is **node-consistent** if no value on the domain of its variables violates the **unary** constraints.

Enforcing node consistency: Algorithm NC-1

- This can be enforced by the very simple algorithm shown below:

```
procedure NC-1(V, D, C);
   for x in V
      for v in Dx do
         for Cx in {C: Vars(Cx) = {x}} do
            if not satisfy(x-v, Cx) then
              Dx <- Dx \ {v}
         end for
      end for
   end for
end procedure
```

Enforcing Node-Consistency

Space Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and assuming that the variable's domains are represented by extension, a space **nd** is required to keep explicitely the domains of the variables.
- Algorithm NC-1 does not require additional space, so its space complexity is **O(nd).**

Time Complexity of NC-1: O(nd).

- Assuming **n** variables in the problem, each with **d** values in its domain, and taking into account that each value is evaluated one single time, it is easy to conclude that algorithm NC-1 has time complexity **O(nd)**.

The low complexity, both temporal and spatial, of algorithm NC-1, makes it suitable to be used in virtual all situations by a solver, despite the low pruning power of node-consistency.

Enforcing Arc-Consistency: AC-1

Definition (**Arc Consistency**):

A constraint satisfaction problem is arc-consistent if it is node-consistent and

- For every label x_i-v_i of every variable x_i, and for all constraints C_{ij}, defined over variables x_i and x_j, there must exist a value v_j that **supports** v_i.

Enforcing arc-consistency: Algorithm AC-1

- The following simple (and inefficient) algorithm enforces arc-consistency:

```
procedure AC-1(V, D, C);
   NC-1(V,D,C);        % node consistency
   Q = {a_ij | c_ij ∈ C ∨ c_ji ∈ C }; % see note
   repeat
     changed <- false;
     for a_ij in Q do
          changed <- changed or revise_dom(a_ij,V,D,C)
     end for
   until not change
end procedure
```

- **Note**: for any constraint c_{ij} two directed arcs, a_{ij} e a_{ji}, are considered.

Enforcing Arc-Consistency: AC-1

Revise-Domain

- Algorithm AC-1 (and others) uses predicate **revise-domain** on some arc a_{ij}, that succeeds if some value is removed from the domain of variable x_i (a side-effect of the predicate).

```
predicate revise_dom(a_ij,V,D,C): Boolean;
   success <- false;
   for v in dom(x_i) do
      if ¬ ∃v_j in dom(x_j): satisfies({x_i-v,x_j-v_j},c_ij) then
         dom(x_i) <- dom(x_i) \ {v};
         success <- true;
      end if
   end for
   revise_dom <- success;
end predicate
```

Enforcing Arc-Consistency: AC-1

Space Complexity of AC-1: $O(ad^2)$

- AC-1 must maintain a queue **Q**, with maximum size **2a**. Hence the inherent spacial complexity of AC-1 is **$O(a)$**.

- To this space, one has to add the space required to represent the domains **$O(nd)$** and the constraints of the problem. Assuming **a** constraints and **d** values in each variable domain the space required is **$O(ad^2)$**, and a total space requirement of

$$O(nd + ad^2)$$

  which dominates $O(a)$.

- For "dense" constraint networks", **$a \approx n^2/2$**. This is then the dominant term, and the space complexity becomes

$$O(ad^2) = O(n^2d^2)$$

Enforcing Arc-Consistency: AC-1

Time Complexity of AC-1: $O(nad^3)$

- Assuming **n** variables in the problem, each with **d** values in its domain, and a total of **a** arcs, in the worst case, predicate revise_dom, checks **d^2** pairs of values.

- The number of arcs a_{ij} in queue Q is **2a** (2 directed arcs a_{ij} and a_{ji} are considered for each constraint C_{ij}). For each value removed from one domain, revise_dom is called **2a** times.

- In the worst case, only one value from one variable is removed in each cycle, and the cycle is executed **nd** times.

- Therefore, the worst-case time complexity of AC-1 is $O(d^2 * 2a * nd)$, i.e.

$O(nad^3)$

Enforcing Arc-Consistency: AC-3

Enforcing node consistency: Algorithm AC-3

- Whenever a value v_i is removed from the domain of some x_i, all arcs are reexamined. However, only the arcs a_{ki} (for k ≠ i§) should be reexamined.
- This is because the removal of v_i may eliminate the support from some value v_k of some variable x_k for which there is a constraint c_{ki} (or c_{ik}).
- Such inefficiency of AC-1 is avoided in **AC-3** below

```
procedure AC-3(V, D, C);
   NC-1(V,D,C);        % node consistency
   Q = {aij | cij ∈ C ∨ cji ∈ C };
   while Q ≠ ∅ do
      Q = Q \ {aij}    % removes an element from Q
      if revise_dom(aij,V,D,C) then    % revised xi
         Q = Q ∪ {aki | (cik ∈ C ∨ cki ∈ C )∧  k ≠ i}
      end if
   end while
end procedure
```

Enforcing Arc-Consistency: AC-3

Space Complexity of AC-3: $O(ad^2)$

- AC-3 has the same requirements than AC-1, and the same worst-case space complexity of $O(ad^2) \approx O(n^2d^2)$, due to the representation of constraints by extension.

Time Complexity of AC-3: $O(ad^3)$

- Each arc a_{ki} is only added to Q when some value v_i is removed from the domain of x_i.
- In total, each of the **2a** arcs may be added to Q (and removed from Q) **d** times.
- Every time that an arc is removed, predicate revise_dom is called, to check at most d^2 pairs of values.
- All things considered, and in contrast with AC-1, with temporal complexity $O(nad^3)$, the time complexity of AC-3, in the worst case, is $O(2ad * d^2)$, i.e.

$$O(ad^3)$$

Enforcing Arc-Consistency: AC-4

Inefficiency of AC-3

- Every time a value v_i is removed from the domain of some variable x_i, **all** arcs a_{ki} ($k \neq i$) leading to that variable are reexamined.
- Nevertheless, only some of these arcs should be examined.
- Although the removal of v_i may eliminate **one** support for some value v_k of another variable x_k (given constraint c_{ki}), other values in the domain of x_i may support the pair x_k-v_k!

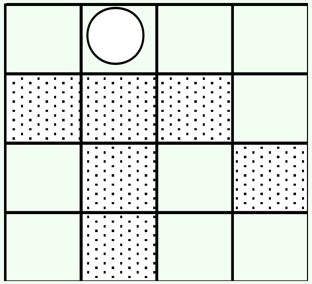
This idea is exploited in algorithm **AC-4,** that uses a number of new data-structures to count supporting values

- **Counters**: For counting support values of label $\{x_i$-$v_i\}$ in x_j
- **Suporting Sets**: That explicitly enumerate the labels $\{x_j$-$v_j\}$ that are supported by label $\{x_i$-$v_i\}$, w.r.t. any constraint c_{ij}.
- **List**: Queue of removed labels to be examined (similar to Q in AC-3)
- **Matrix M**: Maintains information on whether a label $\{x_i$-$v_i\}$ is still present.

Enforcing Arc-Consistency: AC-4

AC-4 Counters

- For example, in the 4 queens problem, the counters that account for the support of value q_1= 2 are initialised as follows
  - $c(2,q_1,q_2) = 1$ % q_2-4 does not attack q_1-1
  - $c(2,q_1,q_3) = 2$ % q_3-1 and q_3-3 do not attack q_1-1
  - $c(2,q_1,q_4) = 3$ % q_4-1, q_4-3 and q_4-4 do not attack q_1



AC-4 Supporting Sets

- To update the counters when a value is eliminated, it is useful to maintain the set of Variable-Value pairs that are supported by each value of a variable.
- AC-4 thus maintain for each Value-Variable pair the set of all Variable-Value pairs supported by the former pair.
  - sup(1, q_1) = [q_2-2, q_2-3, q_3-2, q_3-4, q_4-2, q_4-3]
  - sup(2, q_1) = [q_2-4, q_3-1, q_3-3, q_4-1, q_4-3, q_4-4]
  - sup(3, q_1) = [q_2-1, q_3-2, q_3-4, q_4-1, q_4-2, q_4-4]
  - sup(4, q_1) = [q_2-1, q_2-2, q_3-1, q_3-3, q_4-2, q_4-3]

Enforcing Arc-Consistency: AC-4

Algorithm AC-4 (Overall Functioning) AC-4 is composed of two phases:

a) initialisation, which is executed only once; and

b) propagation, executed after the first phase, and after each enumeration step.

```
procedure initialise_AC-4(V,D,C);
   M <- 1; sup <- ∅; List = ∅;
   for cij in C do
      for vi in dom(xi) do
         ct <- 0;
         for vj in dom(xj) do
            if satisfies({xi-vi, xj-vj}, cij) then
               ct <- ct+1; sup(vj,xj)<- sup(vj,xx) ∪ {xi-vi}
             end if
         endfor
        if ct = 0 then M[xi,vi] <- 0; List <- List ∪ {xi-vi};
                       dom(xi) <- dom(xi)\{vi}
        else c(vi, xi, xj) <- ct;
        end if
      end for
  end for
end procedure
```

Enforcing Arc-Consistency: AC-4

Algorithm AC-4 (propagation phase)

```
procedure propagate_AC-4(List,V,D,R);
    while List ≠ ∅ do
        List <- List\{xi-vi} % remove element from List
        for xj-vj in sup(vi,xi) do
            c(vj,xj,xi) <- c(vj,xj,xi) - 1;
            if c(vj,xj,xi) = 0 ∧ M[xj,vj] = 1 then
                List = List ∪ {xj-vj};
                M[xj,vj] <- 0;
                dom(xj) <- dom(xj) \ {vj}
            end if
        end for
    end while
end procedure
```

Enforcing Arc-Consistency: AC-4

Space Complexity of AC-4: $O(ad^2)$

- As a whole algorithm AC-4 maintains
  - **Counters**: As discussed, a total of 2ad
  - **Suporting Sets**: In the worst case, for each constraint c_{ij}, each of the d x_i-v_i pairs supports d values v_j from x_j (and vice-versa). The space to maintain the supporting sets is thus $O(ad^2)$.
  - **List**: Contains at most 2a arcs
  - **Matrix M**: Maintains nd Boolean values.
- The space required to maintain the supporting sets dominates. Compared with AC-3, where a space of size $O(a)$ was required to maintain the queue, AC-4 has a much worse space complexity of $O(ad^2)$

Enforcing Arc-Consistency: AC-4

Time Complexity of AC-4: O(ad²)

- Analysing the cycles executed in the procedure initialise_AC-4,

```
for c_ij in C do
  for v_i in dom(x_i) do
    for v_j in dom(x_j) do
```

  and assuming that the number of constraints (arcs) is a and the variables have all d values in their domains, the inner cycle of the procedure is executed $2ad^2$ times, which sets the time complexity of the initialisation phase to **$O(ad^2)$**.

- In the inner cycle of procedure propagate_AC-4 a counter for pair x_j-v_j is decremented

```
c(v_j,x_j,x_i) <- c(v_j,x_j,x_i) - 1
```

  Since there are **2a** arcs and each variable has **d** values in its domain, there are **2ad** counters. Each counter is initialised at most to **d**, as each pair x_j-v_j may only have d supporting values in the domain of another variable x_i.

  Hence, the inner cycle is executed at most $2ad^2$ times, which determines the time complexity of the propagation phase of AC-4 to be **$O(ad^2)$**

Enforcing Arc-Consistency: AC-4

The asymptotic complexity of AC-4, cannot be improved by any algorithm!

- To check whether a network is arc consistent it is necessary to test, for each constraint C_{ij}, that the **d** pairs X_i-v_i have support in X_j, for which **d** tests might be required. Since each of the **a** constraints is considered twice, then **$2ad^2$** tests are required, with assymptotic complexity **$O(ad^2)$** similar to that of AC-4.
- However, one should bear in mind that the worst case complexity is *asymptotic*. The data structures of AC-4, namely the counters that enable improving the support detection are too demanding. The initialisation of these structures is also very heavy, namely if the domains have large cardinality, **d**.
- The space required by AC-4 is also problematic, specially when the constraints are represented by intension, rather than by extension (in this latter case, the space required to represent the constraints is of the same order of magnitude...).
- All in all, it has been observed that, in practice (typically),

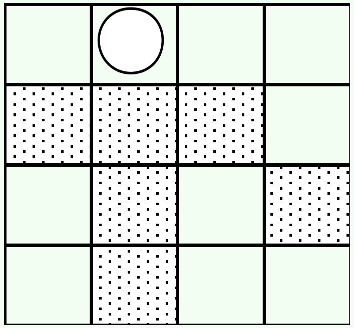
AC-3 is usually more efficient than AC-4!

Enforcing Arc-Consistency: AC-6

- Algorithm **AC-6** avoids the outlined inefficiency of AC-4 with a basic idea: instead of keeping (counting) all values v_i from variable x_i that support a pair x_j-v_j, it simply maintains the lowest such v_i that supports the pair.

- The initialisation of the algorithm becomes “lighter”. Whenever the first value v_i is found, no more supporting values are sought and no counting is required. Also, in AC-6, the supporting sets become singletons.

- **Data Structures of Algorithm AC-6**
  - The **List** is adapted
  - Boolean **matrix M from** AC-4 is kept.
  - The AC-4 **counters** are disposed of;
  - The supporting sets become “singletons”, only keeping the lowest value supported .



- $sup(1,x_1) = [x_2-2, x_2-3, x_3-2, x_3-4, x_4-2, x_4-3]$
- $sup(2,x_1) = [x_2-4, x_3-1, x_3-3, x_4-1, x_4-3, x_4-4]$
- $sup(3,x_1) = [x_2-1, x_3-2, x_3-4, x_4-1, x_4-2, x_4-4]$
- $sup(4,x_1) = [x_2-1, x_2-2, x_3-1, x_3-3, x_4-2, x_4-3]$

Enforcing Arc-Consistency: AC-6

- Both phases of AC-6 use predicate

next_support(x_i, v_i, x_j, v_j, out v)

that succeeds if there is in the domain of $\mathbf{x_j}$ a "next" supporting value **v,** the lowest value, no less than some value, $\mathbf{v_j}$, such that $\mathbf{x_j}$**-v** supports $\mathbf{x_i}$-$\mathbf{v_i}$.

```
predicate next_support(xi,vi,xj,vj, out v): boolean;
     sup_s <- false; v <- vj;
     while not sup_s and v =< max(dom(xj)) do
        if not satisfies({xi-vi,xj-v},cij) then
             v <- next(v,dom(xj))
         else
             sup_s <- true
         end if
      end while
      next_support <- sup_s;
   end predicate.
```

Enforcing Arc-Consistency: AC-6

Algorithm AC-6 (initialisation phase)

```
procedure initialise_AC-6(V,D,C);
  List <- ∅; M <- 0; sup <- ∅;
  for c_ij in C do
  for v_i in dom(x_i) do
    v = min(dom(x_j))
    if next_support(x_i,v_i,x_j,v,v_j) then
       sup(v_i,x_i)<- sup(v_i,x_i) ∪ {x_j-v_j}
    else
       dom(x_i) <- dom(x_i)\{v_i};
       M[x_i,v_i] <- 0;
       List <- List ∪ {x_i-v_i}
    end if
  end for
end for
end procedure
```

Enforcing Arc-Consistency: AC-6

Algorithm AC-6 (propagation phase)

```
procedure propagate_AC-6(List,V,D,C);
    while List ≠ ∅ do
        List <- List\{xj-vj} % removes xj-vj from List
        for xi-vi in sup(vj,xj) do
            sup(vi,xi) <- sup(vi,xi) \ {xj-vj} ;
            if M[xi,vi] = 1 then
                if next_suport(xi,vi,xj,vj,v) then
                    sup(vi,xi)<- sup(vi,xi) ∪ {xj-v}
                else
                    dom(xi) <- dom(xi)\{vi}; M[xi,vi] <- 0;
                    List <- List ∪ {xi-vi}
                end if
            end if
        end for
    end while
end procedure
```

Enforcing Arc-Consistency: AC-6

Space Complexity of AC-6: O(ad)

In total, algorithm AC-6 maintains

- **Supporting Sets**: In the worst case, for each of the **a** constraints c_{ij}, each of the **d** pairs x_i-v_i is supported by a single value v_j form x_j (and vice-versa). Thus, the space required by the supporting sets is **O(ad)**.
- **List**: Includes at most **nd** labels
- **Matrix M**: Maintains **nd** Booleans.

- The space required by the supporting sets is dominant, so algorithm AC-6 has a space complexity of
  - **O(ad)**

  between those of **AC-3** (**O(a)**) and **AC-4** (**$O(ad^2)$**).

Enforcing Arc-Consistency: AC-6

Time Complexity of AC-6: O(ad^2)

- In both phases of initialisation and propagation, AC-6 executes

  next_support(x_i, v_i, x_j, v_j, v)

  in its inner cycle.
- For each pair **x_i-v_i**, variable **x_j** is checked at most **d** times.
- For each arc corresponding to a constraint **C_{ij}**, **d** pairs **x_i-v_i** are considered at most.
- Since there are **2a** arcs (2 per constraint **C_{ij}**), the time complexity, worst-case, in any phase of AC-6 is

  O(ad^2).
- Like in AC-4, this is optimal **assymptotically.**

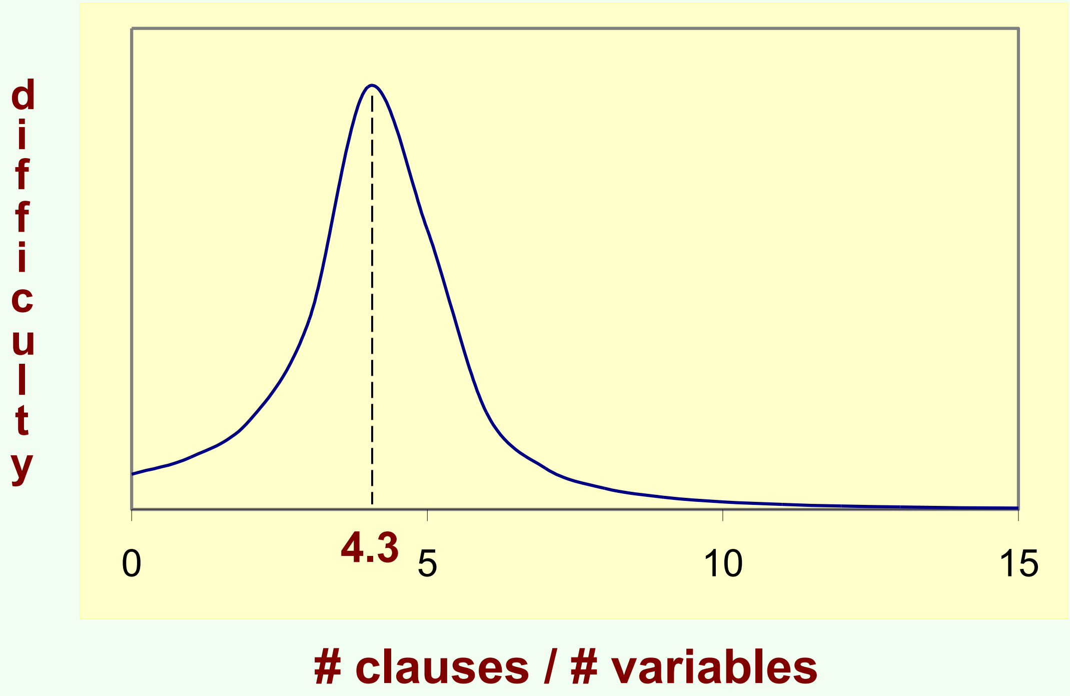
Assessing Typical Complexity

- **Typical** complexity of AC-x algorithms
  - The worst case time complexity that can be inferred from the algorithms do not give a precise idea of their average behaviour in typical situations. For such study, either one tests the algorithms in:
  - A set of "benchmarks", i.e. problems that are supposedly representative of everyday situations (e.g. N-queens); or
  - Randomly generated instances parameterised by
    - their **size** (number of variables and cardinality of the domains) ; and
    - their **difficulty** measured by
      - density of the constraint network - % existing/ possible constraints; and
      - tightness of the constraints - % of allowed / all tuples.
  - The study of these issues has led to the conclusion that constraint satisfaction problems often exhibit a phase transition, which should be taken into account in the study of the algorithms.

Assessing Typical Complexity: Phase Transition

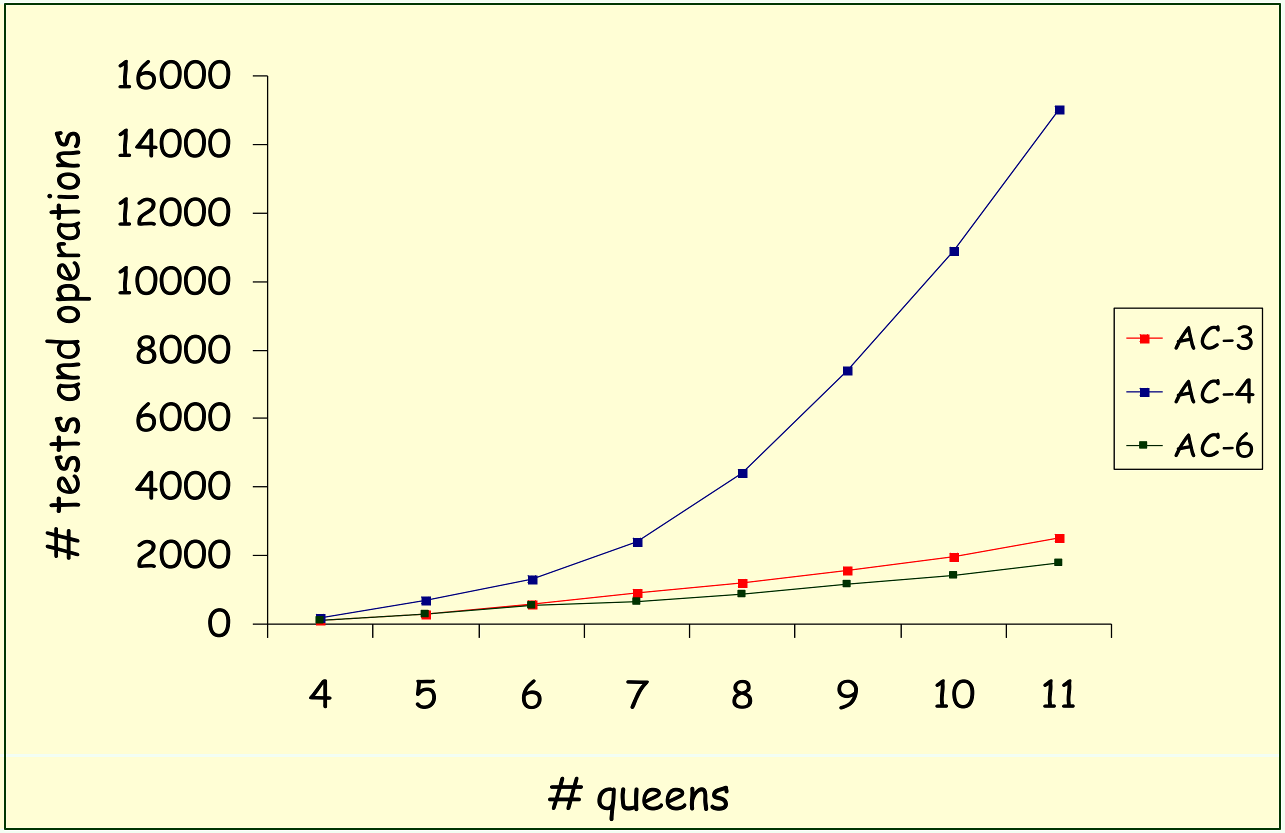
- This phase transition typically contains the most difficult instances of the problem, and separates the instances that are trivially satisfied from those that are trivially insatisfiable.

- For example, in SAT problems, it has been found that the phase transition occurs when the ratio of clauses to variables is around 4.3.



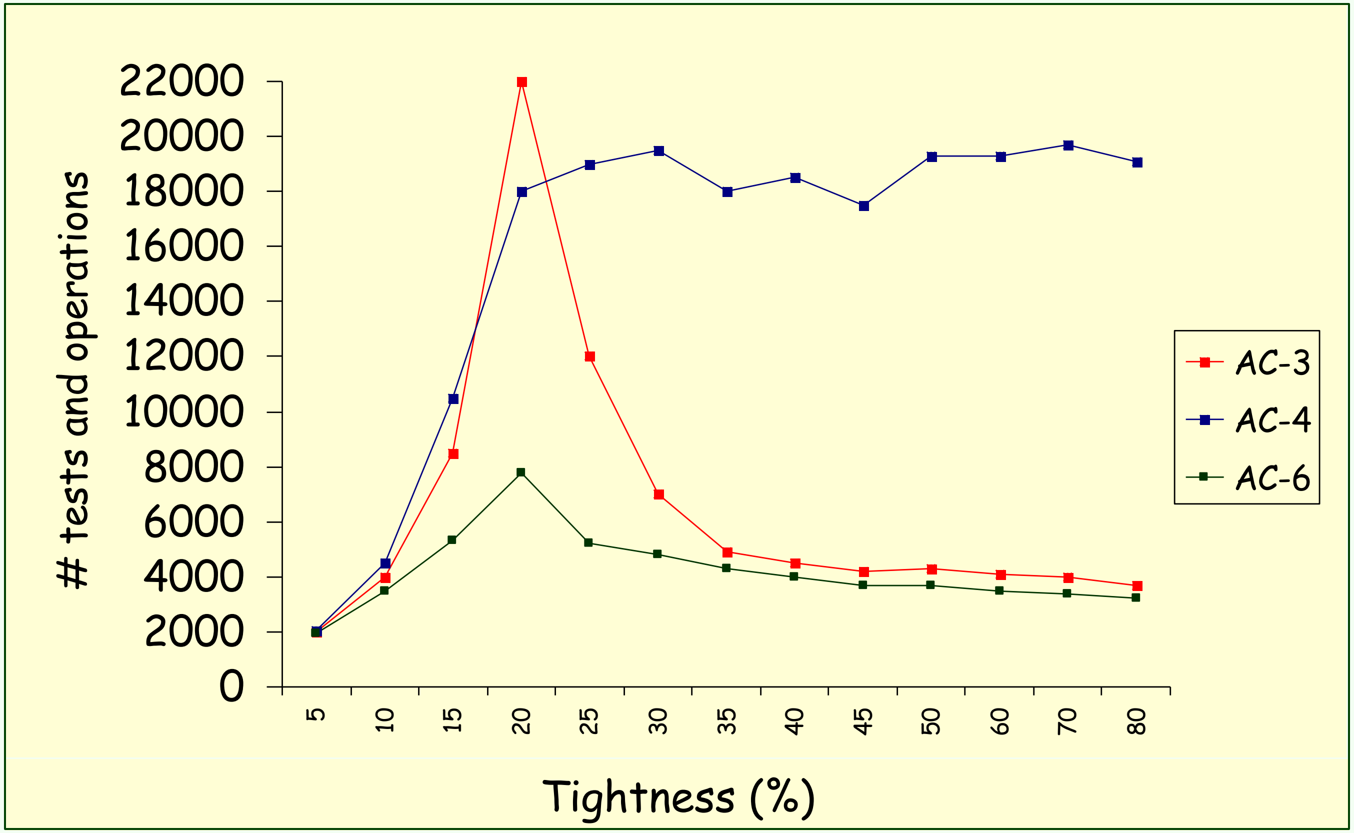
Assessing Typical Complexity

- **Typical Complexity** of algorithms AC-3, AC-4 e AC-6
  - (N-queens)



Assessing Typical Complexity

Typical Complexity of algorithms AC-3, AC-4 e AC-6
(randomly generated problems)
n = 12 variables, d= 16 values, density = 50%



Path-Consistency

Definition (**Path Consistency**):

A constraint satisfaction problem is path-consistent if,

- It is arc-consistent; and
- Every consistent 2-compound label {X_i-v_i, X_{ij}-v_j,} can be extended to a consistent label with a third variable X_k (k ≠ i and k ≠j }.

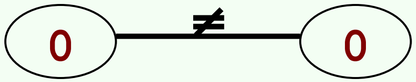
The second condition is more easily understood as

- For every compound label {X_i-v_i, X_{ij}-v_j,} there must be a value v_k that **supports** {X_i-v_i, X_{ij}-v_j,}, i.e. the compound label {X_i-v_i, X_j-$v_{j,}$ X_k-v_k} satisfies constraints C_{ij}, C_{ik}, and C_{kj}.

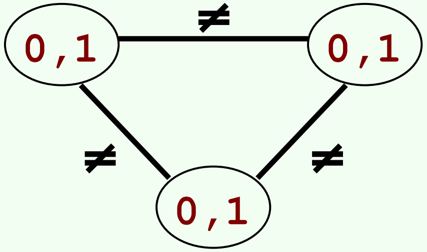
Binary Constraints: i-consistency

- The notions of node-, arc- and path-consistency can be generalised for a common criterion: i-consistency, with increasing demands of consistency.

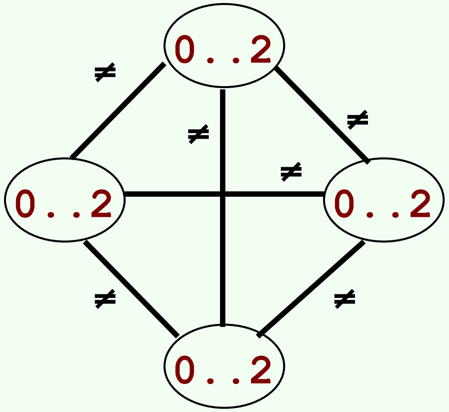
  - A node consistent network, that is not arc consistent



  - An arc consistent network, that is not path consistent



  - A path-consistent network, that is not 4-consistent



Binary Constraints: i-consistency

- The criterion of i-consistency is thus defined as follows.

• A network is **i-consistent** if all compound labels of cardinality i-1 can be extended to any other i-th variable.

1. For example, with k = i-1, any compound label $<x_{a1}-v_{a1}, x_{a2}-v_{a2}, \ldots, x_{ak}-v_{ak}>$, that satisfies the constraints over variables of set S = $\{x_{a1}, x_{a2}, \ldots, x_{ak}\}$ can be extended to another variable x_{ai}, i.e. there is a v_{ai} in the domain of x_{ai} that satisfies all the constraints defined on the set $S \cup \{x_{ai}\}$ of variables.

2. As a special case, when i=1, only the unary constraints must be satisfied.

- Additionally, a network is **strongly** i-consistent if it is k-consistent for all $k \leq i$.

- Given this definitions it is easy to show that the following equivalences:

Node-consistency	$\leftrightarrow$	strong 1-consistency
Arc- consistency	$\leftrightarrow$	strong 2-consistency
Path-consistency	$\leftrightarrow$	strong 3-consistency

Binary Constraints: i-consistency

- Notice that the analogies of node-, arc- and path- consistency were made with respect to **strong** i-consistency.
- This is because a constraint network may be i-consistency but not m-consistent (for some m < i). For example, the network below is 3-consistent, but not 2-consistent. Hence it is not strongly 3-consistent.

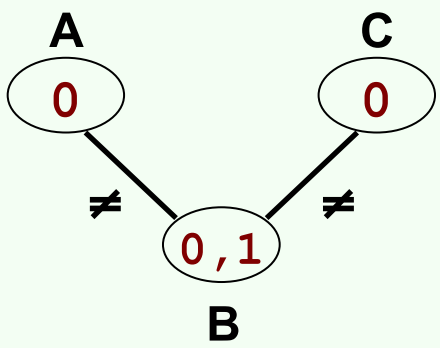
- The only 2-compound labels, that satisfy the constraints

  {A-0,B-1}, {A-0,C-0}, and {B-1, C-0}

  may be extended to the remaining variable

  {A-0,B-1,C-0}

- However, the 1-compound label {B-0} cannot be extended to variables A or C {A-0,B-0} !



Binary Constraints: i-consistency

- For i > 3, i-consistency cannot be implemented with binary constraints alone, In fact:
  - 2-consistency checks whether a 1-label $\{x_i\text{-}v_i\}$ can be extended to some other 2-label $\{x_i\text{-}v_i,\ x_j\text{-}v_j\}$. If that is not the case, label $\{x_i\text{-}v_i\}$ is removed from the domain of X_i.
  - 3-consistency checks whether a 2-label $\{x_i\text{-}v_i,\ x_j\text{-}v_j\}$ can be extended to a 3-label $\{x_i\text{-}v_i,\ x_j\text{-}v_j,\ x_k\text{-}v_k\}$. If that is not the case, label $\{x_i\text{-}v_i,\ x_j\text{-}v_j\}$ is removed.
  - Removing label $\{x_i\text{-}v_i,\ x_j\text{-}v_j\}$ is not achieved by removing values from the domains of the variables, but rather by tightening a constraint C_{ij} on variables x_i and x_j.
- By analogy, to impose 4-consistency 3-labels have to be removed so a constraint on 3 variables has to be created or tightened.
- In general, maintaining i-consistency requires imposing constraints with arity i-1.

Binary Constraints: i-consistency

- The algorithms that were presented for achieving arc-consistency could be adapted to obtain i-consistency, provided that we consider constraints with i-1 arity.
- The adaptation of the AC-1 algorithm (brute-force) would have
  - Time complexity of **$O(2^i (nd)^{2i})$.**
  - Space complexity of **$O(n^i d^i)$.**
- The adaptation of the AC-4 and AC-6 algorithms lead to optimal asymptotic time complexity of **$\Omega (n^i d^i)$** (a lower bound).
- Given the mentioned complexity (even if the typical cases are not so bad) their use in backtrack search is generally not considered.
- The main application of these criteria is in cases where tractability can be proved based on these criteria.

Network Consistency and Satisfiability

All types of i-consistency can be imposed by polinomial algorithms, with asymptotic time complexity $\Omega(n^i d^i)$ even when the corresponding problems (modelled with binary constraints) are NP-complete.

Hence, in general for a network with n variables, i-consistency (for any i < n) i-does not imply satisfiability of the problem, i.e.

> There are unsatisfiable problems modelled with binary constraints whose corresponding network is i-consistent.

Of course, the converse is also true

> There are satisfiable problems modelled with binary constraints whose corresponding network is not i-consistent.

Nevertheless, in some special cases, the two concepts (i-consistency and satisfiability are equivalent).

We will overview two such cases.

Network Consistency and Satisfiability

Case 1: A network of binary constraints, whose variables have only 2 values in their domain, is satisfiable iff it can be made path-consistent.

Proof: By recasting the problem to 2-SAT.

If the network is path-consistent, then

1. all binary constraints are explicit, and
2. the matrices representing the constraints have a maximum of 2 rows and 2 columns.

In this case, the satisfaction of a constraint can be equated to the satisfaction of a Boolean formula in disjunctive normal form (see figure below for an example).

a\b	3	4
2	1	1
5	0	1

(a2 ∧ b3) ∨ (a2 ∧ b4) ∨ (a5 ∧ b4)

Network Consistency and Satisfiability

Now, these formulae can be converted into conjunctive normal form.

(a2 ∧ b3) ∨ (a2 ∧ b4) ∨ (a5 ∧ b4) ⇔

(a2∨a2∨a5) ∧ (a2∨a2∨b4) ∧ (a2∨b4∨a5) ∧ (a2∨b4∨b4)

∧ (b3∨a2∨a5) ∧ (b3∨a2∨b4) ∧ (b3∨b4∨a5) ∧ (b3∨b4∨b4)

The resulting clauses have as many literals as 1´s in the matrix that models a constraint (after imposing path-consistency. In this case the clauses have 3 literals.

But such clauses may be simplified, by adding the semantics associated to the encoding (a variable must have a single value)

a2 ∨ a5 = true; b3 ∨ b4 = true

Yielding, (after simplification) a set of clauses, each having only 2 literals.

true ∧ (a2 ∨ b4) ∧ true ∧ (a2 ∨ b4)

∧ true ∧ true ∧ true ∧ true ⇔

(a2∨b4) ♦

Graph Width

- Before presenting another theorem relating k-consistency and tractability it is convenient to consider constraint networks with n-ary constraints (n>2), either because a problem is specified with such constraints, or because these constraints are induced in a (binary) graph when k-consistency (k>3) is imposed on the constraint network.
- For this purpose we have the following definition:

Definition: Primal Graph of a Constraint Network

The primal graph of a constraint network is a graph where there is an edge between two variables iff there is some constraint with the two variables in its scope.

Given the definition, the primal graph of a constraint satisfaction problem coincides with the problem graph if the only constraints to be considered are binary (or unary).

Graph Width

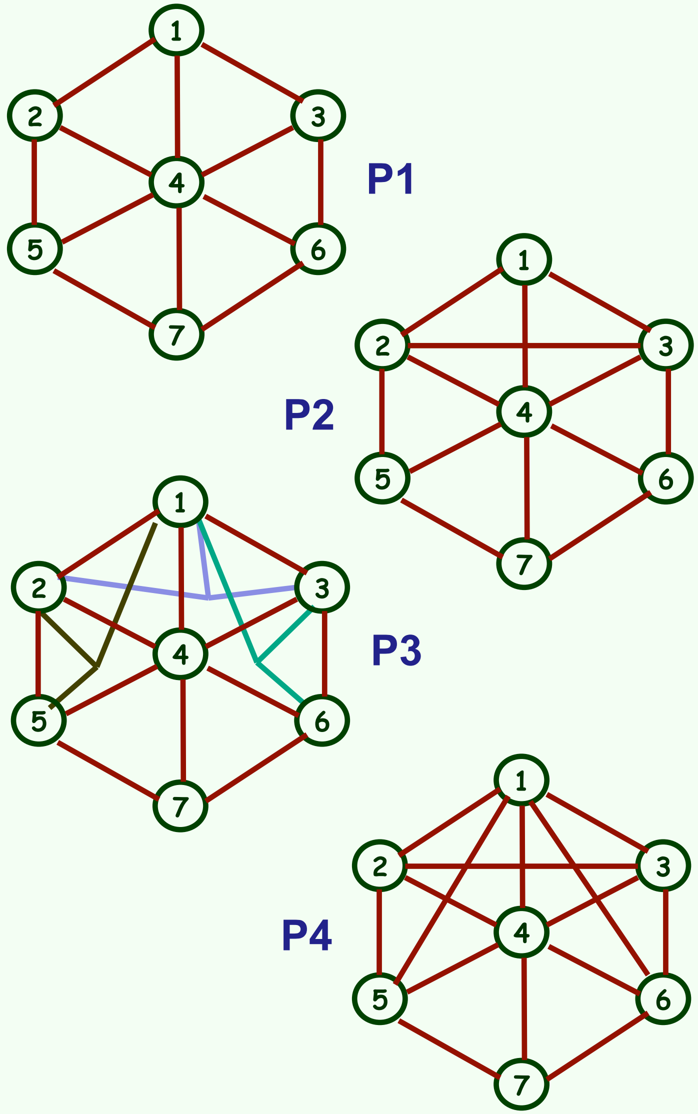
Example:

1. Let us assume that the initial formalisation of a problem leads to the network P1.

2. Imposing path-consistency, arcs are added between variables, e.g. 2-3, resulting in network P2 (still a graph).

3. Imposing 4-consistency, hyper-arcs are imposed on variables 1-2-3, 1-2-5 and 1-3-6, resulting in network P3 (a hyper-graph).

4. The primal graph of the problem is shown as graph P4.



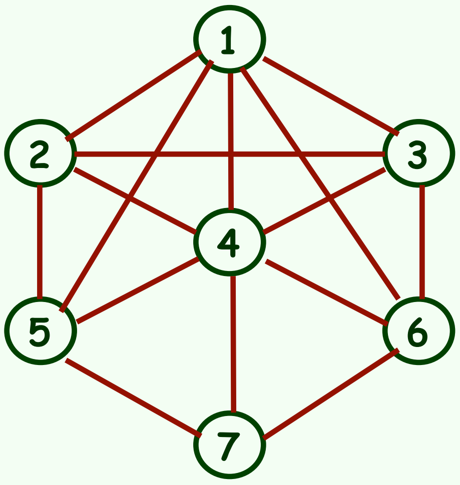
Graph Width

Definition: Node width, given ordering O

Given some total ordering, O, defined on the nodes of a graph, the width of a node N, given ordering O is the number of lower order nodes that are adjacent to N.

Example: For the graph and the ordering O_1 shown we have

- $w(1,O_1) = 0$
- $w(2,O_1) = 1$ (node 1)
- $w(3, O_1) = 2$ (nodes 1 and 2)
- $w(4, O_1) = 3$ (nodes 1, 2 and 3)
- $w(5, O_1) = 3$ (nodes 1, 2 and 4)
- $w(6, O_1) = 3$ (nodes 1, 3 and 4)
- $w(7, O_1) = 3$ (nodes 4, 5 and 6)

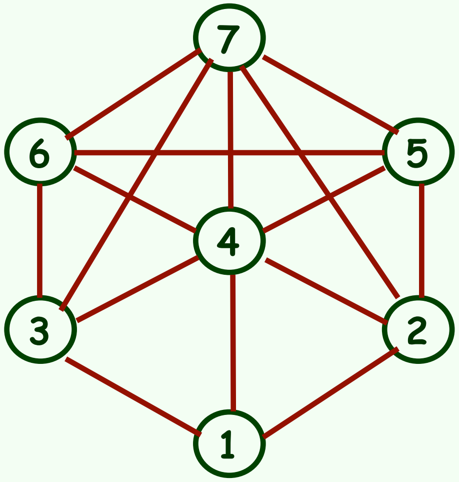


Graph Width

- Different orderings will produce different widths for the nodes of the graphs.

Example: For the same graph but with an “inverted ordering O_2, we have

- $w(1, O_2) = 0$
- $w(2, O_2) = 1$ (node 1)
- $w(3, O_2) = 1$ (node 1)
- $w(4, O_2) = 3$ (nodes 1, 2 and 3)
- $w(5, O_2) = 2$ (nodes 2 and 4)
- $w(6, O_2) = 2$ (nodes 3 and 4)
- $w(7, O_2) = 5$ (nodes 2, 3, 4, 5 and 6)



Graph Width

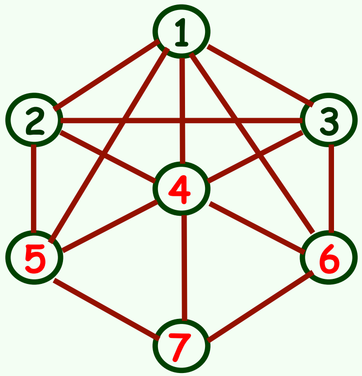
- From the width of the nodes one may obtain the width of a graph.

Definition: Graph width, given ordering O

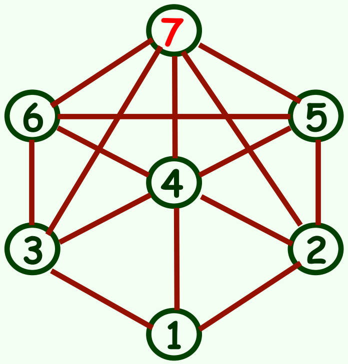
Given some total ordering, O, defined on the nodes of a graph, the width of the graph, given ordering O is the maximum width of its nodes, given ordering O.

Example: For the two orderings we obtain

$W(G,O_1) = 3$



$W(G,O_2) = 5$



Graph Width

- Now we may define the width of a graph, independent of the ordering used.

Definition: Graph width

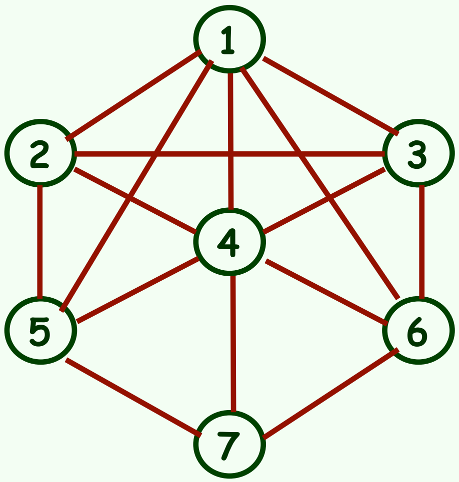
The width of a graph is the lowest width of the graph over all possible total orderings.

In the example, it is easy to see that the width of the graph is 3.

a) Ordering O_1 assigns width 3 to the graph. Hence the graph width is not greater than 3.

b) A width of 2 on a graph with 7 nodes would require the graph to have at most 0+1+5*2 = 11 edges. Hence, the width of the graph cannot be less than 3.

c) From a) and b) the width of graph G is 3.



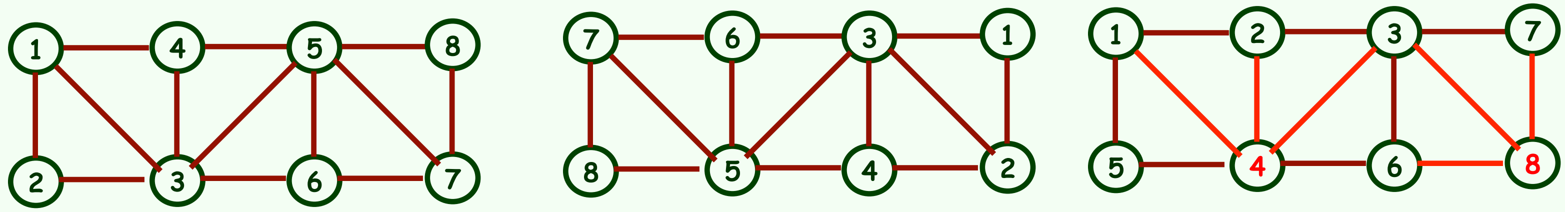
Tractability and i-Consistency

- Now we can present the theorem relating k-consistency and the width of a graph, which indirectly checks whether a problem is tractable.

Theorem: Graph width and Satisfiability

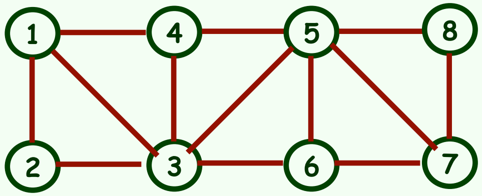
> Let a constraint satisfaction problem be modelled by a constraint network, that after imposing k-consistency leads to a primal graph of width k-1. Under these conditions, any ordering that assigns width k to the primal graph is a backtrack free ordering (BTF).

Example: For the networks below assumed to be path-consistent (3-consistent) O_1 and O_2 are BTF orderings, but O_3 is not.



Tractability and i-Consistency

- In fact, for ordering O3

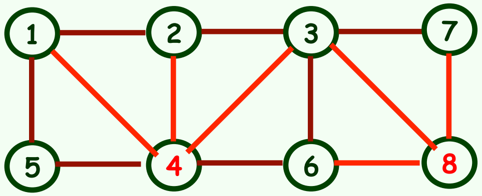


1. every label $\{x_1\text{-}v_1,\ x_2\text{-}v_2\}$, has a support in x_3, say $\{x_3\text{-}v_3\}$.
2. But, label $\{x_1\text{-}v_1,\ x_3\text{-}v_3\}$, has a support in x_4, say $\{x_4\text{-}v_4\}$.
3. Now, label $\{x_3\text{-}v_3,\ x_4\text{-}v_4\}$, has a support in x_5, say $\{x_5\text{-}v_5\}$.
4. Then, label $\{x_3\text{-}v_3,\ x_5\text{-}v_5\}$, has a support in x_6, say $\{x_6\text{-}v_6\}$.
5. And, label $\{x_5\text{-}v_5,\ x_6\text{-}v_6\}$, has a support in x_7, say $\{x_7\text{-}v_7\}$.
6. Finally, label $\{x_5\text{-}v_5,\ x_7\text{-}v_7\}$, has a support in x_8, say $\{x_8\text{-}v_8\}$.

- All things considered, label $\{x_1\text{-}v_1,\ x_2\text{-}v_2,\ x_3\text{-}v_3,\ x_4\text{-}v_4,\ x_5\text{-}v_5,\ x_6\text{-}v_6,\ x_7\text{-}v_7, x_8\text{-}v_8\}$ is a solution of the problem, and was found with no backtracking

Tractability and i-Consistency

- However, for ordering O3
  - every label $\{x_1\text{-}v_1, x_2\text{-}v_2\}$, has a support in x_4, say $\{x_4\text{-}u_4\}$.
  - every label $\{x_1\text{-}v_1, x_3\text{-}v_3\}$, has a support in x_4, say $\{x_4\text{-}v_4\}$.



- But there is no guarantee that v_4 and u_4 are the same!
- In fact, there might be no value in the domain of x_4 that supports both the assignments $\{x_1\text{-}v_1, x_2\text{-}v_2\}$, and $\{x_1\text{-}v_1, x_3\text{-}v_3\}$.
- If this is the case, after assigning values $\{x_1\text{-}v_1, x_2\text{-}v_2, x_3\text{-}v_3\}$, no value exists for x_4 that is compatible with these and one of them must be backktracked!}.
- The same would happen with variable x_8.

Graph Width

- To take advantage of the relation between i-consistency and induced graph width, it is still necessary to find the width of a graph or, equivalently, one optimal ordering, i.e. one that induces a minimal width.

- Fortunately there is a greedy algorithm (thus polinomial) that finds all optimal orderings. The idea is very simple. Always select (nondeterministically) a node with the least number of adjacent nodes (less degree) . Put it in the back of the ordering, delete all the arcs leading to the node, and proceed recursively.

```
function min-width(G: set of Nodes, A: set of Arcs):
        Sequence of Nodes;
   if G.nodes = {n} then
      L ← [n]
   else
      n <- arg_N min {degree(n,G,A)}
      G1.arcs ← G.arcs \ {A: A = (_,N)  ∨ A = (N,_)
      G1.nodes ← G.nodes\{N}
      L ← min-width(G1) + [ n ]
   end if
   min-width ← L
 end function
```

Network Consistency and Satisfiability

- So, in addition to

Case 1: A network of binary constraints, whose variables have only 2 values in their domain, is satisfiable iff it can be made path-consistent.

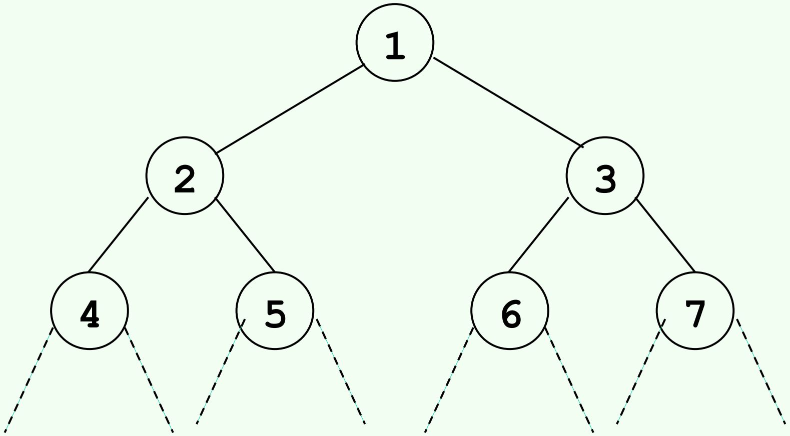
we have

Case 2: A network of constraints (of any arity), whose primal graph has width k is satisfiable iff it is k+1-consistent.

- For example:

2-consistency (i.e. arc-consistency) of the constraint network guarantees the satisfaction of the associated constraint problem, if all constraints are binary and the constraint graph has the topology of a tree.

A BTF ordering proceeds from the root to the leaves



Arc-consistency: special purpose propagators

Some constraints may take advantage of some special features to improve the efficiency of their propagators.

Take for example the propagator for the n-queens problem: **no_attack(i, q_i, j, q_j)**.

The usual arc-consistency would propagate the constraint (i.e. prune each of the values in the domain of q_1/q_2 with no supporting value in q_2/q_1), whenever the constraint is taken from the queue (assuming an AC-3 type algorithm).

However, it is easy to see that a queen with 4 values in the domain offers at least one support value to any other queen.

In fact a queen q_i can only be attacked by 3 queens from another row j. Hence the 4th queen in row j will not attack it.

Hence, the propagator for no_attack should first check the cardinality of the domains, and only check for supports when one of the queens have a domain with cardinality of 3 or less!

Non-Binary Constraints: Bounds-consistency

In numerical constraints (equality and inequality constraints) it is very usual not to impose a too demanding arc-consistency, but rather to impose mere **bounds consistency**.

Take for example the simple constraint **a < b** over variables **a** and **b** with domains 0..1000.

In such inequality constraints, the only values worth considering for removal are related to the bounds of the domains of these variables.

In particular, the above constraint can be compiled into

`max(a) < max(b)` and `min(b) < min(a)`

In practice this means that the values that can be safely removed are

all values of **a** above the maximum value of **b**;

all values of **b** below the minimum value of **a**;

These values can be easily removed from the domains of the variables.

Non-Binary Constraints: Bounds-consistency

It is interesting to note how this kind of consistency detects contradictions.

Take the example of **a < b** and **b > a**, two clearly unsatisfiable constraints. If the domains of **a** and **b** are the range 1..1000, it will take about **500** iterations to detect contradiction

a:: 1 .. 1000, b:: 1 .. 1000	**a < b →**	**a:: 1 .. 999, b:: 2 .. 1000**
a:: 1 .. 999, b:: 2 .. 1000	**a > b →**	**a:: 3 .. 999, b:: 2 .. 998**
a:: 3 .. 999, b:: 2 .. 998	**a < b →**	**a:: 3 .. 997, b:: 4 .. 998**
a:: 3 .. 997, b:: 4 .. 998	**a > b →**	**a:: 5 .. 997, b:: 4 .. 996**
	....	
a:: 499..501, b:: 498..500	**a < b →**	**a::499..499, b::500..500**
a:: 500..500, b:: 500..500	**a > b →**	**a::501..500, b::500..499**

Now, the lower bound is greater than the upper bound of the variables domains, which indicates constradiction!

Non-Binary Constraints: Bounds-consistency

This reasoning can be extended to more complex numerical constraints involving numerical expressions:.

Example: **`a + b ≤ c`**

The usual compilation of this constraint is

`max(a) ≤ max(c) – min(b)`	to prune high values of **a**
`max(b) ≤ max(c) – min(a)`	to prune high values of **b**
`min(c) ≥ min(a) + min(b)`	to prune high values of **a**

Many numerical relations envolving more than two variables can be compiled this way, so that the corresponding propagators achieve bounds consistency.

This is particularly useful when the domains are encoded not as lists of elements but as pairs **min .. max** as is usually the case for numerical variables.

Enforcing generalised arc-consistency: GAC-3

- All algorithms for achieving arc-consistency can be adapted to achieve **generalised arc-consistency** (or **domain-consistency**) by using a modified version of the revise_dom predicate, that for every k-ary constraint checks support values from each variable in the remaining k-1 variables.

```
predicate revise_gac(V,D, c ∈ C): boolean;
   R <- ∅;
   for x_i in vars(c)
      v_i in dom(X_i) do
      Y  = vars(c) \ {x_i} ;
      if ¬ ∃ V in dom(Y): satisfies({x_i-v_i, Y-V}, c) then
         dom(X_i) <- dom(x_i) \ {v_i};
         R <- R ∪ {i};
      end if
   end for
   revise_gac <- R;
end predicate
```

Enforcing generalised arc-consistency: GAC-3

- The GAC-3 algorithm is presented below, as an adaptation of AC-3.
- Any time a value is removed from a variable X_i, all constraints that have this variable in the scope are placed back in the queue for assessing their local consistency.

```
procedure AC-3(V, D, C);
   NC-1(V,D,C);          % node consistency
   Q = { c | c ∈ C};
   while Q ≠ ∅ do
      Q = Q \ {c}   % removes an element from Q
      for i in revise_gac(V,D, c ∈ C) do   % revised xi
         Q = Q ∪ {r | r ∈ C ∧ i ∈ vars(r) ∧ r ≠ c }
      end if
   end while
end procedure
```

Complexity of GAC-3

Time Complexity of GAC-3: $O(a\ k^2\ d^{k+1})$

- Every time that an hyper-arc/n-ary constraint is removed from the queue Q, predicate revise_gac is called, to check at most **$k*d^k$** tuples of values.
- In the worst case, each of the **a** constraints is placed into the queue at most **$k*d$** times.
- All things considered, the worst case time complexity of GAC-3, is $O(kd^k*a*kd)$

$$O(a\ k^2\ d^{k+1})$$

- Of course, when all the constraint are binary the complexity of GAC-3 is the same of AC-3, i.e.

$$O(a\ d^3)$$

Constraint Propagation

Generalised arc-consistency provides a scheme for an architecture of constraint solvers, even when constraints are not binary.

For every constraint a number of propagators are considered. In general, each propagator:

- affects one variable (aiming at narrowing its domain, when invoked);
- Is triggered by some events, namely some change in the domain of some variable;

For example, the posting of the constraint c :: x + y = z creates 3 propagators

$$P_x: x \leftarrow y - z \quad ; \quad P_y: y \leftarrow z - x \quad ; \quad P_z: z \leftarrow x + y$$

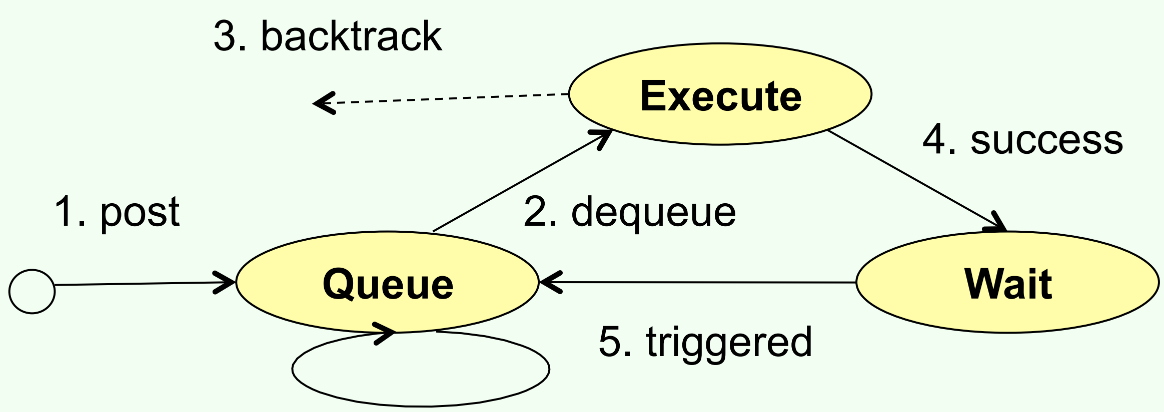
Propagator P_x (likewise for propagators P_y and P_z) is triggered by some change in the domain of variables y or z.

When executed it (possibly) narrows the domain of x. If this becomes empty, a failure is detected and backtracks is enforced.

Constraint Propagation

The life cycle of such propagators can be schematically represented as follows:

1. Propagators are created when the corresponding constraint is posted. They are enqueued and become ready for execution.
2. When they reach the front of the queue they are executed. Upon execution the domain of the propagator variable is possibly narrowed.
3. If the domain is empty, backtracking occurs, and after trailing, the propagator is put back in the queue.
4. Otherwise, the propagator stays waiting for a triggering event.
5. When one such event occurs the propagator is enqueued . While enqueued, other triggering events are possibly "merged" in the queue.



Constraint Propagation

$P_x: x \leftarrow y - z$; $P_y: y \leftarrow z - x$; $P_z: z \leftarrow x + y$

Propagators aim at maintaining some form of consistency, typically domain consistency or bounds consistency, This has a direct influence on the events that trigger them.

For example, with bounds consistency, propagator P_x is triggered when the maximum or minimum values in the domain of variables y and z is changed. These are the only events that change the maximum and minimum values of the domain of variable x.

In contrast, if domain consistency is maintained, propagator P_x is triggered whenever any value is removed from the domain of any of the variables y or z, since these removals may end the support of some value in the domain of x.

This also means that sometimes the activation of the propagator does not lead to the removal of any value in the domain. For example value 3 in x may be supported by either values 5 and 2, or by values 7 and 4 for variables y and z. If 7 is removed from the domain of y, x= 3 still has support in y and z.

Generalised arc-consistency: Global Constraints

The time complexity of generalised arc consistency for n-ary constraints may be too costly. Consider the case of k variables that all have to take different values.

$$x_1 \neq x_2, x_1 \neq x_3 \dots x_1 \neq x_k \dots x_{k-1} \neq x_k$$

These k(k-1)/2 binary constraints can be replaced by a single k-ary constraint

$$\textbf{all\_diffferent}(x_1 , x_2, x_3 , .. , x_k)$$

However, checking the consistency of such constraint by the naïve method presented, would have complexity $O(a\ k^2\ d^{k+1})$, i.e. $O(k^4\ d^{k+1})$.

This is why, some very widely used n-ary constraints are dealt with as **global constraints**, for which special purpose, and much faster, algorithms exist to check the constraint consistency.

In the all_different constraint, an algorithm based in graph theory enforces this checking with complexity $O(d\ k^{3/2})$, much better than the naïve version.

For example for $d \approx k \approx 9$ (sudoku problem!) the number of checks is reduced from $9^2 * 9^{10} \approx 3*10^{10}$ to a much more acceptable number of $9 * 9^{3/2} \approx 243$.