Finite Domains

- Before investigating Propagation Algorithms, we should formalise a number of concepts that will be used
  - Constraints and Constraint Network
  - Partial and Total Solutions
  - Satisfaction and Consistency

Basic Definitions - Domain

Definition (Domain of a Variable):

- The domain of a variable is the (finite) set of values that can be assigned to that variable.

- Given some variable X, its domain will be usually referred to as dom(X) or, simply, Dx.

- Example:

  The N queens problem may be modelled by means of N variables, X_1 to X_n, all with the domain from 1 to n.

  $Dom(X_i) = \{1,2, ..., n\}$ or $X_i :: 1..n.$

Basic Definitions - Labels

- To formalise the notion of the state of a variable (i.e. its assignment with one of the values in its domain) we have the following

Definition (Label):

  - A label is a Variable-Value pair, where the Value is one of the elements of the domain of the Variable.

- The notion of a partial solution, in which some of the variables of the problem have already assigned values, is captured by the following

Definition (Compound Label):

  - A compound label is a set of labels with distinct variables.

Basic Definitions - Constraint

- We come now to the formal definition of a constraint

Definition (Constraint):

  - Given a set of variables, a constraint is a set of compound labels on these variables.

- Alternatively, a constraint may be defined simply as a relation, i.e. a subset of the cartesian product of the domains of the variables involved in that constraint.

- For example, given a constraint C_{ijk} involving variables X_i, X_j and X_k, then

$$C_{ijk} \subseteq dom(X_i) \times dom(X_j) \times dom(X_k)$$

Basic Definitions - Constraint

- Given a constraint C, the set of variables involved in that constraint is denoted by vars(C).
- Simetrically, the set of constraints in which variable X participates is denoted by cons(X).
- Notice that a constraint is a relation, not a function, so that it is always $C_{ij} = C_{ji}$.
- In practice, constraints may be specified by
    - Extension: through an explicit enumeration of the allowed compound labels;
    - Intension: through some predicate (or procedure) that determines the allowed compound labels.

Basic Definitions – Constraint Definition

- For example, the constraint C_{13} involving X_1 and X_3 in the 4-queens problem, may be specified

- By extension (label form):

$$C_{13} = \{\{X_1\text{-}1, X_3\text{-}2\}, \{X_1\text{-}1, X_3\text{-}4\}, \{X_1\text{-}2, X_3\text{-}1\}, \{X_1\text{-}2, X_3\text{-}3\},$$
$$\{X_1\text{-}3, X_3\text{-}2\}, \{X_1\text{-}3, X_3\text{-}4\}, \{X_1\text{-}4, X_3\text{-}1\}, \{X_1\text{-}4, X_3\text{-}3\}\}.$$

or, in tuple (relational) form, omitting the variables

$$C_{13} = \{\langle 1,2\rangle, \langle 1,4\rangle, \langle 2,1\rangle, \langle 2,3\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 4,1\rangle, \langle 4,3\rangle\}.$$

- By intension:

$$C13 = (X_1 \neq X_3) \wedge (1+X_1 \neq 3+X_3) \wedge (3+X_1 \neq 1+X_3).$$

Basic Definitions – Constraint Arity

Definition (Constraint Arity):

- The constraint arity of some constraint C is the number of variables over which the constraint is defined, i.e. the cardinality of the set Vars(C).

- Despite the fact that constraints may have an arbitrary arity, an important subset of the constraints is the set of binary constraints.
- The importance of such constraints is two fold
  - All constraints may be converted into binary constraints
  - A number of concepts and algorithms are appropriate for these constraints.

Basic Definitions – Binary Constraints

Conversion to Binary Constraints

- An n-ary constraint, C, defined by k compound labels on its variables X_1 to X_n, is equivalent to n binary constraints, C_{0i}, through the addition of a new variable, X_0, whose domain is the set 1 to k.

In practice,

- The k n-ary labels may be sorted in some arbitrary order.
- Each of the n binary constraints C_{0i} relates the new variable X_0 with the variable X_i.
- The compound label $\{X_i\text{-}v_i, X_0\text{-}j\}$ belongs to constraint C_{0i} iff $X_i\text{-}v_i$ belongs to the j-th compound label that defines C.

Basic Definitions – Binary Constraints

Example:

Given variables X_1 to X_3, with domains 1 to 3, the following ternary constraint C is composed of 6 labels.

$C(X_1, X_2, X_3) = \{<1,2,3>,<1,3,2>,<2,1,3>,<2,3,1>,<3,1,2>,<3,2,1>\}$

Each of the labels may be associated to a value from 1 to 6

1 : <1,2,3>, 2 : <1,3,2>, ... , 6: <3,2,1>

Now, the following binary constraints, C_{01} to C_{03}, are equivalent to the initial ternary constraint C

$C_{01}(X_0, X_1) = \{<1,1>, <2,1>, <3,2>, <4,2>, <5,3>, <6,3>\}$

$C_{02}(X_0, X_2) = \{<1,2>, <2,3>, <3,1>, <4,3>, <5,1>, <6,2>\}$

$C_{03}(X_0, X_3) = \{<1,3>, <2,2>, <3,3>, <4,1>, <5,2>, <6,1>\}$

Basic Definitions –Constraint Satisfaction

Definition (Constraint Satisfaction 1):

- A compound label satisfies a constraint if their variables are the same and if the compound label is a member of the constraint.

- In practice, it is convenient to generalise constraint satisfaction to compound labels that strictly contain the constraint variables.

Definition (Constraint Satisfaction 2):

- A compound label satisfies a constraint if its variables contain the constraint variables and the projection of the compound label to these variables is a member of the constraint.

Basic Definitions –Constraint Satisfaction Problem

Definition (Constraint Satisfaction Problem):

- A constraint satisfaction problem is a triple <V, D, C> where
  - V is the set of variables of the problem
  - D is the domain(s) of its variables
  - C is the set of constraints of the problem

Definition (Problem Solution):

- A solution to a Constraint Satisfaction Problem P, defined as the tuple <V, D, C>, is a compound label over the variables V of the problem, which satisfies all the constraints in C.

Basic Definitions – Constraint Graph

- For convenience, the constraints of a problem may be considered as forming a special graph.

Definition (Constraint Graph or Network):

  - The Constraint Graph or Network of a binary constraint satisfaction problem is defined as follows

    - There is a node for each of the variables of the problem.

    - For each non-trivial constraint of the problem, involving one or two variables, the graph contains an arc linking the corresponding nodes.

- When the problems include constraints with arbitrary arity, the Constraint Network may be formed after converting these constraints on its binary equivalent.

Basic Definitions – Constraint Hiper-Graph

- When it is not convenient for some reason to convert constraints to their binary equivalent, the problem may be represented by an hiper-graph.

Definition (Constraint Hyper-Graph):

  - Any constraint satisfaction problem with arbitrary n-ary constraints may be represented by a Constraint Hyper-Graph formed as follows:

    - There is a node for each of the variables of the problem.

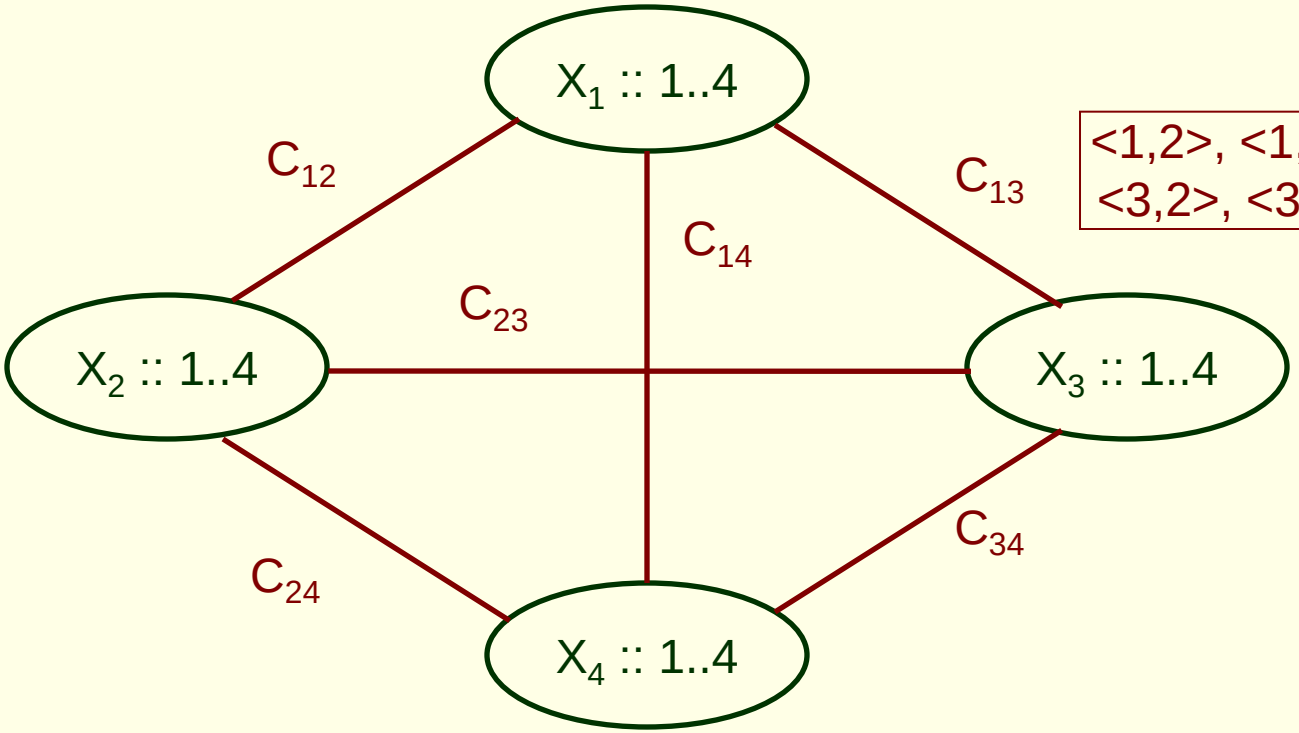
    - For each constraint of the problem, the graph contains an hiper-arc linking the corresponding nodes.

- Of course, when the problem has only binary constraints, the hiper-graph degenerates into the simpler graph.

Example: 4 - Queens

Example:

The 4 queens problem may be specified by the following constraint network:



<1,2>, <1,4>, <2,1>, <2,3>,
<3,2>, <3,4>, <4,1>, <4,3>

Basic Definitions – Graph Density

Definition (Complete Constraint Network):

- A constraint netwok is complete, when there is an arc connecting any two nodes of the graph (i.e. when there is a constraint over any pair of variables).

- The 4 queens problem (in fact, any n queens problem) has a complete graph.

- However, this is often not the case: most graphs are sparse. In this case, it is important to measure the density of the constraint network.

- Definition (Density of a Constraint Graph):

  - The density of a constraint graph is the ratio between the number of its arcs and the number of arcs of a complete graph on the same number of nodes

Basic Definitions – Problem Difficulty

- In principle, the “difficulty” of a constraint problem is related to the density of its graph.
- Intuitively, the denser the graph is, the more difficult the problem becomes, since there are more opportunities to invalidate a compound label.
- It is important to distinguish between the difficulty of a problem and the difficulty of *solving* the problem.
  - In particular, a difficult problem may be so difficult that it is trivial to find it to be impossible!
- The “difficulty” of a problem is also related to the difficulty of satisfying each of its constraints. Such difficulty may be measured through its “tightness”.

Basic Definitions – Constraint Tightness

Definition (Tightness of a Constraint):

- Given a constraint C on variables X_1 ... X_n, with domains D_1 to D_n, the tightness of C is defined as the ratio between the number of labels that define the constraint and the size (i.e. the cardinality) of the Cartesian product D_1 x D_2 x .. D_n.

- For example, the tightness of constraint C_{13} of the 4 queens problem, on variables X_1 and X_3 with domains 1..4

  C_{13} = {<1,2>, <1,4>, <2,1>, <2,3>, <3,2>, <3,4>, <4,1>, <4,3>}

  is 1/2, since there are 8 tuples in the relation out of the possible 16 (4*4).

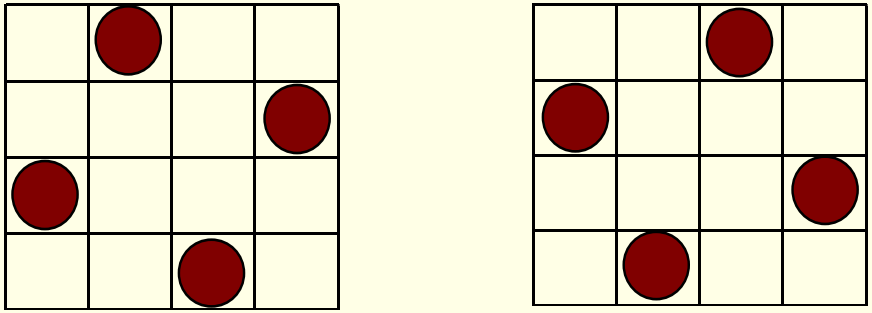
- The notion of tightness may be generalised for the whole problem.

Basic Definitions – Problem Tightness

- Definition (Tightness of a Problem):
  - The tightness of a constraint satisfaction problem with variables X_1 ... X_n, is the ratio between the number of its solutions and the cardinality of the cartesian product

    D_1 x D_2 x .. D_n.

- For example, the 4 queens problem, that has only two solutions <2,4,1,3> e <3,1,4,2>



has tightness 2/(4*4*4*4) = 1/128.

Basic Definitions – Constraint Tightness

- The difficulty in solving a constraint satisfaction problem is related both with the
  - The density of its graph and
  - Its tightness.

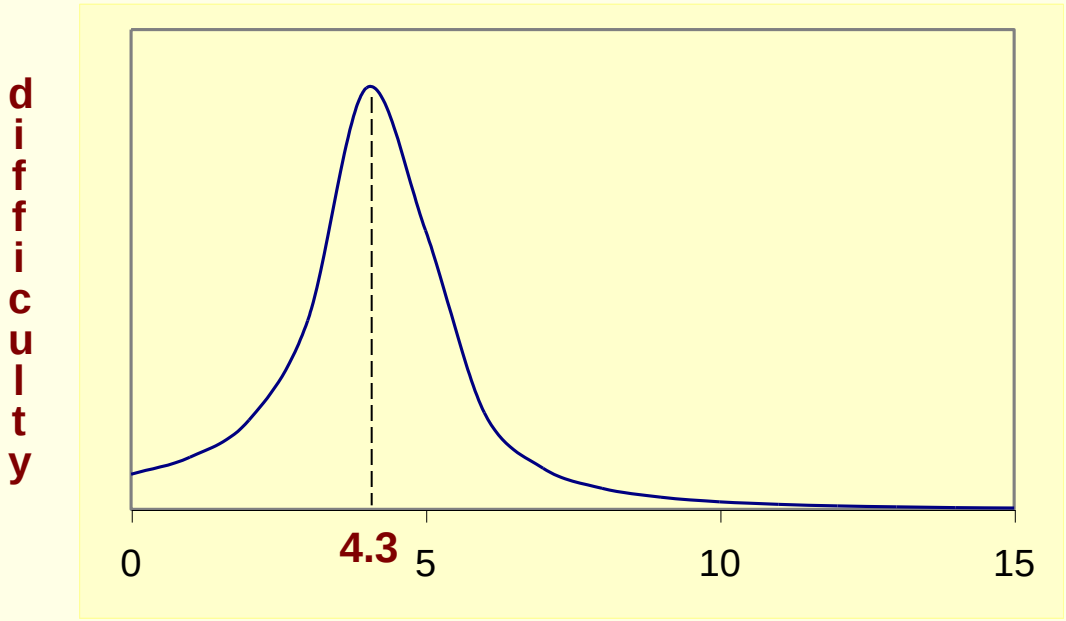
- As such, when testing algorithms to solve this type of problems, it is usual to generate random problem instances, parameterised not only by the number of variables and the size of their domains, but also by the density of its graph and the tightness of its constraints.

- 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.

- 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.

Basic Definitions – Phase Transition in SAT

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



Basic Definitions – Redundancy

- Another issue to consider in the difficulty of solving a constraint satisfaction problem is the potential existence of redundant values and labels in its constraints.

Definition (Redundant Value):

  – A value in the domain of a variable is redundant, if it does not appear in any solution of the problem.

Definition (Redundant Label):

  – A compound label of a constraint is redundant if it is not the projection to the constraint variables of a solution to the whole problem.

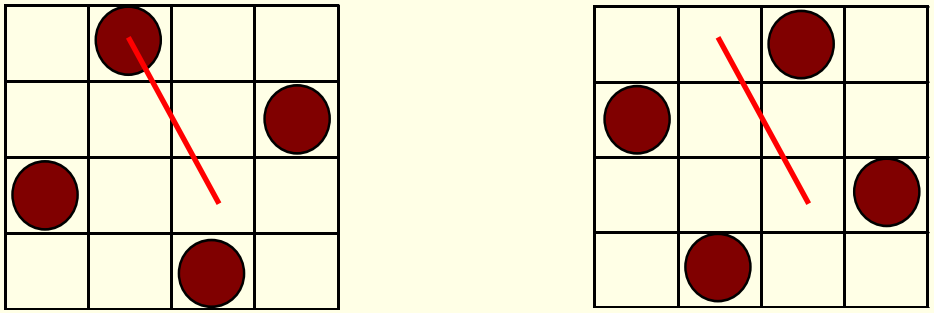
- Redundant values and labels increase the search space uselessly, and should thus be avoided. There is no point in maintaining a value that does not appear in any solution !

Basic Definitions – Redundancy

Example:

The 4 queens problem only admits two solutions:

<2,4,1,3> and <3,1,4,2>.



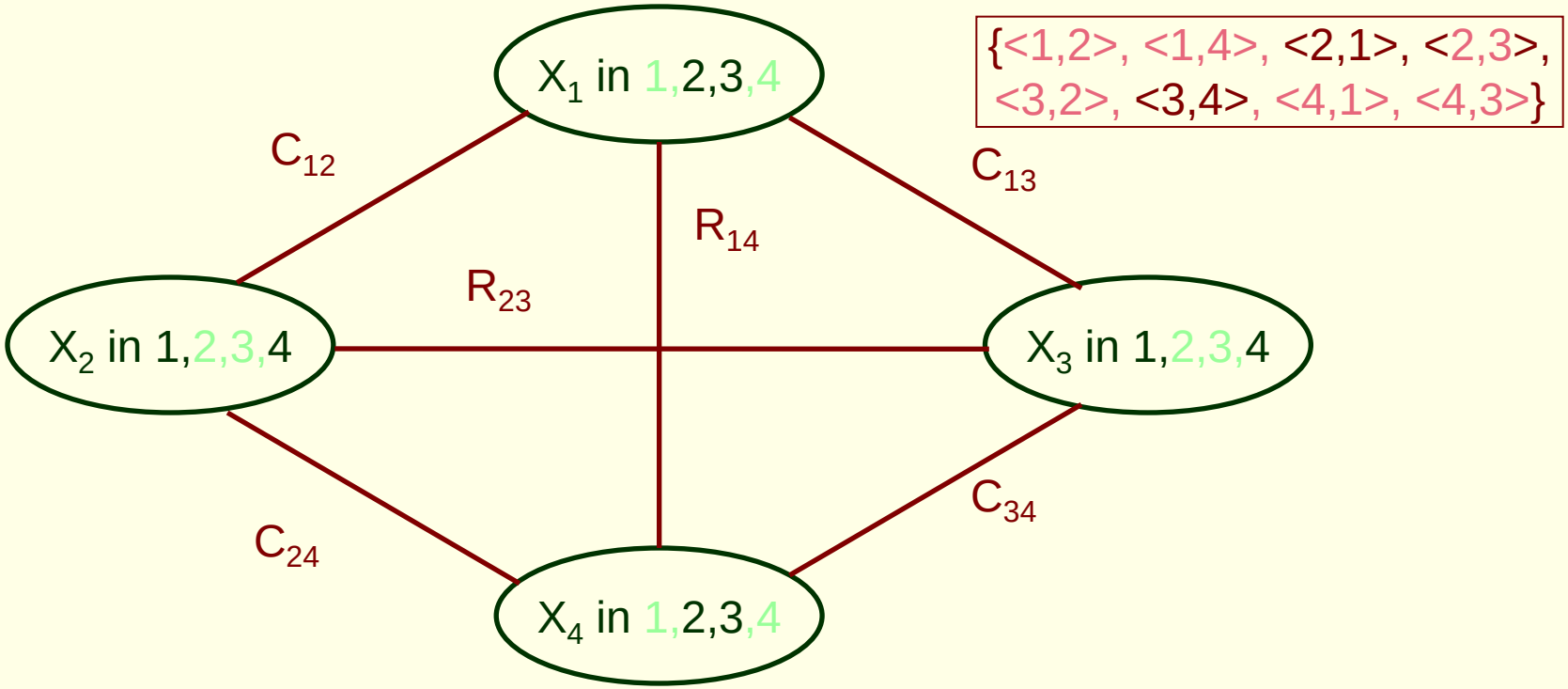
- Hence, values 1 and 4 are redundant in the domain of variables X_1 and X_4, and values 2 and 3 are redundant in the domain of variables X_2 and X_3.

- Regarding redundant labels, labels <2,3> and <3,2> are redundant in constraint C_{13}.

Basic Definitions – Redundancy

Example:

The 4 queens problem, which only admits the two solutions <2,4,1,3> e <3,1,4,2> may be "simplified" by elimination of the redundant values and labels.



Basic Definitions – Equivalent Problems

- Of course, any problem should be equivalent to any of its simplified versions. Formally,

Definition (Equivalent Problems):

  - Two problems P1 = <V1, D1, C1> and P2 = <V2, D2, C2> are equivalent iff they have the same variables (i.e. V1 = V2) and the same set of solutions.

- The "simplification" of a problem may also be formalised

Definition (Reduced Problem):

  - A problem P=<V,D,C> is reduced to P'=<V', D', C'> if
    - P e P' are equivalent;
    - The domains D' are included in D; and
    - The constraints C' are at least as restrictive as those in C.

Constraint Solving Methods

- As shown before, and independently of any problem reduction, a generate and test procedure to solve a Constraint Satisfaction Problem is usually very inefficient.
  - Nevertheless, this is the approach taken in local search methods (simulated annealing or genetic algorithms) – although mostly in an optimisation context !
- It is thus often preferable to use a solving method that is constructive and incremental, whereby a compound label is being completed (constructive), one variable at a time (incremental), until a solution is reached.
- However, one must check that at every step in the construction of a solution the resulting label has still the potential to reach a complete solution.

Constraint Solving Methods

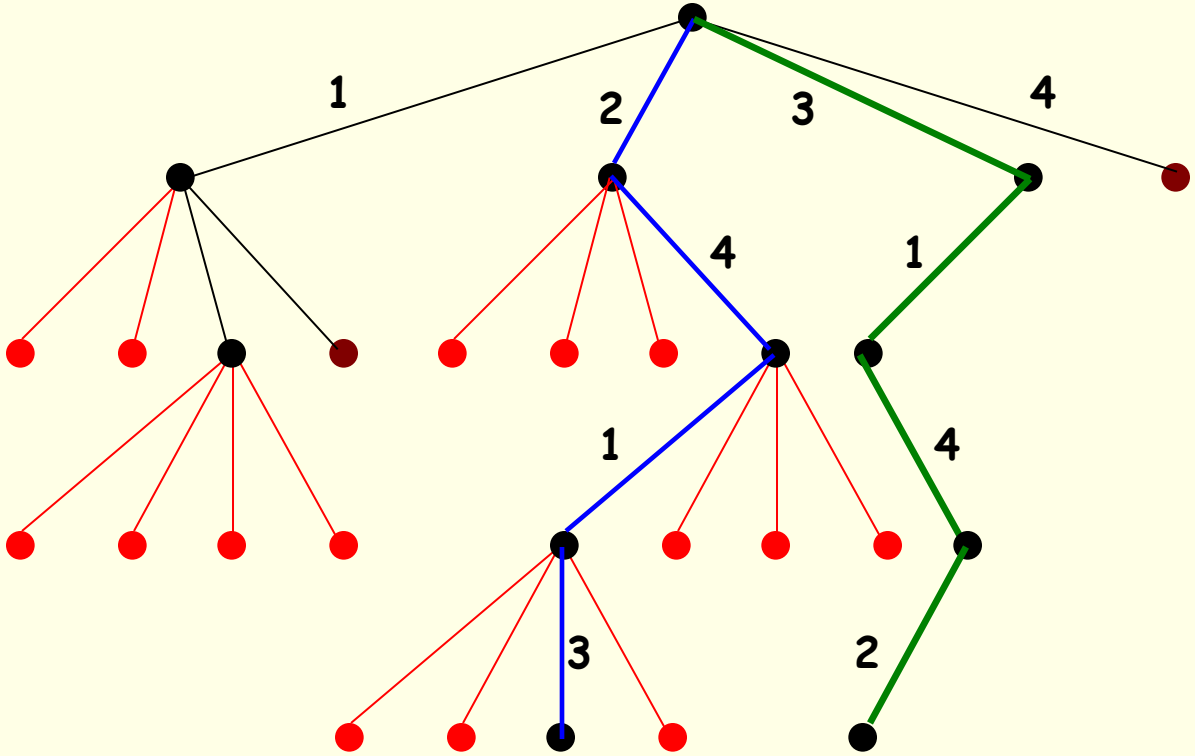
- Ideally, a compound label should be the projection of some problem solution.
- Unfortunately, in the process of solving a problem, its solutions are not known yet!
- Hence, one may use a notion weaker than that of a (complete) solution, namely

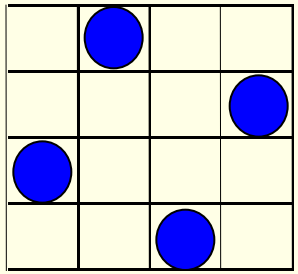
Definition (k-Partial Solution):

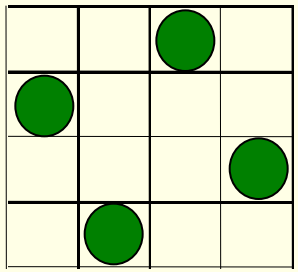
– A k-partial solution of a constraint solving problem P, defined as P = <V,D,C>, is a compound label on a subset of k of its variables, V_k, that satisfies all the constraints in C whose variables are included in V_k.

Constructive Solving Methods

- This is of course, the basis of the solving methods that use some form of backtracking. If conveniently performed, backtracking may be regarded as a tree search, where the partial solutions correspond to the internal nodes of the tree and complete solutions to its leaves.







Problem Search Space

- Clearly, the more reduced a problem is, the easier it is, in principle, to solve it.

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

$$\#S = \#D_1 * \#D_2 * ... * \#D_n$$

- Assuming identical cardinality (or some kind of average of the domains size) for all the variable domains, ($\#D_i = d$) the search space has cardinality

$$\#S = d^n$$

which is exponential on the “size” n of the problem.

Reduction of the Search Space

- 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 decreases may be very significant for "reasonably" large values of n, as shown in the table.

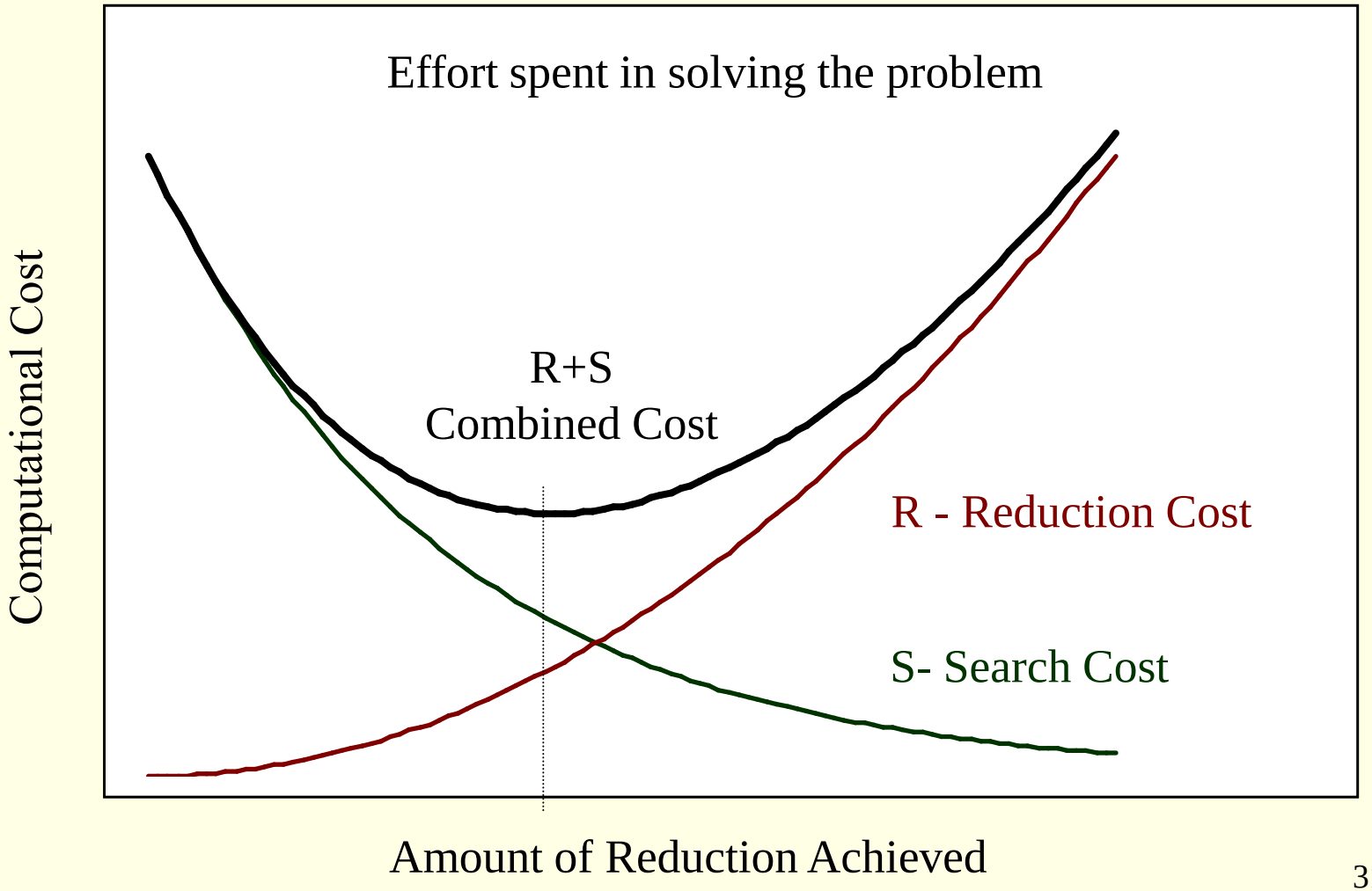
		n									
S/S'		10	20	30	40	50	60	70	80	90	100
7	6	4.6716	21.824	101.95	476.29	2225	10395	48560	226852	1E+06	5E+06
6	5	6.1917	38.338	237.38	1469.8	9100.4	56348	348889	2E+06	1E+07	8E+07
5	4	9.3132	86.736	807.79	7523.2	70065	652530	6E+06	6E+07	5E+08	5E+09
4	3	17.758	315.34	5599.7	99437	2E+06	3E+07	6E+08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

Reduction of the Search Space

- 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.

Reduction of the Search Space

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



Reduction of the Search Space

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

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

```
Problem(Vars):-
        Declaration of Variables and Domains,
        Specification of Constraints,
        Labelling of the Variables.
```

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

Reduction of the Search Space

- Given a problem with n variables X_1 to X_n the execution 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)
```

Domain Reduction – Consistency Criteria

- Once formally defined the notion of problem reduction, one must discuss the actual procedures that may be used to achieve it.
- First of all, one must ensure that whatever procedure is used, the reduction keeps the problem equivalent to the initial one.
- Here we have a small problem, since the definition of equivalence requires the solutions to be the same and we do not know in general what the solutions are!
- Nevertheless, the solutions will be the same if in the process of reduction we have the guarantee that "no solutions are lost".
- Such guarantees are met by several criteria.

Domain Reduction – 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.

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

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

Domain Reduction – Node Consistency

Definition (**Node Consistency**):

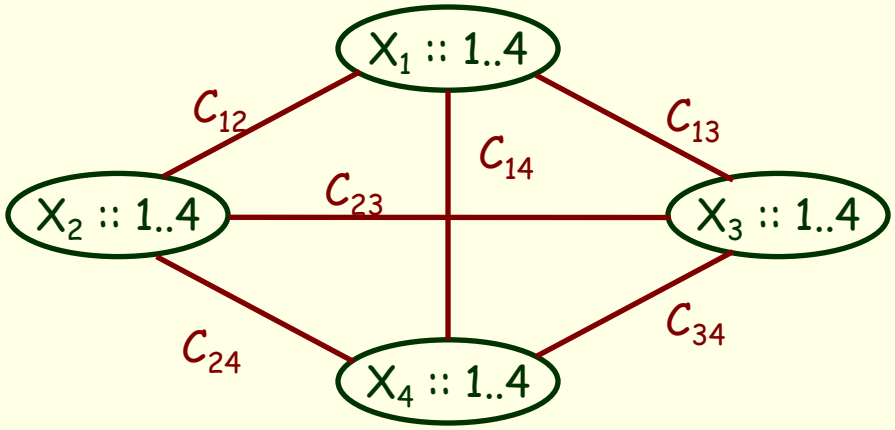
- A constraint satisfaction problem is node-consistent if no value on the domain of its variables violates the unary constraints.

- 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.

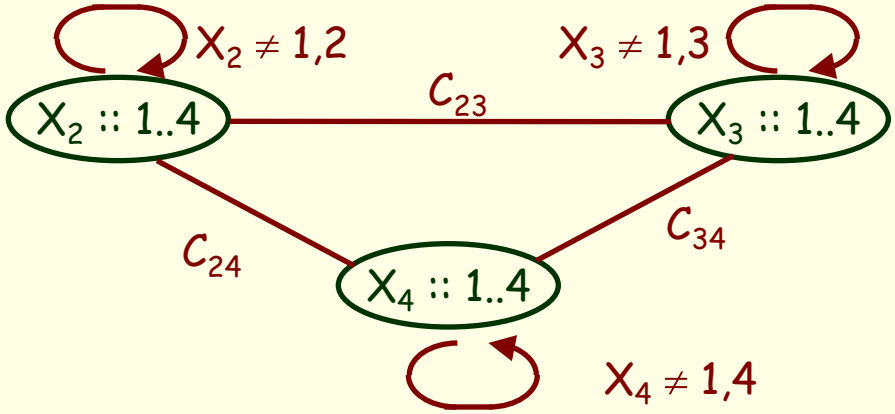
Domain Reduction – Node Consistency

Example:

- After the initial posting of all the constraints, the constraint network model in the left models the 4 queens problem.

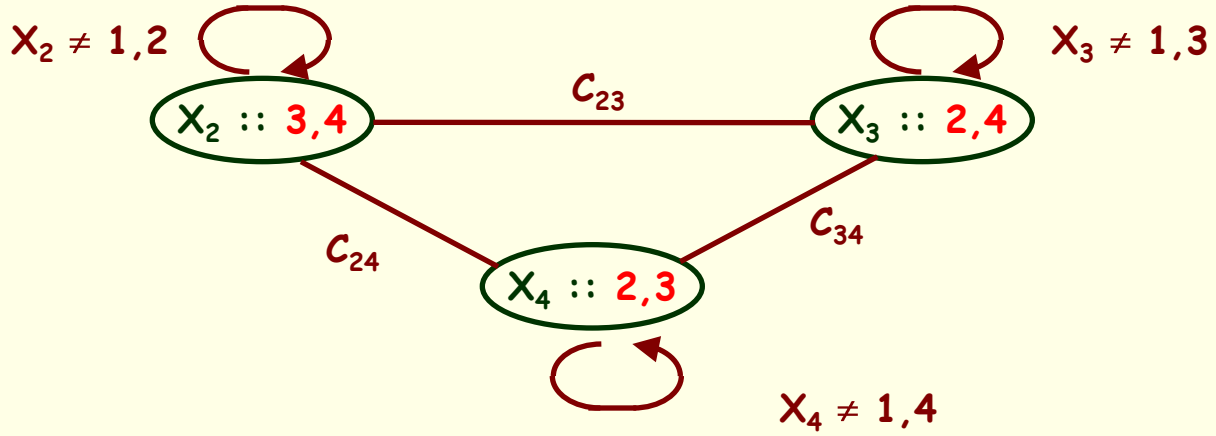


- After enumeration of the variable X_1, i.e. X_1=1, constraints C_{12}, C_{13} and C_{14} become unary !!



Domain Reduction – Node Consistency

- An algorithm that maintains node consistency should remove from the domains of the "future" variables the appropriate values.



- Maintaining node consistency achieves a domain reduction similar to what is achieved in propagation with a Boolean formulation.

●			
1	1		
1		1	
1			1

$X_2 \neq 1,2$

$X_3 \neq 1,3$

$X_4 \neq 1,4$

Maintaining Node Consistency

- Given the simplicity of the node consistency criterion, an algorithm to maintain it is very simple and with low complexity.
- A possible algorithm is NC-1, below.

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

Maintaining Node Consistency

Space Complexity of NC-1

- 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 explicitly the domains of the variables.

- For example, the initial domain 1..4 of variables X_i in the 4 queens problem is represented by a Boolean vector (where 1 means the value is in the domain) X_i = [1,1,1,1] or 1111.

- After enumeration X_1=1, node consistency prunes the domain of other variables to X_1 = 1000, X_2 = 0011, X_3 = 0101 and X_4 = 0110

- Algorithm NC-1 does not require additional space, so its space complexity is **O(nd).**

Maintaining Node Consistency

Time Complexity of NC-1

- 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 virtually all situations by a solver.

- However, node consistency is rather incomplete, not being able to detect many possible reductions.

Domain Reduction – Arc Consistency

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

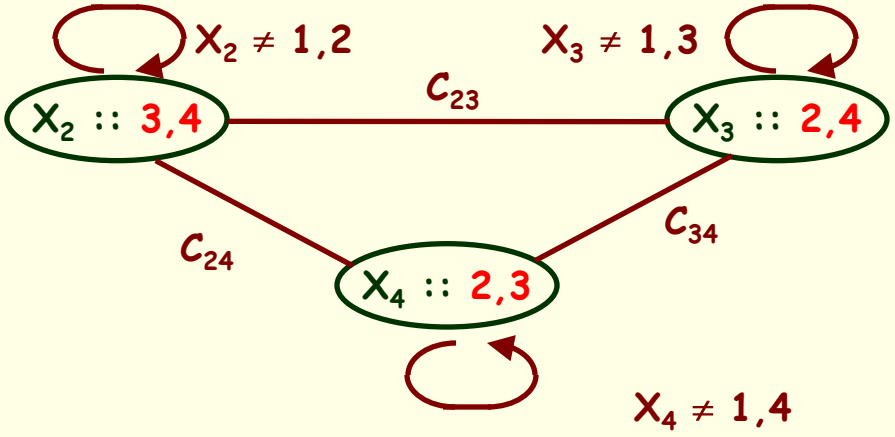
Definition (**Arc Consistency**):

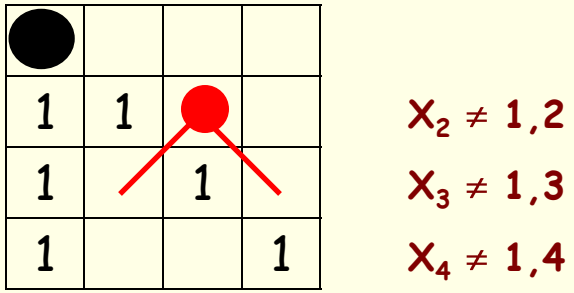
- A constraint satisfaction problem is arc-consistent if,
  - It is node-consistent; and
  - For every label X_i-v_i of every variable X_i, and for all constraints C_{ij}, defined over variables X_i and X_j, there must exist a value v_j that supports v_i, i.e. such that the compound label $\{X_i\text{-}v_i,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

Domain Reduction – Arc Consistency

Example:

- After enumeration of variable X_1=1, and making the network node-consistent, the 4 queens problem has the following constraint network:



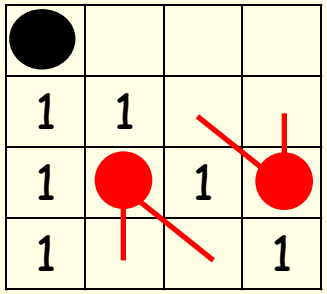


- However, label X_2-3 has no support in variable X_3, since neither compound label {X_2-3 , X_3-2} nor {X_2-3 , X_3-4} satisfy constraint C_{23}.

- Therefore, value 3 can be safely removed from the domain of X_2.

Domain Reduction – Arc Consistency

- In fact, none (!) of the values of X_3 has support in variables X_2 and X_4., as shown below:



$X_2 \neq 1,2$

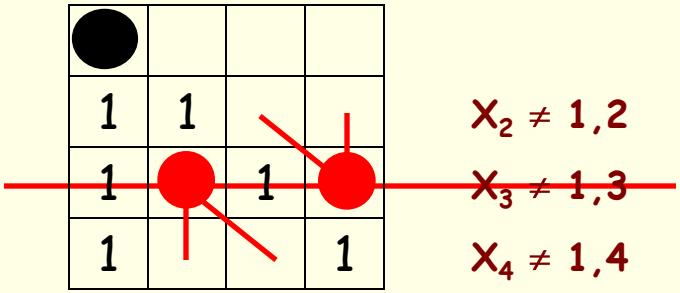
$X_3 \neq 1,3$

$X_4 \neq 1,4$

  - label X_3-4 has no support in variable X_2, since none of the compound labels {X_2-3, X_3-4} and {X_2-4, X_3-4} satisfy constraint C_{23}.
  - label X_3-2 has no support in variable X_4, since none of the compound labels {X_3-2, X_4-2} and {X_3-2, X_4-3} satisfy constraint C_{34}.

Domain Reduction – Arc Consistency

- Since none of the values from the domain of X_3 has support in variables X_2 e X_4, maintenance of arc-consistency "empties" the domain of X_3!

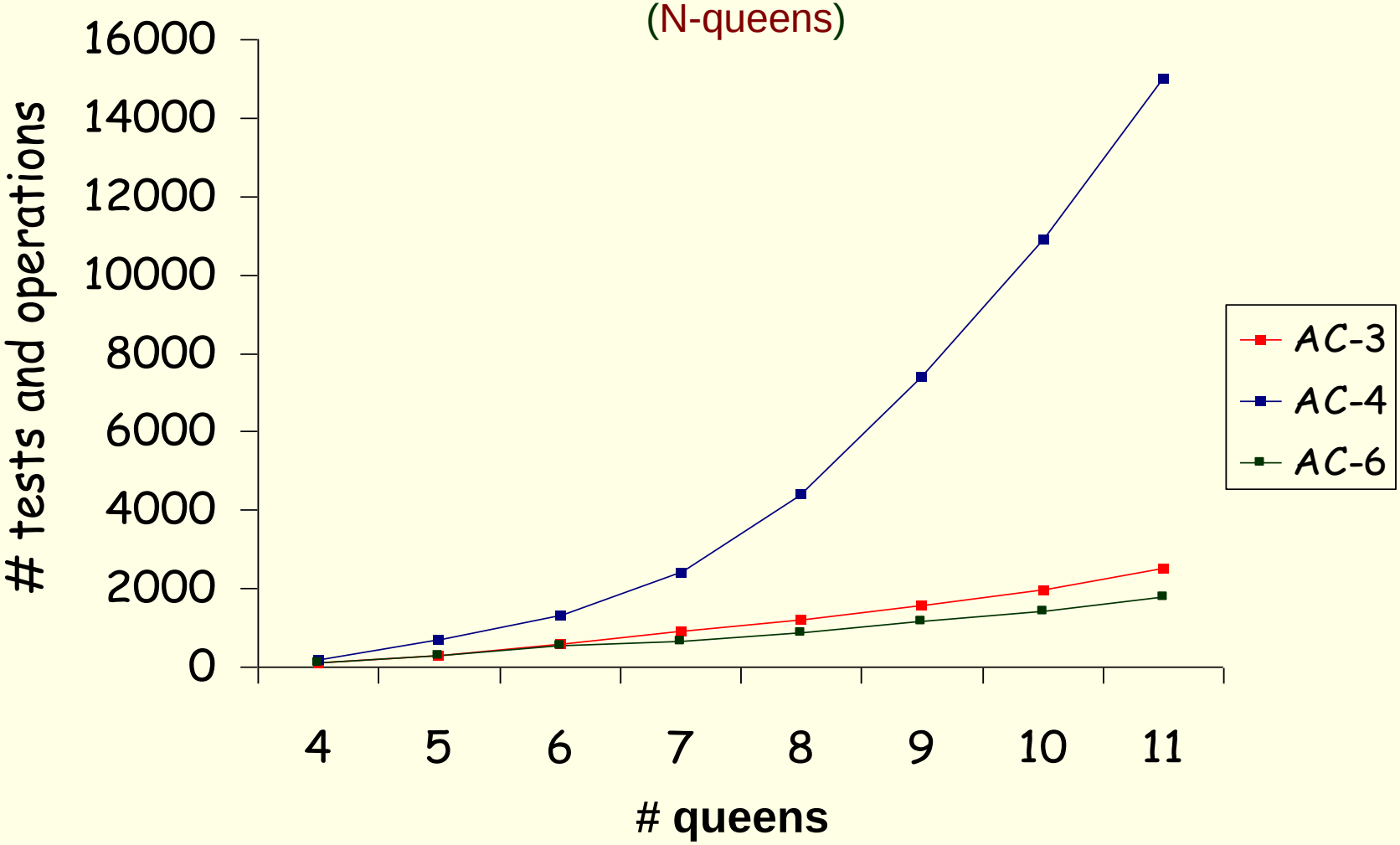


- Hence, maintenance of arc-consistency not only prunes the domain of the variables but also antecipates the detection of unsatisfiability in variable X_3 ! In this case, backtracking of X_1=1 may be started even before the enumeration of variable X_2.

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

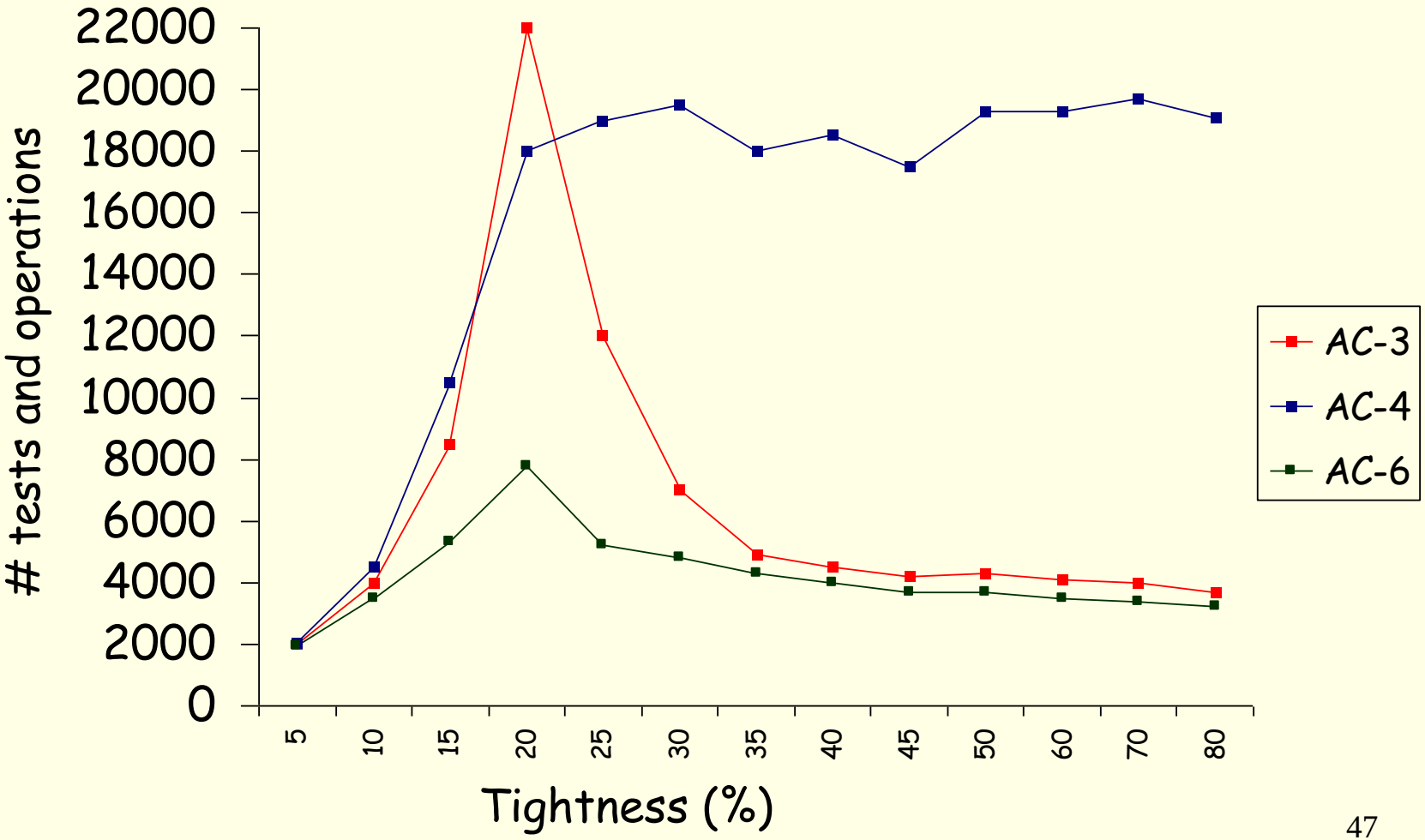
Maintaining Arc Consistency: AC-6

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



Maintaining 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%



Path Consistency

In addition to arc consistency, other types of consistency may be defined for binary constraint networks.

Path consistency is a “classical” type of consistency, stronger than arc consistency.

The basic idea of path consistency is that, in addition to check support in the arcs of the constraint network between variables $\mathbf{X_i}$ and $\mathbf{X_j}$, further support must be checked in the variables $\mathbf{X_{k1}}$, $\mathbf{X_{k2}}$... $\mathbf{X_{km}}$ that form a path between $\mathbf{X_i}$ and $\mathbf{X_j}$, i.e. whenever there are constraints $\mathbf{C_{i,k1}}$, $\mathbf{C_{k1,k2}}$, ..., $\mathbf{C_{km-1,\,km}}$ and $\mathbf{C_{km,j}}$.

In fact, it is possible to show this is equivalent to seek support in any variable $\mathbf{X_k}$,connected to both $\mathbf{X_i}$ and $\mathbf{X_j}$.

Path Consistency

Definition (**Path Consistency**):

A constraint satisfaction problem is path-consistent if,

1. It is arc-consistent; and
2. For every pair of variables $\mathbf{X_i}$ and $\mathbf{X_j}$, and paths $\mathbf{X_i – X_{i1} – X_{i2}}$ **- ... -** $\mathbf{X_{in} – X_j}$, the direct constraint $C_{i,j}$ is tighter than the composition of the constraints in the path $\mathbf{C_{i,i1}, C_{i1,i2}, \ldots, C_{in,j}}$.

In practice, every value $\mathbf{v_i}$ in variable $\mathbf{X_i}$ must have support, not only in some value $\mathbf{v_j}$ from variable $\mathbf{X_j}$ but also in values $\mathbf{v_{i1}, v_{i2}}$, ... , $\mathbf{v_{in}}$ from the domain of the variables in the path.

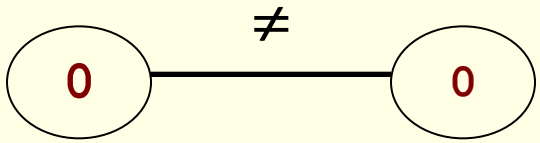
k-Consistency

Node, arc and path consistency, are all instances of the more general case of k-consistency.

Informally, a constraint network is k-consistent when for a group of k variables, X_{i1}, X_{i2}, ... , X_{ik} the values in each domain have support in those of the other variables, considering this support in a global form.

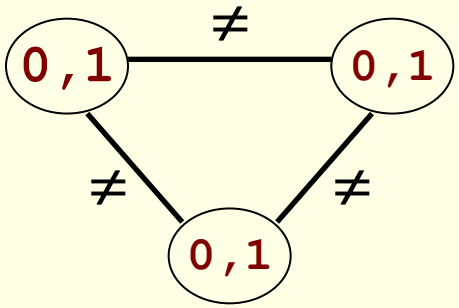
The following is a classical example of the advantages of keeping a global view on constraints

A node consistent network, that is not arc consistent

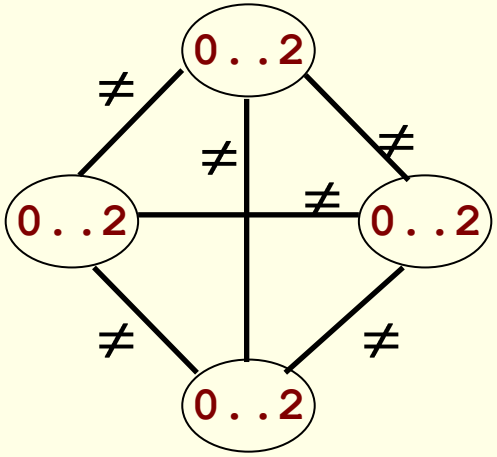


k-Consistency

An arc consistent network, that is not path consistent



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



k-Consistency

Definition (k-Consistency):

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

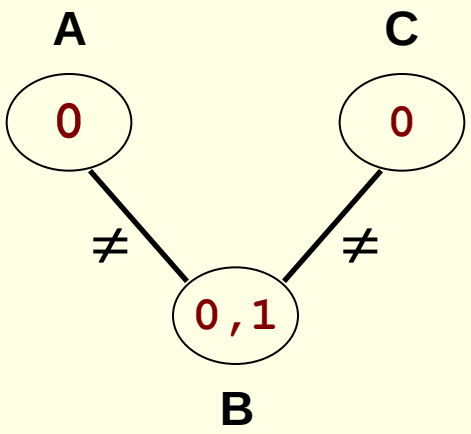
A network is **k-consistent** iff all its **(k-1)-compound labels** (i.e. formed by **k-1** pairs **X-v**) that satisfy the relevant constraints can be extended with some label on any other variable, to form some **k-compound** labels that make the network **k-1** consistent (satisfy the relevant constraints)

k-Consistency

Definition (Strong k-Consistency):

A constraint problem is strongly **k-consistent**, if and only if it is **i-consistent**, for all **i** $\in$ **1 .. k**.

For example, the network below is 3-consistent, but not 2-consistent. Hence it is not strongly 3-consistent.



The only 2-compound labels, which satisfy the inequality 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} !

k-Consistency

As can be seen from the definitions, the three types of consistency studied so far (**node**, **arc** and **path** consistency) are all instances of the more general **k-consistency** or, rather, strong k-consistency).

Hence, a constraint network is

- **node consistent** iff it is **strongly 1-consistent**.
- **arc consistent** iff it is **strongly 2-consistent**.
- **path consistent** iff it is **strongly 3-consistent**.

Generalised Arc Consistency

In the general case of non-binary constraints and constraint networks, the notion of node consistency and its enforcing algorithm **NC-1** may be kept, since they only regard unary constraints.

The generalisation of arc consistency for general n-ary constraint networks (**n > 2**) is quite simple.

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).

Although less intuitive (what is a path in a hyper-graph?), path consistency may still be defined by means of strong 3-consistency for hyper-graphs.

Constraint Dependent Consistency

The algorithms and consistency criteria considered so far take into no account the semantics of specific constraints, handling them all in a uniform way.

In practice, such approach is not very useful, since important gains may be achieved by using specialised criteria and algorithms.

For example, regardless of more general schemes, it is not useful, for inequality constraints ($\neq$), **when considered separately**, to go beyond simple node consistency.

In fact, for $\mathbf{X_i} \neq \mathbf{X_j}$, one may only eliminate a value from a variable domain, when the domain of the other is reduced to a singleton.

Constraint Dependent Consistency

Other more specific types of consistency may be used and enforced. For example in the n queens problem, the constraint

no_attack(X_i, X_j)

should only be considered when the size of the domain of one of the variables becomes reduced to 3 or less than 3 values.

In fact, if $\mathbf{X_i}$ maintains 4 values in the domain, all values in the domain of $\mathbf{X_j}$ will have, at least, one supporting value in the domain of $\mathbf{X_i}$.

This is because the constraint is symmetrical and any queen may only attack 3 queens in a different row.

Bounds consistency

Another specific type of consistency that is very useful in numerical constraints is “bounds-consistency”, often used with constraints such as **=<, <, =** in ordered domains.

For example, consider constraint **A = B**, when variables **A** and **B** have domains

A:: {1, 2, 3, 4, 5, 8, 10} and **B :: {4, 6, 8, 12, 15, 20}**

Whereas arc consistency would reduce the domains of the variables to **{4, 8}** with some computational cost, enforcing simpler bounds consistency would result in the domains

A:: {4, 5, 8} and **B :: {4, 6, 8}**

Although these domains keep redundant values, bounds consistency is much easier to maintain.

Bounds consistency

Bounds consistency is specially interesting when dealing with very large domains, and with n-ary constraints, as is common in numerical problems.

Consider for example the ternary constraint **A + B =< C**, where variables have domains 1..1000. Arc consistency would require checking **1000^3** triples $<v_A, v_B, v_C>$!!!

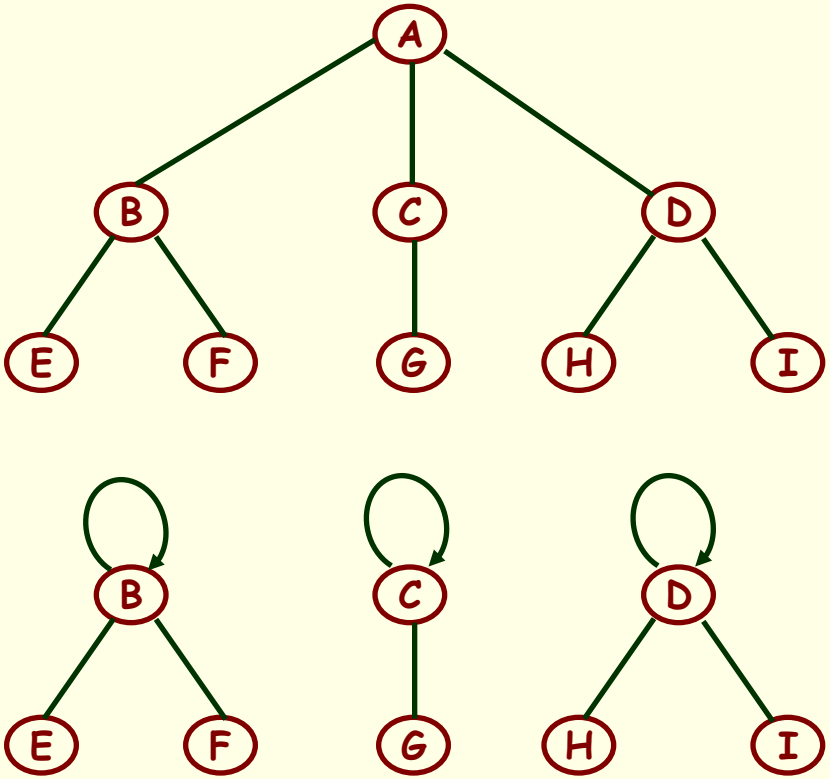
Bounds consistency is much easier to achieve, by enforcing

```
min dom(C) >= min dom(A) + min dom(B)
max dom(A) =< max dom(C) - min dom(B)
max dom(B) =< max dom(C) - min dom(A)
```

Special case of Constraint Trees

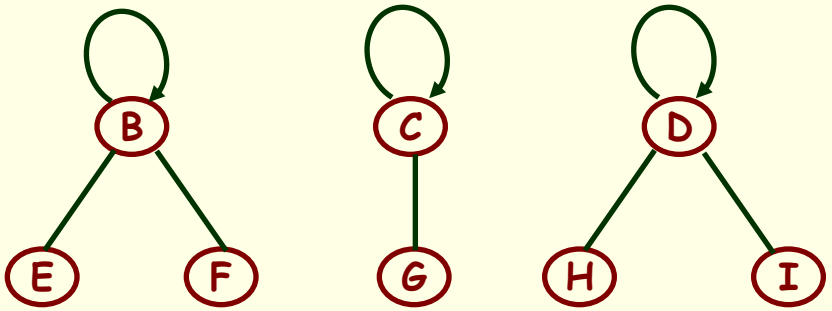
Before addressing the need for heuristic search (even in reduced networks), namely the importance of the order in which variables are selected for enumeration, let us consider the special case of constraint graphs that take the form of a tree.

If variable **A** is enumerated first, with some value ***a*** from its domain (**A = a**), the problem is decomposed into 3 independent subproblems.



Special case of Constraint Trees

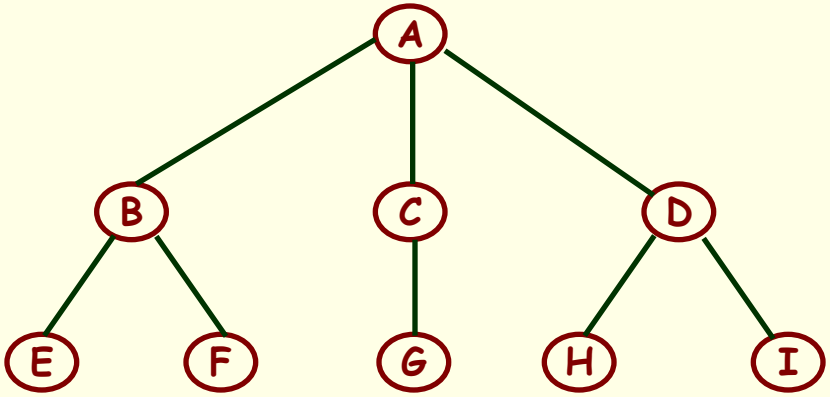
In this case, to guarantee that the enumeration does not lead immediately to unsatisfiability, value a from variable A must have support in variables B, C e D.



This reasoning may proceed recursively, for each of the subtrees with roots B, C and D. For example, enumeration of B (say, B = b) does not lead to an immediate “dead end” if value b has support in E and F.

Special case of Constraint Trees

In general, if enumeration of the variables in the tree is made from root to leaves, such enumeration will not lead to unsatisfiability if the values kept in any node have support in their children nodes.



Hence, a good heuristic to select the variable to enumerate is choosing variables in the root of the tree (or subtree).

Special case of Constraint Trees

Although this is not the case in general constraint graphs, as seen before, constraint trees that are arc-consistent are also **satisfiable**.

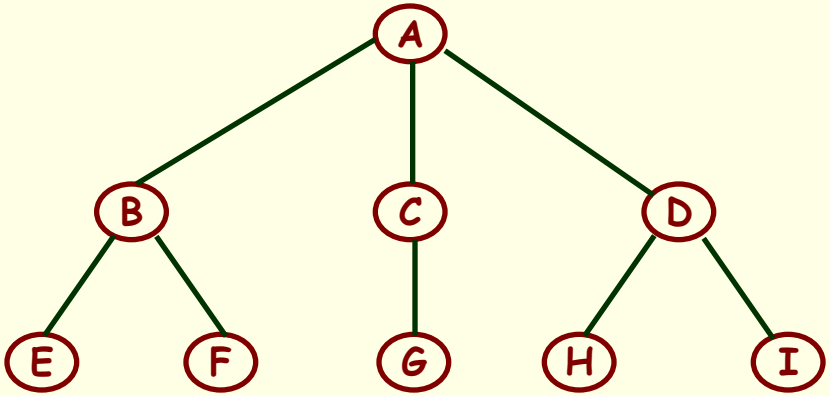
In fact,

1. Using any heuristics that selects the root of the tree and its subtrees,
2. This choice never leads to contradiction, since they have always support in their children,
3. Such support is recursively guaranteed until the leaves.

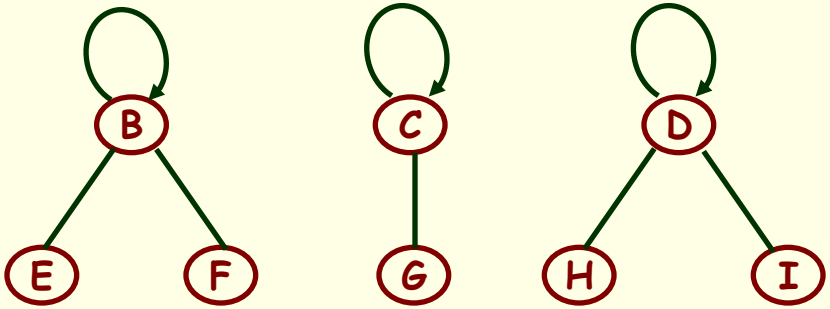
Moreover, this directionality in the enumeration (from root to leaves) enables the consideration of **directed arc consistency**, a weaker criterion than full arc consistency, but sufficient to handle constraint trees.

Special case of Constraint Trees

In the previous example, **it is not necessary** that all values in the domain of B, C and D have support in the domain of A. The only requirement is that **all values in the domain of A have support** in the domains of B, C and D.



When a value is chosen for A, constraints on A and its descendents become unary, and the values from the domains of B, C and D are removed by simple node consistency.



Directed Arc Consistency

In general, one may define a criterion of directed arc consistency if, in contrast to what has been considered, we consider a **directed graph** to represent the constraint network (assuming some direction to each constraint of the problem).

Definition (Directed Arc Consistency):

A constraint problem is **directed arc** consistent iff

1. It is node consistent; and
2. For every label $\mathbf{X_i}$-$\mathbf{v_i}$ of any variable $\mathbf{X_i}$, and for any **directed arc** $\mathbf{a_{ij}}$ (from $\mathbf{X_i}$ to $\mathbf{X_j}$) corresponding to a constraint $\mathbf{C_{ij}}$, there is a supporting value $\mathbf{v_j}$ in the domain of $\mathbf{X_j}$, i.e. the compound label $\mathbf{\{X_i\text{-}v_i, X_j\text{-}v_j\}}$ satisfies constraint $\mathbf{R_{ij}}$.

Maintaining Directed Arc Consistency: DAC

As can be seen, algorithm DAC does not guarantee the elimination of all redundant values, since it does not reexamine variables that *could* loose support.

If in the general case this might be innadequate, this is not a problem in the case of trees, if the arcs, which are directed in descending order of the variables (the tree root has lowest number) are revised in descending order, i.e. arcs closer to the leaves are revised first.

Then all the values of a node have guaranteed support in all the nodes down to the leaves of a tree, although not necessarily on those upto the tree root!

As shown before, if enumeration starts from the root, the unsupported values are eliminated by node consistency.

Maintaining Directed Arc Consistency: DAC

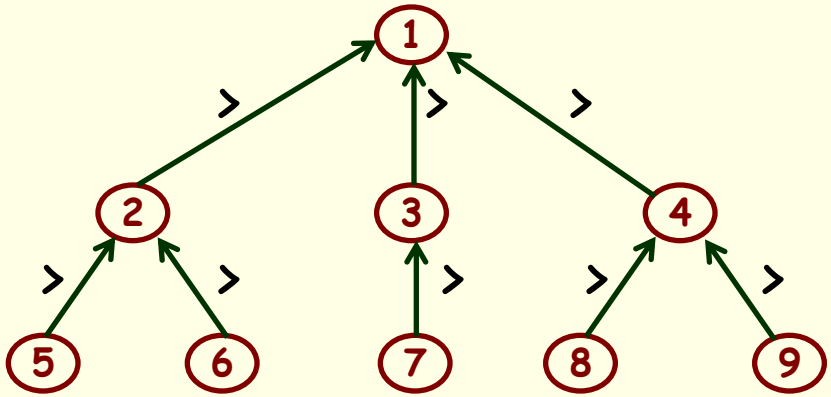
Example: Take the constraint tree below

X_1 in {1, 3, 5, 7, 9},

X_2 in {2, 4, 6}, X_5 in {4,8}, X_6 in {3, 9},

X_3 in {1, 5, 7}, X_7 in {3, 9},

X_4 in {1, 5, 8}, X_8 in {3, 7}, X_9 in {2, 9}



After revising arc	X_9 -> X_4	X_4 in {1, 5, 8}	% X_9 >= 2
	X_8 -> X_4	X_4 in {1, 5, 8}	% X_8 >= 3
	X_7 -> X_3	X_3 in {1, 5, 7}	% X_7 >= 3
	X_6 -> X_2	X_2 in {2, 4, 6}	% X_6 >= 3
	X_5 -> X_2	X_2 in {2, 4, 6}	% X_5 >= 4
	X_4 -> X_1	X_1 in {1, 3, 5, 7, 9}	% X_4 >= 5
	X_3 -> X_1	X_1 in {1, 3, 5, 7, 9}	% X_3 >= 5
	X_2 -> X_1	X_1 in {1, 3, 5, 7, 9}	% X_2 >= 6

Maintaining Directed Arc Consistency: DAC

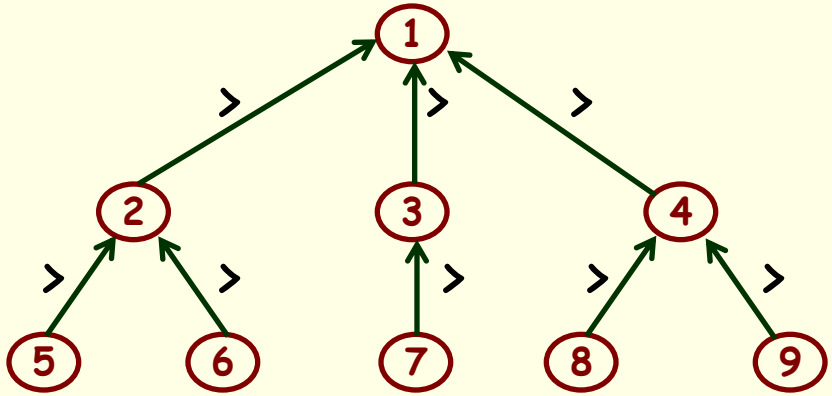
Example (cont): The enumeration became **backtrack free** !

X_1 in {7, 9},

X_2 in {6}, X_5 in {4, 8}, X_6 in {3, 9},

X_3 in {5, 7}, X_7 in {3, 9},

X_4 in {5, 8}, X_8 in {3, 7}, X_9 in {2, 9}



$X_1 = 7$ NC enforces X_2 in {6}, X_3 in {5}, X_4 in {5}
$X_2 = 6$ NC enforces X_5 in {4}, X_6 in {3},
$X_3 = 5$ NC enforces X_7 in {3},
$X_4 = 4$ NC enforces X_8 in {3}, X_9 in {2},

$X_1 = 9$ NC enforces X_2 in {6}, X_3 in {5,7}, X_4 in {4, 8}
$X_2 = 6$ ...., X_3 in {5,7}, enforces $X_7 = 3$
$X_4 = 4$ NC enforces X_8 in {3}, X_9 in {2}
or $X_4 = 8$ NC enforces X_8 in {3, 7}, X_9 in {2}