Search and Optimisation

- Heuristic Search

- • Variable and Value selection

- • Static and Dynamic Heuristics

- • User Defined 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)

```
Solver<CP> cp();
... // declaration of variables
solve<cp> {
... // declaration of constraints
} using {
... // 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

$$x[1] \text{ in } 1..2, \; x[2] \text{ in } 1..3 \text{ and } x[3] \text{ in } 1..4.$$
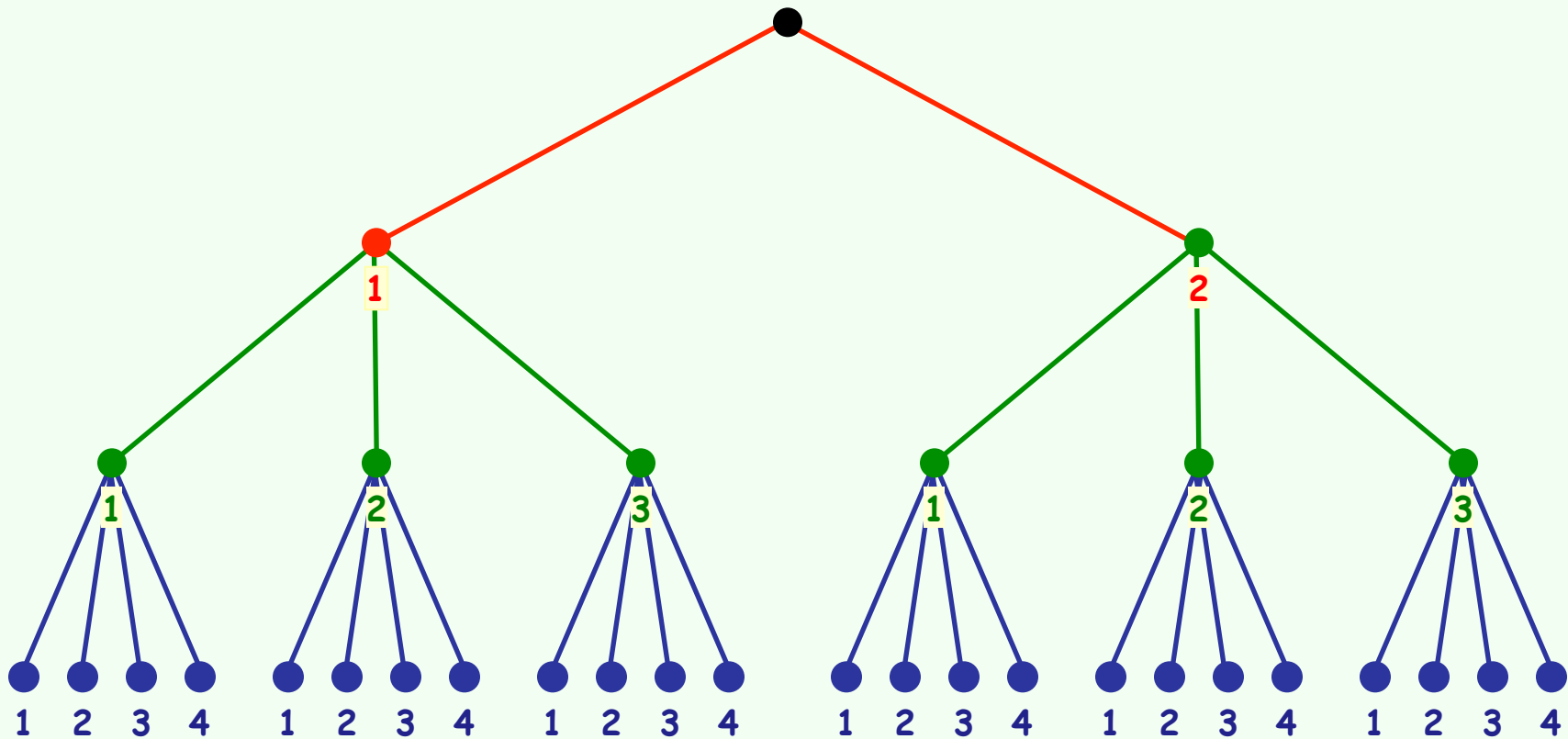
Complete Search

- A depth first search is specified in Comet with the predefined function label, that can be applied to all FD variables of the solver or to a more specific a set or array of variables (in this case all variables are in array x, so both can be used)

```
Integer i(0);
Solver<CP> cp();
var<CP>{int} x[1..3](cp,1..4);
solveall<cp>{
  cp.post(x[1] <= 2);
  cp.post(x[2] <= 3);
} using {
  label(cp);  // or label(x);
  i := i+1;
  cout << " solution " << i << ": " << x << endl;
}
```

- Note: In this example we search for **all** solutions. Integer `i` counts the solutions which are reported before closing the using part.

Complete Search

- In fact the label function performs two decisions:

  - Select a variable to label

  - Select the value used to assign to the variable

- Such decisions are made explicit by means of the following equivalent specification of labeling with the **try** command (and the function that returns a domain of a variable)

```
function set{int} domainOf(var<CP>{int} x){
  int v1 = x.getMin();
  int v2 = x.getMax();
  set{int} dom = collect(v in v1..v2: x.memberOf(v))(v);
  return dom;
}

forall(i in x.getRange())
  tryall<cp>(v in domainOf(x[i]) cp.label(x[i],v);
```

Complete Search

- Labeling all FD variables of a solver is similar as shown below. Note that the variable and value choices of function label are not arbitrary. In fact, this function selects:

  - Variables: in increasing "declaration" order
  - Values: in increasing order:

- Although these orderings are made by default in the select and try commands, they are made explicit in the following specification:

```
// Solver<CP> sv();
…
// label(sv);
forall(i in sv.getIntVariables().getRange()) by (i)
  tryall<cp>(v in domainOf(sv.getVariable(i))) by (v)
    cp.label(sv.getVariable(i),v);
```

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.

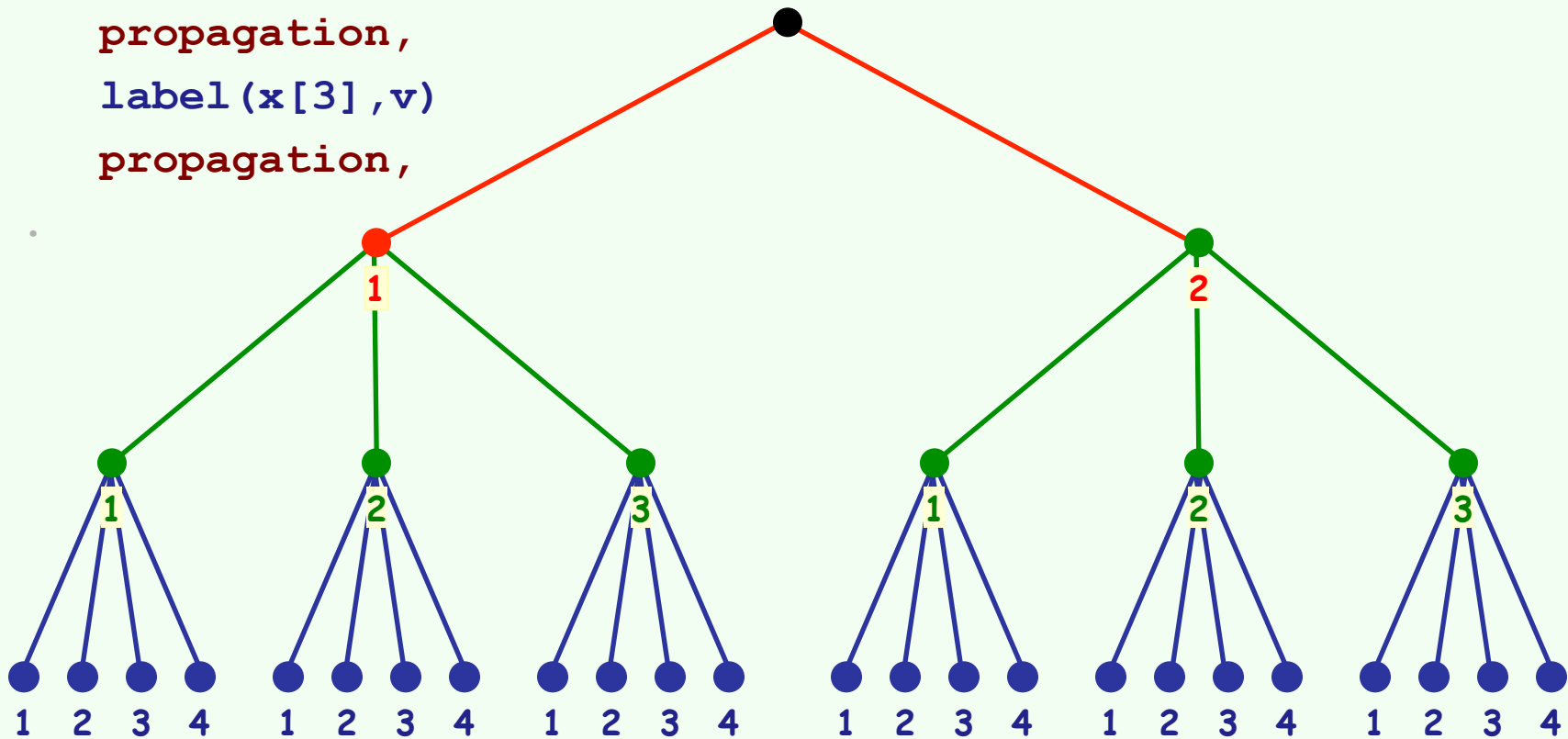# Complete Search

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

# of nodes = 32
    (2 + 6 + 24)

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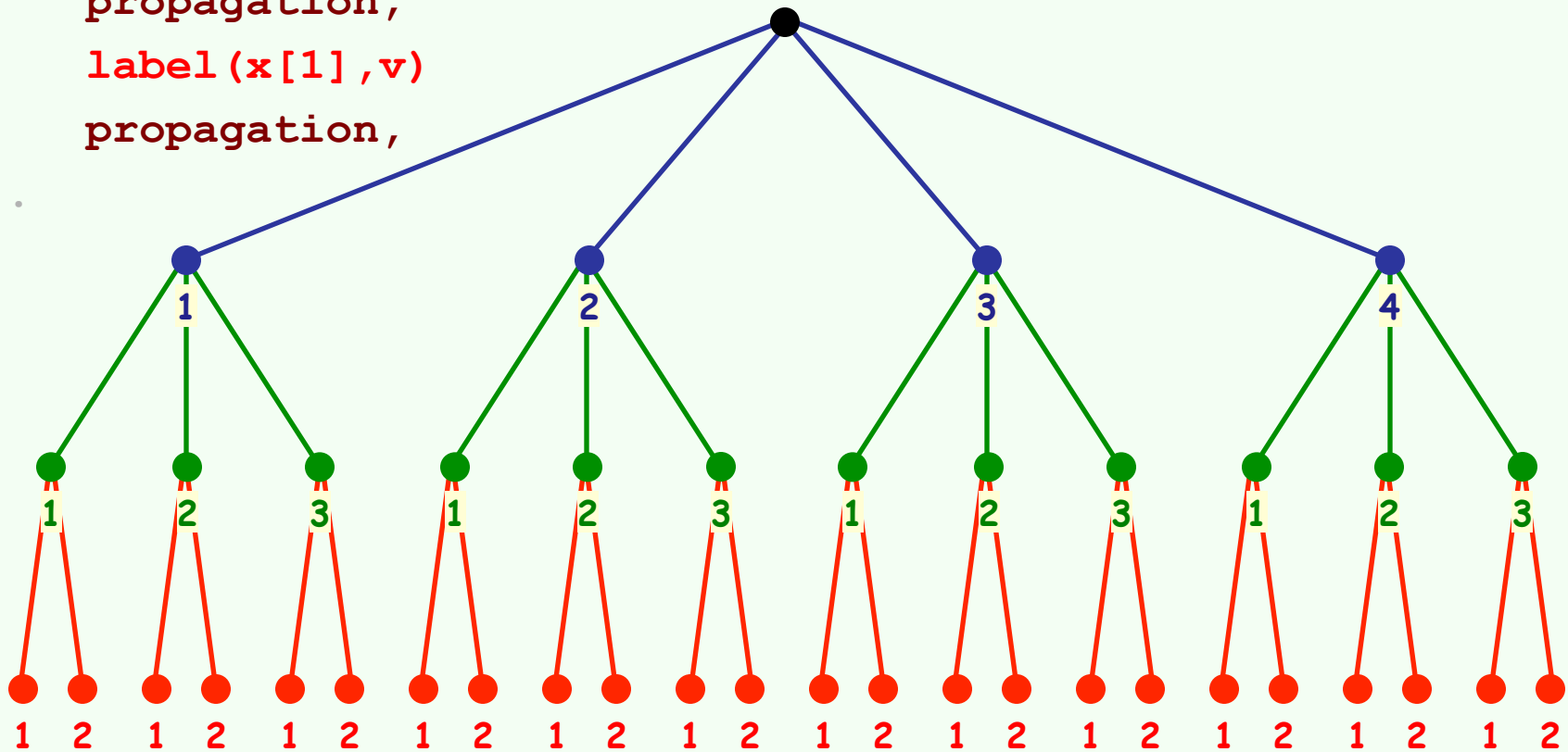
```
label(x[3],v)
propagation
label(x[2],v),
propagation,
label(x[1],v)
propagation,
```

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

Complete Search

- This is one of the reasons that explains the efficiency of the first-fail heuristic available in Comet as function labelFF, applicable to a set or array **x** of variables

```
// labelFF(x);    x is an array of FD variables
while(!bound(x)) {
  selectMin(i in x.getRange(): x[i].getSize() > 1)
        (x[i].getSize()){
    set{int} Dom = domainOf(x[i]);
    tryall<cp>(v in domainOf(x[i]) cp.label(x[i],v);
}}
```

... or to all variables of the solver **cp**:

```
// labelFF(cp);    cp is a Solver<CP>
set{int} Vars = cp.getIntVariables().getRange();
while(!bound(cp.getIntVariables())) {
  selectMin(i in Vars: cp.getVariable(i).getSize() > 1)
                      (cp.getVariable(i).getSize()){
    set{int} Dom = domainOf(cp.getVariable(i));
    tryall<cp>(v in Dom) cp.label(cp.getVariable(i),v);
}}
```

Complete Search

- More complex heuristics may be specified in Comet taking into account specific information that is known about variables and their dependencies. In the example:

  - Variables wh1 are labeled 1;

  - The corresponding variables wh2 are labeled 1 as well; If one of the latter assignments fails, it is backtracked to value 0;

  - If an assignement of wh1 to 1 fails (possibly after failing all possible assignments of the corresponding wh2 variables), it backtracks to 0;

  - Variable if3 is only enumerated (starting with 1) after all variables wh1 and wh2;

  - Variables a and b (in this order) are only enumerated after all the other variables;

```
forall(i in 0..20) try<cp> {
  cp.post(wh1[i] == 1);
  forall(j in 0..30) try<cp>
    cp.post(wh2[i,j] == 1); | cp.post(wh2[i,j] == 0);
} | cp.post(wh1[i] == 0);
try<cp> cp.post(if3 == 1); | cp.post(if3 == 0);
label(a);
label(b);
```

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.

    - **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.
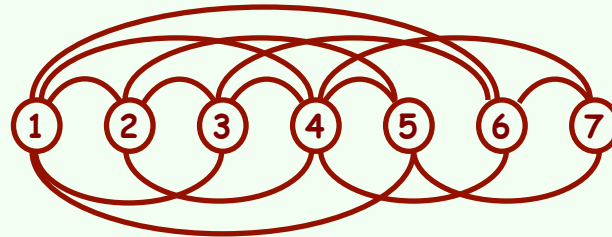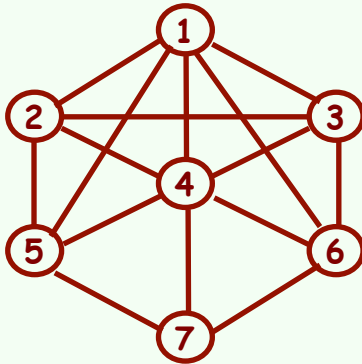
- **Width of a graph G:**

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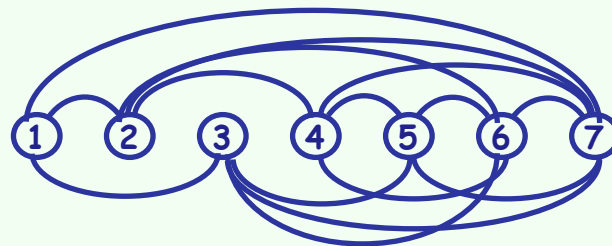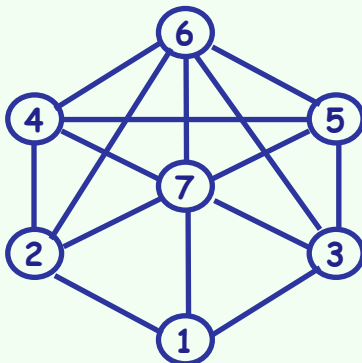
Static Heuristics

- In the graph below, we may consider various orderings of its nodes, inducing different widths.
- 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

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

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 adopt 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.
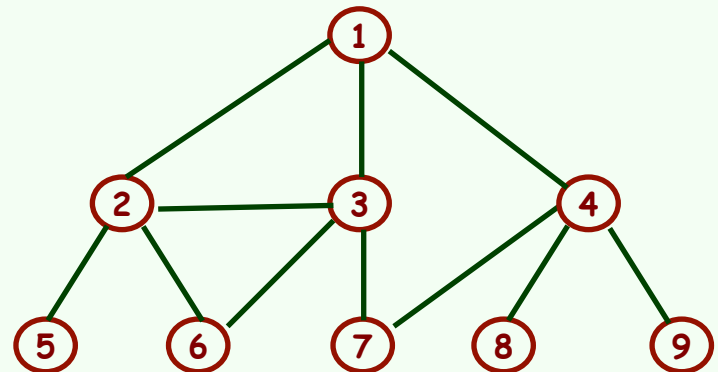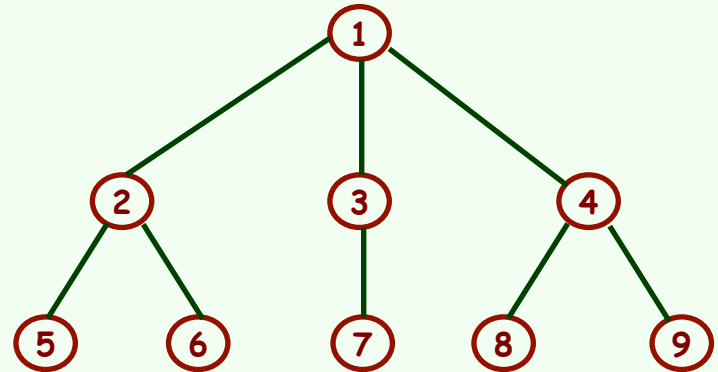
Static Heuristics

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

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

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

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

 induce different widths (4 and 3, respectively).
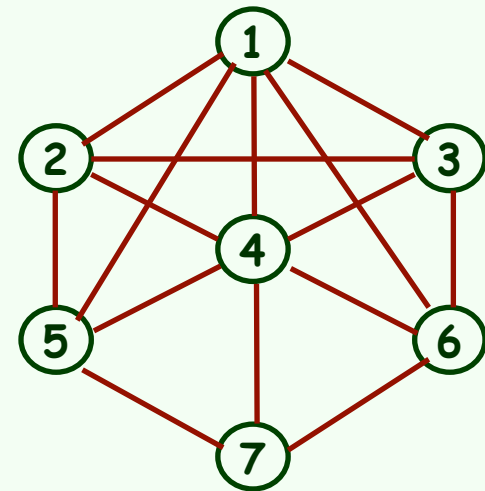
Static Heuristics

- 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.

- 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?
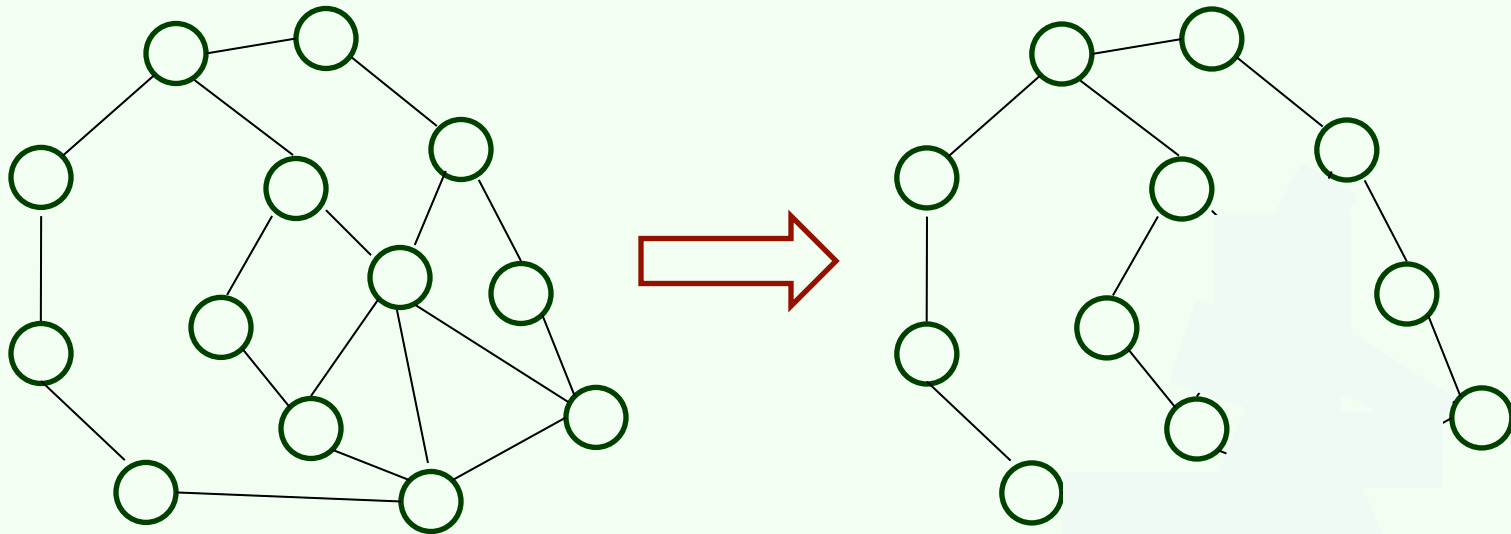
**CCS Heuristics (*Cycle Cut Set Heuristic*):**

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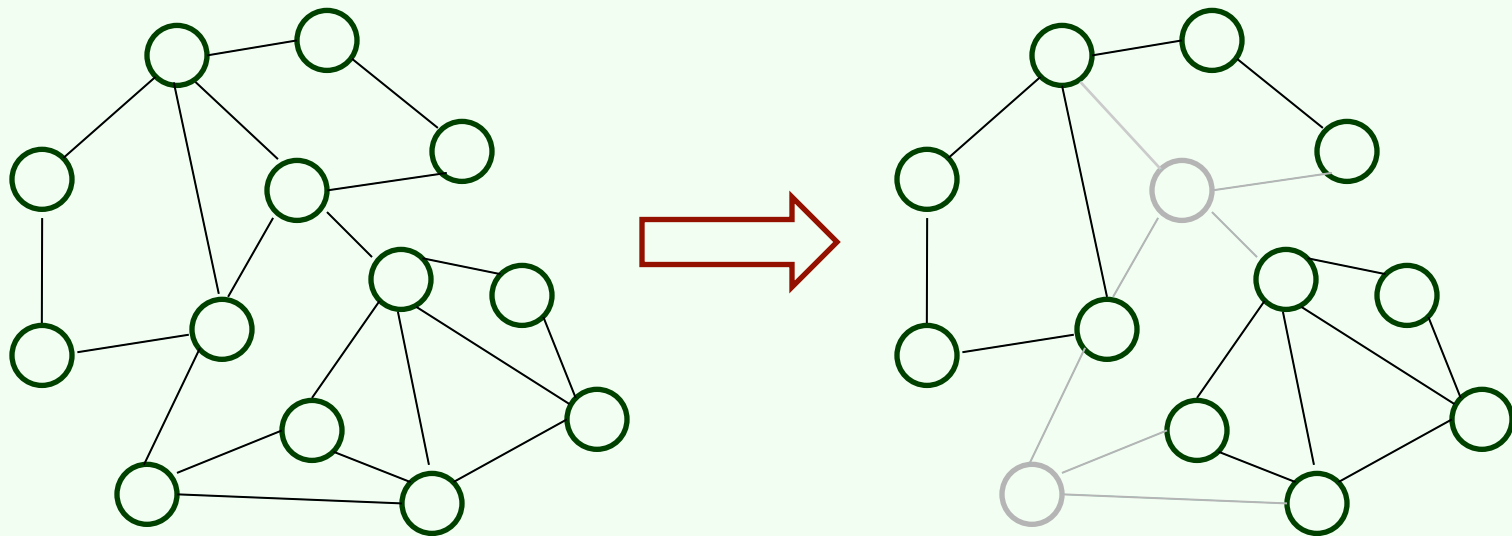
- The rationale is of course to transform a problem with worst case complexity of $O(d^n)$ into two problems with "half" the variables of complexity $O(2\,d^{n/2})$.

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).
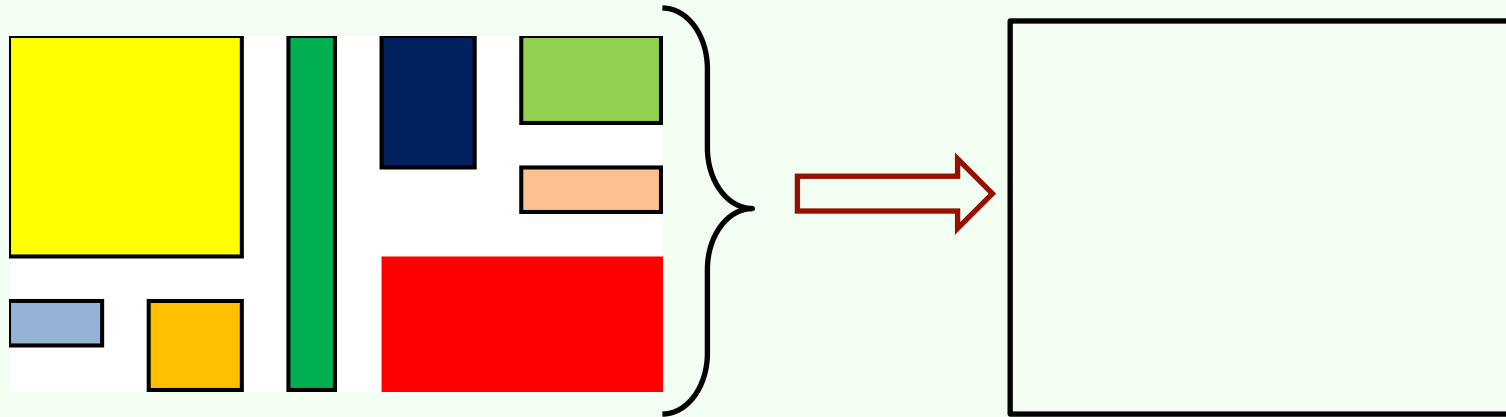
- A sensible heuristics will start by *placing the larger rectangles first*, and the rationale might be explained as follows:

- 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

- 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

- **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 assigning a wrong value 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 – Look Present

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 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 COMET and in SICStus Prolog)

Dynamic Heuristics – Look Present

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. No support seems to be given in COMET.

Dynamic Heuristics – Look Back

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 adjusting 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 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.

Dynamic Heuristics – Look Back

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)

  - 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 – Look Back

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 – Look Back

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 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.

Dynamic Heuristics – Look Back

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 approximation) of the search space is the product of the cardinality of the domains of the variables still not enumerated.

- An enumeration **e** (i.e. where some value **v** is assigned to variable **x**) has an impact **i(x,e)** measured by the relative reduction of the search space.

- Specifically, denoting by $S_a(e)$ (resp. $S_b(e)$) the size of the search space **after** (resp. **before**) the enumeration (considering only the other variables) the impact is measured as

$$i(x,e) = (S_b(e) - S_a(e)) / S_b(e) = 1 - S_a(e)/S_b(e)$$

Dynamic Heuristics – Look Back

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

$$I(x) = average_e(i(x,e))$$

- 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 with 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 (note: the contribution of variable x_5 to the search space is not considered)

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

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

Impact Heuristics

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

Dynamic Heuristics – Look Ahead

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.
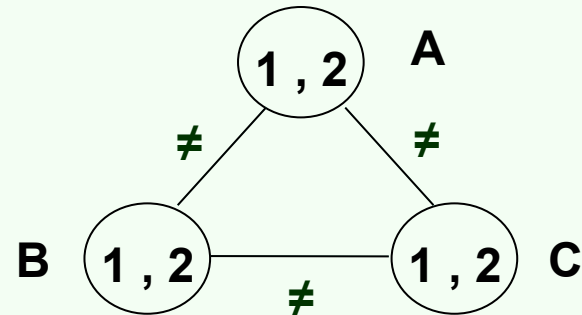
- This type of look-ahead consistency, named **singleton arc-consistency** (or **shaving**), was already previously proposed as a consistency criterion, but it was not considered efficient.

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

Dynamic Heuristics – Look Ahead

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 as before (the ratio between the reduction of the search space and the search space before the assignment).

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

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

$$I(x) = average_e(i(x,e))$$

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

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 the Dom heuristics. Several variables were quickly restricted to 2 values and we used the heuristics only on these values.

- Moreover, full arc consistency is *not* 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 (forward-checking). It does not detect possible inconsistent values as SAC does, but is much faster to get approximate values of S_b.

- In general all heuristics require some tuning, and this technique is more of an art than a science, 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
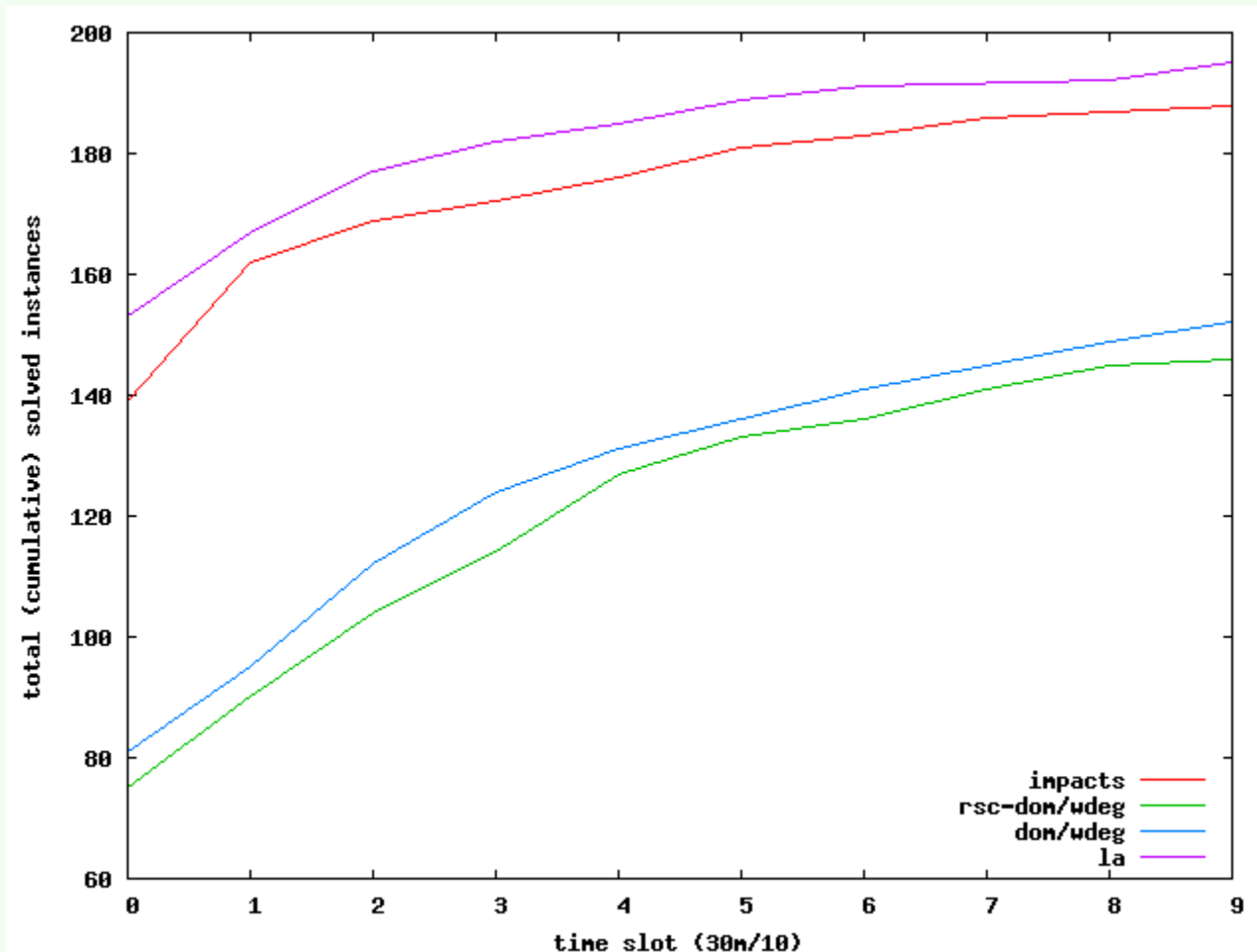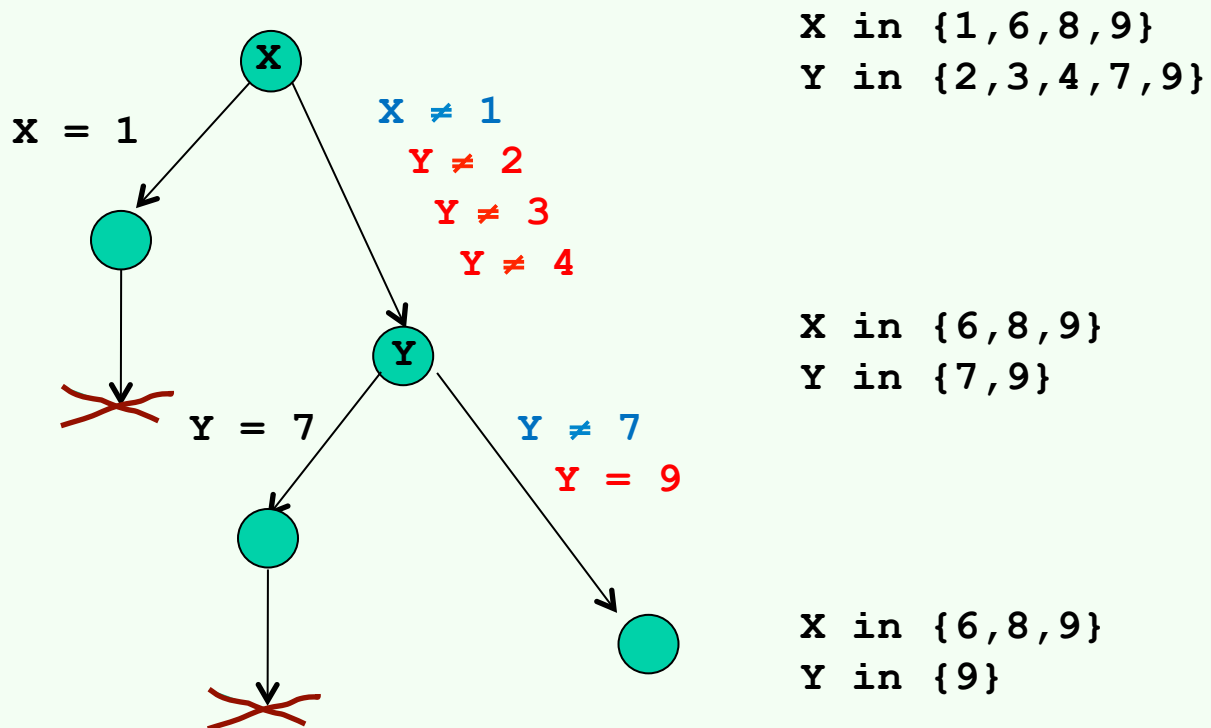
Different Branching Strategies

- 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.
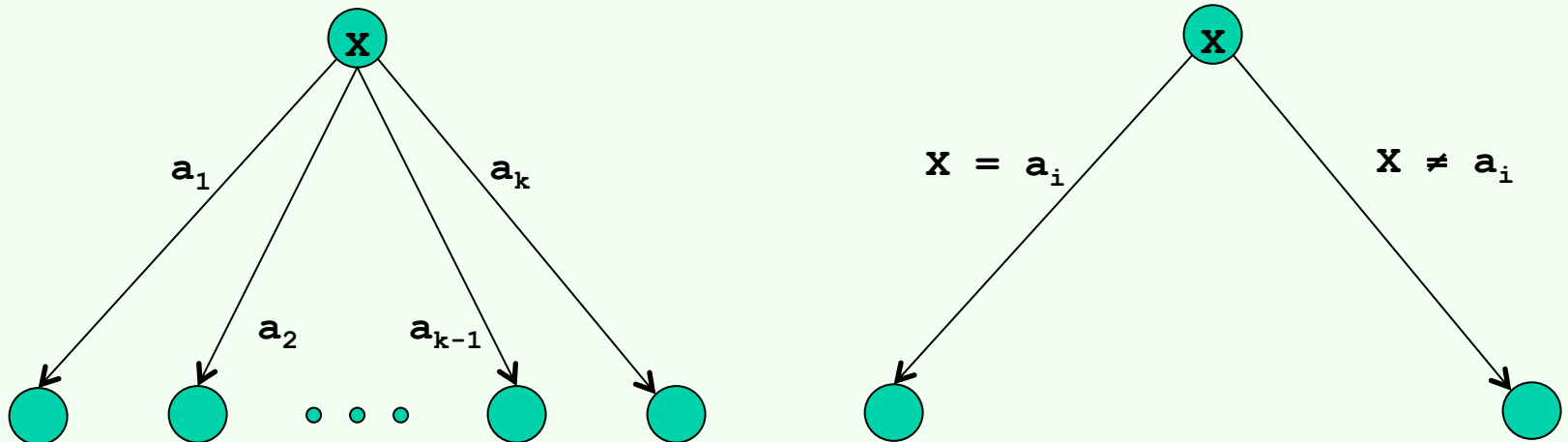
Different Branching Strategies

- 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)

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

X = 1

X ≠ 1
Y ≠ 2
Y ≠ 3
Y ≠ 4

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

Y = 7

Y ≠ 7
Y = 9

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

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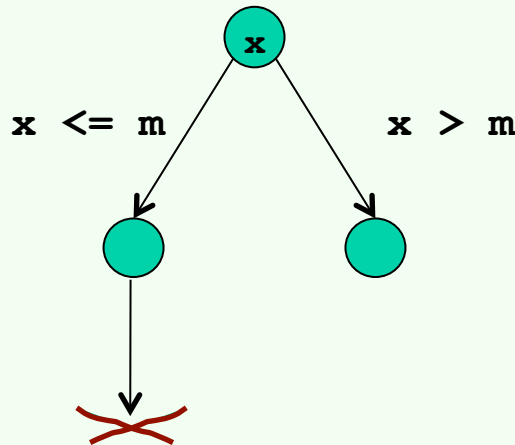
- This main use of this technique is in optimisation problems using a branch & bound strategy. Once a good solution has been found in one branch, we are interested in eliminating all branches that do not lead to a better solution. Domain splitting often makes such discarding more efficient.

Different Branching Strategies

- The changes in the codes shown before are straightforward. Instead of trying all values as done in of n-way branching,

```
while(!bound(x)) {
  selectMin(i in x.getRange(): x[i].getSize() > 1)
        (x[i].getSize()){
    set{int} Dom = domainOf(x[i]);
    tryall<cp>(v in Dom) cp.label(x[i],v);
  }
}
```

- 2-way branching replaces the try command with an alternative such as

```
try<cp>(v in Dom) {cp.label(x[i],v) | cp.diff(x[i],v)}
```

- 3-way branching, as well as domain splitting may be specified as below (although the choice of v should be improved)

```
try<cp>(v in Dom)
    {cp.lthen(x[i],v) | cp.label(x[i],v) | cp.gthen(x[i],v)}
```

```
try<cp>(v in Dom) {cp.post(x[i] <= v) | cp.post(x[i]> v) }
```

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
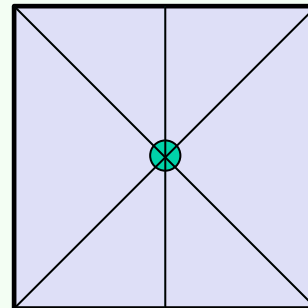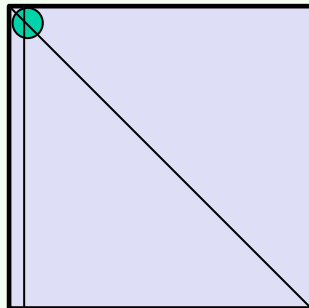
User-Defined Heuristics

- 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);
}
```
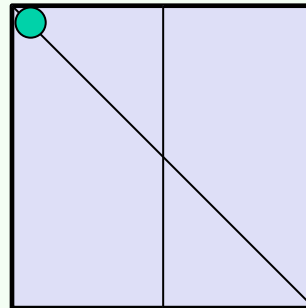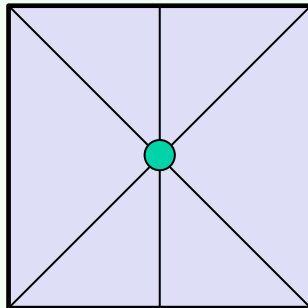

- In fact, this heuristics makes it possible to solve, almost without backtracking, **very large** instances of the n-queens problem (e.g. $n \approx 5000$)

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. Of course, in the new execution the choices must not be the same (otherwise the failures would be repeated).

- Stochastic selections could use the command **select()** rather than **selectMin**() or **SelectMax()**, which implements a random choice.

- To abort the search before a solution is found the following methods of the Solver<CP> class can be used, to specify conditions to abort search:

  - **void limitFailures(int f)** : sets **f** as the maximum number of failures

 - **void limitTime(int s)**: sets **t** as the maximum number of seconds

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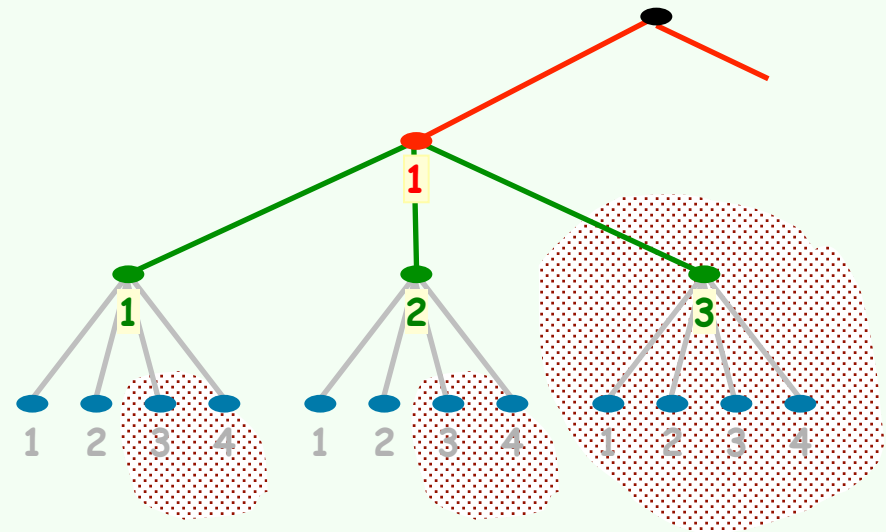
- To guarantee completeness, if the search fails for a given b, this value is *iteratively increased*, hence the iterative broadening qualification.

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.

- Comet implements this strategy with method:
    - void setSearchController(**BDSController(cp)**)
    - (cf. AbstractSearchController class, for more details)
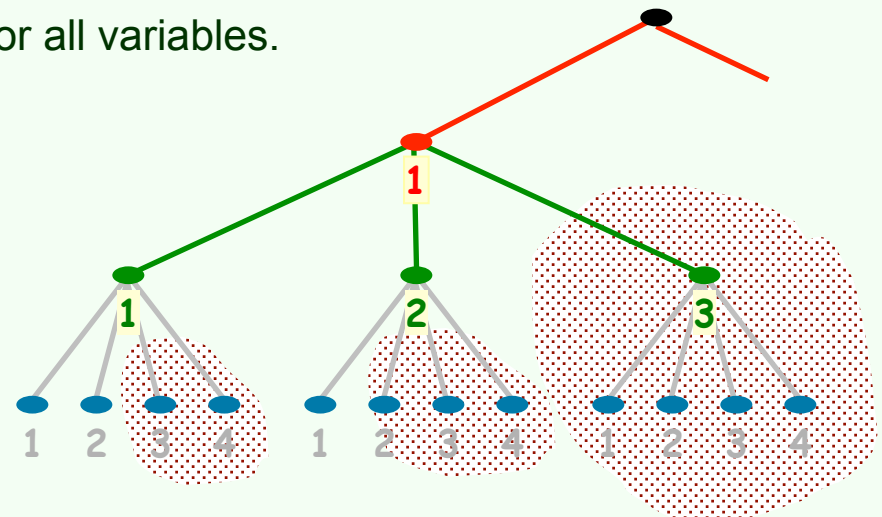
Incomplete Search Strategies

Depth-Bound Discrepancy

- This type of discrepancy search, allows any number of bad choices, but only to a limited depth, **d**. Once this depth is obtained, no backtracking is accepted. In the figure the allowed depth is 2 (i.e. 2 enumerations)

  - In the 1st iteration, no backtrack is accepted

  - The 2nd iteration backtracks to the 1st choice point (1st enumeration)

  - The 3rd / 4th iterations backtrack to the 2/3 choice point (1st enumeration)

  - No further backtracking is allowed.

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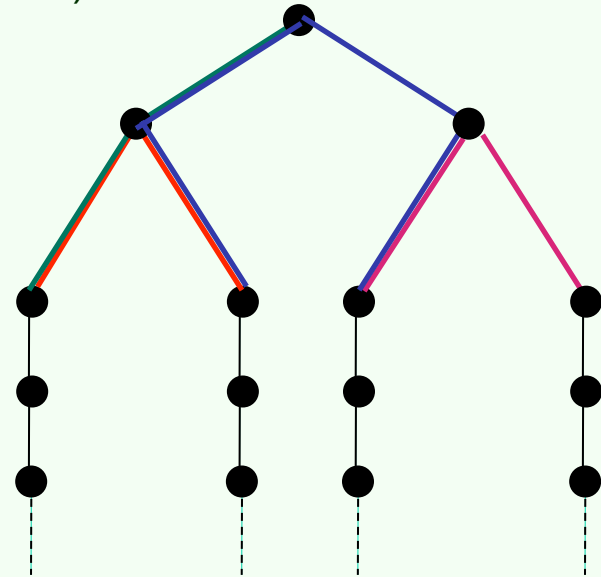
Incomplete Search Strategies

Depth - Bound (Best First)

- A different implementation of depth bounded search also takes into account that heuristics are more precise at higher depths, but adopts a best first approach.

- In the 1st phase all nodes of depth **d** are generated.

- They are sorted by their heuristic value.

- In the second phase, the search proceeds with no backtracking, starting from each of the nodes of depth d.
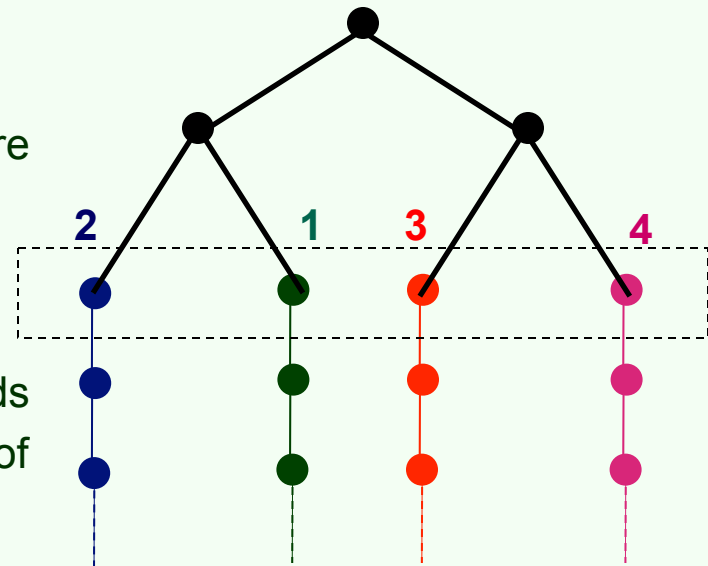
- Again, if the search fails, **d** may be incremented and the search space is increasingly incremented.

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:

  o Backjumping ;

  o Backchecking ; and

  o 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

Backchecking and Backmarking

- These techniques may be useful when the testing of constraints on different variables is very costly. The key idea is to memorise previous conflicts, in order to avoid repeating them.

  - In backchecking, only the assignments that caused conflicts are memorised.

  - In backmarking the assignments that did not cause conflicts are also memorised.

- The use of these techniques with constraint propagation is usually not very effective (with the important exception of **SAT solvers**, with nogood clause learning), since propagation anticipates 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 (e.g. ASP).

- 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,

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

- 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.

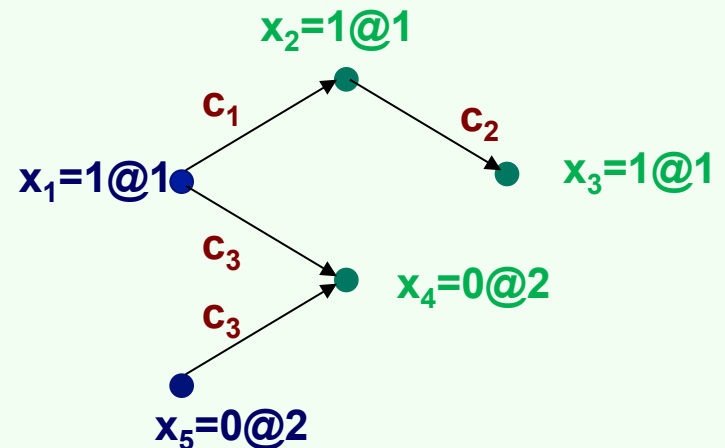 • Other assignments are due to propagation on the former.

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)
...
```
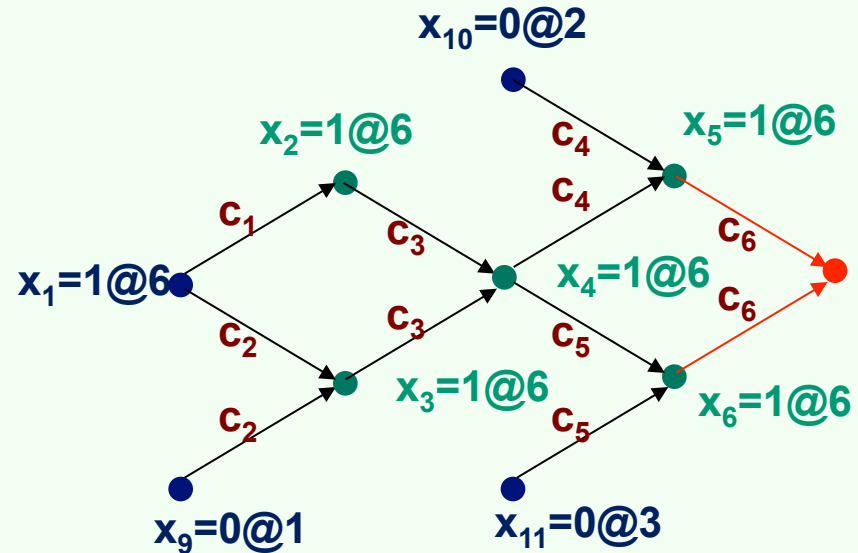

- Clearly, the conflict has been caused by assignments

  - $x_1=1$, $x_9=0$, $x_{10}=0$ 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

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

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).