Heuristic Search

- Heuristic Search

  - Variable and Value selection
  - Static and Dynamic Heuristics
  - Advanced Search Techniques

Complete 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, maintains 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) – and traversing the tree to a constructive approach for finding solutions.

Complete 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 during labelling)

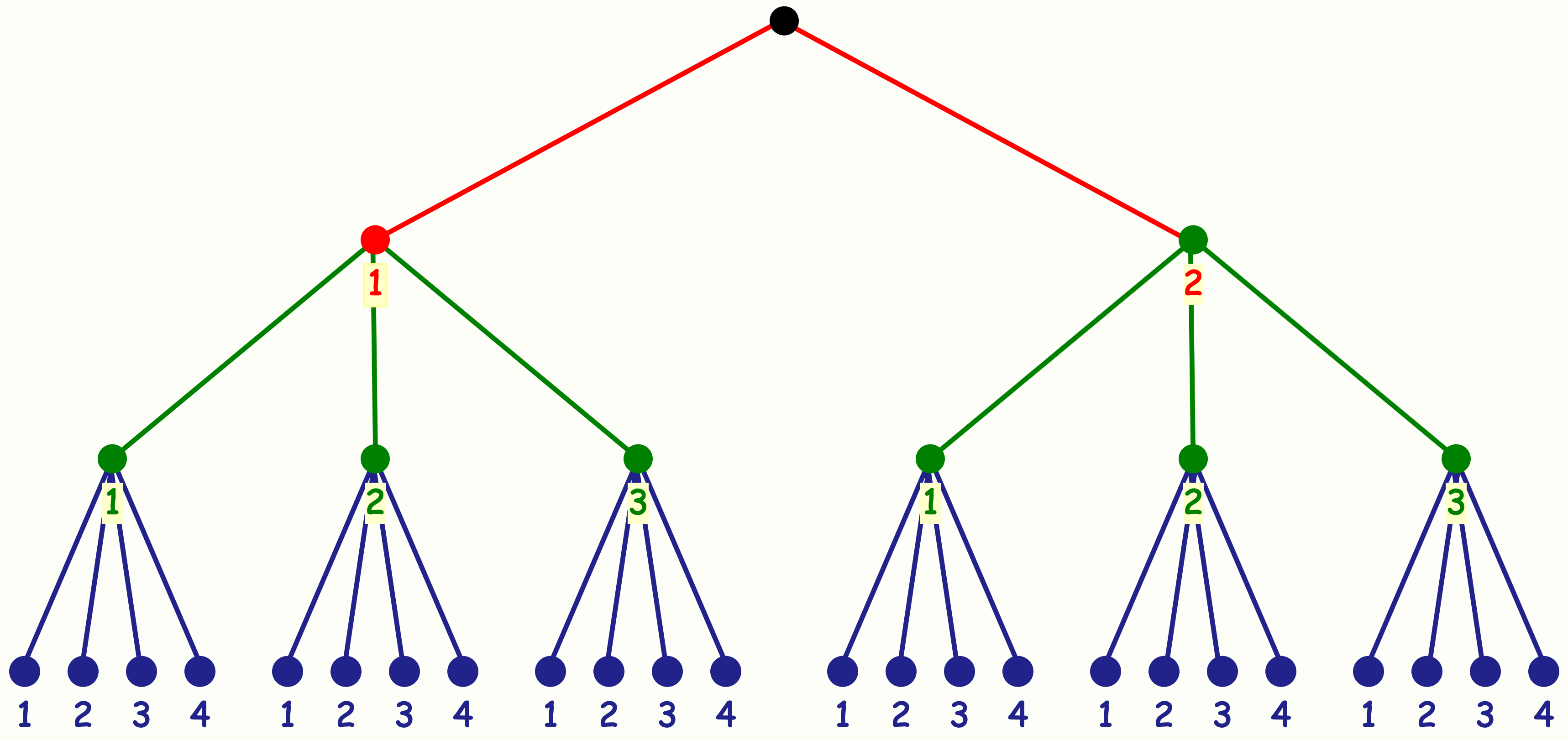
```
... // declaration of Model
... // declaration of Solver
... // declaration of variables
... // declaration of constraints
... // labelling of variables
... // report solution
```

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.

Complete Search

- Take for example a problem with variables of array `x` with domains

`x[0] in 1..2`, `x[1] in 1..3` and `x[2] in 1..4`.



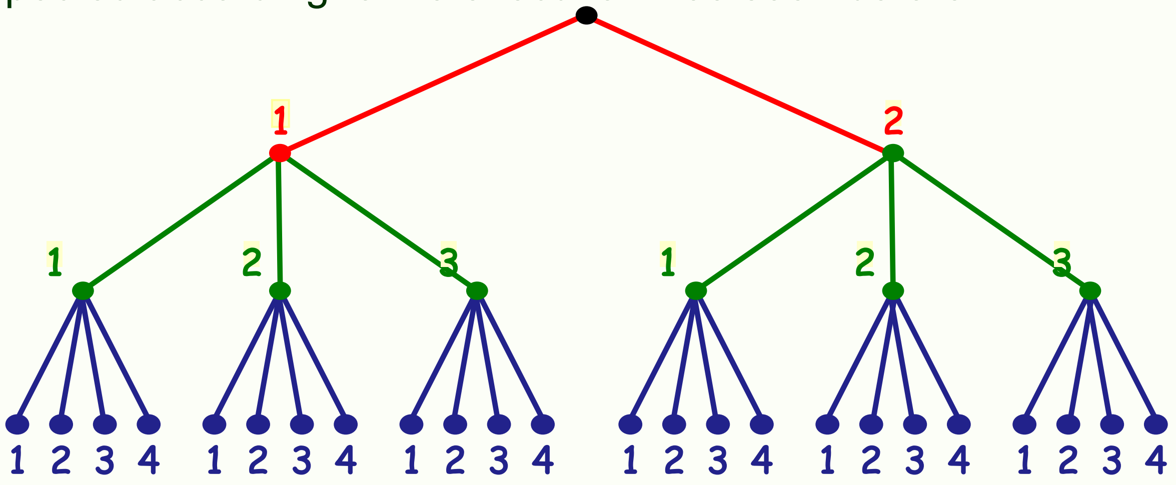
Complete Search

- A depth first search, left to right is typically performed **by default** in Choco, if no specific solving strategy strategy is set. The previous example, encoded as

```
Model model = new Model(" X123");
Solver s = model.getSolver();
IntVar [] x = model.intVarArray("X", 3, 1, 4);
model.arithm(x[0], "<=", 2).post();
model.arithm(x[1], "<=", 3).post();
model.arithm(x[2], "<=", 4).post();
while (s.solve()){
    // show solutions
}
```

provides all solutions as expected according to the execution tree seen before.

```
-- solution 1: 1 1 1
-- solution 2: 1 1 2
-- solution 3: 1 1 3
-- solution 4: 1 1 4
.......
-- solution 21: 2 3 1
-- solution 22: 2 3 2
-- solution 23: 2 3 3
-- solution 24: 2 3 4
```



Complete Search

- In fact the labelling strategy used by the solver iterates two decisions:
  - Select a variable to label
  - Select the value used to assign to the variable

- Different strategies can be explicitly set in Choco. All that is required is to define the variables to label and the values to be used.

- The typical depth first strategies, **left to right** and **right to left** can be imposed on an array of variables by explicitly setting the corresponding methods of the solver, after importing the strategy classes, as shown

```
//import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Solver s = model.getSolver();
       IntVar [] x = model.intVarArray("X", 3, 1, 4);
       ...
       s.setSearch(inputOrderLBSearch(x)) // left to right
       //s.setSearch(inputOrderUBSearch(x)) // right to left
.....
```

Complete 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 pruning of the search tree.

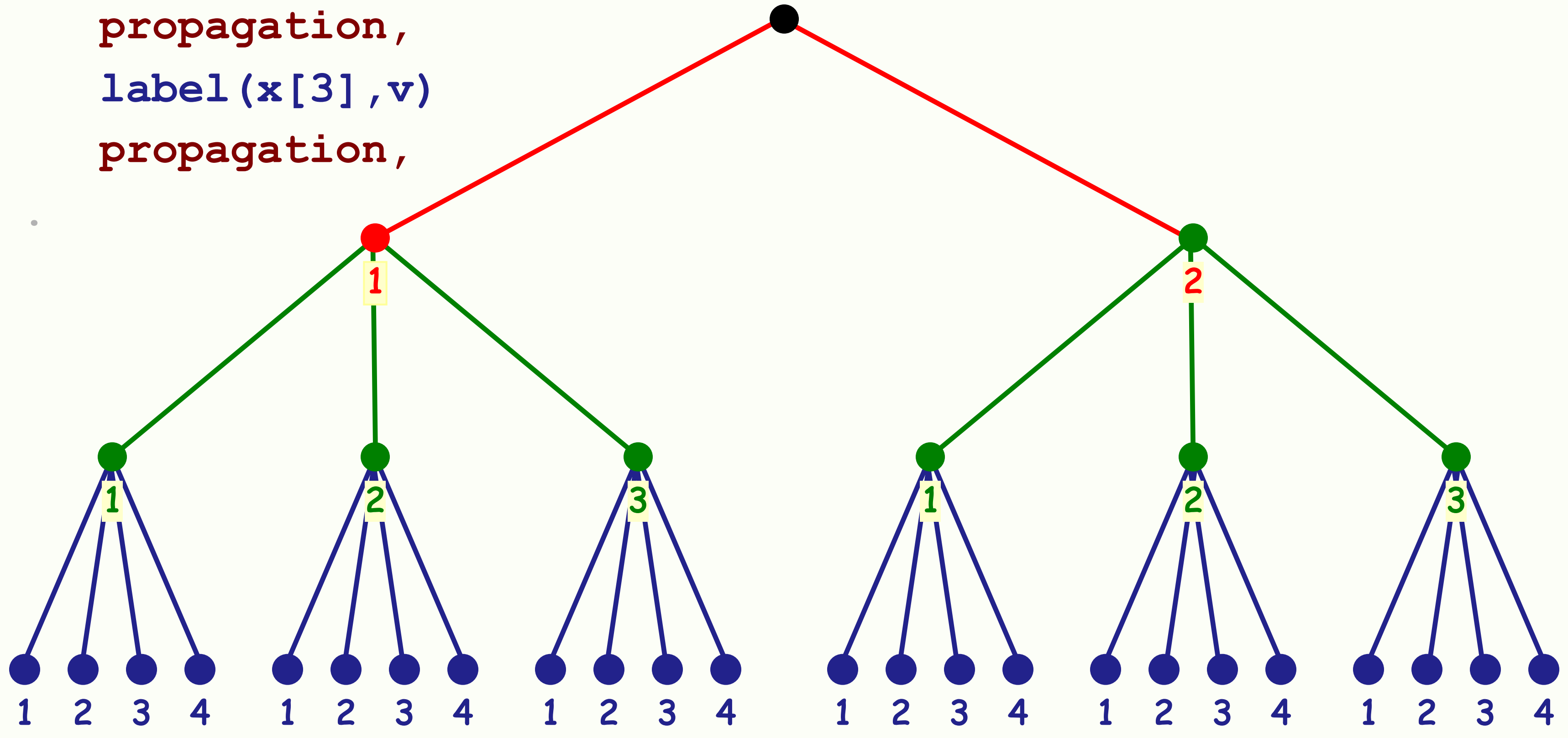
- The **ordering of the domains** has no direct influence on the search space, although it may have great importance in finding the first solution.

Complete Search

```
label(x[1],v)
propagation
label(x[2],v),
propagation,
label(x[3],v)
propagation,
```

# of nodes = 32

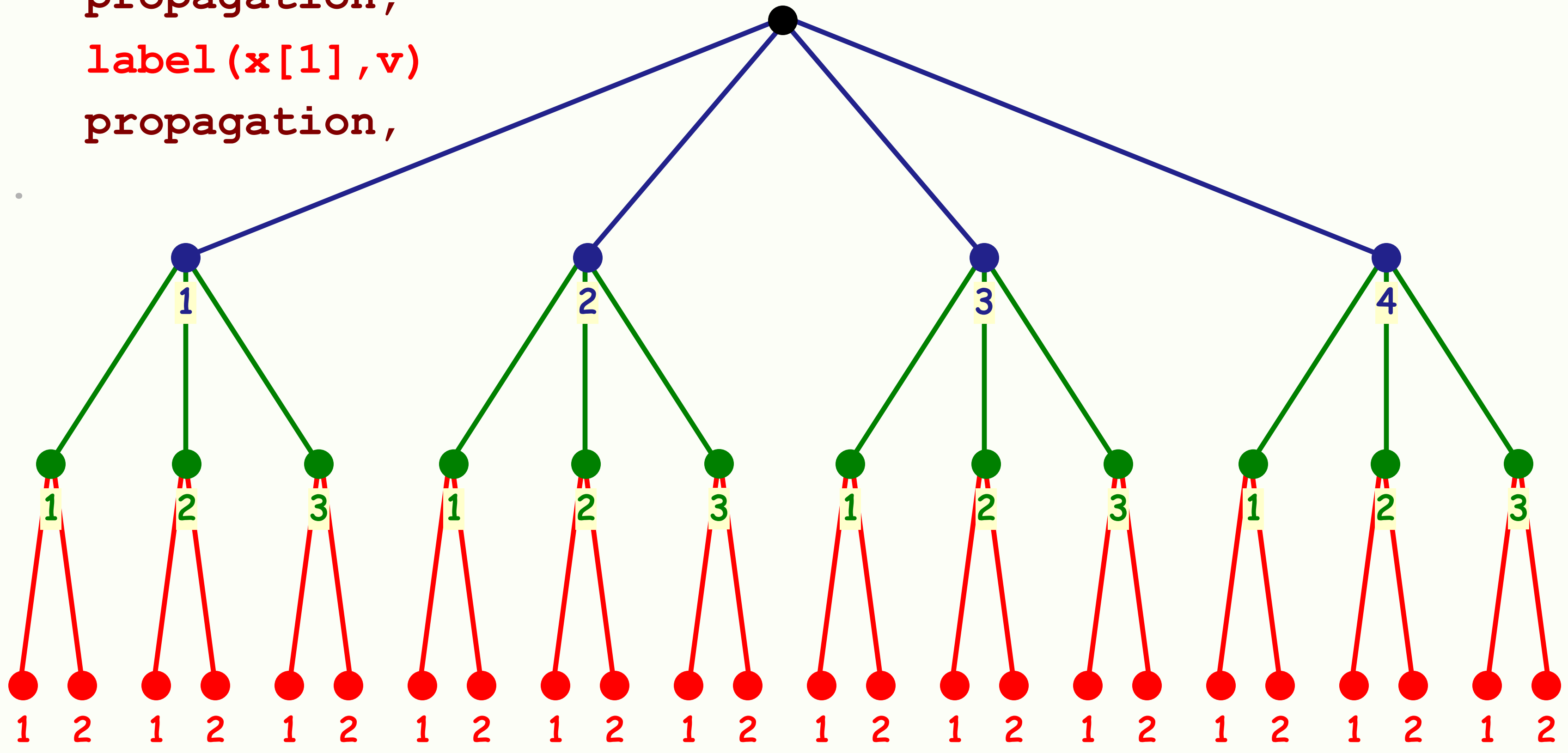
(2 + 6 + 24)



Complete Search

```
label(x[3],v)
propagation
label(x[2],v),
propagation,
label(x[1],v)
propagation,
```

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



Complete Search

- This latter strategy may be imposed by setting the strategy on a vector y of variables that corresponds to vector x in reverse order, and label the y variables

```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Solver s = model.getSolver();
       IntVar [] x = model.intVarArray("X", 3, 1, 4);
       ...
       IntVar [] y = model.intVarArray("Y", 3, 1, 4);
       for(int i = 0; i < 3; i++)
           y[2-i] = x[i];

       //s.setSearch(inputOrderLBSearch(y)) // left to right
       s.setSearch(inputOrderUBSearch(y)) // right to left


       while (s.solve()){
           // show solutions
       }
.....
```

- Note that starting with the upper bounds the search tree is explored right to left.

Complete Search

- In general, if variables to be labelled belong to different vectors, one may gather them all together in a single vector to which the appropriate strategy is set.
- In some cases, the variables may be labelled in sequence, possibly with different strategies.
- The alternatives are shown below

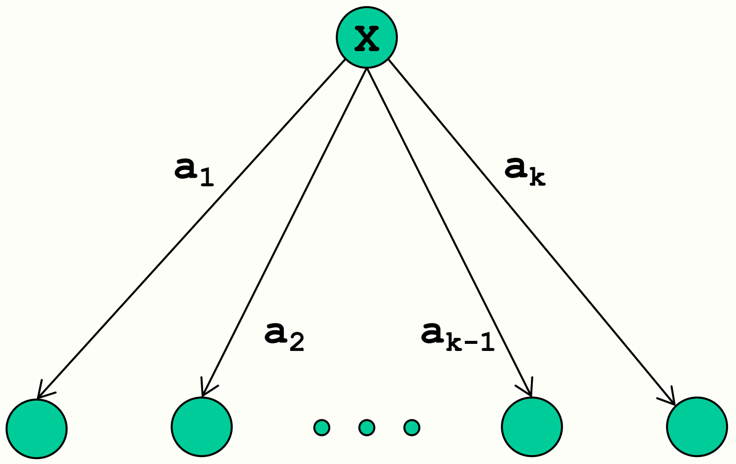
```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Solver s = model.getSolver();
       IntVar [] x = model.intVarArray(...);
       IntVar [] y = model.intVarArray(...);
       IntVar [] z = model.intVarArray(...); // x and y appended

       //s.setSearch(inputOrderLBSearch(z));    // left to right in z
       s.setSearch(inputOrderLBSearch(x),       // left to right in x
                   inputOrderUBSearch(y));      // right to left in y

       while (s.solve()){
           // show solutions
       }
.....
```

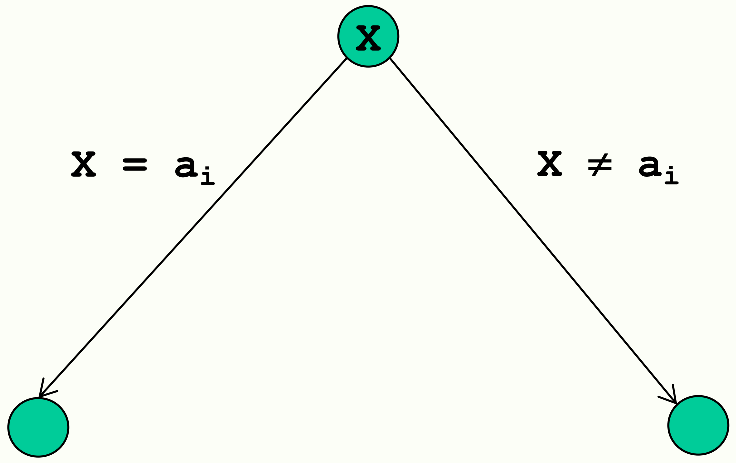
Different Branching Strategies

- So far it has been assumed n-way branching: when a variable is selected for labelling and has several values in its domain all these values are tried in some order.

- However, this is often *not* the most efficient way of organizing choice points. At least two other alternatives are quite common in practice: 2-way branching and domain splitting.

- 2-way branching simply imposes that a variables either takes or not some selected value.

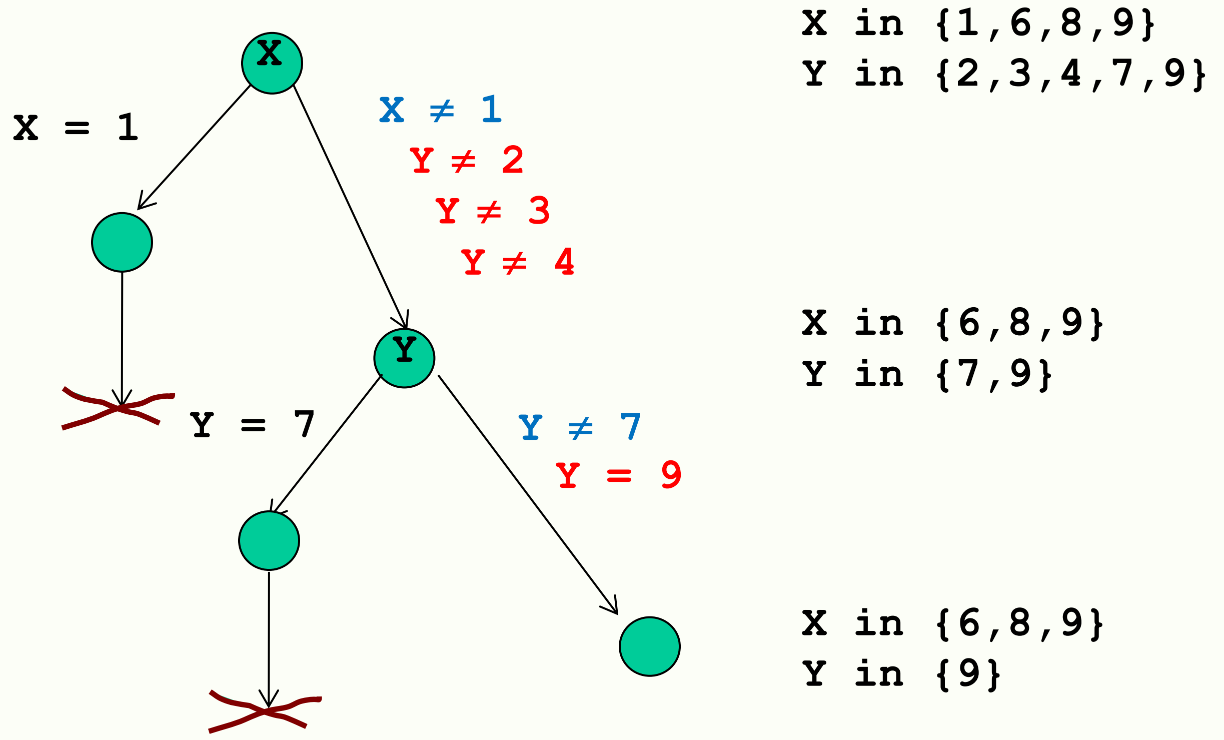




Different Branching Strategies

- The main advantage of this method is to prevent heuristics to be stuck at specific variables. Once a value is not considered convenient for a variable, this should not force another value for the same variable to be selected. It might be better to select another variable/value pair as shown below

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



Different Branching Strategies

- Choco assumes this type of branching by default. This can be seen in the execution of the program

```
IntVar x = md.intVar("x", new int[] {1,6,8,9});
IntVar y = md.intVar("y", new int[] {2,3,4,7,9});
md.arithm(x, "<=", y).post();
          sv.setSearch(minDomLBSearch(x,y));
while (s.solve()){
    // show solution
    System.out.println(sv.getDecisionPath());
}
```

where the chosen heuristic, `minDomLBSearch(x,y)`, (cf.later) always picks the variable with less values in the domain to be enumerated next.

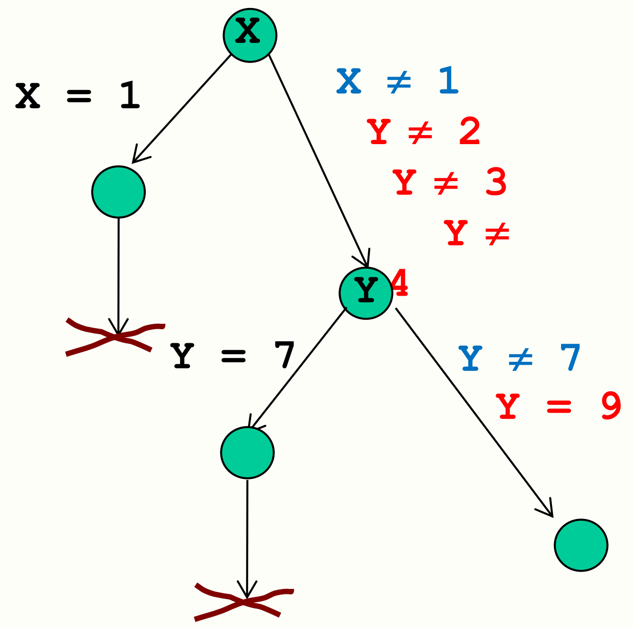
The solver method

`getDecisionPath());`

provides the path with all the decisions made until a solution is reached.

Different Branching Strategies

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



```
- solution 1: 1 2 path = Path[3]: ROOT, d_1: x=1, d_2: y=2
- solution 2: 1 3 path = Path[4]: ROOT, d_1: x=1, d_2: y\2, d_3: y=3
- solution 3: 1 4 path = Path[5]: ROOT, d_1: x=1, d_2: y\2, d_3: y\3, d_4: y=4
- solution 4: 1 7 path = Path[6]: ROOT, d_1: x=1, d_2: y\2, d_3: y\3, d_4: y\4, d_5: y=7
- solution 5: 1 9 path = Path[6]: ROOT, d_1: x=1, d_2: y\2, d_3: y\3, d_4: y\4, d_5: y\7
- solution 6: 6 7 path = Path[3]: ROOT, d_1: x\1, d_2: y=7
- solution 7: 6 9 path = Path[4]: ROOT, d_1: x\1, d_2: y\7, d_3: x=6
- solution 8: 8 9 path = Path[5]: ROOT, d_1: x\1, d_2: y\7, d_3: x\6, d_4: x=8
- solution 9: 9 9 path = Path[5]: ROOT, d_1: x\1, d_2: y\7, d_3: x\6, d_4: x\8
```

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.
    - As seen before, in Choco, such ordering may be imposed by aggregating all the decision variables in a single vector, sorted by the ordering to be imposed.
  - **Dynamic** - the selection of the variable is determined after analysis of the problem that resulted from previous enumerations (and propagation).

Static Heuristics

- Static heuristics are based on some properties of the underlying constraint graphs, namely their **width**.

- **Node width, given ordering O**:

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

- **Width of a graph G, induced by O:**

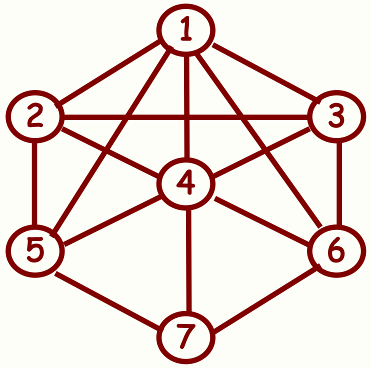
  Given some ordering, O, of the nodes of a graph, G, the width of G induced by ordering O, is the maximum width of its nodes, given that ordering.

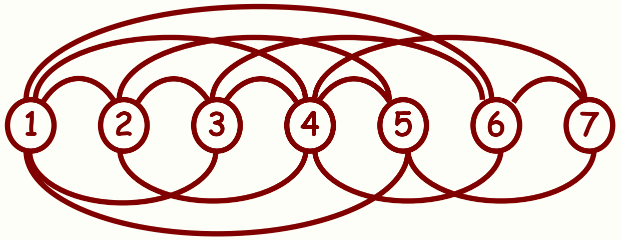
- **Width of a graph G:**

  The width of a graph G is the lowest width of the graph induced by any of its orderings O.

Static Heuristics

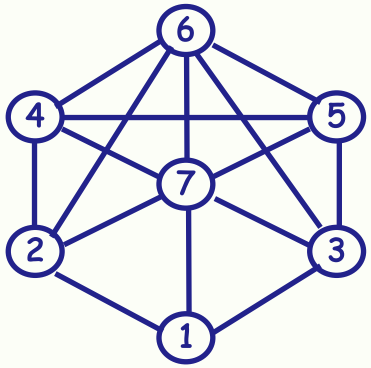
- In the graph below, we may consider various orderings of its nodes, inducing different widths, as discussed before.
- The width of the graph is 3, as is, for example the width induced by ordering O1,

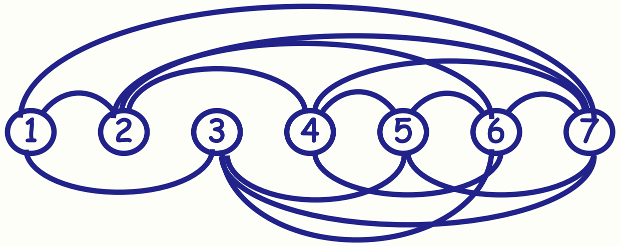




1: 0 / { }
2: 1 / {1}1
3: 2 / {1,2}
4: 3 / {1,2,3}
5: 3 / {1,2,4}
6: 3 / {1,3,4}
7: 3 / {4,5,6}

- although order O2 induces a width of 6 on the graph).





1: 0 / { }
2: 1 / {1}
3: 1 / {1}
4: 1 / {2}
5: 2 / {3,4}
6: 4 / {2,3,4,5}
7: 6 / {1,2,3,4,5,6}

Static Heuristics

- The **MWO** is a static heuristics based on the width of the underlying constraint graphs.

MWO Heuristics (*Minimum Width Ordering*):

The *Minimum Width Ordering* heuristics suggests that the variables of a constraint problem are chronological enumerated, in some ordering that leads to a minimal width of the *primal* constraint graph.

- This heuristic is “justified” by the relationship between graph (induced) width and satisfiability, through the initial imposition of i-consistency (remind that i = 1, 2 or 3 correspond, respectively to node-, arc- and path-consistency).

- Even if, given its computational cost, one cannot impose i-consistency with values of **i** high enough to guarantee backtrack free search, it is likely that low width inducing orderings will lead to some acceptably low backtracking.

Static Heuristics

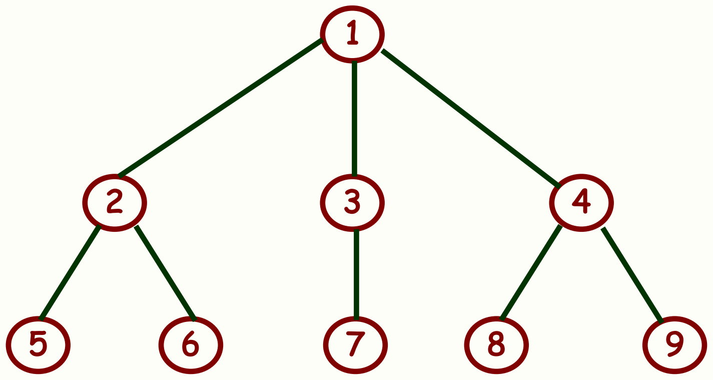
Special cases:

Trees are graphs of width 1. Hence a **backtrack free search** (*even with no propagation during search*) is obtained after imposing strong 2-consistency (i.e. arc-consistency) and enumerating variables in an width-1 inducing order:

(1, 2, 3, 4, 5, 6, 7, 8, 9) and also

(1, 2, 5, 6, 3, 7, 4, 8, 9), but beware of

(2, 3, 4, **1** , 5, 6, 7, 8, 9).



Static Heuristics

Special cases:

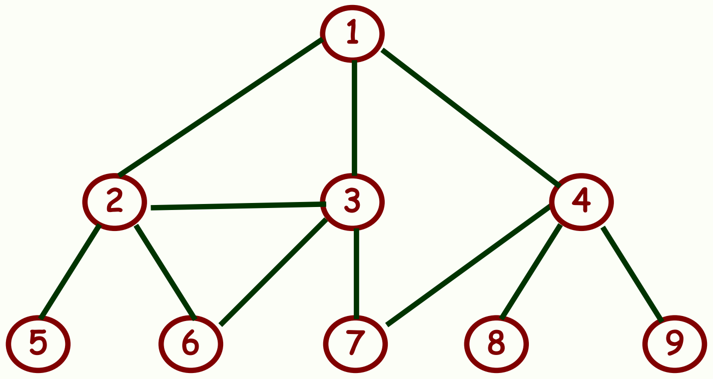
In graphs of width i-1, backtrack free search (*with no propagation during search*) is obtained by imposing strong i-consistency and enumerating variables in an width-i inducing order. In the example (i=3, path-consistency)

(1, 2, 3, 4, 5, 6, 7, 8, 9), and also

(1, 3, 2, 4, 5, 6, 7, 8, 9), but not

(1, 2, 5, 6, **3**, 4, 7, 8, 9), nor

(1, 4, 9, 8, 7, 3, 6, **2**, 5).



Static Heuristics

- An approximation of the **MWO** heuristics is the **MDO** heuristics that avoids the computation of ordering O leading to lowest constraint graph width.

MDO Heuristics (*Maximum Degree Ordering*):

The *Maximum Degree Ordering* heuristics suggests that the variables of a constraint problem are enumerated, by decreasing order of their degree in the constraint graph.

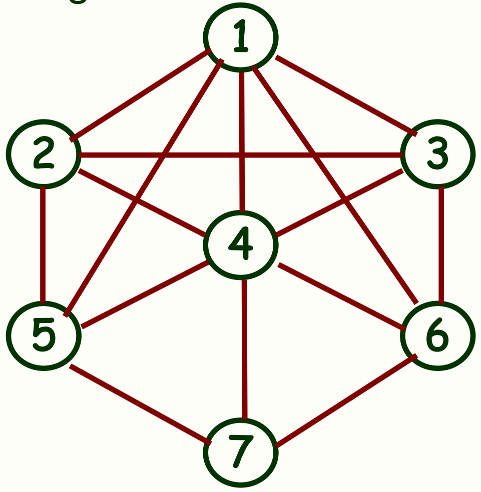
- This heuristic avoids computing an ordering from the outset, and computes the number of adjacent neighbours of the nodes in the remaining graph, with a similar algorithm to compute minimum width orderings, but
  - placing nodes with **higher** degrees in the **beginning** of the ordering; rather than
  - placing nodes with **lower** degrees in the **end** of the ordering.

Static Heuristics

- Both the **MWO** and the **MDO** heuristic tend to start the enumeration by those variables with more variables adjacent in the graph, **resulting in the early detection of dead ends.**

- Having a common rational, **MWO** and **MDO** orderings are not necessarily coincident.

Example:



- In the graph shown, the two MDO orderings

  O1 = [4/6, **1**/5, **5**/4, **6**/4, **2**/4, **3**/4, **7**/3**]** , and

  O2 = [4/6, **1**/5, **2**/4, **3**/4, **5**/4, **6**/4, **7**/3**]**

  induce different widths (4 and 3, respectively).

  O1 = [4/0{}, **1**/1{4}, **5**/2{1,4}, **6**/2{1,4}, **2**/3{1,4,5}, **3/4**{1,2,4,6}, **7**/3{4,5,6}**]** , and

  O2 = [4/{}, **1**/1{4}, **2**/2{1,4}, **3**/3{1,2,4}, **5**/3{1,2,4}, **6**/3{1,3,4}, **7**/3{4,5,6}**]**

Static Heuristics

- When assuming constraint propagation algorithms are interleaved with enumeration, another property of the underlying graph can be exploited: **cycle-cut sets**.

- In fact, given the complexity of the algorithms to impose i-consistency, only small values of i are usually considered. In particular, i=2 corresponds to arc-consistency, usually a good trade of between cost of maintaining consistency and search efficiency improvement.

- Nevertheless, in the worst case, backtracking is needed in (almost) all variables. However, this is not always the case.

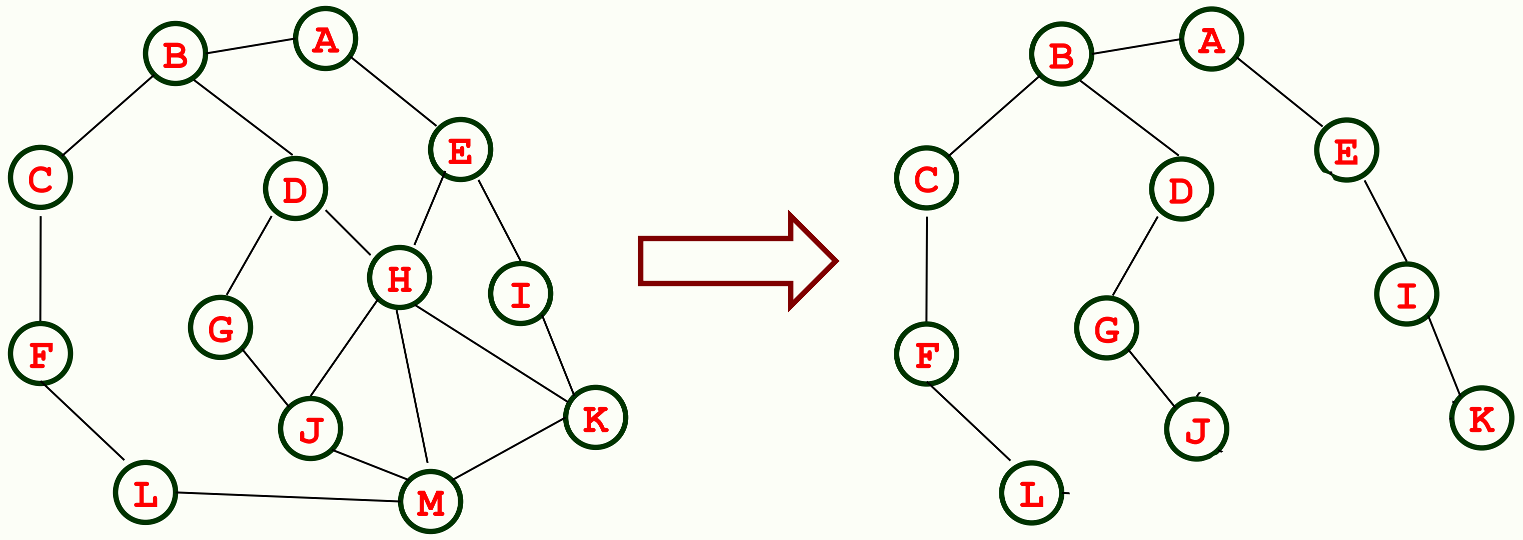
- In particular, when the underlying graph is a tree, we have noticed that backtrack free search is possible if (directional) arc-consistency is imposed.

- This is the idea of a cycle-cut set – find a set of nodes that when removed cut all cycles in the graph, i.e. convert the graph into a tree!

Static Heuristics

Example:

- In the graph shown, as soon as some of the variables are enumerated the graph becomes a tree. Which variables?



CCS Heuristics (*Cycle Cut Set Heuristic*):

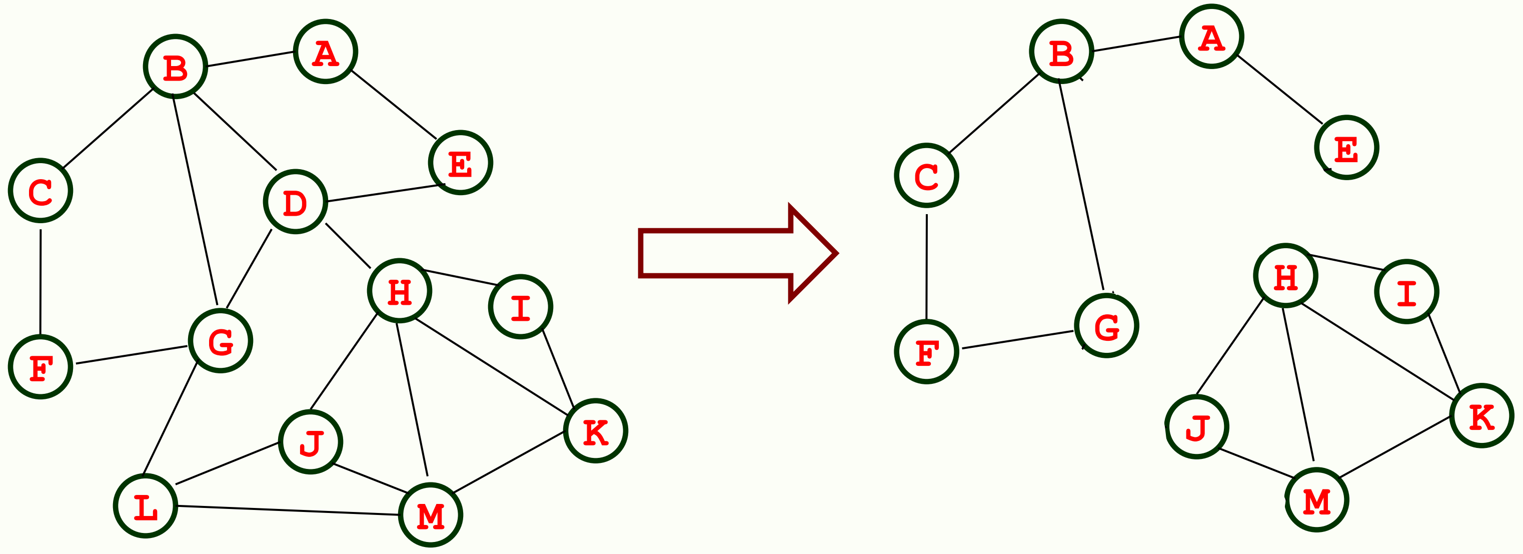
The ***Cycle Cut Set Heuristic*** heuristics suggests the variables of a lowest cardinality cycle-cut set to be enumerated first, reducing the problem to a tree shaped network.

Static Heuristics

- Unfortunately, there is no known polynomial algorithm to obtain cycle-cut sets with lowest cardinality.
- But a good guess is to start with highest degree nodes. In a way, the MDO heuristic may hence lead somehow to a CCS heuristic.
- Of course, this strategy assumes that arc-consistency is maintained interleaved with enumeration, and tree-checking of the remaining nodes of the graph.
- Note that checking whether a graph is a tree can be done in polynomial time.
- As such, as soon as a tree is reached, arc-consistency automatically guarantees either
  - a solution is found with no backtracking (no extra computational work is needed as soon as an arc-consistency tree is reached); or
  - unsatisfiability is proved, and backtracking is performed **only** on the variables of the cycle cut set alone.
- **Note:** In some cases it may pay off to impose/maintain path consistency, if a graph of induced width of two is likely to be found.

Static Heuristics

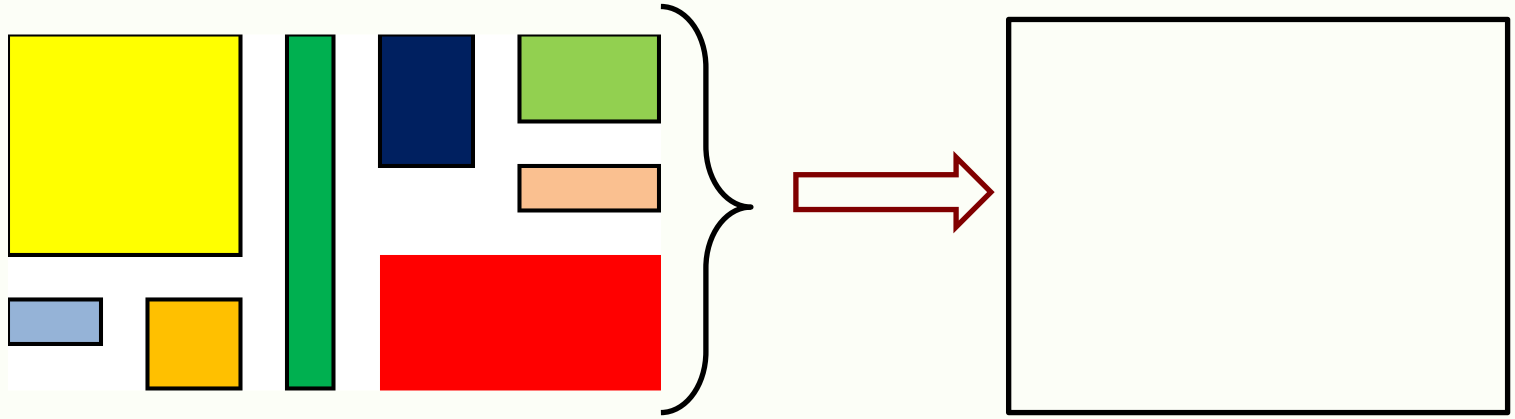
- In other cases, namely when the graph has components close to be disconnected, a decomposition strategy may pay-off, i.e. selecting an order of enumeration that decomposes a problem in smaller problems. In the example of the figure, after enumerating some of variables in the "border" the problem is decomposed into independent problems.



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

Dynamic Heuristics

- The basic principle followed by most 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*, and the rationale might be explained as follows:
- Larger rectangles are harder to place than the smaller ones, and have less possible choices. If one starts with the smaller (and easy) rectangles, they further restrict these choices, possibly making them impossible, thus inducing some avoidable (?) backtracking.

Dynamic Heuristics

- In Choco an arbitrary selection of a variable and a value may be achieved by the solver method `setSearch`, parameterized by the search discipline to be used, as in the following example, equivalent to `setSearch(inputOrderLBSearch(x))`.

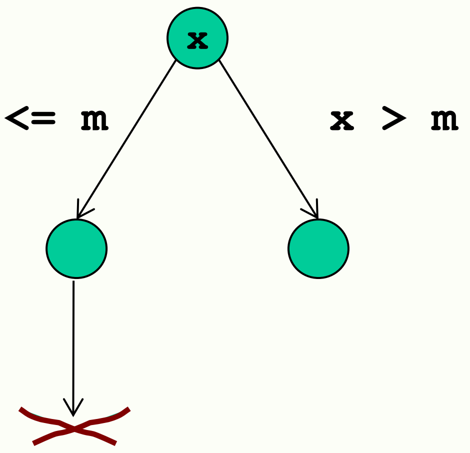
```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
    sv.setSearch(intVarSearch(
        // variable selector
            (VariableSelector<IntVar>) variables -> {
                for(IntVar v:variables){
                    if(!v.isInstantiated()){
                        return v;
                }}
                return null;
            },
        // value selector
            (IntValueSelector) var -> var.getLB(),
        // apply equality (var = val)
            DecisionOperator.int_eq,
        // variables to branch on
            X
    ));
        .....
```

- Note: this method does not seem to be working, namely the Decision operator, but should be used to enforce domain splitting and user defined heuristics.

Different Branching Strategies

- Domain splitting does not impose any specific value for the selected variable, rather it splits the domain, usually in two halves.

- This technique allows that several values are discarded in a single step, as shown (it should use `DecisionOperator.int_le`)



- This main use of this technique is in optimisation problems using a branch & bound strategy. Once a good solution has been found in one branch, we are interested in eliminating all branches that do not lead to a better solution. Domain splitting often makes such discarding more efficient.

Dynamic Heuristics

- The **first-fail** principle that dynamic variable select heuristics follow in general.
- 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 (and value);
    - **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

- **Look Present Heuristics.**

- 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.

- If there are many choices (as there are for the smaller rectangles in the example), the likelihood of being able to assign a right choice increases, and the difficulty can thus be assessed from 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 (FirstFail - ff)

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 are less choices available !
- In the limit, if variable x1 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 COMET and in SICStus Prolog).

Dynamic Heuristics – DOM (FirstFail - ff)

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

- The **DOM** heuristic (also know as **first-fail**) selects the variables to enumerate by the increasing size of their domains.
  - In case of ties, one variable is chosen randomly.

- In Choco this is achieved with the methods, that select the values for the variables in increasing or decreasing order.
  - `s.setSearch(minDomLBSearch(X));`
  - `s.setSearch(minDomUBSearch(X));`

```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Solver s = model.getSolver();
       IntVar [] x = model.intVarArray("N", 3, 1, 4);
       ...
       //s.setSearch(inputOrderLBSearch(x)) // left to right
       //s.setSearch(inputOrderUBSearch(x)) // right to left
       s.setSearch(minDomLBSearch(X));      // ff left to right
       //s.setSearch(minDomUBSearch(X));    // ff right to left
       .....
```

Dynamic Heuristics – Deg

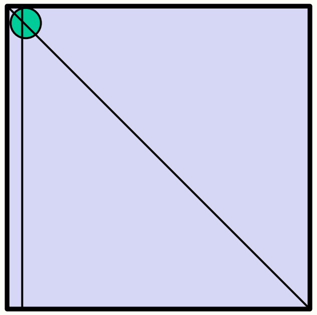
Deg Heuristics: The **number of constraints** (degree) of the variables

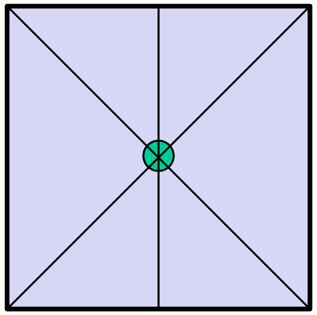
- This heuristics is basically the Maximum Degree Ordering (MDO) heuristics, but now the degree of the variables is assessed dynamically, after each variable enumeration.
- Clearly, the more constraints a variable is involved in, the more difficult it is to assign a good value to it, since it has to 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 referred to as **ffc** heuristics.
- No support seems to be given in Choco (nor in COMET).

User-Defined Heuristics

- In some cases the user has some knowledge about the variables and values to select and thus not rely on general purpose heuristics.
- For example, in the n-queens problem, a very successful heuristics selects variables and values from the "centre" of the **n×n** board, **in addition** to selecting the variable with least domain size (ff). Once selected the variable, the value selected should also be near the middle of its domain.
- The rationale is simple: by propagation (say node-consistency)
  - whereas queen in a corner only eliminates O(2n) values from the domains of the other variables
  - a queen placed in the center of the board eliminates O(3n) values from the domains of the other variables

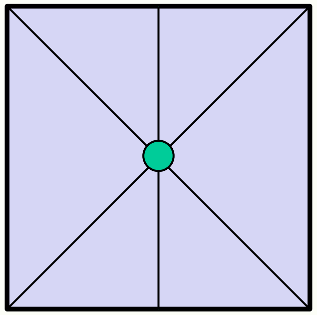


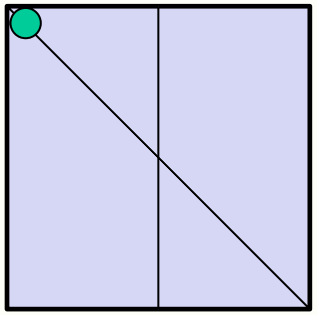


User-Defined Heuristics

- Such heuristics can be easily specified in **COMET** as shown below:

```
range R = 1..n;
var<CP>{int} q[R](s,R);
...
while(! bound(q)){
   selectMin(i in R:!q[i].bound())(x[i].getSize(),abs(i-n/2))
      selectMin(v in domainOf(q[i]) (abs(v-n/2))
         try<s> s.label(q[i],v); | s.diff(q[i],v);
}
```





- In fact, this heuristics makes it possible to solve, almost without backtracking, **very large** instances of the n-queens problem (e.g. n ≈ 5000)

Dynamic Heuristics – Look Back

Look Back Heuristics.

- These heuristics aim to learn the difficulty of the variables from past experience in the search process, and require that information about the search is collected and made accessible to inform decisions.

- Learning the difficulty of a variable, and adjusting it during search, has been tried successfully in the past in three 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 the 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.
    - Note: Different techniques may be used to approximate the search space reduction.

Dynamic Heuristics - Wdeg

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)
    - P_1: `max(c) ← max(a) + max(b)`
    - P_2: `min(a) ← min(c) - max(b)`
    - P_3: `min(b) ← min(c) - max(a)`
- Propagators may fail. For example, if propagator P1 makes **max(c) < min(c),** not only the domain of variable **c** becomes empty, but also a failure is registered for propagator P_1.
- Failures of any propagator are assigned to the corresponding constraints.

Dynamic Heuristics – Wdeg

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 in, 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 domain 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

Dom/Wdeg Heuristics

- Similarly to the Deg, also the Wdeg heuristics 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 the more difficult to enumerate
  - the **lowest** its Dom is;
  - the **highest** Wdeg is.
- Hence, the Dom/Wdeg heuristics selects for enumeration the variable with the **lowest Dom / Wdeg** ratio.

In Choco this heuristics is set, on a vector X of decision variables, with the predefined method `setSearch(domOverWDegSearch(X))` as in

```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Model m = new Model(...);
       Solver s = m.getSolver();
       s.setSearch(domOverWDegSearch(X));
.....
```

Dynamic Heuristics - Impact

- A different type of heuristics attempts to assign the degree of difficulty to a variable by the **impact** it had in previous enumerations. A often used heuristics measures the impact as the **reduction of the search space** when an enumeration is made.
- An (upper bound) of the search space is the product of the cardinality of the domains of the variables still not enumerated.

$$S = \prod_{x \in X} |Dom(x)|$$

- At every decision node, **k**, a variable **x** is selected for enumeration, and a value **v** is assigned to it. The impact $i_k(x = v)$ of such assignment is measured as the ratio :

$$i_k(x = v) = \frac{S_b(x = v) - S_a(x = v)}{S_b\,(x = v)} = 1 - \frac{S_a(x = v)}{S_b\,(x = v)}$$

- This impact varies in the range 0..1, where
  - $i_k(x = v)$ **=** 1 when there is a failure, i.e. $S_a(x = v) = 0$**;**
  - $i_k(x = v)$ **≈** 0 when there is "no impact", i.e. $S_a(x = v) \approx S_b(x = v)$

Dynamic Heuristics – Impact

- Typically, the impact of a given assignment might take into consideration previous such assignments, that may be weighted by some factor ω (≥ 1).

$$I_k(x = v) = \frac{(1 - \omega) I_{<k}(x = v) \; + \; i_k(x = v)}{\omega}$$

- The influence of the previous impacts ranges from
  - No influence, when ω = 1;
  - Increasing influence, when ω > 1;

- Now, the **variable-value impact** $I(x = v)$ (i.e. the impact of assigning value **v** to variable **x**) is given by some aggregation of the impact of all such assignments in past decision nodes. For example the average, as in

$$I(x = v) = \sum_{k:(x=v) \in k} \frac{I_k(x = v)}{|k|}$$

- Finally the **variable** impact $I(x)$ of some variable **x**, is the aggregation of all the impacts for values in its domain. For example the average, as in

$$I(x) = \sum_{v:\ v \in Dom(x)} \frac{I\ (x = v)}{|Dom(x)|}$$

Dynamic Heuristics - Impact

Impact Heuristics

- The **impact** heuristics selects both a variable and a value for the next decision node, according to the heuristic “more difficult, best promise” principle.
    - Variable Selection: with **highest** variable impact $I(x)$.
    - Value Selection: with **lowest** variable-value impact $I(x = v)$.
- In Choco this heuristics is set, on a vector X of decision variables, with the predefined method *

```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Model m = new Model(...);
       Solver s = m.getSolver();
       s.setSearch(s.setSearch(impactBasedSearch(X));
.....
```

* **Note**: It does not seem to be working

Dynamic Heuristics – Activity

- An heuristics that has some similarities to impact is **activity**.
- Instead of measuring the ”impact” of an assignment by the reduction of the search space, the impact of such assignment takes into account the number of variables that have their domain reduced after the assignment.
- Given a problem with X variables, an assignment $x = v$, defines a subset X’ of those variables with their domain reduced by propagation.
- The activity of a variable assignment at a search node k is given by the number of variables that have their domain reduced

$$a_k(x = v) = |X'|$$

- The impact, based on this activity can be measured by an aggregation of all previous activities in the search tree

$$a(x = v) = \sum_{k:(x=v) \in k} \frac{a_k(x = v)}{|k|}$$

Dynamic Heuristics – Impact

- Again the activity of a variable may consider past activities by means of a weighted sum (where ω ≥ 1).

$$A_k(x = v) = \frac{(1 - \omega)A_{<k}(x = v) \; + \; a_k(x = v)}{\omega}$$

where the importance of past activities increases with with ω, and is nil for ω =1.

- Now, the **variable-value** activity $A(x = v)$ (i.e. the impact of assigning value **v** to variable **x**) is given by some aggregation of the impact of all such assignments in past decision nodes. For example the average, as in

$$A(x = v) = \sum_{k:(x=v) \in k} \frac{A_k(x = v)}{|k|}$$

- Finally the **variable** activity $A(x)$ of some variable **x**, is the aggregation of all the impacts for values in its domain. For example their sum, as in

$$A(x) = \sum_{v:\; v \in Dom(x)} A(x = v) \frac{A(x = v)}{|Dom(x)|}$$

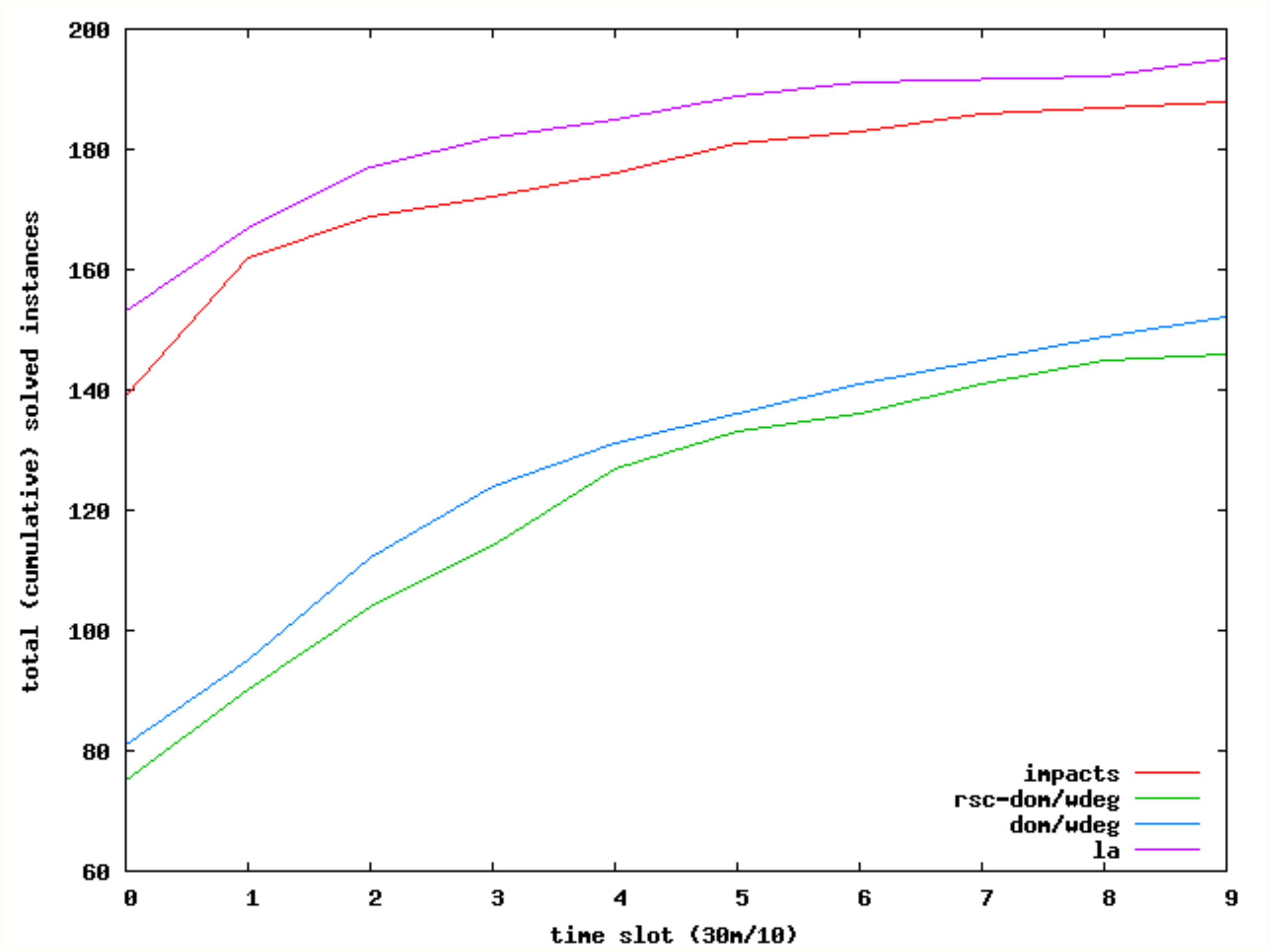
Dynamic Heuristics - Activity

- To reduce the influence of quite distant in the past assignments, the activity of a variable, after an assignment $x = v$ is updated (by a forgetting factor $\boldsymbol{\gamma}$) as :
  - $A(x) = \gamma\, A(x)$ for $x \in X$
  - $A(x) = A(x) + 1$ for $x \in X'$
- The **activity** heuristics selects both a variable and a value for the next decision node, according to the heuristic "more difficult, best promise" principle.
  - Variable Selection: with **highest** activity per value impact, A_x.
  - Value Selection: with **lowest** variable-value impact $A(x = v)$.
- In Choco this heuristics is set, on a vector X of decision variables, with the predefined method `setSearch(activityBasedSearch(X))`, as in

```
import static org.chocosolver.solver.search.strategy.Search.*;
.....
       Model m = new Model(...);
       Solver s = m.getSolver();
       s.setSearch(s.setSearch(activityBasedSearch(X));
.....
```

Dynamic Heuristics - Lookahead

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



Incomplete 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 on 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 to pure depth first search have been proposed so as to allow the search to focus on the most promising parts of the search space, at the potential cost of becoming incomplete, namely
  - **Restarts;**
  - **Limited Discrepancy**;
  - **Iterative Broadening;**
  - **Depth-Bounded Discrepancy;** and
  - **Depth - Bound (Best First)**

Incomplete Search Strategies

Restarts

- If bad values are selected in the first choices, it will be very difficult to backtrack them, since the space that needs to be searched before such backtracking is of the order of magnitude of the whole search space of the program.
- In this case, it is more convenient to start a new search. To abort the search before a solution is found the following methods of the solver class can be used:
    - `s.limitTime(limit);`
    - `s.limitFail(limit);`
    - `s.limitBacktrack(limit);`
- Of course, in the new execution the choices must not be the same (otherwise the failures would be repeated).
- Stochastic selections explicit selecting variables and values randomly, as possible with method
    - `s.setSearch(randomSearch(X,5));`

where the second argument is the seed.

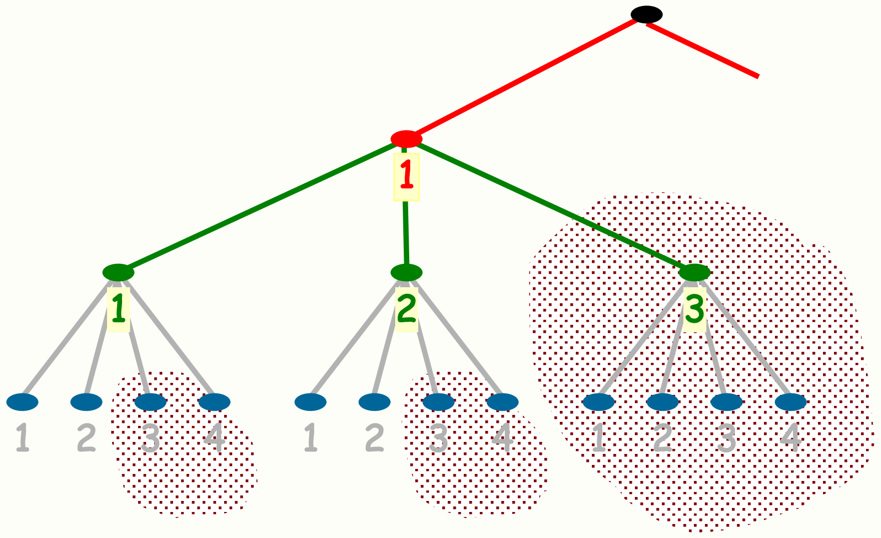
Incomplete Search Strategies

Iterative broadening,

- In iterative broadening a limit **b** is assigned to the number of children of a node that are visited , i.e. to the number of values that may be chosen for a variable.

- If this value is exceeded, the node and its successors are not explored any further.

• In the example, assuming that b=2, the search space that is pruned is shadowed.



- To guarantee completeness, if the search fails for a given b, this value is *iteratively increased*, hence the iterative broadening qualification.

- Note: Choco does not seem to offer support for this search policy.

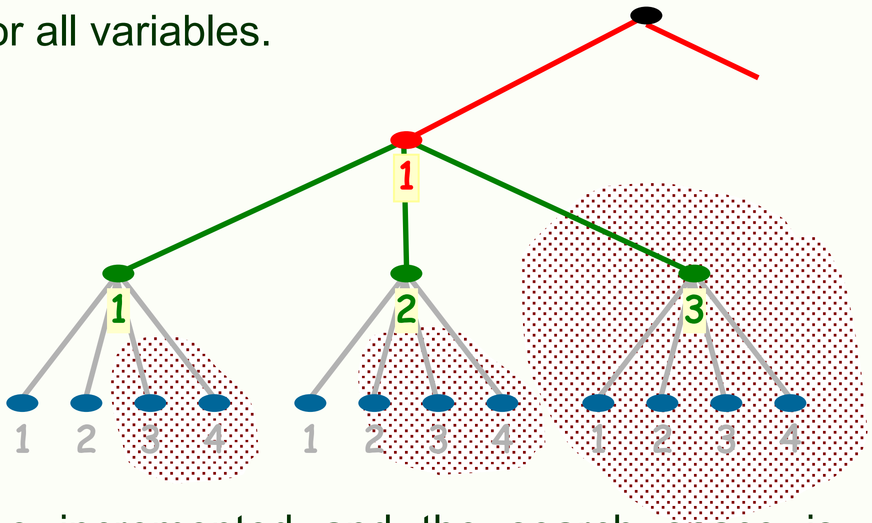
Incomplete Search Strategies

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.

- It sets a limit **d**, to the number of times that the heuristic is not taken into account.

- In the example, assuming heuristic options at the left and d=2 the search space prunned is shadowed.



- Again, if the search fails, **d** may be incremented and the search space is increasingly incremented.

- Choco (and Comet) provide tools to implement this strategy

  -cf. `org.chocosolver.solver.search.loop.move.MoveBinaryLDS`

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 intelligent 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

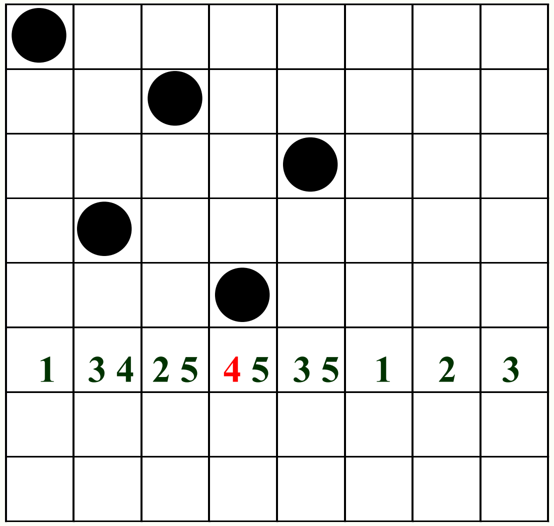
- Failing the labeling of a variable, all variables that cause the failure of each of the values are analysed, and the “highest” of the “least” variables is backtracked.

- In the example shown, analysis of why variable Q6, could not be labeled, leads to the conclusion that ...

- All possible positions are prevented by queen Q4 or some 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 Q5, would simply lead to new failures!



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 by some high-level language that is subsequently encoded in a *Boolean* language (e.g. ASP).

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

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

- 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

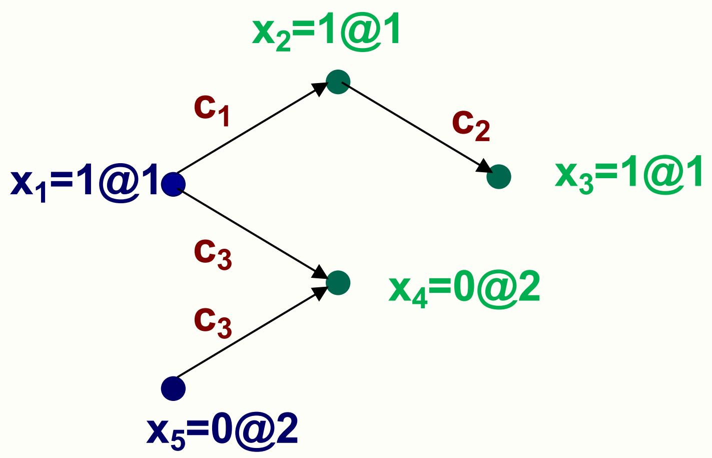
- To illustrate these techniques, we should consider the assignment of values to variables are made. Two different situations occur:
  - Some assignments are explicit decisions made by the solver, selecting the variable and the value.
  - Other assignments are due to propagation on the former.

Example: Take the following labeling on these 3 clauses

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

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

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

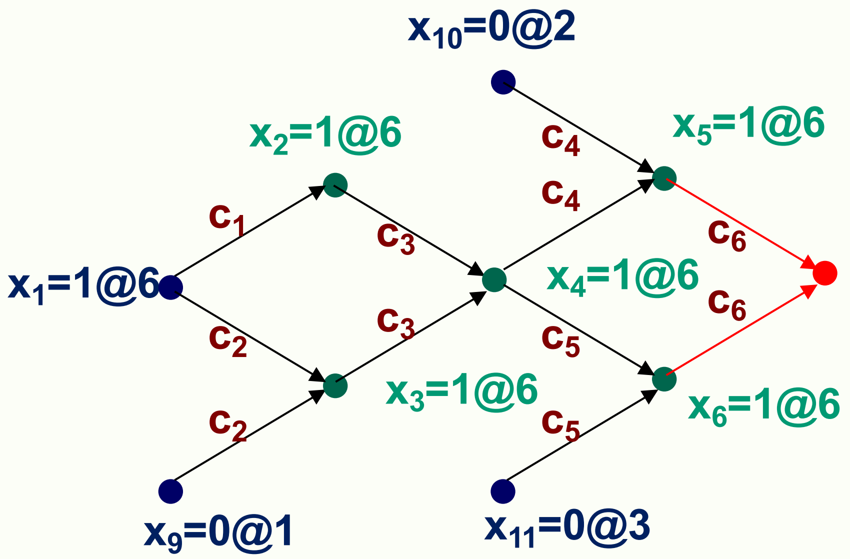


1. A first decision (at level 1) makes x1 = 1
2. Propagation enforces x2 = 1 and x3 = 1
3. A second decision (at level 2) makes x5 = 0
4. Propagation enforces x4 = 0

Intelligent Backtracking

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

```
c1: (¬x1 ∨ x2)
c2: (¬x1 ∨ x3 ∨ x9)
c3: (¬x2 ∨ ¬x3 ∨ x4)
c4: (¬x4 ∨ x5 ∨ x10)
c5: (¬x4 ∨ x6 ∨ x11)
c6: (¬x5 ∨ ¬x6)
c7: (x1 ∨ x7 ∨ ¬x12)
c8: (x1 ∨ x8)
c9: (¬x7 ∨ ¬x8 ∨ ¬x13)
...
```



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

Intelligent Backtracking

- Since, the conflict has been caused by the assignments

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

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

  $c_0: (\neg x_1 \lor x_9 \lor x_{10} \lor x_{11})$

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

In fact, one may notice that this clause could have been obtained through resolution on clauses

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

- but this no-good learning technique aims at only learn useful no-good clauses, i.e. those that were “found” **at run time.**

Intelligent Backtracking

- Not all learned clauses are useful. SAT solvers are usually further parameterised in order to determine which learned clauses are to be maintained (e.g. discard long clauses, clauses not used for a long time, ...).
- 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.
- 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. Most excitingly, SAT solvers have been used quite successfully to implement FD propagation and even global constraints (**lazy clause generation** approach).

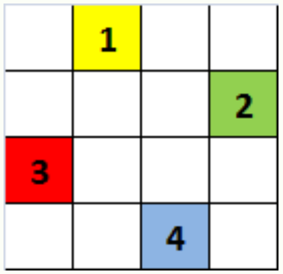
Symmetry Breaking

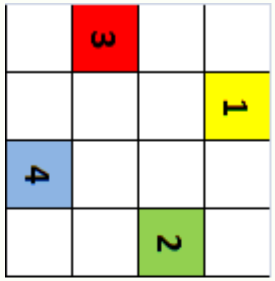
Types of Symmetries

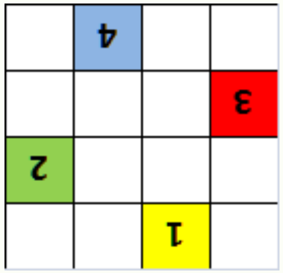
- Although, in an abstract way, symmetries are well defined as permutations, different types of symmetries have been defined in Constraint Programming, namely because the methods used to break them are not completely general.
- The informal definition below captures the essence of symmetry:
  - a symmetry is some particular transformation of the solution of a problem into another solution of the problem, or from a non-solution into a another non-solution.
- This informal definition, is very useful for direct application, since the purpose of defining symmetries is exactly to avoid **searching** for the symmetric **(partial) solutions**.
- Symmetries are a property of the problem. An example clarifies this

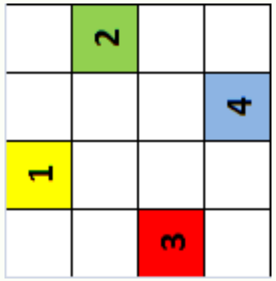
Symmetry Breaking

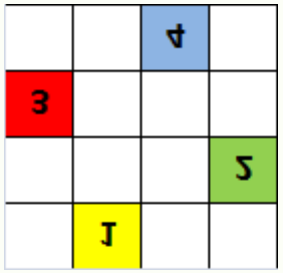
- The n-queens, as most geometrical board problems (latin squares, sudoku) presents 8 symmetries.

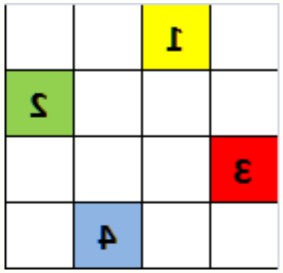

id
Identity

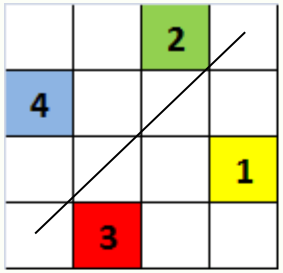

r_{90}
90° rotation

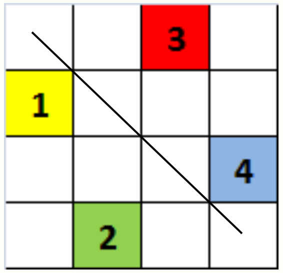

r_{180}
180° rotation


r_{270}
270° rotation


h:
horizontal
reflection


v:
vertical
reflection

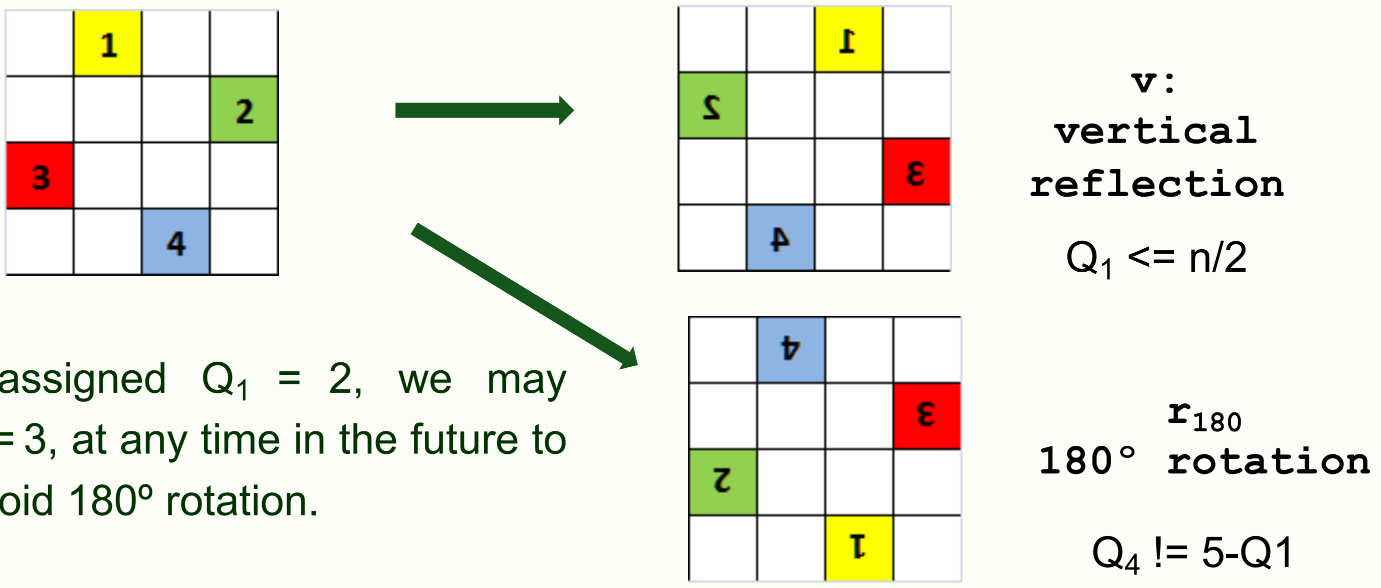

d1:
Diagonal-1
reflection


d2:
Diagonal-2
reflection

Static Symmetry Breaking

Static Symmetry Breaking

- Symmetries can be broken statically, by posting problem specific symmetry breaking constraints that avoid finding solutions that may be obtained by symmetry, not search.
- For example, in the n-queens problem we may restrict values of the first queen to be in the first "half-row", so as to avoid finding solutions that can be obtained by vertical reflection.



- Or, once assigned $Q_1 = 2$, we may prevent $Q_4 = 3$, at any time in the future to avoid to avoid 180º rotation.

Dynamic Symmetry Breaking

Dynamic Symmetry Breaking (DSB)

- Rather than posting constraints before the search starts, as done in ad hoc and other methods, 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

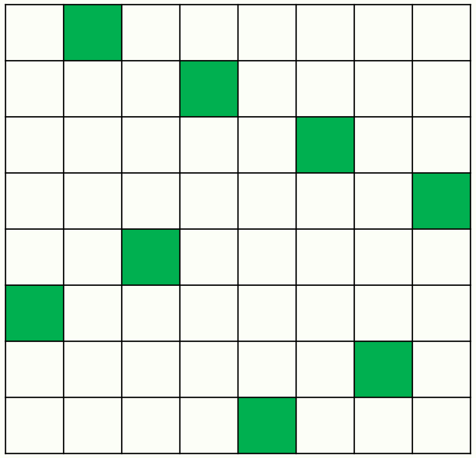
SBDS – Symmetry Breaking During Search

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

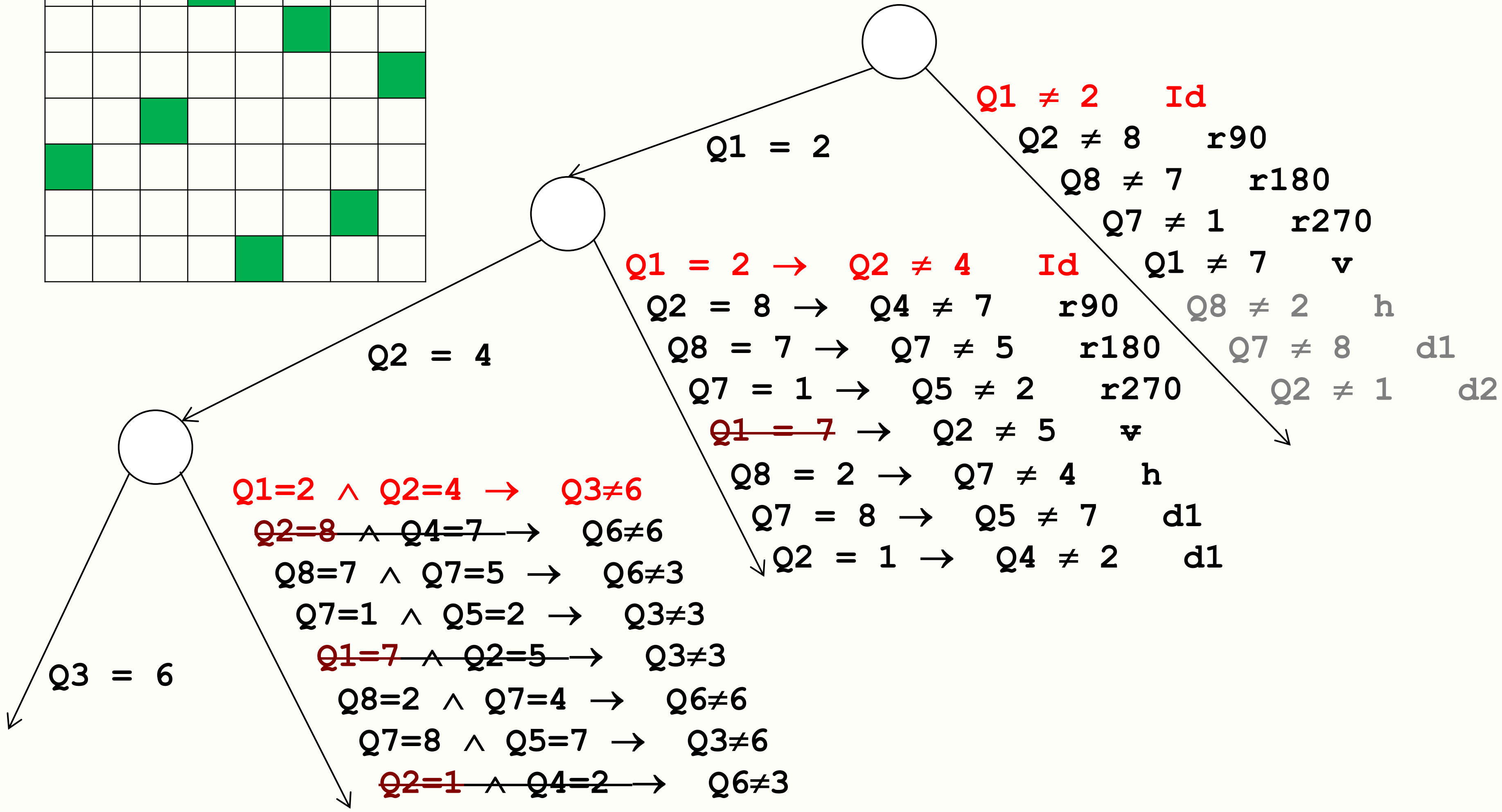
SBDD – Symmetry Breaking via Dominance Detection

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.

Symmetry Breaking During Search



8-queens



Q1 ≠ 2	Id
Q2 ≠ 8	r90
Q8 ≠ 7	r180
Q7 ≠ 1	r270
Q1 ≠ 7	v
Q8 ≠ 2	h
Q7 ≠ 8	d1
Q2 ≠ 1	d2

Q1 = 2 →	Q2 ≠ 4	Id
Q2 = 8 →	Q4 ≠ 7	r90
Q8 = 7 →	Q7 ≠ 5	r180
Q7 = 1 →	Q5 ≠ 2	r270
~~Q1 = 7~~ →	Q2 ≠ 5	v
Q8 = 2 →	Q7 ≠ 4	h
Q7 = 8 →	Q5 ≠ 7	d1
Q2 = 1 →	Q4 ≠ 2	d1

Q1=2 ∧ Q2=4 →	Q3≠6
~~Q2=8 ∧ Q4=7~~ →	Q6≠6
Q8=7 ∧ Q7=5 →	Q6≠3
Q7=1 ∧ Q5=2 →	Q3≠3
~~Q1=7 ∧ Q2=5~~ →	Q3≠3
Q8=2 ∧ Q7=4 →	Q6≠6
Q7=8 ∧ Q5=7 →	Q3≠6
~~Q2=1 ∧ Q4=2~~ →	Q6≠3

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 $Y \neq 4$ 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

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