Constraint Programming

- Heuristics in Search

  - Variable and Value selection
  - Static and Dynamic Heuristics
  - Incomplete Search Strategies
  - Symmetry Breaking (introduction)

Heuristic Search

- Algorithms that maintain some form of consistency, remove redundant values but, not being complete, do not eliminate the need for search, except in the (few) cases where i-consistency guarantees not only satisfiability of the problem but also a backtrack free search. In general,
  - A satisfiable constraint may not be consistent (for some criterion); and
  - A consistent constraint network may not be satisfiable

- All that is guaranteed by maintaining some type of consistency is that the initial network and the consistent network are equivalent - solutions are not “lost” in the reduced network, that despite having less redundant values, has all the solutions of the former. Hence the need for search.

- Complete search strategies usually organise the search space as a tree, where the various branches down from its nodes represent assignment of values to variables. As such, a tree leaf corresponds to a complete compound label (including all the problem variables) – a constructive approach to solution finding.

Heuristic Search

- A depth first search in the tree, resorting to backtracking when a node corresponds to a dead end, corresponds to an incremental completion of partial solutions until a complete one is found.

- Given the execution model of constraint programming (or any algorithm that interleaves search with constraint propagation)

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

  the enumeration of the variables (labeling) determines the shape of the search tree, since its nodes depend on the order in which variables are enumerated.

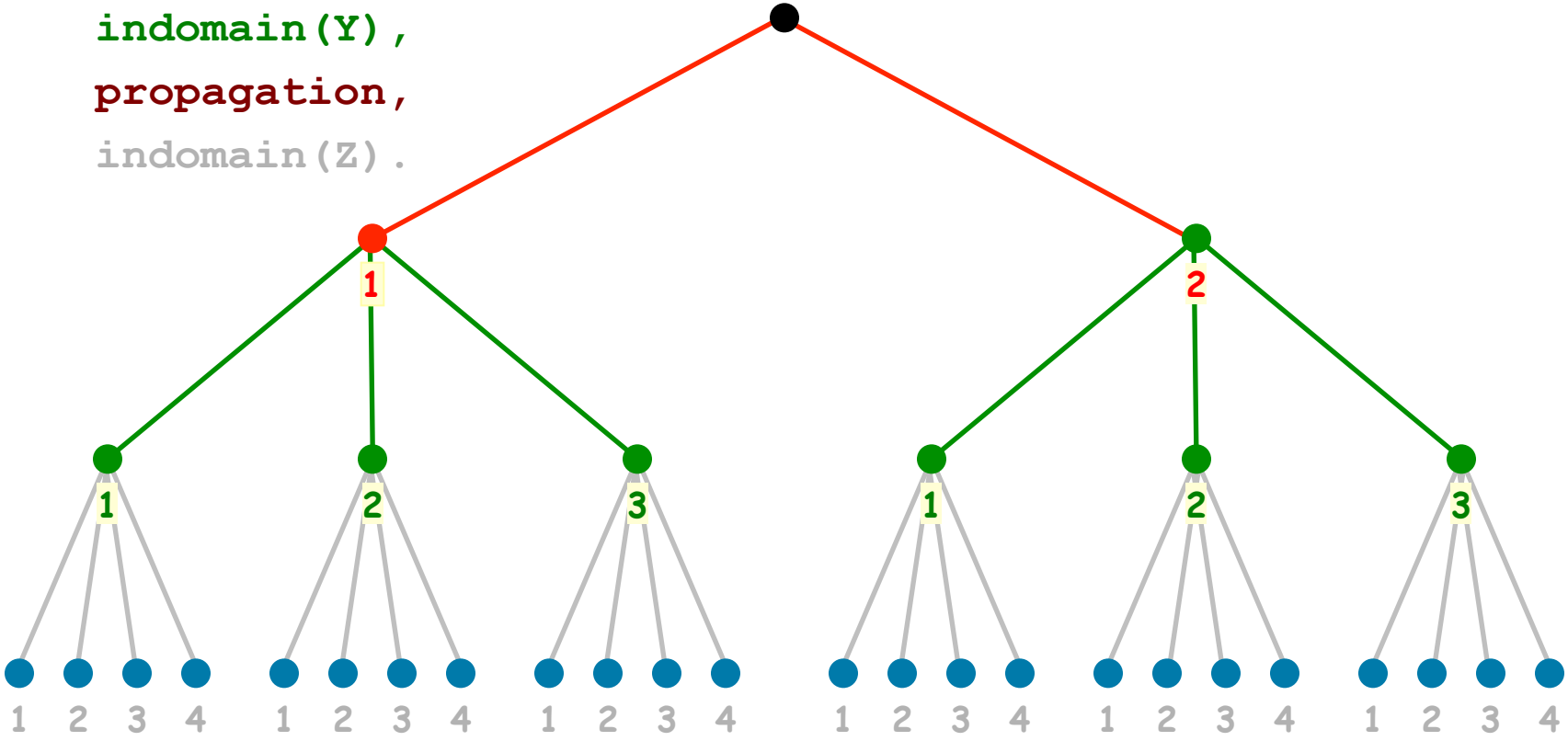
- Take for example two distinct enumerations of variables whose domains have different cardinality, e.g. `X in 1..2, Y in 1..3` and `Z in 1..4.`

Heuristic Search

```
enum([X,Y,Z]):-
    indomain(X)
    propagation
    indomain(Y),
    propagation,
    indomain(Z).
```

# of nodes = 32

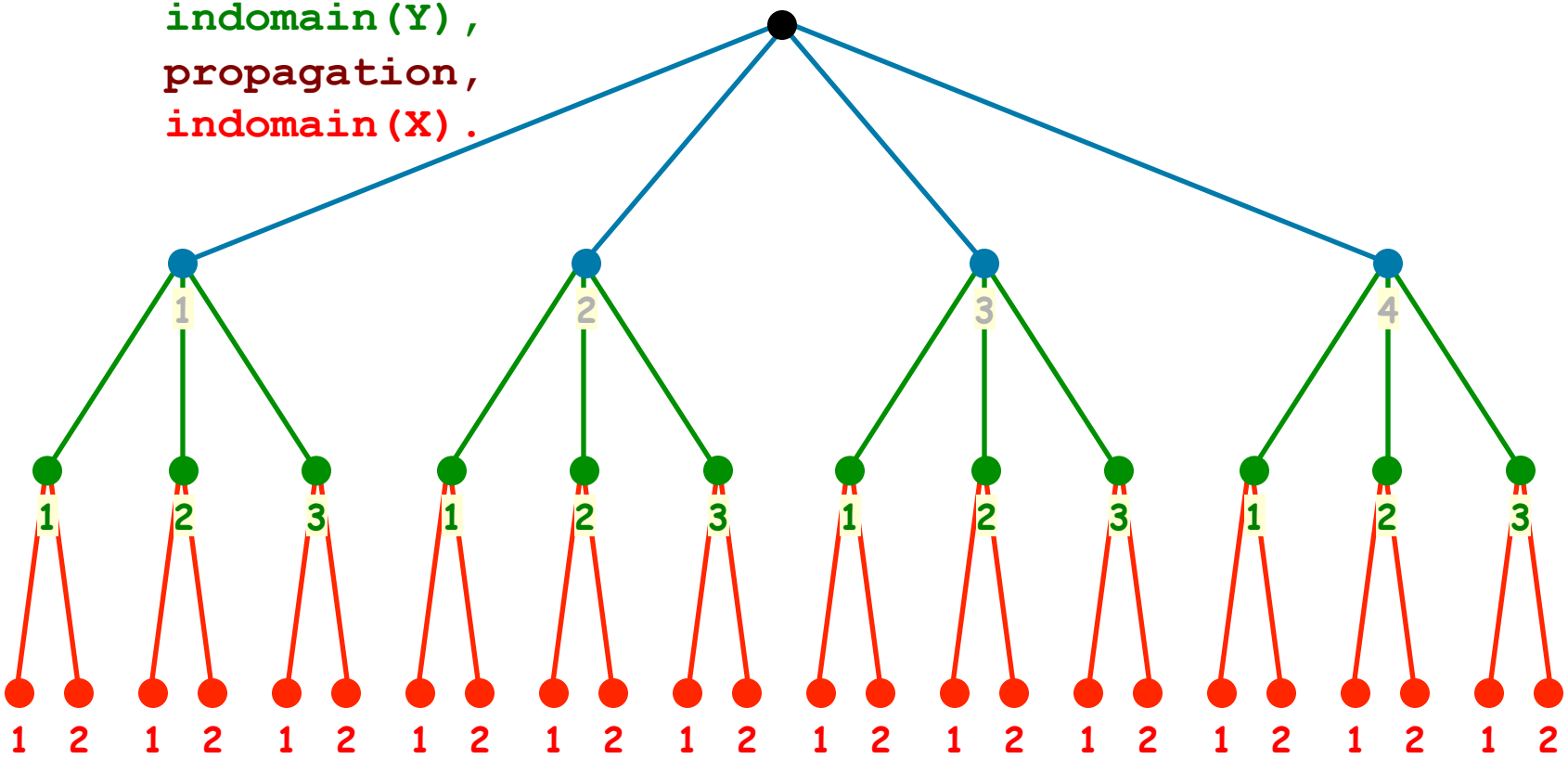
(2 + 6 + 24)



Heuristic Search

```
enum([X,Y,Z]):-
    indomain(Z),
     propagation
    indomain(Y),
    propagation,
    indomain(X).
```

# of nodes = 40
(4 + 12 + 24)



Heuristic Search

- The order in which variables are enumerated may have an **important** impact on the efficiency of the tree search, since
  - The number of internal nodes is different, despite the same number of leaves, or potential solutions, Π **#D**$_i$.
  - Failures can be detected differently, favouring some orderings of the enumeration.
  - Depending on the propagation used, different orderings may lead to different prunings of the tree.
- The **ordering of the domains** has no **direct** influence on the search space, although it may have great importance in finding the first solution.

Heuristic Search

- To control the efficiency of tree search one should in principle adopt appropriate heuristics to select
    - The next **variable** to label
    - The **value** to assign to the selected variable

- Since heuristics for **value choice** will not affect the size of the search tree to be explored, particular attention will be paid to the heuristics for **variable selection**, where two types of heuristics can be considered:

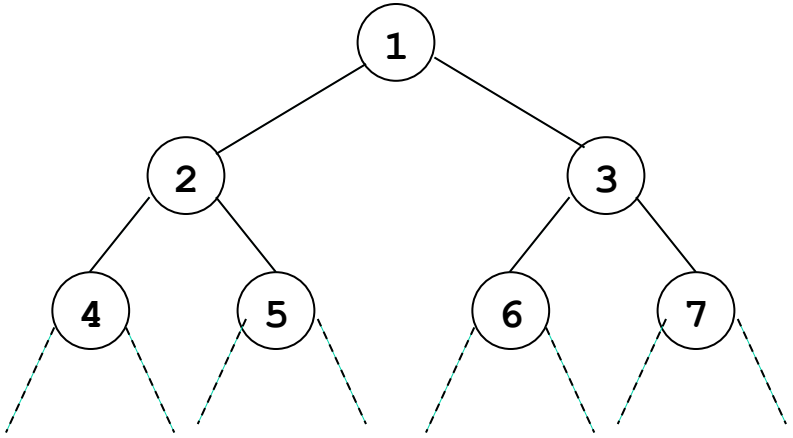
    - **Static** - the ordering of the variables is set up before starting the enumeration, not taking into account the possible effects of propagation.

      Static heuristics are interesting when the constraint networks are sparse, and have particular topologies that can be exploited usefully (e.g. problem decomposition).

    - **Dynamic** - the selection of the variable is determined after analysis of the problem that resulted from previous enumerations (and propagation).

Static Heuristics

- As a special case, it is possible to show that tree-shaped CSPs are tractable.



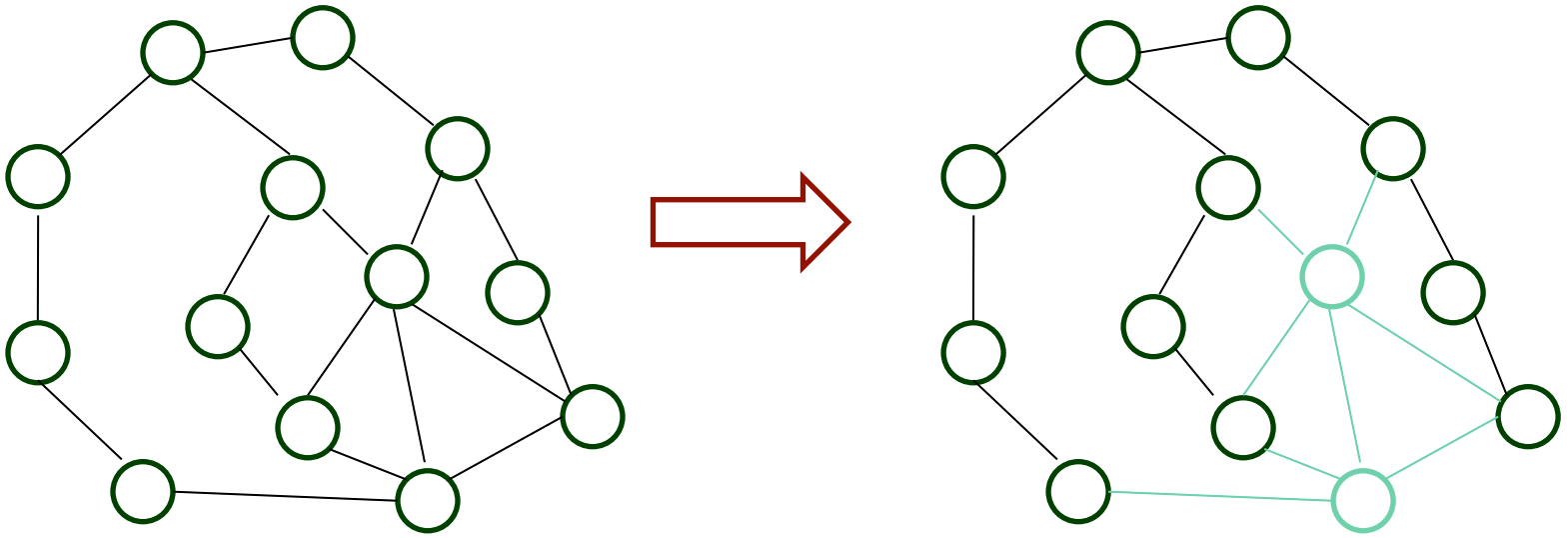
- Given an enumeration of the variables from the root “downwards”, a **backtrack free** search is guaranteed **if arc-consistency is maintained**.
- After the enumeration X1-v1, arc-consistency guarantees that there are supports in X2-v2 and X3-v3.
- In turn, value X2-v2 has support in X4-v4 and X5-v5, and X3-v3 X2-v2 has support in X6-v6 and X7-v7.,,,,

Static Heuristics

An example of Static Heuristics:

CSS Heuristics (*Cycle Cut Set Heuristic*):

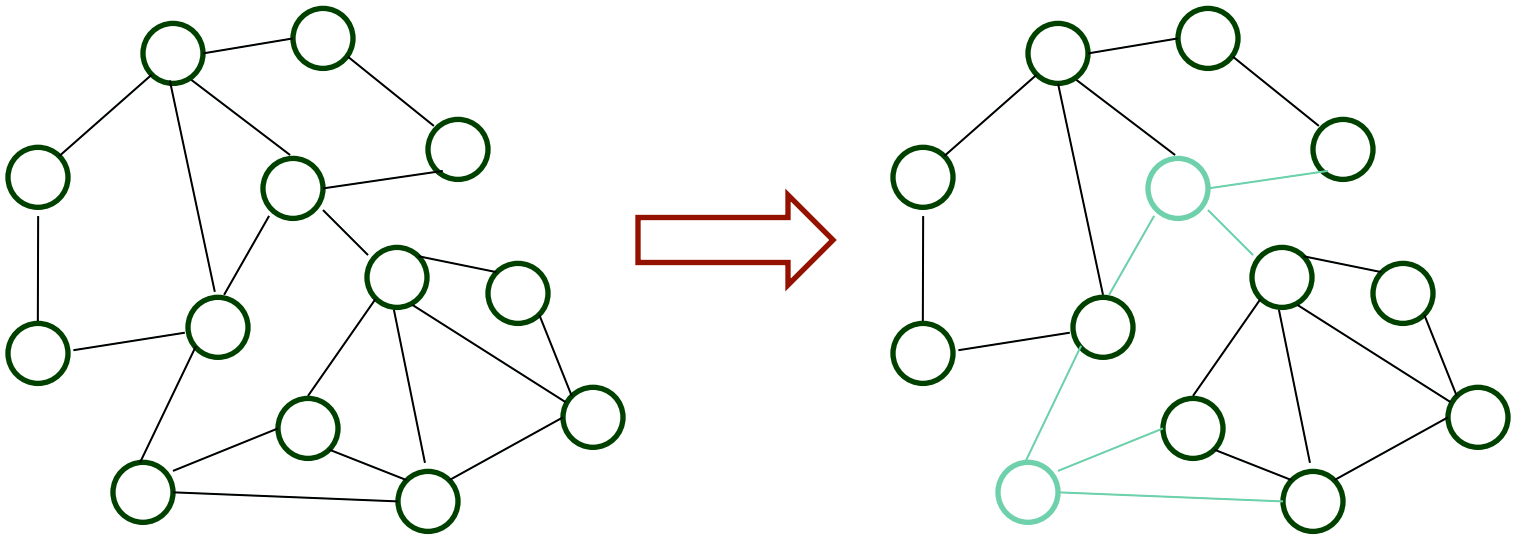
The ***Cycle Cut Set Heuristic*** suggests the variables of a lowest cardinality cycle-cut set to be enumerated first, reducing the problem to a tree network.



- In the graph shown, as soon as the highlighted variables are enumerated the graph becomes a tree.

Static Heuristics

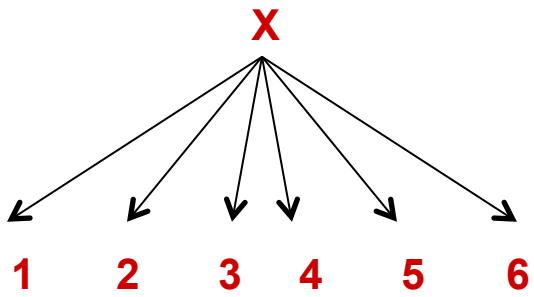
- In other cases, a decomposition strategy may pay-off, i.e. selecting an order of enumeration that decomposes a problem in smaller problems. In the example of the figure, after enumerating the variables in the common “frontier” the problem is decomposed into independent problems.



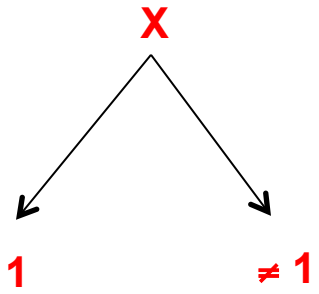
- The rationale is of course to transform a problem with worst case complexity of $O(d^n)$ into two problems of complexity $O(d^{n/2})$.

Complete Branch & Bound Search

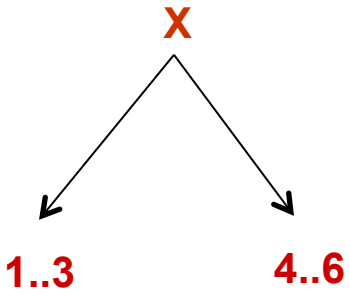
- Before analysing dynamic heuristics is is important to note that enumeration may be done in several ways, different from that presented earlier.



- **K-way branching**: Leaves of the search tree have all the same depth.



- **2-way branching**: The same variable can be selected multiple times in a path to a solution. Usually it is more efficient than the previous variable selection heuristics

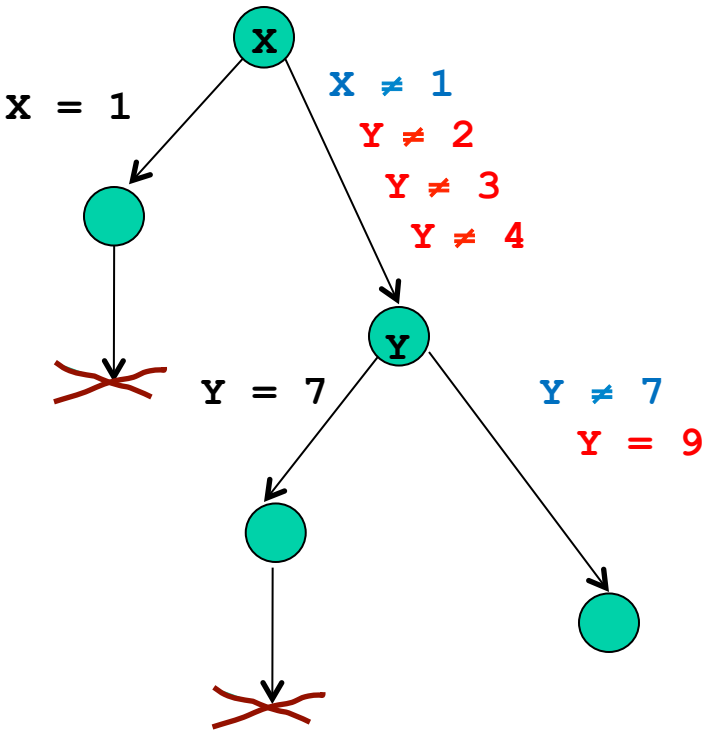


- **Bipartite selection**: Half (or other selected fraction) of the search space might be prunned in a **branch.** Usually used in branch-and-bound optimisation.

Dynamic Symmetry Breaking

- 2-way branching allows for a more flexible application of heuristics. Different variables may be picked in alternance.

- **Example**: X in {1, 6, 8, 9} Y in {2, 3, 4, 7, 9} , X =< Y, p(X,Y)



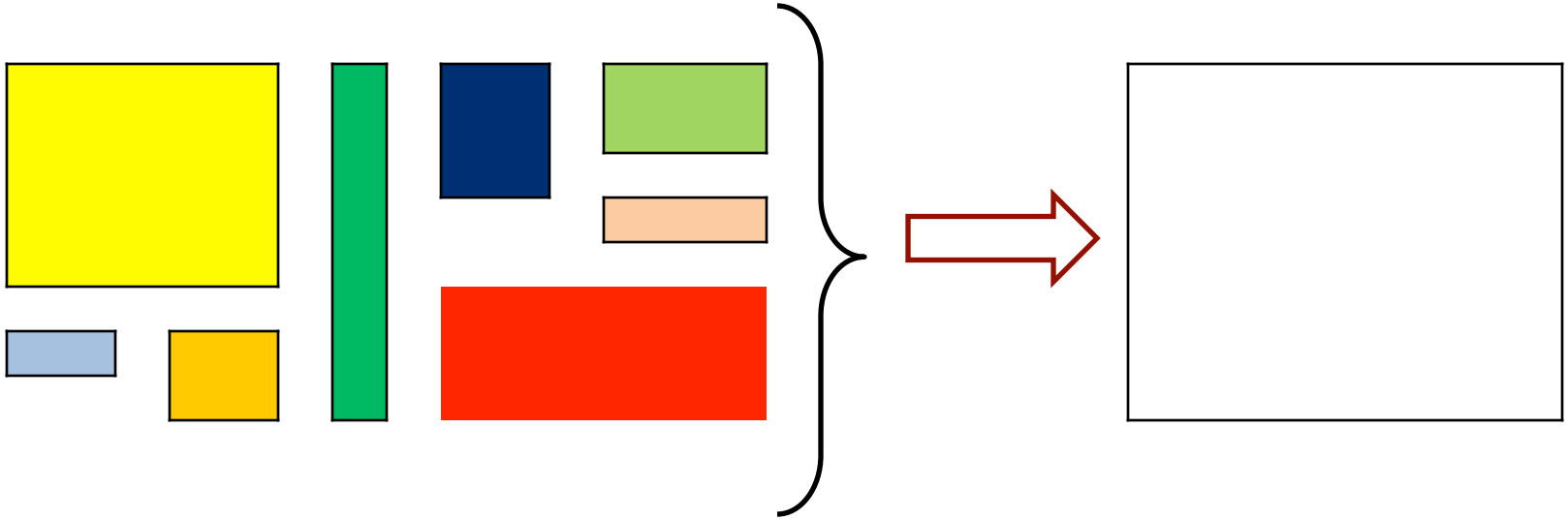
X in {1,6,8,9}
Y in {2,3,4,7,9}

X in {6,8,9}
Y in {7,9}

X in {6,8,9}
Y in {9}

Dynamic Heuristics

- The basic principle followed by most (if not all) dynamic variable selection heuristics can be illustrated with the following placement problem:
- Fill the large rectangle in the right with the 8 smaller rectangles (in the left).



- A sensible heuristics will start by placing the larger rectangles first. The rationale is simple: They are harder to place than the smaller ones, and heve less possible choices. If one starts with the easy ones, they are likely tofurther restrict these choices, soa s to make them impossible, thus resulting in avoiudable backtracking.

Dynamic Heuristics

- This is the principle that dynamic variable select heuristics follow in general: the **first-fail** principle.
- When selecting the variable to enumerate next, try the one that is more difficult, i.e. start with the variables more likely to fail (hence the name).
- If the principle is simple, there are many possible ways of implementing it. As usual, many apparently good ideas do not produce good results, so a relatively small number of implementations is considered in practice.
- They can be divided in three distinct groups:
  - Look-Present heuristics: the difficulty of the variable to be selected is evaluated taking into account the current state of the search process;
  - Look-Back heuristics: they take into account past experience for the selection of the most difficult variable;
  - Look-Ahead Heuristics: the difficulty of the variable is assessed taking into account some probing of future states of the search.

Dynamic Heuristics

- **Look Present Heuristics.**

- Of course, enumerating a variable is a simple task that is equally difficult for all variables. The important issue here is the likelihood that the assignment is a correct one.

- Of course if there are many choices (as there are for the smaller rectangles in the example), the likelihood of assigning a wrong value increases, and the difficulty can thus be assessed in this number of choices.

- If this assessment is to be based solely on the current state of the search, it should consider features that are easy to measure, such as
  - The domain of the variables (its cardinality)
  - The number of constraints (degree) they participate in.

Dynamic Heuristics

Dom Heuristics: The **domain** of the variables (cardinality)

- Take variables X_1 / X_2 with m_1 / m_2 values in their domains, and $m_2 > m_1$. Intuitively, it is more difficult to assign values to X_1, because there is less choice available !

- In the limit, if variable X_1 has only one value in its domain, ($m_1 = 1$), there is no possible choice and the best thing to do is to immediately assign the value to the variable.

- Another way of seeing this choice (but from a value-selection perspective) is the following:
  - On the one hand, the "chance" to assign a good value to X_1 is higher than that for X_2.
  - On the other hand, if that value proves to be a bad one, a larger proportion of the search space is eliminated.

- This heuristics is also referred to as **ff** (e.g. In SICStus Prolog)

Dynamic Heuristics

Deg Heuristics: The **number of constraints** (degree) of the variables

- This heuristics is basically the Maximum Degree Ordering (MDO) heuristics, but now the degree of the variables is assessed dynamically, after each variable enumeration.

- Clearly, the more constraints a variable is involved in, the more difficult it is to assign a good value to it, since it has to satisfy a larger number of constraints.

- In practice this heuristic is not used alone, but as a form of breaking ties (for variables with domains of the same cardinality).

- This heuristics is also referred to as **c** (e.g. in SICStus Prolog), namely when associated with the Dom heuristics to break ties in this latter. The association is referrred to as **ffc** heuristics.

Dynamic Heuristics in Comet

```
function void labelQueensCentre(var<CP>{int}[] X){
  Solver<CP> s = X[X.getLow()].getSolver();
  int n = X.getSize();
  while(! bound(X)){
    selectMin(i in X.getRange(): !X[i].bound())
                      (X[i].getSize(),abs(i-n/2)*n+i){
      range D = X[i].getMin()..X[i].getMax()
      selectMin (v in D: X[i].memberOf(v)) (v)
         try<s> s.label(X[i],v); | s.diff(X[i],v);}
    }
}
```

```
function void labelQueensDouble(var<CP>{int}[]C, var<CP>{int}[] R){
  Solver<CP> cp = C[C.getLow()].getSolver();
  int n = C.getSize();
  range S = C.getRange();
  forall(r in S) by(C[r].getSize(), abs(r-n/2))
    tryall<cp>(c in S:C[r].memberOf(c)) by
                                  (R[c].getSize(),abs(c-n/2))
      cp.post(C[r] == c);
}
```

Dynamic Heuristics

Look Back Heuristics.

- These heuristics aim to learn the difficulty of the variables from past experience in the search process.

- Learning the difficulty of a variable, and adjust it during search, has been tried successfully in the past in two different directions:

  - To measure the difficulty of the constraints, through the number of failures they induce during the search process - the more failures are detected, the more difficult a constraint is considered. The difficulty of a variable is then obtained indirectly from the difficulty of the constraints it belongs to.

  - To measure th difficulty of the variables, by the reduction of the search space they induce. The more this search space has been reduced in the past, the more difficult is considered the variable.

Dynamic Heuristics

Look Back Heuristics.

Wdeg Heuristics: The **weighted degree** of the variables

- This heuristics is a variation of the **Deg** heuristics, and updates the weigths of the constraints of the problems during search, taking into account constraint propagation, as follows.
- Every constraint is implemented through propagators, usually one for every variable appearing in the constraint. For example, constraint c: A+B >= C is implemented with 3 propagators (for bounds consistency)
  - `c1: max(C) <- max(A) + max(B)`
  - `c2: min(A) <- min(C) - max(B)`
  - `c3: min(B) <- min(C) - max(A)`
- Propagators may fail. For example, if propagator c1 makes **max(C) < min(C),** not only the domain of variable C becomes empty, but also a failure is registered for propagator c1.
- Failures of any propagator are assigned to the corresponding constraints.

Dynamic Heuristics

Wdeg Heuristics: The **weighted degree** of the variables

- The Wdeg heuristics is thus implemented as follows:
- All constraints of the problem start with weight w = 1
- Whenever a propagator leads to a failure, the weight of the corresponding constraint is increased by 1 (w = w +1).
- Every variable X, still to enumerate, is assigned a weighted degree, **Wdeg**, which is the sum of the weights of all the constraints it participates, that are still n-ary (n >1) at that state of the search (it is assumed that unary constraints are easily checked and do not influence this counting scheme).
- For example, if A+B > C and A and B are fixed, then the domian of C is updated and the constraint is not considered any longer (for variable C).
- At each enumeration step, the variable selected is that with highest **Wdeg**.

Dynamic Heuristics

Dom/Wdeg Heuristics

- Similarly to the Deg, also the Wdeg hweuristics can be combined with the Dom (cardinality of the domain) heuristics.

- The most successful combination is the Dom/Wdeg heuristics, that takes into account that a variable is more difficult to enumerate if

  - Its Dom is lower.

  - Its Wdeg is higher

- Hence, the Dom/Wdeg heuristics selects for enumeration the variable with the lowest ratio **Dom / Wdeg**.

Dynamic Heuristics

Impact Heuristics

- A different type of heuristics attempts to assign the degree of difficulty to a variable by the **impact** it had in previous enumerations. The most successful heuristics measures the impact as the **reduction of the search space** when an enumeration is made, under the assumption that difficult variables will reduce more the possibilities to the other variables.
- A simple measure (upper bound) of the search space is the product of the cardinality of the domains of the variables still not enumerated.
- An enumeration **e** (i.e. of some value v to some variable **x**) has an impact **i (x,e)** measured by the relative reduction of the search space. Denoting by $\mathbf{S_a(e)}$ (resp. $\mathbf{S_b(e)}$) the size of the search space **after** (resp. **before**) the enumeration (considering only the other variables) the impact is measured as

$$\mathbf{i(x,e) = (S_b(e) - S_a(e)) / S_b(e) = 1 - S_a(e)/ S_b(e)}$$

- The total impact factor of a variable is the average of all the impacts of previous enumerations of that variable, i.e.

$$\mathbf{i(x) = average_e(i(x,e))}$$

Dynamic Heuristics

Impact Heuristics

- The **impact** heuristics selects for enumeration the variable with the highest impact factor I(x).

- For example, assume that the problem has variables X_1 to X_6, all with 5 values in their domain. If variable X_5 is enumerated some value in its domain and the size of the variables becomes [3,4,1,5,1,6] then the impact of this enumeration on X_5 is

$$1- (3*4*1*5*6)/(6*6*6*6*6) = 1- 360/6^5 = 1-0.04(629) =0.95(370)$$

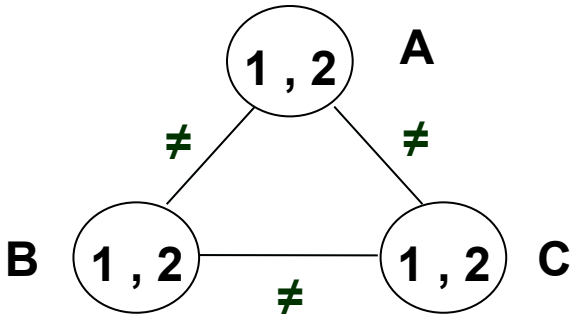
- Of course the highest this value (that ranges in [0,1]) the largest the impact is, which justifies the heuristics.

Dynamic Heuristics

Look Ahead heuristics

- An heuristics that we have tested successfully in some problems (e.g. latin squares) selects a variable assessing the effect of its possible assignments (not done yet).

- Although arc-consistency does not detect unsatisfiability of the network shown, this would be detected if each of the values of a variable is “tried”, as a form of look-ahead.



- This type of look-ahead consistency, named singleton arc-consistency, was already previously proposed, but it was not considered efficient.

- We noticed however that its adoption, not to merely prune values from the domains of the variables but also as an heuristics to take into account the impact of each of the tried values.

Dynamic Heuristics

Look Ahead heuristics

- For each variable x under consideration, a look-ahead is made, assigning each of the values v of the variable domain and assessing its impact, i(x,v) defined much like before (the ratio between the reduction of the search space and the search space before the assignment, *but now only considering all the current values in the domain of the variable*).

$$\mathbf{i(x=v) = (S_b(x=v) - S_a(x=v)) / S_b(x=v) = 1 - S_a(x=v)/ S_b(x=v)}$$

- The impact of each variable x is obtained as the averave over the different values

$$\mathbf{i(x) = average_v(i(x=v))}$$

SAC-LA Heuristics

- The **SAC-LA** heuristics selects for enumeration the variable with the highest impact factor I(x) obtained after imposing Singleton Arc Consistency.

Dynamic Heuristics

Look Ahead heuristics

- In practice this heuristics has provided the best results in the latin-squares problem (hard instances of size 35, 1/3 of positions filled in).

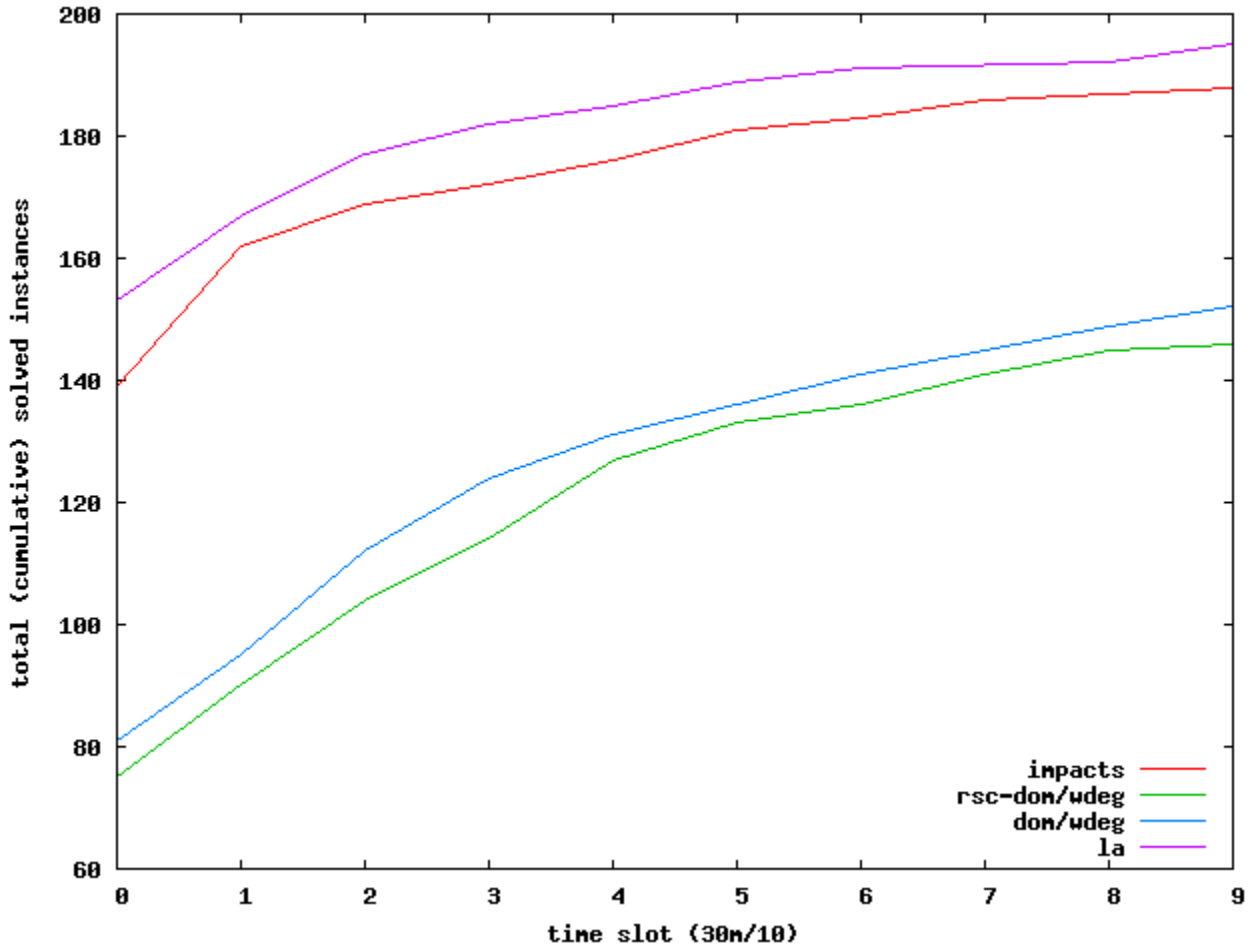
- Moreover it was adapted, to breack ties in teh Dom heuristics. Several variables were quickly restricted to 2 value and we use the heuristics in these values.

- Moreover, not full arc consistency is used. Instead, we used a restricted form of arc-consistency that does not reach a fix point in reduction but only makes a round robin visit to all the constraint propagators.

- In general all heuristics require some tuning, and this is more an art than a technique, but is often the best (only?) way of solving a problem (namely its hard instances).

Dynamic Heuristics

- An example of comparison of heuristics - 35 latin square completion



Intelligent Backtracking

- When the enumeration of a variable fails, backtracking is usually performed on the variable that immediately preceded it.

- This is the so-called chronological backtracking.
- However, it is possible that *this last variable is not to blame for the failure*, in which case, chronological backtracking will only re-discover the same failures.

- Various techniques for inteligent backtracking, or dependency directed search, aim at identifying the causes of the failure and backtrack directly to the first variable that participates in the failure.

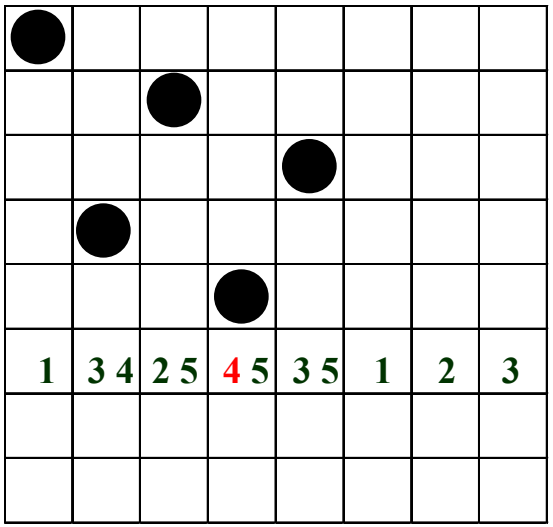
- Some variants of intelligent backtracking are:
    - Backjumping ;
    - Backchecking ; and
    - Backmarking .

Intelligent Backtracking

Backjumping

- Failing the labeling of a variable, all variables that cause the failure of each of the values are analysed, and the “highest” of the “least” variables is backtracked.

- In the example shown, analysis of why variable Q6, could not be labeled, leads to the conclusion that in all possible positions are prevented by a queen Q4 or an earlier queen.



- Hence Q4 is the “last of the prior“ (max-min) variables involved in the failure of Q6.

- Hence backtracking, should be made to Q4, not Q5. The assignment of values 5,6,7 or 8 to Q4, would simply lead to new failures!

- The use of these techniques with constraint propagation is usually not very effective (with a possible exception of **SAT solvers**), since propagation antecipates the conflicts, somehow avoiding irrelevant backtracking.

Intelligent Backtracking

Sat Solvers

- Among all possible finite domains, the Booleans is a specially interesting case, where all variables take values 0/1.

- In Computer Science and Engineering the importance of this domain is obvious: ultimately, all programs are compiled into machine code, i.e. to specifications coded in bits, to be handled by some processor.

- More pragmatically, a vast number of problems may be naturally specified through a set of boolean constraints, coded in a variety of forms.

- Among these forms, a quite useful one is the clausal form, which corresponds to the Conjunctive Normal Form (CNF) of any Boolean function. For example,

$$c_1: \quad \neg x_1 \vee x_2$$

- In such cases, Boolean SATisfiability is often referred to as SAT.

Intelligent Backtracking

Sat Solvers

- Advanced SAT solvers use techniques common to other Finite Domains solvers, namely
  - Boolean constraint propagation (BCP)
  - Heuristics to select the next variable and value to select, so that search is guided towards most promising regions of the search space.

- The specificity of SAT, enables specialised solvers to use additional advanced techniques, not commonly used in more general FD solvers, namely
  - Diagnosis of failure
  - Non-chronological backtracking
  - Learning of “nogood” clauses

Intelligent Backtracking

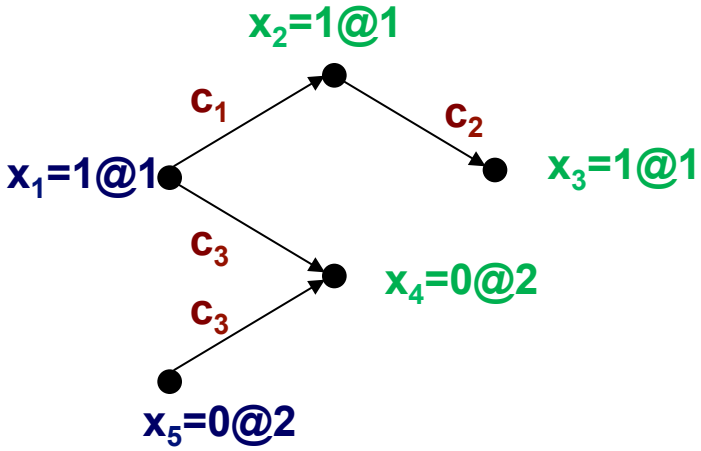
- To illustrate these techniques, we should consider the assignment of values to variables are made. Two different situations occur:
  - Some assignments are explicit decisions made by the solver, selecting the variable and the value.
  - Other assignments are due to propagation on the former.

Example: Take the following labeling on these 3 clauses

$c_1 : (\neg x_1 \vee x_2)$

$c_2 : (\neg x_2 \vee x_3)$

$c_3 : (\neg x_1 \vee \neg x_4 \vee x_5)$

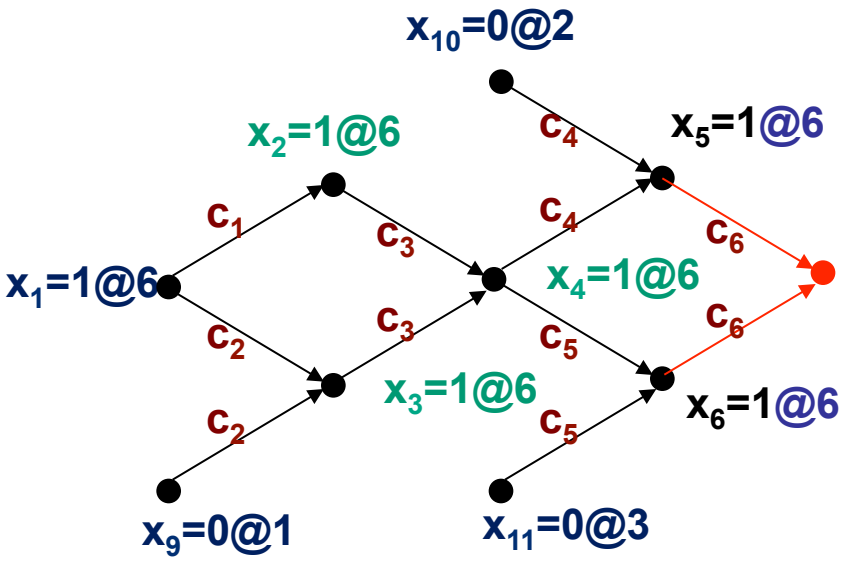


1. A first decision (at level 1) makes $x_1 = 1$
2. Propagation enforces $x_2 = 1$ and $x_3 = 1$
3. A second decision (at level 2) makes $x_5 = 0$
4. Propagation enforces $x_4 = 0$

Intelligent Backtracking

- By maintaining such graph it is easy to detect the real causes of the failures, as illustrated in the graph below.

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

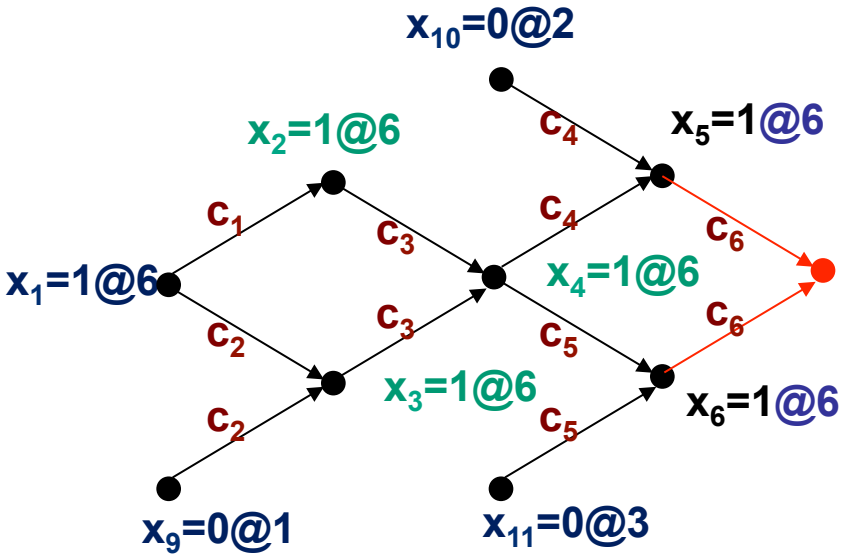


- Clearly, the conflict has been caused by assignments
  - x_1=1 and x_9=0 and x_{10}=0 and x_{11} = 0,
- although it was detected in clause c6, envolving variables x_4 and x_5.

Intelligent Backtracking

- SAT solvers usually do better than these by exploiting Unique Implication Points (UIPs) , as illustrated in the graph below.

$c_1 : (\neg x_1 \vee x_2)$
$c_2 : (\neg x_1 \vee x_3 \vee x_9)$
$c_3 : (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4 : (\neg x_4 \vee x_5 \vee x_{10})$
$c_5 : (\neg x_4 \vee x_6 \vee x_{11})$
$c_6 : (\neg x_5 \vee \neg x_6)$
$c_7 : (x_1 \vee x_7 \vee \neg x_{12})$
$c_8 : (x_1 \vee x_8)$
$c_9 : (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...



- In fact, rather than tracing the decisions, we may notice that the conflict has been caused , not only by decisions assignments

$$x_1 = 1 \text{ and } x_9 = 0 \text{ and } x_{10} = 0 \text{ and } x_{11} = 0,$$

- but also by assignments

$$x4 = 1 \text{ and } x_{10} = 0 \text{ and } x_{11} = 0,$$

Intelligent Backtracking

- Since, the conflict has been caused by the assignments

  x_1=1 and x_9=0 and x_{10}=0 and x_{11} = 0, // x_4=1 and x_{10}=0 and x_{11} = 0,

  the no-good clause below prevents repetition of this impossible assignment

  $c_{0a}: (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$ // $c_{0b}: (\neg x_4 \vee x_{10} \vee x_{11})$

- .

$c_1: (\neg x_1 \vee x_2)$
$c_2: (\neg x_1 \vee x_3 \vee x_9)$
$c_3: (\neg x_2 \vee \neg x_3 \vee x_4)$
$c_4: (\neg x_4 \vee x_5 \vee x_{10})$
$c_5: (\neg x_4 \vee x_6 \vee x_{11})$
$c_6: (\neg x_5 \vee \neg x_6)$
$c_7: (x_1 \vee x_7 \vee \neg x_{12})$
$c_8: (x_1 \vee x_8)$
$c_9: (\neg x_7 \vee \neg x_8 \vee \neg x_{13})$
...

In fact, one may notice that clause c_{0a} could have been obtained throug resolution on clauses

c_1 & $c_3: (\neg x_1 \vee \neg x_3 \vee x_4)$
& $c_4: (\neg x_1 \vee \neg x_3 \vee x_5 \vee x_{10})$
& $c_6: (\neg x_1 \vee \neg x_3 \vee \neg x_6 \vee x_{10})$
& $c_5: (\neg x_1 \vee \neg x_3 \vee \neg x_4 \vee x_{10} \vee x_{11})$
& $c_3: (\neg x_1 \vee \neg x_2 \vee \neg x_3 \vee x_{10} \vee x_{11})$
& $c_1: (\neg x_1 \vee \neg x_3 \vee x_{10} \vee x_{11})$
& $c_2: (\neg x_1 \vee x_9 \vee x_{10} \vee x_{11})$

but this technique attempts to only learn useful no-good clauses

Intelligent Backtracking

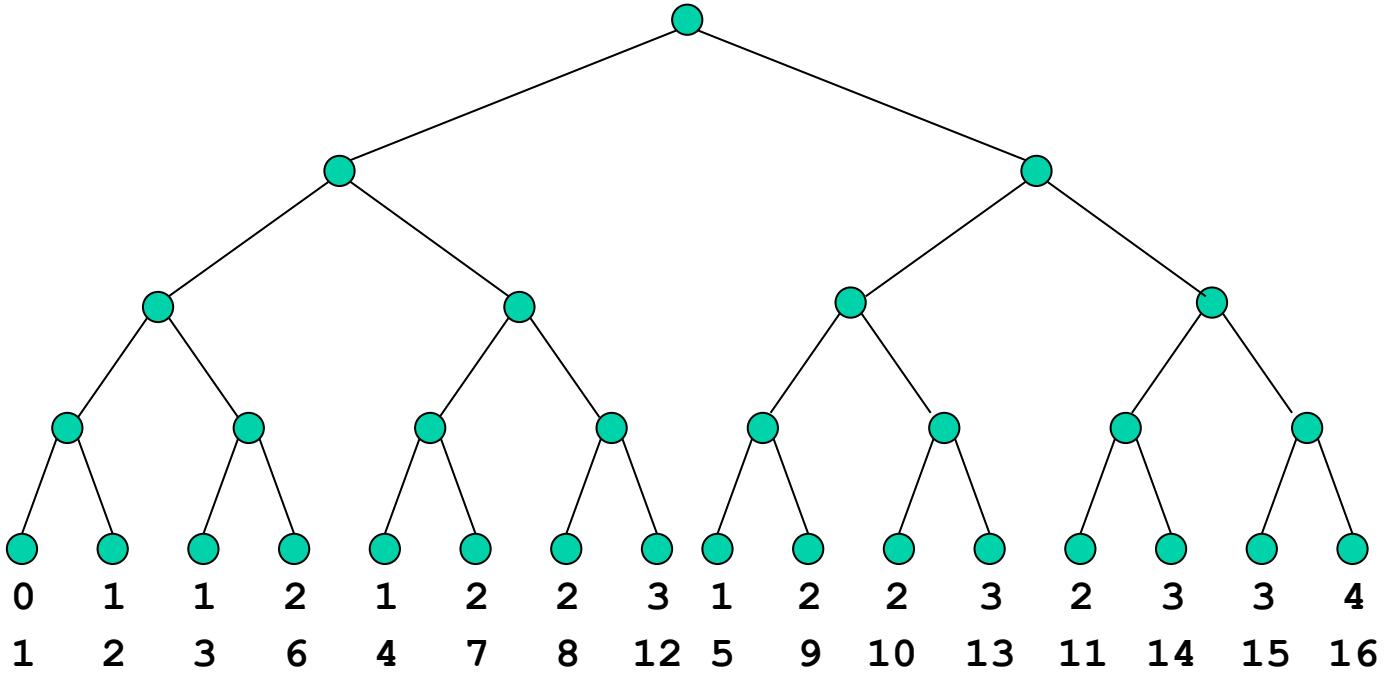
- Not all learned clauses are useful. SAT solvers are ususally paramaterised in order to determine which learned clauses are to be maintained.
- The overhead for carrying on with the analysis for non chronological backtracking might not pay off (which may depend on the heuristics used for variable/value selection). Parameterisation may help, but tuning all these parameters may be very difficult, and differ considerably for aparently similar problems.
- Nevertheless current solvers may handle benchmark instances with tens of millions of clauses on around one million variables (not random instances).
- However:
  - These numbers are misleading. Much less variables and constraints are required if problems are modelled with FD constraints.
  - Processing nogoods is simply learning a structured model that was destroyed when encoding the problem into the “poorly expressive ” SAT clauses.
- Despite these criticisms, SAT solving is quite competitive with FD solvers, and offers possibilities for hybridization with them.

Non Depth-First Search Strategies

- Constraint Programming uses, by default, depth first search with backtracking in the labelling phase.
- Despite being “interleaved” with constraint propagation, and the use of heuristics, the efficiency of search depends critically of the first choices done, namely the values assigned to the first variables selected.
- If the first variable has 2 values, and the wrong one is selected, half the search space is computed and visited uselessly!
- Hence, alternatives have been proposed to pure depth first search so as to allow the search to focus on the most promising parts of the search space.
- Among all possibilities, the most used is
    - **Limited Discrepancy**

Limited Discrepancy

- Limited discrepancy assumes that the value choice heuristic may only fail, **globally**, a (small) number of times. Hence, rather than limiting at each variable the number of bad choices it does so for all variables.



- Again, if the search fails, **d** is incremented and the search space is thus increasingly incremented. In the limit, the search is complete.

Incomplete Search Strategies - Restarts

- Several stidies have shown that heuristics solvers present a heavy-tail distribution, i.e. many instances are not solved with the same heuristic. However, different heuristics solve different instances.

- Hence, if completeness is not a major concern (we know that there are solutions) an incomplete strategy is ofetn used with success – restarts.

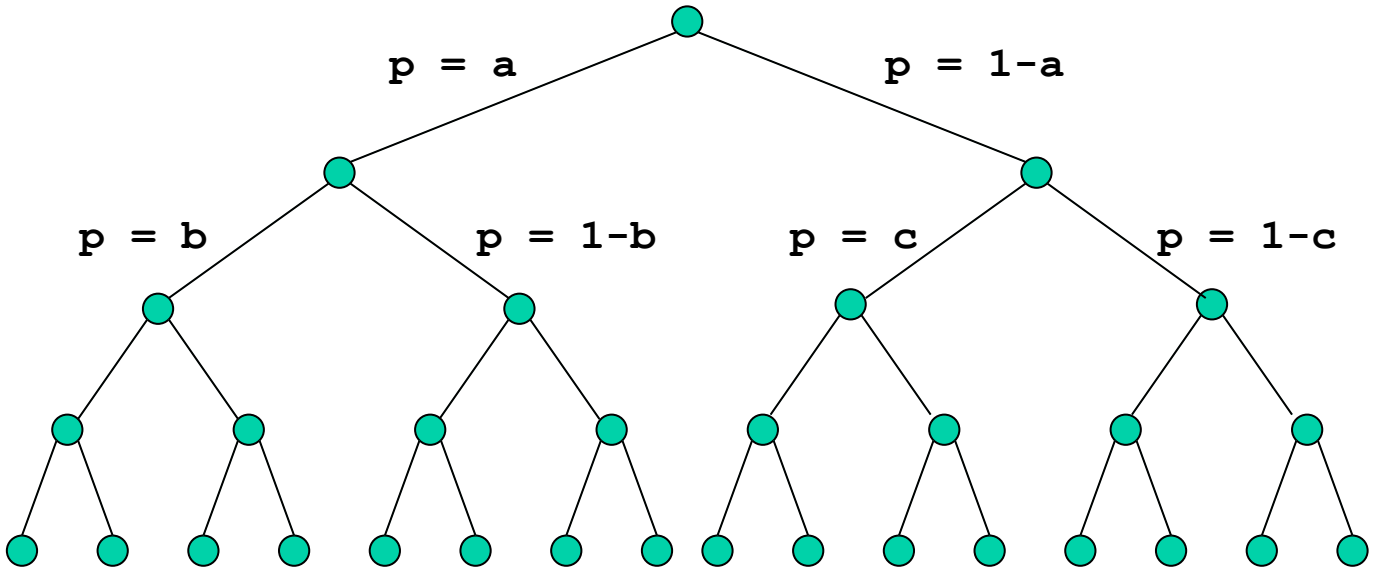
- The idea is simple. After a certain “time” without finding a solution, the search is stoped and restarted.

- Of course, the scheme requires that the heuristics used is not deterministic, otherwise the same search space would be exploited with similar un-success.

- To circumvent this problem, stochastic heuristic choices are used, allowing that different paths in the search space are investigated for different executions.

Incomplete Search Strategies - Restarts

- For every decision (binary in the figure) the preferred value is chosen with a probability a (0.5 < a <1) , whereas the alternative is selected with probability 1-a.



- This technique can be used with both k-way or 2-way branching (shown in the figure) as well as with limited discrepancy.

Why Symmetries in CP

- The main motivation for the study of symmetries in constraint programming is that finding solutions is difficult, whereas applying symmetries is usually easy.

- Hence symmetries can be used in two main situations:

- **Optimisation:**

  In optimisation (i.e. in problems requiring the computation of all solutions, at least implicitely), the use of symmetries may avoid the finding of many solutions that are symmetrical of other solutions already obtained.

- **No-goods:**

  Whenever a failed path is found in a complete tree search, symmetrical paths can be avoided if properly (and efficiently) identified. Thus the search space can be significantly reduced even when the problem consists of finding one single solution.

Why Symmetries in CP

- Of course to take advantage of symmetries, they must be identified first.

- In some cases this is an easy task that users can identify, but in other situations, users might be unable to elicit important symmetries, namely in cases where the number of symmetries is very large.

- We will thus be concerned in the study of symmetries in constraint Programming in two complementary problems:

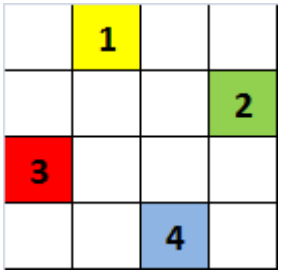
  - How to detect and specify symmetries; and

  - How to break symmetries, i.e. how to take advantage of them to simplify the search for solution(s).

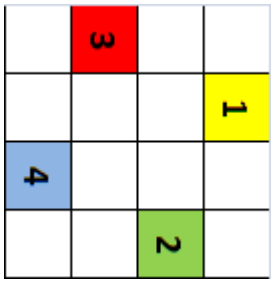
- Before being more formal, the ubiquous n-queens problem is going to be used as a motivating example.

Symmetries by Example

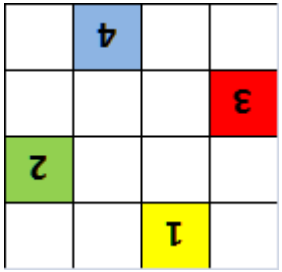
- The n-queens, as most geometrical board problems (latin squares, sudoku) presents 8 symmetries.



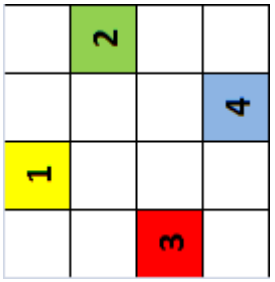
id
Identity



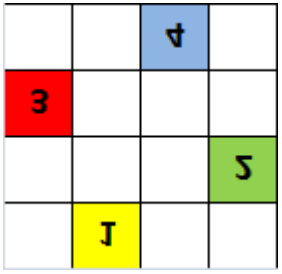
r_{90}
90° rotation



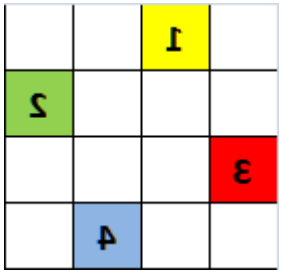
r_{180}
180° rotation



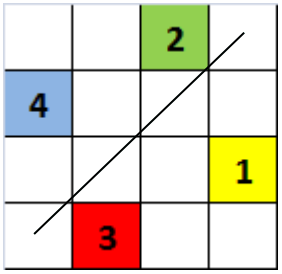
r_{270}
270° rotation



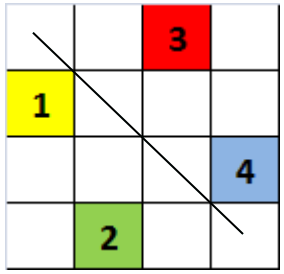
h:
horizontal reflection



v:
vertical reflection



d1:
diagonal 1 reflection



d2:
diagonal 2 reflection

Types of Symmetries

- Symmetries may be classified in several types.
- In particular, two kinds of symmetries are very important as they are often present and detectable in many problems.

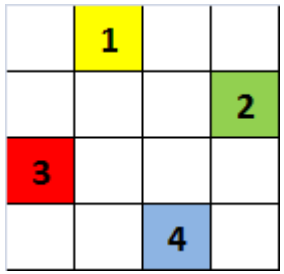
- **Value Symmetries**

  A value symmetry is a symmetry **s** that is a permutation on the values of the variables of the problem, i.e.

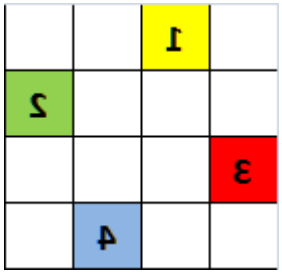
$$s(x = a) \equiv x = s(a)$$

- These symmetries define permutations of values that are applicable to all variables. For example, the symmetry **v** in n*n board problems are value symmetries (the case for n = 4 is shown below)

v → v
1 → 4
2 → 3
3 → 2
4 → 1



id



v

Types of Symmetries

- The next definition is dual of the value symmetry.

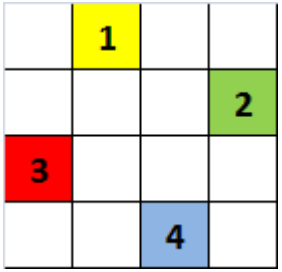
Variable Symmetries

A variable symmetry is a symmetry **s** that is a permutation on the variables of the problem, i.e.

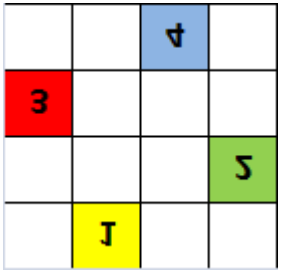
s(x = a) ≡ s(x) = a

- These symmetries define permutations of variables that are applicable to all values. For example, the symmetry **h** in n*n board problems are value symmetries (the case for n = 4 is shown below)

x → x
x1 → x4
x2 → x3
x3 → x2
x4 → x1



id



h:

Symmetry Detection and Symmetry Breaking

- There are two main issues in dealing with symmetries in Constraint Programming, namely

Symmetry Detection

- Symmetries have to be detected, before they can be efficiently addressed. In some cases this is an obvious task, in others, specially when their number is very large, this is not so easy.

Symmetry Breaking

- Once detected, methods have been studied to take advantadge of the symetries, namely to avoid the exploitation of useless parts of the search tree, whose positive or negative findings can be obtained by applying symmatrical reasoning on the exploited parts.

Symmetry Breaking

- Breaking the symmetries can be performed by a number of techniques. They can be classified as follows:

Problem Reformulation

- A new model of the problem may be studied, that eliminates some (or all) of the symmetries that were found. For example, several symmetries can be broken by adopting sets as the domain of certain variables.

Addition of Symmetry Breaking Constraints

- When symmetries are detected in a model, additional constraint may be added to the model in order to eliminate these symmmetries. Such addition can be static (before execution) or dynamic (during execution).

Symmetry Breaking before Search

- When there is no priviliged solution, one may simply impose an arbitrary solution as canonic. Usually, the canonical solution imposes an increased ordering on the lexicographic order of the variables, i.e.

$$x1 < x2 < x3$$

  i.e. the canonical solution is <1,2,3>.

- This technique may be used in applications envolving several sets, such as the well known example of the

Social Golphers problem (g,s,w):

- The goal is to schedule **g*s** golphers, in **g** groups of **s** players each, such that they can play for **w** weeks and two players never play in the same group more than once.

- This problem presents a huge number of symmetries that prevents an efficient execution, even for small numbers of g, s and w, unless these symmetries are not broken. To simplify, we will use the values g = 3, s= 2, and w = 4.

Symmetry Breaking before Search

- **Value symmetries** [(g*s)!]: If a solution assigns to the variables x1, x2 , ... , xn (with n = g*s) different values in the range 1, 2, ..., n, any permutation of these values is also a solution.

  This is a typical situation in sport tournaments: a schedule is prepared in terms of virtual teams t1, t2, tn, which are assigned an arbitrary assignmnet (by a random selection – e.g drawing the names from a box)

- **Group Symmetries** [w*g* (s !)]: : Within each group, a set, the elements can be permuted

- **Group Symmetries within each weak** [w*(g!)]: the order of the groups in each week is arbitrary.

- **Week Symmetries** [w!] : The weeks are arbitrarily permuted.

- All together, there are (w!) * w*(g!) * w*g*(s!) * (s*g)! Symmetries. This may be a huge number. Even in the small instances that is considered here, there are 4! * 4*3! * 4*3* 2! * (3*2!) = 24*24*24*720 ≈ 10^7 symmetries.

Symmetry Breaking before Search

- A number of ad hoc constraints can be used to eliminate these constraints. Let us represent a solution by variables x_{wgp} where w is the week, g the group and p the position in the group of the player

$$\{ x_{111} , x_{112} \} , \{ x_{121} , x_{122} \} , \{ x_{131} , x_{132} \}$$

$$\{ x_{211} , x_{212} \} , \{ x_{221} , x_{222} \} , \{ x_{231} , x_{232} \}$$

$$\{ x_{311} , x_{312} \} , \{ x_{321} , x_{322} \} , \{ x_{331} , x_{332} \}$$

$$\{ x_{411} , x_{412} \} , \{ x_{421} , x_{422} \} , \{ x_{431} , x_{432} \}$$

- Taking into account an implicit all-different constraint on the variables in each week:

Value symmetries:

Some of them can be broken by fixing the values for the first week, which are completely arbitrary as, for example, { 1 , 2 } , { 3 , 4 } , { 5, 6 }

Symmetry Breaking before Search

Group symmetries:

- The standard technique can be applied to sort the sets in increasing order, by adding constraints $\mathbf{x_{wgp} < x_{wgq}}$ for all ellements (**p < q** in 1..2) of all groups (**g** in 1..3) of all weeks (**w** in 1..4).

Group in Weeks symmetries:

Taking into account that the first elements of the groups in the same week must be different and smaller than the other elements of the corresponding groups, a standard order of the groups may be imposed by adding constraints $\mathbf{x_{wg1} < x_{wh1}}$ for all all weeks (**w** in 1..4). 1..4 and for all groups in each week (**g< h** in 1..3).

Week symmetries:

- The previous constraints force, for all weeks (w in 1..4), the first element of the first group to be 1 ($x_{w11} = 1$). Then the second elements must be difefernt. Week symmetries may be broken by imposing $x_{w12} < x_{v12}$ for each pair of weeks w < k in 1..4.

Dynamic Symmetry Breaking

Dynamic Symmetry Breaking (DSB)

- Rather than posting constraints before the search starts, as done in lex-leader and ad hoc methods that have been used in specific problems, DSB methods, analyse the search at every choice point and perform some extra operations to avoid the exploration of paths leading to symmetric solutions. Two main methods have been proposed

SBDS – Symmetry Breaking During Search

Assuming that an assignment has been tested, all the symmetric assignments are excluded from the search by addition of symmetry breaking constraints.

SBDD – Symmetry Breaking via Dominance Detection

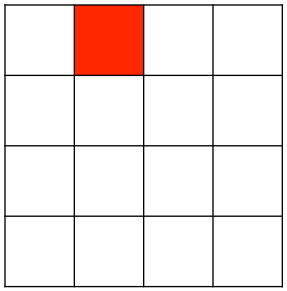
At every node of the search tree, SBDD checks whether the node is symmetric of some other node already exploited, in which case it does not expand the node.

SBDS by example

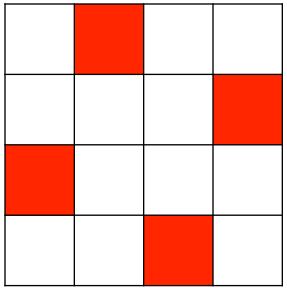
- In SBDS, the negations done in the right branches take into account not only the actual assignments made but also their symmetrical.

Example: 4 queens

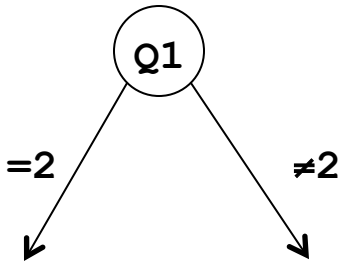
- Let us assume that the first assignment made is Q1 = 2 (left branch).



- This assignment is fully explored (it leads to the single solution shown) in the left hand branch.



- Then the right hand branch introduces constraint Q1 ≠ 2, aiming at some optimisation (by propagation)



SBDS by example

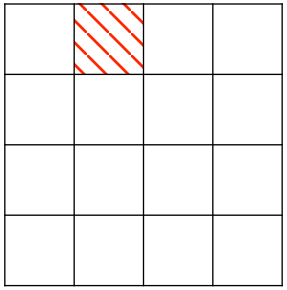
Example: 4 queens

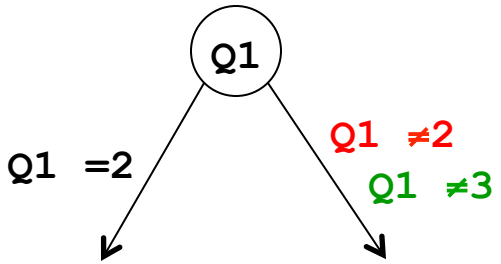
- The added constraint Q1 ≠ 2 guarantees that already found solution(s) (with Q1 = 2) are not found again).

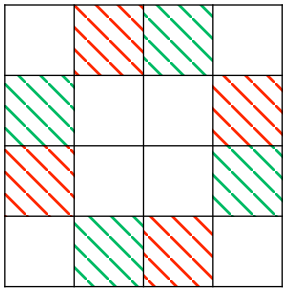
- But finding symmetrical solutions may also be avoided if symmetrical constraints are posted.

- For example, the vertical reflection of assignment Q1 = 2 is Q1 = 3, so this constraint can be added as well.

- In fact this eliminates finding the other solution, which can be obtained later by (vertical reflection) symmetry.



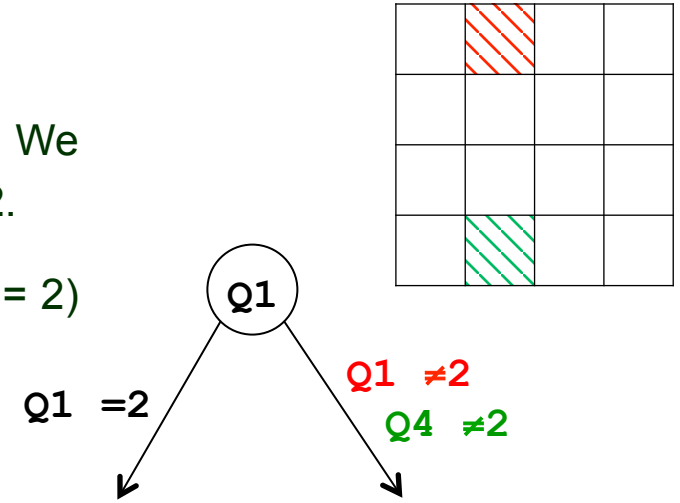




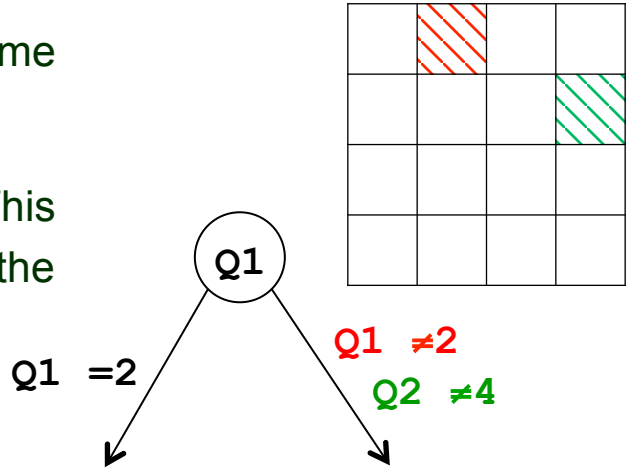
SBDS by example

Example: 4 queens

- The technique works with any symmetry. We could use vertical symmetry to exclude Q4 = 2.
- Again the other solution (with Q1 = 3 and Q4 = 2) would not be rediscovered.



- **Inneficiency**
- If the technique works with any symmetry, some will lead to useless prunning.
- For example, 90º rotation will exclude Q2 = 4This wouldsimply eliminate the rediscovery of the same solution already found.



SBDS by example

- The situation is a bit more complex when the node where the symmetry breaking constraint is added is not the root node.

- Take the example shown.In this case, we cannot simply state Y ≠ 4 on the right branch of node Y, because it could lead to loss of solutions

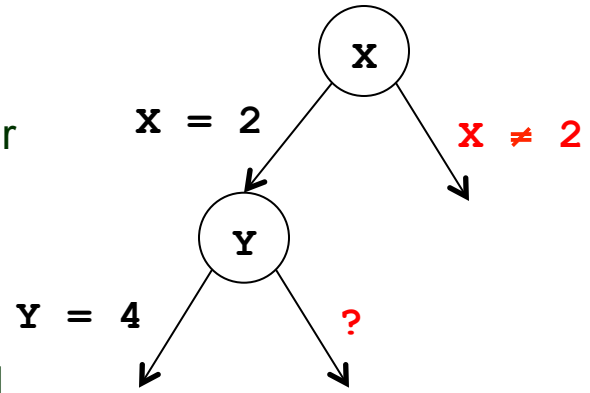
- For example, a solution with X = 3 and Y = 4.

- All that is safe to is impose that Y = 4 is no longer acceptable in the context of X = 2, i.e.

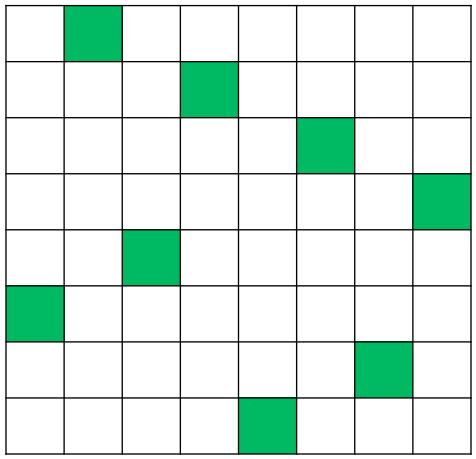
$$X = 2 \rightarrow Y \neq 4.$$

- Again, symmetrical conclusions should be inferred. Given a symmetry s, we could add on the right branch the constraint

$$s(X = 2 \rightarrow Y \neq 4)$$



Symmetry Breaking During Search



8-queens

Successful Applications

- Many of the symmetry breaking methods outlined in the previous sections have been applied successfully to a variety of problems, and new results were obtained using symmetry breaking in constraint programming. Some successful applications include

  • Maximum Density Still Life (CSPLib – 38)

  • Balanced Incomplete Block Design (BIBD) generation (CSPLib – 28)

  • Social Golfers Problem (CSPLib – 10)

  • Fixed Length Error Correcting Codes (CSPLib – 36)

  • Peg Solitare (CSPLib – 37)

Further reading (a survey)

- Ian Gent, Karen Petrie and J.-François Puget, Symmetry in Constraint Programming, in **Handbook of Consraint Programming**, Elsevier, 329-376, 2006.