Constraint Programming

- An overview

  - Node-consistency
  - Arc-consistency
  - Path-consistency
  - l-consistency
  - Generalised Arc-consistency
  - Bounds-Consistency
  - Propagators: The COMET example

Complexity of Search

- Given a problem **P = <X, D, C>** with **n** variables $\mathbf{X_1,..,X_n}$ the potential search space where solutions can be found (i.e. the leaves of the search tree with compound labels $\mathbf{\{<X_1\text{-}v_1>, ..., <X_n\text{-}v_n>\}}$) has cardinality

$$\mathbf{\#S = \#D_1 * \#D_2 * ... * \#D_n}$$

- Assuming the domains of all variables have the same cardinality ($\mathbf{\#D_i = d}$) the search space has cardinality

$$\mathbf{\#S = d^n}$$

  which is exponential on the "size" **n** of the problem.

- In general the exponential nature of the search space cannot be avoided (in NP complete problems), its size can nonetheless be significantly reduced .

Complexity of Search

- If instead of the cardinality **d** of the initial problem, one solves a reduced problem whose domains have lower cardinality **d'** (<d) the size of the potential search space also decreases exponentially!

$$S'/S = d'^{n} / d^{n} = (d'/d)^{n}$$

- Such exponential decrease may be very significant for "reasonably" large values of **n**, as shown in the table.

		n									
S/S'		**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**	**100**
7	**6**	**4.6716**	**21.824**	**101.95**	**476.29**	**2225**	**10395**	**48560**	**226852**	**1E+06**	**5E+06**
6	**5**	**6.1917**	**38.338**	**237.38**	**1469.8**	**9100.4**	**56348**	**348889**	**2E+06**	**1E+07**	**8E+07**
5	**4**	**9.3132**	**86.736**	**807.79**	**7523.2**	**70065**	**652530**	**6E+06**	**6E+07**	**5E+08**	**5E+09**
4	**3**	**17.758**	**315.34**	**5599.7**	**99437**	**2E+06**	**3E+07**	**6E+08**	**1E+10**	**2E+11**	**3E+12**
3	**2**	**57.665**	**3325.3**	**191751**	**1E+07**	**6E+08**	**4E+10**	**2E+12**	**1E+14**	**7E+15**	**4E+17**
d	**d'**										

Propagation in Search

- The effort in reducing the domains must be considered within the general scheme to solve the problem.

- In Constraint Programming, the specification of the constraints precedes the enumeration of the variables.

```
Problem :=
    Declaration of Variables and Domains,
    Specification of Constraints,
    Labelling of the Variables.
```

- In general, search is performed exclusively on the labelling of the variables.

- The execution model alternates enumeration with propagation, making it possible to reduce the problem at various stages of the solving process.

Propagation in Search

- The execuction model follows the following pattern:

```
Declaration of Variables and Domains,
Specification of Constraints,
   indomain(X1), % value selection with backtraking
   propagation,  % reduction of problem X2 to Xn
   indomain(X2),
   propagation,  % reduction of problem X3 to Xn
      ...
   indomain(Xn-1)
   propagation,  % reduction of problem Xn
  indomain(Xn)
```

Complexity of Search

- In practice, this potential narrowing of the search space has a cost involved in finding the redundant values (and labels).

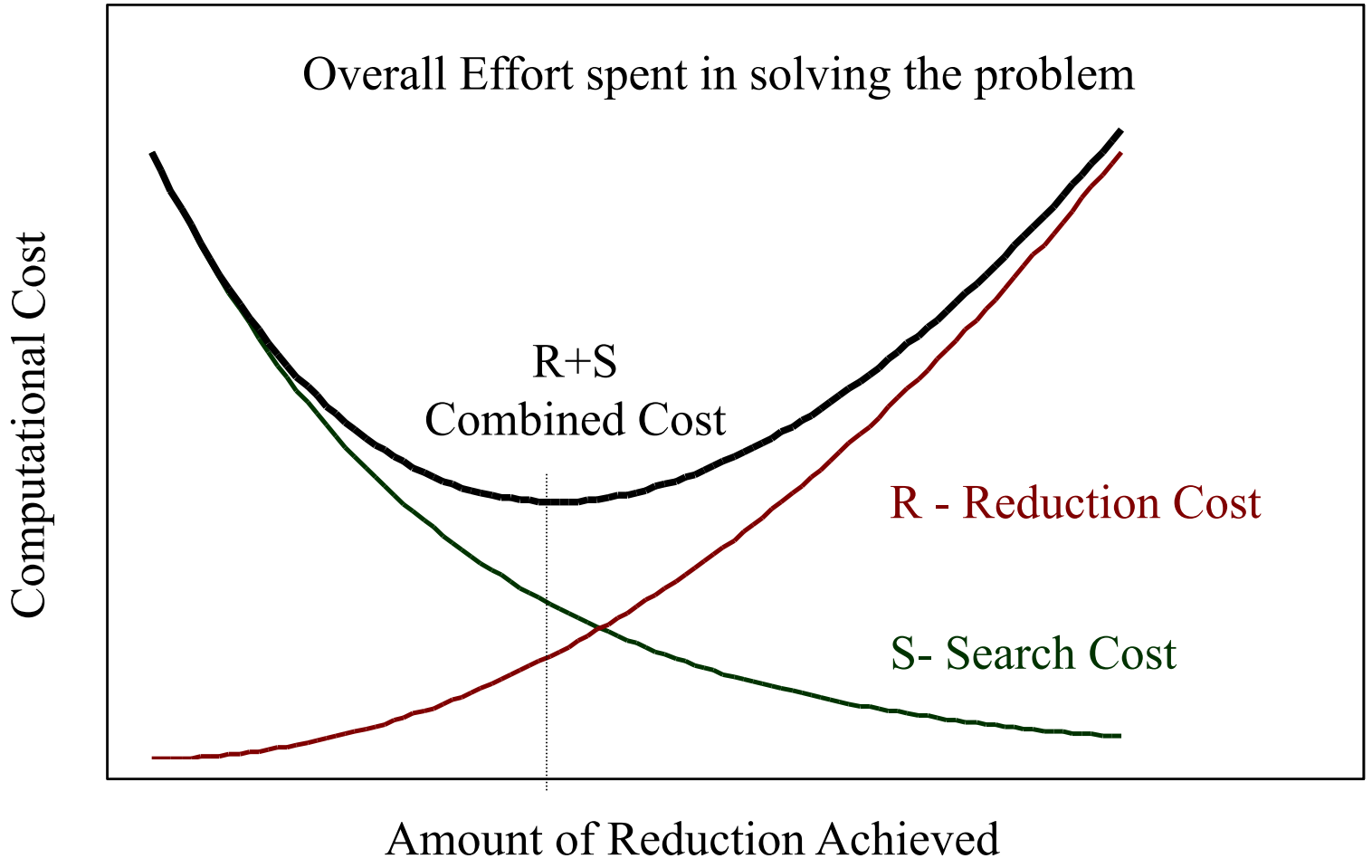
- A detailed analysis of the costs and benefits in the general case is extremely complex, since the process depends highly on the instances of the problem to be solved.

- However, it is reasonable to assume that the computational effort spent on problem reduction is not proportional to the reduction achieved, becoming less and less efficient.

- After some point, the gain obtained by the reduction of the search space does not compensate the extra effort required to achieve such reduction.

Complexity of Search

- Qualitatively, this process may be represented by means of the following graph



Propagation: Consistency Criteria

- Consistency criteria enable to establish redundant values in the variables domains in an indirect form, i.e. requiring no prior knowledge on the set of problem solutions.

- Hence, procedures that maintain these criteria during the “propagation” phases, will eliminate redundant values and so decrease the search space on the variables yet to be enumerated.

- For constraint satisfaction problems with binary constraints, the most usual criteria are, in increasingly complexity order,

  - **Node Consistency**
  - **Arc Consistency**
  - **Path Consistency**
  - **Consistency-i**

Node - Consistency

Definition (**Node Consistency**):

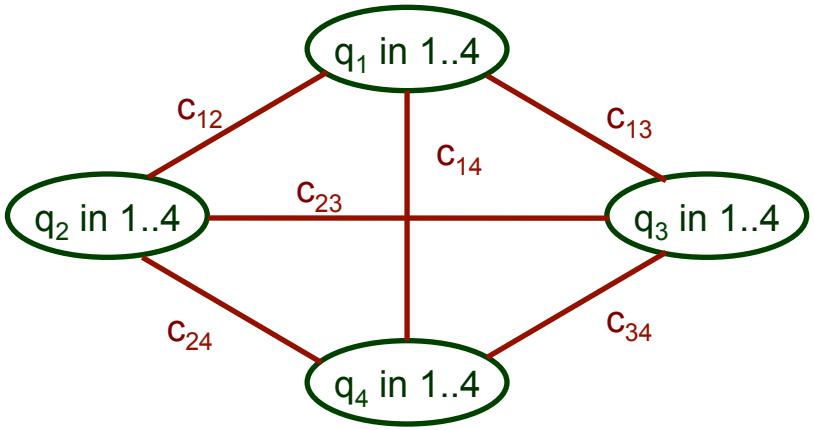
> A constraint satisfaction problem is **node-consistent** if no value on the domain of its variables violates the unary constraints.

- This criterion may seem both obvious and useless. After all, who would specify a domain that violates the unary constraints ?!
- However, this criterion must be regarded within the context of the execution model that incrementally completes partial solutions. Constraints that were not unary in the initial problem become so when one (or more) variables are enumerated.

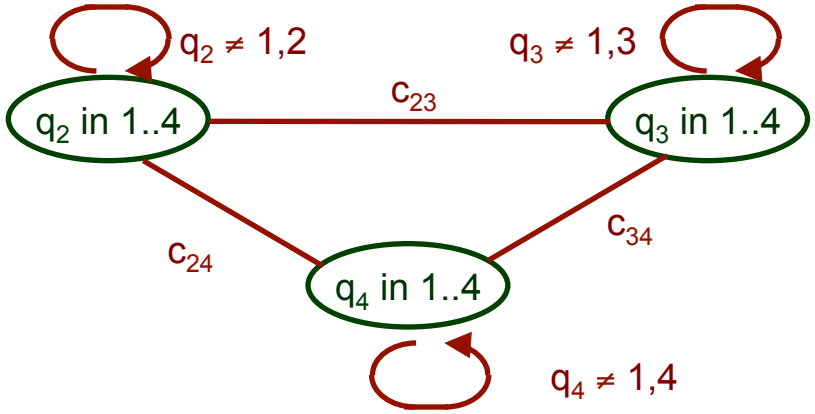
Node - Consistency

Example:

- After the initial posting of the constraints, the constraint network model at the right represents the 4-queens problem.

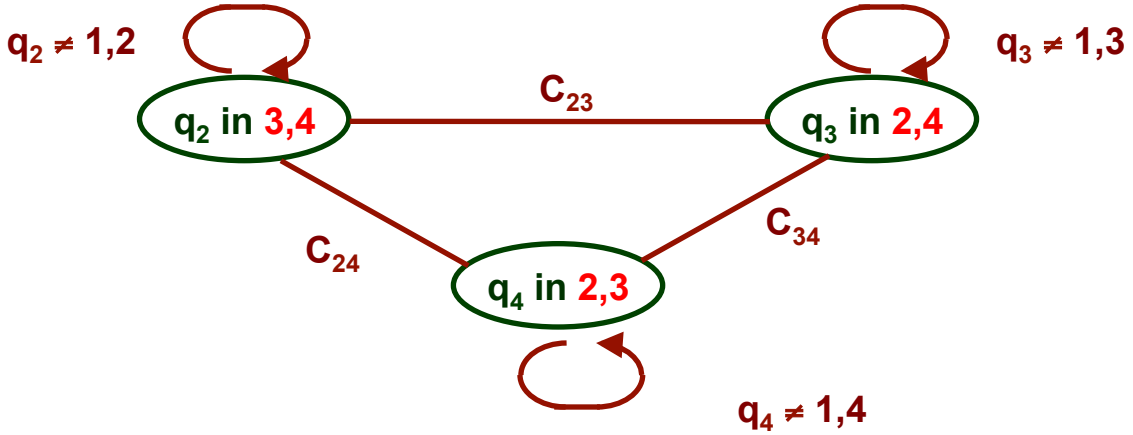


- After enumeration of variable q_1, i.e. q_1=1, constraints c_{12}, c_{13} and c_{14} become **unary** !!



Node - Consistency

- An algorith that maintains node consistency should remove from the domains of the “future” variables the appropriate values.



- Maintaining node consistency achieves the following domain reduction.

●			
1	1		
1		1	
1			1

$q_2 \neq 1,2$

$q_3 \neq 1,3$

$q_4 \neq 1,4$

Enforcing Node-Consistency

- Before discussing other more demanding criteria (path consistency, consistency-i) we now address some algorithms to enforce node- and arc-consistency.

Enforcing node consistency: Algorithm NC-1

- This is a very simple algorithm shown below:

-

```
procedure NC-1(X, D, R);
   for x in X
      for v in dom(x) do
         for c in {cons(x): vars(c) = {x}} do
            if not satisfy(x-v, c) then
              dom(x) <- dom(x) \ {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.

Arc- Consistency

- A **more** demanding and complex criterion of consistency is that of arc-consistency

Definition (**Arc Consistency**):

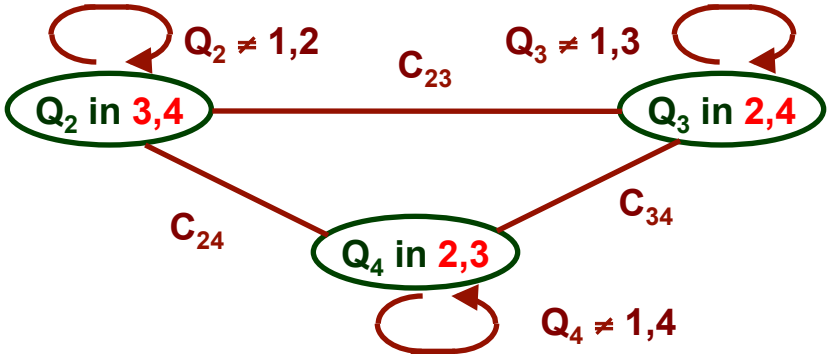
A constraint satisfaction problem is arc-consistent if,

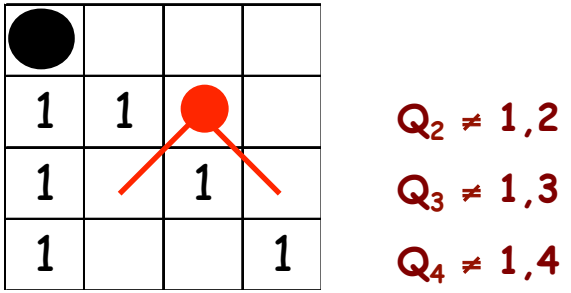
- It is node-consistent; and
- For every label X_i-v_i of every variable X_i, and for all constraints C_{ij}, defined over variables X_i and X_j, there must exist a value v_j that **supports** v_i, i.e. such that the compound label {X_i-v_i, X_j-v_j} satisfies constraint Cij.

Arc- Consistency

Example:

- After enumeration of variable Q_1=1, and making the network node-consistent, the 4 queens problem has the following constraint network:





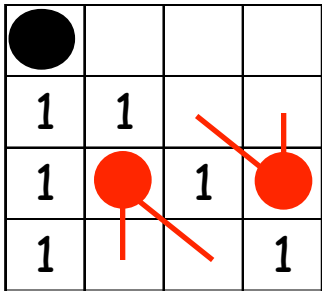
- However, label Q_2-3 has no support in variable Q_3, since neither compound label {Q_2-3 , Q_3-2} nor {Q_2-3 , Q_3-4} satisfy constraint C_{23}.

- Therefore, value 3 can be safely removed from the domain of Q_2.

Arc- Consistency

Example (cont.):

- In fact, none (!) of the values of Q_3 has support in variables Q_2 and Q_4, as shown below:



$Q_2 \neq 1,2$

$Q_3 \neq 1,3$

$Q_4 \neq 1,4$

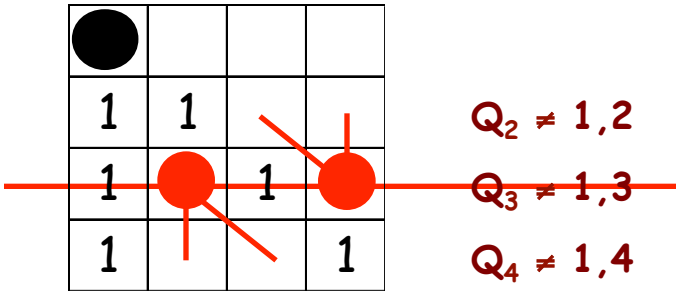
- label Q_3-4 has no support in variable Q_2, since none of the compound labels {Q_2-3, Q_3-4} and {Q_2-4, Q_3-4} satisfy constraint C_{23}.

- label Q_3-2 has no support in variable Q_4, since none of the compound labels {Q_3-2, Q_4-2} and {Q_3-2, Q_4-3} satisfy constraint C_{34}.

Arc- Consistency

Example (cont.):

- Since none of the values from the domain of Q_3 has support in variables Q_2 and Q_4, maintenance of arc-consistency **empties** the domain of Q_3!



- Hence, maintenance of arc-consistency not only prunes the domain of the variables but also antecipates the detection of unsatisfiability in variable Q_3 ! In this case, backtracking of Q_1=1 may be started even before the enumeration of variable Q_2.

- Given the good trade-of between pruning power and simplicity of arc-consistency, a number of algorithms have been proposed to maintain it.

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 a_{ki} ($k \neq i$) 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}).

```
procedure AC-3(X, D, C);
    NC-1(X,D,C);          % node consistency
    Q = {a_ij | c_ij ∈ C ∨ c_ji ∈ C };
    while Q ≠ ∅ do
        Q = Q \ {a_ij}    % removes an element from Q
        if revise_dom(a_ij,X,D,C) then    % revised X_i
            Q = Q ∪ {a_ki | c_ki∈ C ∨ c_ik∈ C ∧ k≠i ∧ k≠j}
        end if
    end while
end procedure
```

Enforcing Arc-Consistency: AC-3

Revise-Domain

- Algorithm AC-3 (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,X,D,C): Boolean;
    success <- false;
    for v_i in dom(x_i) do
        if ¬ ∃v_j in dom(x_j): satisfies({x_i-v_i,X_j-v_j},c_ij) then
            dom(x_i) <- dom(x_i) \ {v_i};
            success <- true;
        end if
    end for
    revise_dom <- success;
end predicate
```

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 $\mathbf{O(ad^2)} \approx \mathbf{O(n^2d^2)}$, 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, in contrast with AC-1, with timel complexity $\mathbf{O(nad^3)}$, the worst-case time complexity of AC-3 is $O(2ad * d^2)$, i.e.

$$\mathbf{O(ad^3)}$$

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} ($k \neq i$ and $k \neq j$) leading to that variable are reexamimed.
- 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 value v_k of x_k!

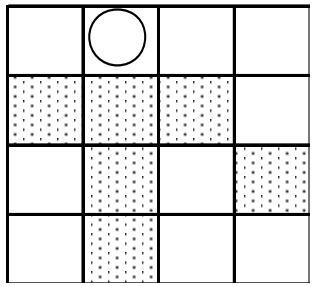
This idea is exploited in algorithm **AC-4,** that uses a number of new data-structures

- **Counters**: For counting support values of label $\{x_i\text{-}v_i\}$ in x_j
- **Suporting Sets**: That explicitly enumerate the labels $\{x_j\text{-}v_j\}$ that are supported by label $\{x_i\text{-}v_i\}$, 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 $\{x_i\text{-}v_i\}$ 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_1-2
  - $c(2, q_1, q_3) = 2$ % q_3-1 and q_3-3 do not attack q_1-2
  - $c(2, q_1, q_4) = 3$ % q_4-1, q_4-3 and q_4-4 do not attack q_1-2



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_2-3, q_2-4, 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_3-3, q_3-4}**
  - **sup(3,q_1) = {q_2-1, q_3-2, q_4-4, 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(X,D,C);
   M <- 1; sup <- ∅; List = ∅;
   for c_ij in C do
      for v_i in dom(x_i) do
         ct <- 0;
         for v_j in dom(x_j) do
            if satisfies({x_i-v_i, x_j-v_j}, c_ij) then
               ct <- ct+1; sup(v_j,x_j)<- sup(v_j,x_j) ∪ x_i-v_i}
             end if
         endfor
        if ct = 0 then M[x_i,v_i] <- 0; List <- List ∪ {x_i-v_i};
                       dom(x_i) <- dom(x_i)\{v_i}
        else c(v_i, x_i, x_j) <- ct;
        end if
      end for
  end for
end procedure
```

Enforcing Arc-Consistency: AC-4

Algorithm AC-4 (propagation phase)

```
procedure propagate_AC-4(X,D,C);
    while List ≠ ∅ do
       List <- List\{x_i-v_i} % remove element from List
       for x_j-v_j in sup(v_i,x_i) do
           c(v_j,x_j,x_i) <- c(v_j,x_j,x_i) - 1;
           if c(v_j,x_j,x_i) = 0 ∧ M[x_j,v_j] = 1 then
              List = List ∪ {x_j-v_j};
              M[x_j,v_j] <- 0;
              dom(x_j) <- dom(x_j) \ {v_j}
           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 the much worse space complexity of **$O(ad^2)$**

Enforcing Arc-Consistency: AC-4

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

- 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(ad2).

- In the inner cycle of the propagation procedure 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 $\mathbf{2ad^2}$ times, which determines the time complexity of the propagation phase of AC-4 to be $\mathbf{O(ad^2)}$

Enforcing Arc-Consistency: AC-4

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

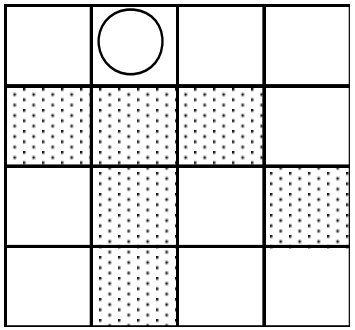
- To check whether a network is arc consistent it is necessary to test, for each constraint c_{ij}, that the **d** pairs x_i-v_i have support in x_j, each requiring at most **d** tests. Since each constraint is considered twice, $\mathbf{2ad^2}$ tests are required, with assymptotic complexity $\mathbf{O(ad^2)}$ similar to that of AC-4.
- However, one should bear in mind that the worst case complexity is *assymptotic*. 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 counting the values v_i from variable x_i that support a pair x_j-v_j, it simply maintains the lowest such v_i.

- 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. Hence, in AC-6, the supporting sets become singletons.



- **Data Structures of Algorithm AC-6**
  - The **List** is adapted
  - Boolean **matrix M from** AC-4 iskept.
  - ~~The AC-4 **counters** are disposed of~~;
  - The supporting sets become "singletons" .

  - $sup(1,q_1)$ = {q_2-3, q_2-4, 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_3-4}
  - $sup(3,q_1)$ = {q_2-1, q_3-2, q_4-4, 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-6

- Both phases of AC-6 use predicate

```
next_support(xi,vi,xj,vj, out v)
```

that succeeds if there is in the domain of x_j a "next" supporting value **v**, i.e the lowest value, no less than v_j, such that x_j-**v** supports x_i-v_i.

```
predicate next_support(x_i,v_i,x_j,v_j, out v): boolean;
     sup_s <- false; v <- v_j;
     while not sup_s and v =< max(dom(x_j)) do
        if not satisfies({x_i-v_i,x_j-v},c_ij) then
            v <- next(v,dom(x_j))
         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 inicialise_AC-6(X,D,C);
  List <- ∅; M <- 0; sup <- ∅;
  for c_ij in C do
  for v_i in dom(x_i) do
    v_j = min(dom(x_j))
    if next_support(x_i,v_i,x_j,vj,v) then
       sup(v_i,X_i)<- sup(v_i,X_i) ∪ {X_j-v}
    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(X,D,C);
   while List ≠ ∅ do
      List <- List\{x_j-v_j} % removes x_j-v_j from List
      for x_i-v_i in sup(v_j,x_j) do
         sup(v_i,x_i) <- sup(v_i,x_i) \ {x_j-v_j} ;
         if M[x_i,v_i] = 1 then
            if next_suport(x_i,v_i,x_j,v_j,v) then
               sup(v,x_j)<- sup(v,x_j) ∪ {x_i-v_i}
            else
               dom(x_i) <- dom(x_i)\{v_i}; M[x_i,v_i] <- 0;
               List <- List ∪ {x_i-v_i}
            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 $\mathbf{c_{ij}}$, each of the **d** pairs $\mathbf{x_i}$-$\mathbf{v_i}$ is supported by a single value $\mathbf{v_j}$ form $\mathbf{x_j}$ (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

  $$\texttt{next\_support}(x_i, v_i, x_j, v_j, v)$$

  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.**

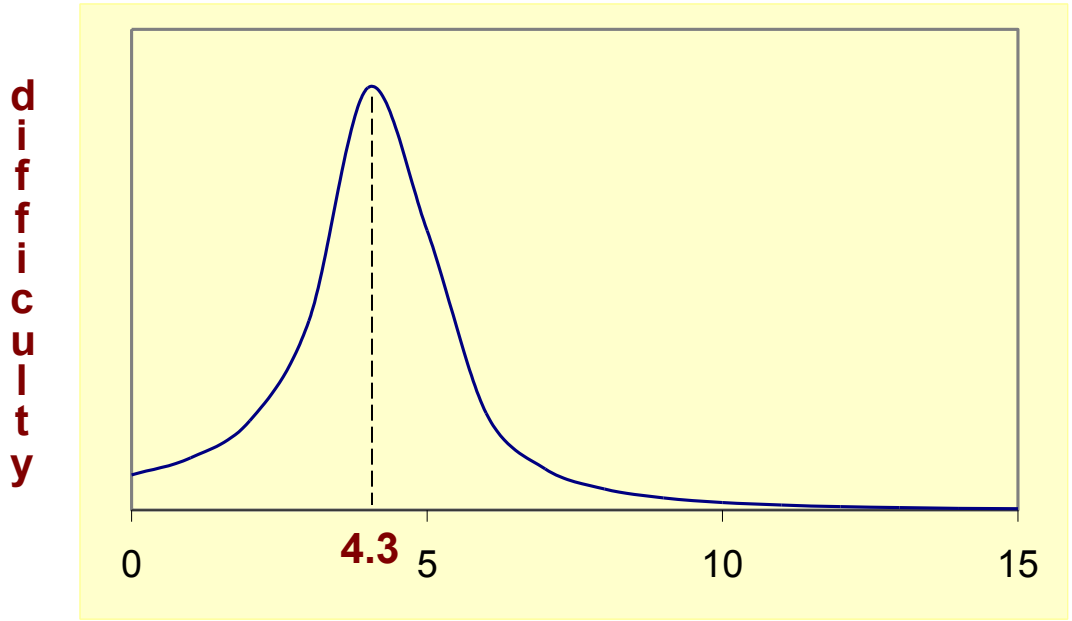
Enforcing Arc-Consistency: AC-6

- **Typical** complexity of AC-6
  - 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.

Enforcing Arc-Consistency: AC-6

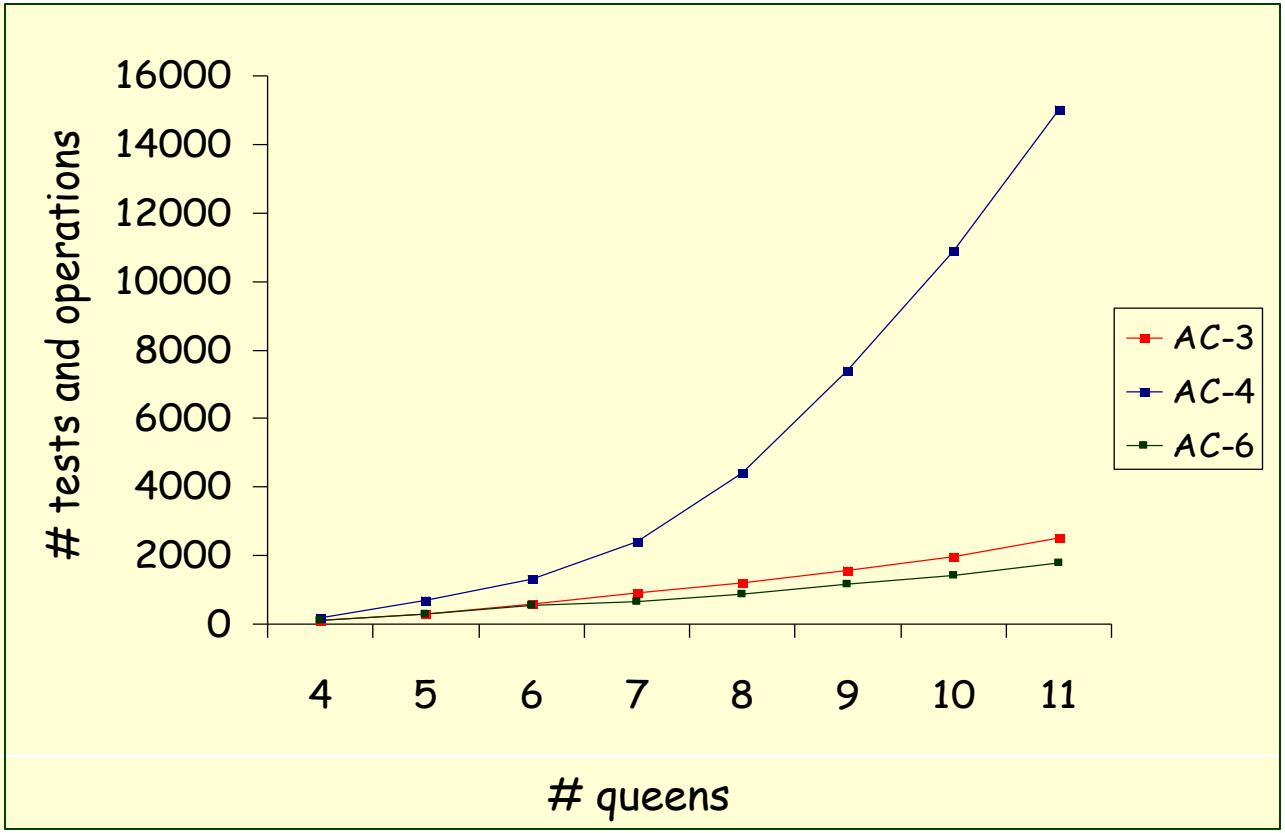
- This phase transition typically contains the most difficult instances of the problem, and separates the instances that are trivially satisfied from those that are trivially insatisfiable.

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



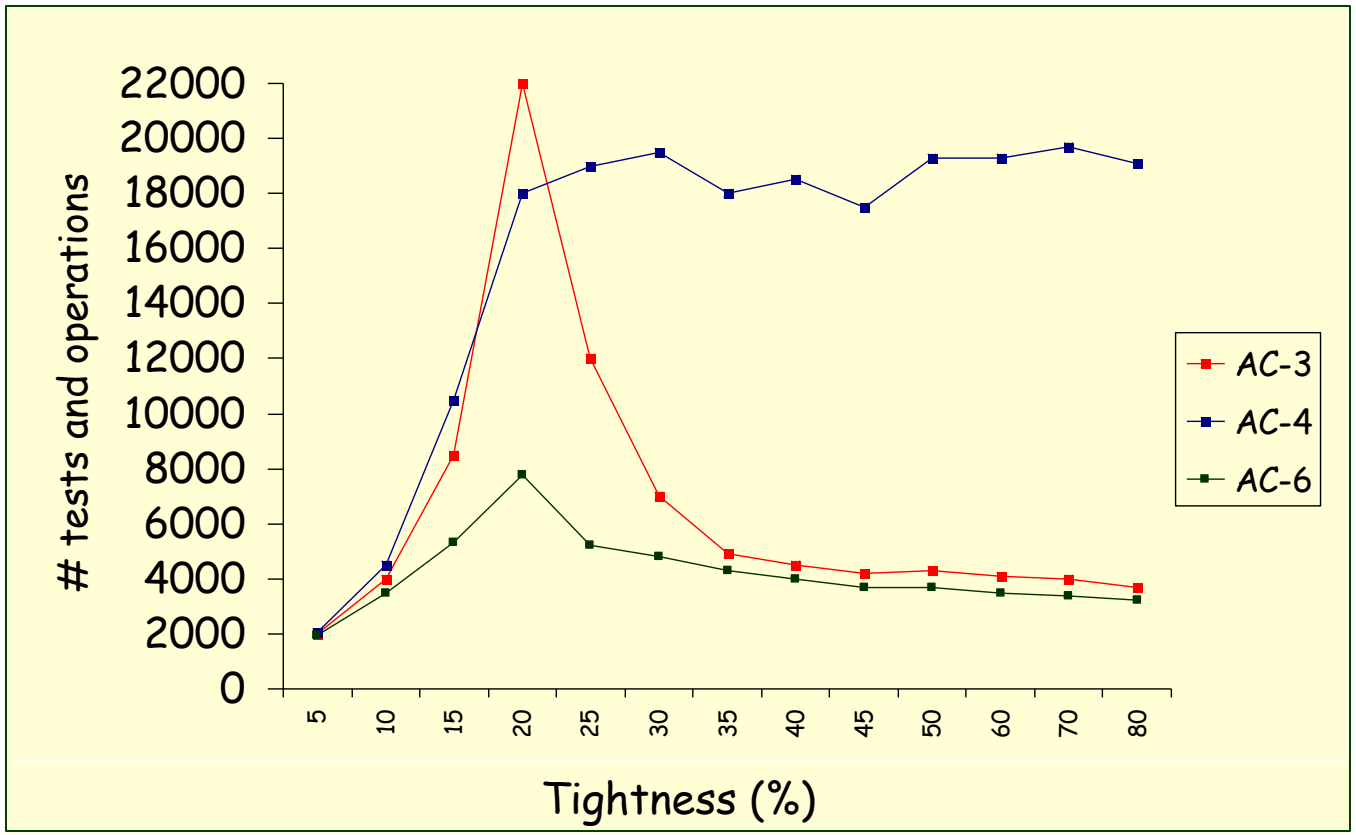
Enforcing Arc-Consistency: AC-6

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



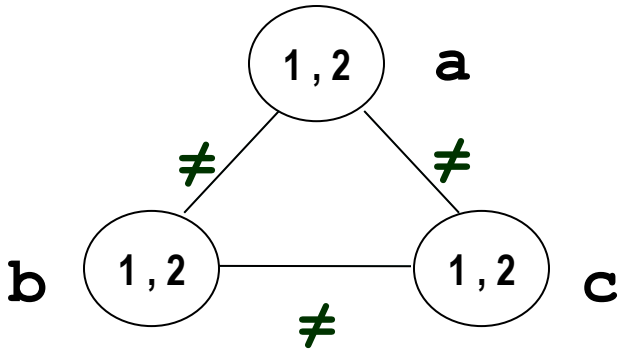
Enforcing Arc-Consistency: AC-6

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



Constraints: Path-Consistency

- The following constraint network is obviously inconsistent:



- Nevertheless, it is arc-consistent: every binary constraint of difference (≠) is arc-consistent whenever the constraint variables have at least 2 elements in their domains.

- However, is is not path-consistent: no consistent label {<a-v_a>, <b-v_b>} can be extended to the third variable (c).

  {<a-1>, <b-2>} → c <>1, c <> 2 {<a-2>, <b-1>} → c <>1, c <> 2

- This property is captured by the notion of path-consistency.

Path- Consistency

Definition (**Path Consistency**):

A constraint satisfaction problem is arc-consistent if,

- It is arc-consistent; and
- Every consistent compound label $\{x_i\text{-}v_i,\ x_j\text{-}v_j,\}$ can be extended to consistent label with a third variable x_k ($k \neq i$ and $k \neq j$).

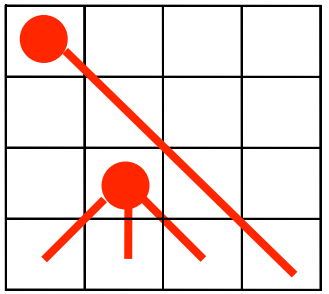
The second condition is more easily understood as

- For every compound label $\{x_i\text{-}v_i,\ x_{ij}\text{-}v_j,\}$ there must be a value v_k that **supports** $\{x_i\text{-}v_i,\ x_{ij}\text{-}v_j,\}$, i.e. the compound label $\{x_i\text{-}v_i,\ x_j\text{-}v_j,\ x_k\text{-}v_k\}$ satisfies constraints C_{ij}, C_{ik}, and C_{kj}.

Path- Consistency

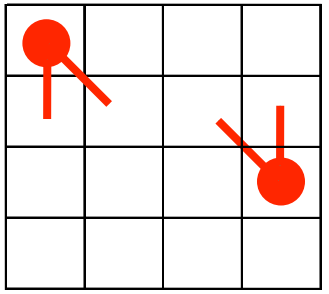
Example:

- By enforcing path consistency it is possible to avoid backtracking in the 4-Queens problem.
- In fact, q_1-1 has only two supports in variable q_3, namely q_3-2 and q_3-4.



However:

- $<q_1\text{-}1, q_3\text{-}2>$ cannot be extended to variable q_4
- $<q_1\text{-}1, q_3\text{-}4>$ cannot be extended to variable q_2

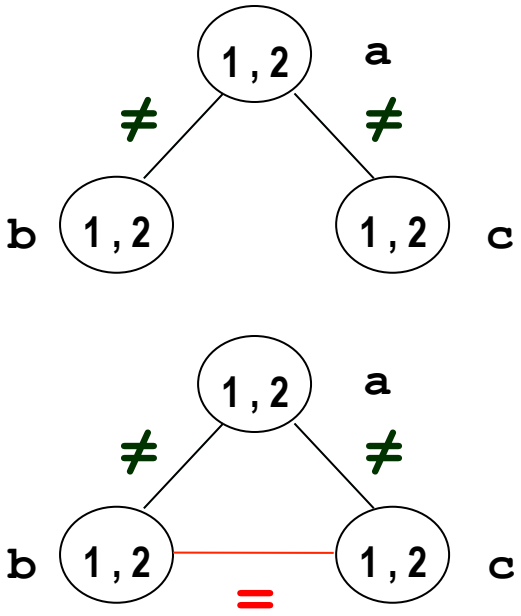


- Hence, 1 can be safely removed from the domain of variable q_1.
- With similar reasoning, it may be shown that none of the corners, and none of the centre positions can have a queen.

Path- Consistency

- Despite the previous example, in general maintaining path consistency does not prune the domain of a variable, but rather “forbids” compound labels with cardinality 2.
- This means that imposing arc-consistency on variables x_i and x_j through variable x_k, will tighten the (possible non-existing) constraint between x_i and x_j.

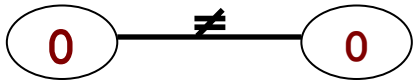
- In the example, a constraint of equality is imposed on variables b and c, because the compound labels { b-1 , c-1 } and { b-2 , c-2 } cannot be extended to variable a.



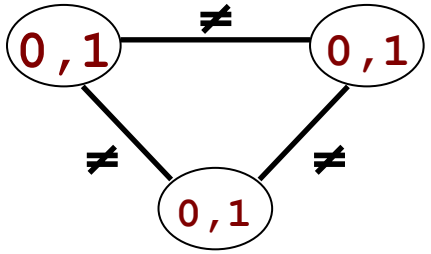
Binary Constraints: i-consistency

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

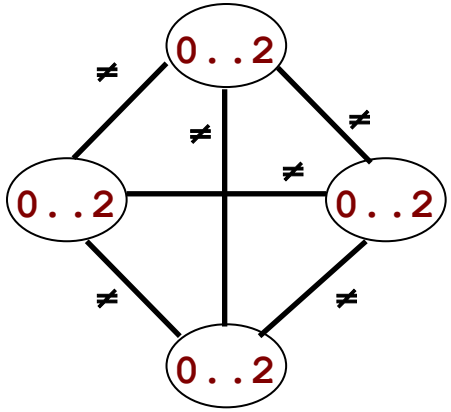
  - A node consistent network, that is not arc consistent



  - 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.

  1. A constraint satisfaction problem (or constraint network) is 1-consistent if the values in the domain of its variables satisfy all the unary constraints.

  2. A network is i-consistency if all labels $<X_{a1}\text{-}v_{a1}, X_{a2}\text{-}V_{a2}, ..., X_{ak}\text{-}v_{ak}>$ of cardinality k = i-1, can be extended to any i-th variable X_{ai}, i.e. there is a v_{ai} in the domain of X_{ai} that satisfies all constraints C defined over any set $S \cup \{X_{ai}\}$ of variables, where S is a strict subset of the i-1 variables X_{a1} to X_{ak} ($S \subset \{X_{a1}, X_{a2}, ..., X_{ak}>$).

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

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

Node-consistency	↔	strong 1-consistency
Arc- consistency	↔	strong 2-consistency
Path-consistency	↔	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.

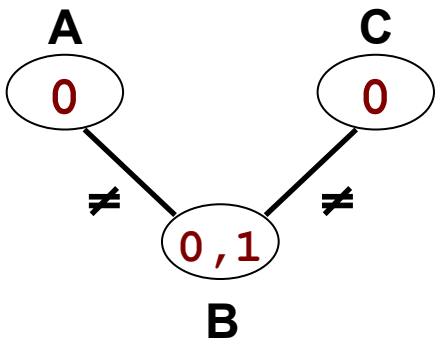
- The only 2-compound labels, that satisfy the constraints

  {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

- The algorithms that were presented for achieving arc- and path- consistency could be adapted to obtain i-consistency, provided that we consider constraints with i-1 arity.
- The adaptation of the AC-4 or AC-6 algorithms leads to optimal assymptotic 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.

Non-Binary Constraints: Generalised arc-consistency

- In networks with binary constraints, arc-consistency can be regarded as imposing consistency in each constraint, seen independently of the others.
- This idea can be generalised for networks where some constraints are non-binary, leading to the criterion of generalised arc-consistency.
- The concept of an arc must be replaced by that of an hyper-arc, and it must be guaranteed that each value in the domain of a variable must be supported by values in the other variables that share the same hyper-arc (constraint).

Definition (Generalised Arc Consistency):

A constraint satisfaction problem is generalised arc-consistent if,

- It is node-consistent; and
- For every label X_i-v_i of every variable X_i, and for every constraint, defined over variables X_i, X_j ... X_k , there is a tuple {X_i-v_i, X_j-v_j ..., X_k-v_k} that satisfies the constraint.

Enforcing generalised arc-consistency: GAC-3

- All algorithms for achieving arc-consistency can be adapted to achieve generalised arc consistency, by using a modified version of the revise_dom predicate, shown below. For every k-ary constraint, it checks support values from each variable in the remaining k-1 variables.

```
predicate revise_gac(c,X,D,C): boolean;
   R <- ∅;
   for x_i in vars(c)
      for v_i in dom(x_i) do
      {y_1..y_k} = vars(c) \ {x_i} ;
      if ¬ ∃ u_1..u_k in dom{y_1..y_k}:
                satisfies({x_i-v_i,y1-u_1 ... y_k-u_k},c) then
         dom(x_i) <- dom(x_i) \ {v_i};
         R <- R ∪ {x_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 GAC-3(X, D, C);
   NC-1(X,D,C);          % node consistency
   Q = { c | c ∈ C};
   while Q ≠ ∅ do
      Q = Q \ {c}    % removes an element from Q
      for x in revise_gac(c,V,D,C) do    % revised X_i
         Q = Q ∪ {c | c∈ C ∧  x ∈ vars(c) ∧ 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.

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

- But this is not acceptable for many n-ary constraints!

Generalised arc-consistency: Global Constraints

- Take for example the case of k variables that all have to take different values.

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

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

$$\texttt{all\_different([}x_1\texttt{ , }x_2\texttt{, }x_3\texttt{ , .. , }x_k\texttt{])}$$

- However, checking the consistency of such constraint by the naïve method presented, would have complexity $\mathbf{d^k}$.

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

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

- For example for d ≈ k ≈ 9 (sudoku problem!) the number of checks is reduced from $9^9 \approx 400*10^6$ to a much more acceptable number of $9*\ 9^{3/2} \approx 243$.

Non-Binary Constraints: Bounds-consistency

- In numerical constraints (equality and inequality constraints) it is very usual no to impose a demanding arc-consistency but 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

- 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 often the case with numerical 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 contradiction!

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,Q1,j,Qj)**.
- The usual arc-consistency would propagate the constraint (i.e. prune each of the values in the domain of Q1/Q2 with no supporting value in Q2/Q1), 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 the support of one value to any other queen.
- 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!

From Abstract models to Propagators

Algorithms such as AC-3 or AC-6 provide the scheme for constraint propagation, 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

$$P1: X \leftarrow Y - Z \quad ; \quad P2: Y \leftarrow Z - Y \quad ; \quad P3: Z \leftarrow X + Y$$

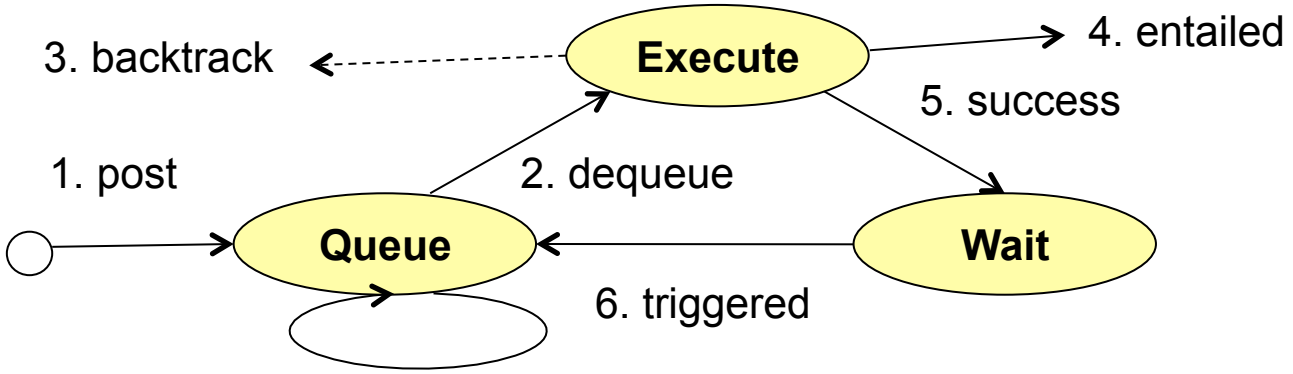
Propagator C1 (likewise for propagators P2 and P3) is triggered by some change in the domain of variables Y or Z.

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

From Abstract models to Propagators

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. In some cases, the constraint is guaranteedly satisfied.
5. Otherwise, the propagator stays waiting for a triggering event.
6. When one such event occurs the propagator is enqueued . While enqueued, other triggering events are possibly “merged” in the queue.



Propagators in COMET

- COMET: In addition to default propagators for the most common constraints, user-defined propagators can be defined by extending the **`UserConstraint<CP>`** Class.
- We illustrate this with the n queens problem below.

```
Solver<CP> cp();
  int n = 8;
  range S = 1..n;
  var<CP>{int} q[i in S](cp,S);
  solve<cp> {
    forall(i in S, j in S: j > i)
      cp.post(noAttack(i, q[i], j, q[j]));
  }
  using {
    forall(i in S) by (/* q[i].getSize(),*/ i)
      tryall<cp> (v in S: q[i].memberOf(v)) by (v)
        cp.label(q[i],v);
  }
```

Propagators in COMET

- The specification of a constraint must include its associated propagators, as well as the actions to perform when the constraint is posted
- The constructor of the constraint simply copies the variables and integers to its internal variables.

```
class noAttack extends UserConstraint<CP> {
  var<CP>{int} _x1;   var<CP>{int} _x2;
  int _r1;            int _r2;

   noAttack(int r1, var<CP>{int} x1,
            int r2, var<CP>{int} x2):UserConstraint<CP>() {
   _x1 = x1;    _x2 = x2;    _r1 = r1;    _r2 = r2;
  }
   Outcome<CP> propagate() { … }
   Outcome<CP> post(Consistency<CP> cl) { … }
}
```

- Two other methods are needed for the posting and propagators.

Propagators in COMET

- The specification of a constraint must include its associated propagators. In this case, we should specify 2 propagators:
  - How `q[i]` propagates to `q[j]`
  - How `q[j]` propagates to `q[i]`
- They are quite similar so below is only shown how variable _x2 supports _x1.

```
Outcome<CP> propagate() {
  forall(c1 in _x1.getMin().._x1.getMax(): _x1.memberOf(c1)){
    bool support = false;
      forall(c2 in _x2.getMin().._x2.getMax(): _x2.memberOf(c2))
        if((c1 != c2 && c1-_r1 != c2-_r2 && c1+_r1 != c2+_r2)){
          support = true; break;
        }
      if(!support && _x1.removeValue(c1) == Failure){
    return Failure;}
  forall (…) {…}
  return suspend; }
```

(wipe out of the domain of _x1) a failure is returned as the outcome of propagation.

- Otherwise the defauld Suspend outcome is returned.

Propagators in COMET

- The specification of the constraint posting, simply propagates and defines the type of consistency to be achieved.
- In the figure, node-consistency is specified with the **`_addBind(this)`** method that associates the constraint to the event Bind event of the variable e.g.. enques the propagator).

```
Outcome<CP> post(Consistency<CP> cl) {
   if(propagate() == Failure)
     return Failure;
   _x1.addBind(this);
   _x2.addBind(this);
   return Suspend;
}
```

- Alternatively, other types of consistency can be by trigerring the propagators with other events:
  - arc-consistency: **`_x1.addDomain(this);`**
  - bounds-consistency: **`_x1.addBounds(this);`**
    (but also : **`_x1.addMin (this)`** and **`_x1.addMax (this);`**