JaCoP v. 4.5
Java Constraint Programming Libraray

Krzysztof Kuchcinski
Krzysztof.Kuchcinski@cs.lth.se

Department of Computer Science
Lund Institute of Technology
Sweden

October 25, 2017

Outline

JaCoP library

- constraint programming paradigm implemented in Java.
- Provides finite domain, set and floating-point constraints
  - *primitive constraints*, such as arithmetical constraints (+, *, div, mod, etc.), equality (=) and inequalities ($<, >, \leq, \geq, \neq$).
  - *logical, reified and conditional constraints*
  - *global constraints*, such as `alldifferent`, `circuit`, `cumulative` and `diff2`.
  - *set constraints*, such as =, $\bigcup$, $\bigcap$.
  - floating-point constraints
- Java API, provided as a JAR file or a class directory
- `http://www.jacop.eu`
- `https://github.com/radsz/jacop`
- `http://sourceforge.net/projects/jacop-solver/`
- Maven repository

JaCoP library (cont'd)

- compilation and execution

Commands

```
javac -cp .:path_to_JaCoP Main.java
java -cp .:path_to_JaCoP Main
```

- in an application one should specify an `import` statement

```java
import org.jacop.core.*;
import org.jacop.constraints.*;
import org.jacop.search.*;

import org.jacop.set.core.*;
import org.jacop.set.constraints.*;
import org.jacop.set.search.*;
```


MiniZinc for JaCoP

- MiniZinc front-end for JaCoP
- requires MiniZinc compiler "mzn2fzn" from `http://www.minizinc.org`

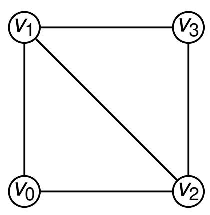
Commands

```
mzn2fzn -G jacop model.mzn

java -cp .:path_to_JaCoP org.jacop.fz.Fz2jacop
                         [options] model.fzn
```


Example

Consider coloring of a graph depicted below.



Example (cont'd)

Constraint programming encoding

for (**int** i=0; i <4; i++)
  v_i :: {1..4};
impose $v_0 \neq v_1$;
impose $v_0 \neq v_2$;
impose $v_1 \neq v_2$;
impose $v_1 \neq v_3$;
impose $v_2 \neq v_3$;
search(v, dfs(), indomain(min), delete());

Example in Java

```java
import org.jacop.core.*;
import org.jacop.constraints.*;
import org.jacop.search.*;

public class Main {
    static Main m = new Main ();

    public static void main (String[] args) {
        Store store = new Store();  // define store
        int size = 4;
        // define finite domain variables
        IntVar[] v = new IntVar[size];
        for (int i=0; i<size; i++)
            v[i] = new IntVar(store, "v"+i, 1, size);
        // define constraints
        store.impose( new XneqY(v[0], v[1]) );
        store.impose( new XneqY(v[0], v[2]) );
        store.impose( new XneqY(v[1], v[2]) );
        store.impose( new XneqY(v[1], v[3]) );
        store.impose( new XneqY(v[2], v[3]) );

        // search for a solution and print results
        Search<IntVar> search = new DepthFirstSearch<IntVar>();
        SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(store, v,
                                                                 new IndomainMin<IntVar>());
        boolean result = search.labeling(store, select);

        if ( result )
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+v[2] +", "+v[3]);
        else System.out.println("*** No");
    }
}
```


Example (cont'd)

The program produces the following output indicating that vertices v0, v1 and v3 get different colors (1, 2 and 3) while vertex v3 gets color 1.

```
Solution: v0 = 1, v1 = 2, v2 = 3, v3 = 1
```

Example in MiniZinc

```
array [0..3] of var 1..4: v;

constraint
   v[0] != v[1] /\
   v[0] != v[2] /\
   v[1] != v[2] /\
   v[1] != v[3] /\
   v[2] != v[3];
solve :: int_search(v, input_order, indomain_min, complete)
    satisfy;

output[ show(v) ];
```


Store

- the problem is specified using variables (finite domain, boolean, set and floating-point) and constraints over these variables.
- both variables and constraints are stored in the store (`Store`).
- the Store needs to be defined before defining variables and constraints. Typically it is defined using the following statement.

  ```
  Store store = new Store();
  ```

- The store has the method `toString()` redefined but printing large stores can be a very slow process. Be careful!!!

Variables

- Finite Domain Variables (FDVs) including Boolean Variables (0/1 variables) – `IntVar`, `BooleanVar`
- Set Variables – `SetVar`
- Floating-Point Variables – `FloatVar`

- Each variable in JaCoP is defined by a Java class

Finite Domain Variables

- Variable $X :: 1..100$ is specified in JaCoP as

```
IntVar x = new IntVar(store, "X", 1,100);
```

- Access of the actual domain– `dom()`.
- minimal and maximal– `min()` and `max()`, and the value– `value()`.
- the domain can contain "holes". This is specified by adding intervals to variable domain, as done below

```
IntVar x = new IntVar(store, "X", 1,100);
x.addDom(120,160);
```

  which represents $X :: 1..100 \vee 120..160$.
- default min/max values for the domain defined in IntDomain class.

Finite Domain Variables (cont'd)

- Variables without identifiers– JaCoP creates an identifier that starts with "_" and followed by a sequential number of this variable, for example "_123".

```
IntVar x = new IntVar(store, 1,100);
```

- Variables can be printed using Java primitives since the method `toString()` is redefined for them.

```
IntVar X = new IntVar(store, "X", 1,2);
X.addDom(14,16);
System.out.println(X);
```

  produces the following output.

```
X::{1..2, 14..16}
```


Finite Domains of Variables

- `IntervalDomain` – default domain

```
IntervalDomain d = new IntervalDomain(1, 10);
d.addDom(30, 40);
IntVar X = new IntVar(store, "X", d);
```

- `BoundDomain` – specifies only min..max values

```
BoundDomain d = new BoundDomain(1, 10);
IntVar X = new IntVar(store, "X", d);
```

- `SmallDenseDomain` – represents a domain as bits (limit: $max - min \leq 64$)

```
SmallDenseDomain d = new SmallDenseDomain(1,10);
IntVar X = new IntVar(store, "X", d);
```

- `BooleanVar` – 0/1 varibales

```
BooleanVar X = new BooleanVar(store, "X");
```

Set Domains of Variables

- Set is defined as an ordered list of non-repeating elements, for example

  s = $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = \{1..10\}$

- SetDomain is defined by its greatest lower bound (glb) and its least upper bound (lub); $glb \subseteq lub$

```
BoundSetDomain sd = new BoundSetDomain(1, 10);

sd = {{}..{1..10}}[card={0..10}]
```

- another BoundSetDomain

```
IntervalDomain s1 = new IntervalDomain(1, 2);
IntervalDomain s2 = new IntervalDomain(1, 10);
BoundSetDomain sd = new BoundSetDomain(s1, s2);

sd = {{1..2}..{1..10}}[card={2..10}]
```


Set Variables

- Typical definition

```
SetVar v = new SetVar(store, "v", 1, 10);

v::{{}..{1..10}}[card={0..10}]
```

- Empty set

```
SetVar v = new SetVar(store, "v", new BoundSetDomain());

v = {}
```

- A set domain

```
SetVar v = new SetVar(store, "v",
     new BoundSetDomain(new IntervalDomain(1, 2),
     new IntervalDomain(1, 10)));

v::{{1..2}..{1..10}}[card={2..10}]
```


Constraints

- Constraint – most constraints, including global constraints.
- PrimitiveConstraint – constraints that can be arguments to other constraints.
- DecomposedConstraint – constraints created automatically by solver using other two classes of constraints.

- Each constraint in JaCoP is defined by a Java class,
  - for example equality of two variables is defined by XeqY class.

Constraints

- Constraints and primitive constraints are imposed using `impose` method of store as defined below.

```
store.impose( new XeqY(x1,x2));
```

or

```
Constraint c = new XeqY(x1,x2);
c.impose(store);
```

Both methods are totally equivalent.
- The methods `impose(constraint)` and `constraint.impose(store)` create all data structures in the store which are needed for the solver.
- Decomposed constraints are imposed using `imposeDecomposition`

```
store.imposeDecomposition(new Stretch(...))
```


Constraints (cont'd)

- `impose` methods do not make consistency checking that may determine inconsistenty of the store.
- If checking consistency is needed, the method `imposeWithConsistency(constraint)` should be used instead. This method throws FailException() if the store is inconsistent.
- the similar can be achieved by calling the procedure `store.consistency()` explicitly (returns `false` if the store is inconsistent and `true` if inconsistency is not determined).

Constraints (cont'd)

- Constraints can have as an argument a primitive constraint.
- For example, *reified constraints* of the form $X = Y \Leftrightarrow B$ can be defined in JaCoP in the following way.

```
IntVar X = new IntVar(store, "X", 1, 100);
IntVar Y = new IntVar(store, "Y", 1, 100);
IntVar B = new IntVar(store, "B", 0, 1);
store.impose( new Reified( new XeqY(X, Y), B) );
```

Constraints (cont'd)

- disjunctive constraints can be imposed in a similar way.
- For example, the disjunction of three constraints can be defined as follows.

```
Constraint[] c = {c1, c2, c3};
store.impose( new Or(c) );
```

or

```
ArrayList<Constraint> c = new ArrayList<Constraint>();
c.add(c1); c.add(c2); c.add(c3);
store.impose( new Or(c) );
```


Search for solutions

- When all the variables and constraints are defined a search for a solution can be started.
- JaCoP offers a number of search methods
  - search for a single solution,
  - find all solutions, and
  - find a solution which minimizes a given cost function.
- This is achieved by using depth-first-search together with consistency checking.

Can only find minimum in JaCop (Minzinc can find max)
- If you want to find max, you need to negate
- eg.

IntVar cost = …
IntVar negCost = …
XplusYeq(cost, negCost, 0) to get the max cost

Search for solutions (cont'd)

- Consistency checking is achieved by using the following method.

```
boolean result = store.consistency();
```

- When the procedure returns false the `store` is inconsistent and no solution can be found.
- The result `true` indicates that inconsistency cannot be found. Since the solver is not complete it does not mean that the store is really consistent.

Search for solutions (cont'd)

To find a single solution the method `DepthFirstSearch` can be used.

```
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select =
                             new SimpleSelect<IntVar>(
                                    var,
                                    varSelect,
                                    tieBreakerVarSelect,
                                    indomain);
boolean result = label.labeling(store, select);
```


Search for solutions (cont'd)

The method `SimpleSelect` requires the following information:

- *var* is a vector of variables,
- *varSelect* a comparator method for selecting a variable, and
- *tieBreakerVarSelect* is a tie breaking comparator method. The tie breaking method is used when the *varSelect* method cannot decide ordering of two variables.
- *indomain* selects a value that will be assigned to a selected variable.

Search for solutions (cont'd)

- Example 1

```
Search<IntVar> label = new DepthFirstSearch<IntVar>();
IntVar[] var = {v1, v2, v3, v4};
SelectChoicePoint<IntVar> select =
                        new SimpleSelect<IntVar>(
                          var,
                          null, // input order
                          new IndomainMin<IntVar>());
boolean result = label.labeling(store, select);
```

Search for solutions (cont'd)

- Example 2

```
Search<IntVar> label = new DepthFirstSearch<IntVar>();
IntVar[] var = {v1, v2, v3, v4};
SelectChoicePoint<IntVar> select =
                        new SimpleSelect<IntVar>(
                          var,
                          new SmallestDomain<IntVar>(),
                          new IndomainMin<IntVar>());
boolean result = label.labeling(store, select);
```


Search for solutions (cont'd)

In some situations it is better to group variables and assign the values to them within a group. JaCoP supports this by another method.

```
IntVar[][] vars;
. . .
SelectChoicePoint<IntVar> select =
              new SimpleMatrixSelect<IntVar>(
                    vars,
                    new SmallestMin<IntVar>(),
                    new MostConstrainedStatic<IntVar>(),
                    new IndomainMin<IntVar>(),
                    0);
```


Set Search

```
import org.jacop.search.*;
import org.jacop.set.core.*;
import org.jacop.set.constraints.*:
import org.jacop.set.search.*;

SetVar[] vars;
. . .
Search<SetVar> label = new DepthFirstSearch<SetVar>();

SelectChoicePoint<SetVar> select =
                  new SimpleSelect<SetVar>(
                         vars,
                         new MinGlbCard<SetVar>(),
                         new IndomainSetMin<SetVar>());

Result = label.labeling(store, select);
```

Optimization

- Optimization requires definition of a cost function and use of minimization methods (maximization can be achieved by minimizing `-cost`);.
- The cost function is defined by a variable which by the assigned constraints gets a correct cost value.
- A typical optimization for defined constraints and a cost FDV is specified below.

```
IntVar cost;
 ...
boolean result = label.labeling(store, select, cost);
```


Optimization (cont'd)

- Minimization method can have additional parameters.
- The time-out parameter can be specified. The search is interrupted after the specified number of seconds if the search does not finish earlier.

```
// 10 seconds time-out
label.setTimeOut(10);
```


Search useful hints

print information on search

```
label.setPrintInfo(true);
```

print out intermediate results

```
label.setSolutionListener(new PrintOutListener<IntVar>());
```

Alldifferent, Alldiff and Alldistinct constraints

- The alldifferent constraint assures that all FDVs on a given list have different values assigned.
- Alldifferent constraint uses a simple consistency technique which removes a value which is assigned to a given FDV from the domains of the other FDVs.
- Alldiff uses bounds consistency,

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar[] v = {a, b, c};
Constraint ctr = new Alldifferent(v);
store.impose(ctr);
```


Alldifferent, Alldiff and Alldistinct constraints

```
IntVar a = new IntVar(store, "a", 2, 3);
IntVar b = new IntVar(store, "b", 2, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar[] v = {a, b, c};

store.impose( new Alldifferent(v) );

a :: {2..3}, b :: {2..3}, c :: {1..3}
```

and

```
store.impose( new Alldiff(v) );

a :: {2..3}, b :: {2..3}, c = 1
```

`Alldistinct` constraint is complete (complexity $O(n^{\frac{5}{2}})$)

Circuit constraint

- The circuit constraint tries to enforce that FDVs which represent a directed graph will create a Hamiltonian circuit.
  - the graph is represented by the FDV domains
  - nodes of the graph are numbered from 1 to N.
  - each position in the list defines a node number.
  - each FDV domain represents a direct successors of this node.
- For example, if FDV x at position 2 in the list has domain 1, 3, 4 then nodes 1, 3 and 4 are successors of node x. Finally, if the i'th FDV of the list has value j then there is an arc from i to j.

Circuit constraint – example

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar[] v = {a, b, c};
Constraint ctr = new Circuit(v);
store.impose(ctr);
```

can find a Hamiltonian circuit [2, 3, 1], meaning that node 1 is connected to 2, 2 to 3 and finally, 3 to 1.

Subcircuit constraint

- Same principle as Circuit but not all nodes must be part of the circuit.
- Nodes that are not part of the circuit point to themself.

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar[] v = {a, b, c};
store.impose( new Subcircuit(v));
```

Possible solution:

```
a = 2, b = 1, c = 3
```


Element constraint

`Element`(*I*, *List*, *V*) enforces a finite relation between *I* and *V*, $V = List[I]$. The vector of values, *List*, defines this finite relation. For example, the constraint

```
int[] el = {3, 44, 10};
Constraint ctr = new Element(I, el, V) ;
store.impose(ctr);
```

or

```
int[] el = {3, 44, 10};
Constraint ctr = Element.choose(I, el, V) ;
store.impose(ctr);
```

Element constraint, cont'd

- imposes the relation on the index variable $I :: \{1..3\}$, and the value variable $V :: \{3, 10, 44\}$.
- any change of one variable propagates to another variable. Imposing the constraint $V < 44$ results in $I :: \{1, 3\}$.
- Used, for example,
  - to define discrete cost functions of one variable or
  - a relation between task delay and its implementation resources.

Cumulative constraint

- expresses the fact that at any time instant the total use of these resources for the tasks does not exceed a given limit.
- It has four parameters:
  - a list of tasks' starts O_i,
  - a list of tasks' durations D_i,
  - a list of amount of resources AR_i required by each task, and
  - the upper limit of the amount of available resources *Limit*.

```
IntVar[] o = {O1, ..., On};
IntVar[] d = {D1, ..., Dn};
IntVar[] r = {AR1, ..., ARn};
IntVar Limit = new IntVar(store, "limit", 0, 10);
Constraint ctr = Cumulative(o, d, r, Limit)
```

`org.jacop.constraints.Cumulative` $O(n^2)$,
`org.jacop.constraints.cumulative.Cumulative` $O(k \cdot n \cdot logn)$,
`org.jacop.constraints.cumulative.CumulativeUnary` $O(n \cdot logn)$.

Cumulative constraint (cont'd)

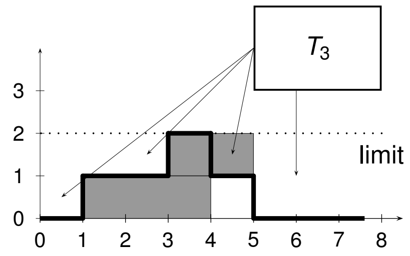
Formally, it enforces the following constraint:

$$\forall t \in [\min_{1 \le i \le n}(O_i), \max_{1 \le i \le n}(O_i + D_i)] : \sum_{k: O_k \le t \le O_k + D_k} AR_k \le Limit$$

$$\exists t \in [\min_{1 \le i \le n}(O_i), \max_{1 \le i \le n}(O_i + D_i)] : \sum_{k: O_k \le t \le O_k + D_k} AR_k = Limit$$

Cumulative constraint (cont'd)

cumulative([T_1, T_2, T_3],[D_1, D_2, D_3],[1,1,2],2)
where
$T_1 :: \{0..1\}$, $D_1 :: \{4..5\}$, $T_2 :: \{1..3\}$, $D_2 :: \{4..7\}$,
$T_3 :: \{0..10\}$, $D_3 :: \{3..4\}$
After consistency checking $T_3 :: \{5..10\}$



Diff2/Diffn constraint

- takes as an argument a list of 2-dimensional rectangles and assures that for each pair i, j ($i \neq j$) of such rectangles, there exists at least one dimension k where i is after j or j is after i, i.e., the rectangles do not overlap.
- The rectangle is defined by a 4-tuple [O_1, O_2, L_1, L_2], where O_i and L_i are respectively called the origin and the length in i-th dimension.

```
IntVar[][] rectangles = {{O11, O12, L11, L12}, ...,
                        {On1, On2, Ln1, Ln2}};
Constraint ctr = new Diff2(store, rectangles)
```

- available as `org.jacop.constraints.Diff2` an `org.jacop.constraints.diffn.Diffn`.

Diff2 constraint (cont'd)

- `Diff2` constraint can be used to express requirements for packing and placement problems, and
- can define constraints for scheduling and resource assignment.

Min and Max constraints

These constraints enforce that a given FDV is minimal or maximal of all variables present on a defined list of FDVs.
For example, a constraint

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar min = new IntVar(store, "min", 1, 3);
IntVar[] v = {a, b, c};
Constraint ctr = new Min(v, min);
store.impose(ctr);
```


SumInt constraint

- sum of elements of FDVs' vector is equal to a given FDV `sum`, that is $x_1 + x_2 + \cdots + x_n \Re\ sum$ where $\Re \in \{<, \leq, >, \geq, =, \neq\}$

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar sum = new IntVar(store, "sum", 1, 10);
IntVar[] v = {a, b, c};
Constraint ctr = new SumInt(store, v, "==", sum);
store.impose(ctr);
```

- There exists also SumBool constraint.

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar sum = new IntVar(store, "sum", 1, 10);
IntVar[] v = {a, b, c, sum};
Constraint ctr = new LinearInt(store, v,
          new int[] {1, 1, 1, -1}, "==", 0);
```


LinearInt constraints

- *primitive* constraint that defines relation $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n \Re\ sum$, where $\Re \in \{<, \leq, >, \geq, =, \neq\}$ and *sum* is constant.

Example:

```
IntVar a = new IntVar(store, "a", 1, 3);
IntVar b = new IntVar(store, "b", 1, 3);
IntVar c = new IntVar(store, "c", 1, 3);
IntVar[] v = {a, b, c};
PrimitiveConstraint ctr = new LinearInt(store,
           v, new int[] {1, -2, 1}, ">", 0);
BooleanVar b = new BooleanVar(store, "b");
store.impose( new Reified(ctr, b) );
```

More global constraints

- Table, ExtensionalSupport and ExtensionalConflict
- Assignment (inverse)
- Count
- Values (nvalue)
- Global cardinality (GCC)
- Among and AmongVar
- Regular
- Knapsack
- Geost
- NetworkFlow
- Binpacking
- LexOrder
- Decomposed constraints
  - Sequence
  - Stretch
  - Soft-Alldifferent
  - Soft-GCC

Depth First Search

- a solution satisfying all constraints– a depth first search (DFS) algorithm.
- DFS organizes the search space as a search tree.
- In every node a value is assigned to a variable and a decision to extended (consistent) or to cut (not consistent) the search is made.
- The search is cut if the assignment to the selected variable produces inconsistent model.
- An assignment of a value to a domain variable triggers the constraint propagation.

Combining searches– sequence of searches

```
Search<IntVar> slave = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> selectSlave =
          new SimpleSelect<IntVar>(vars2,
                                  new SmallestMin<IntVar>(),
                                  new SmallestDomain<IntVar>(),
                                  new IndomainMin<IntVar>());
slave.setSelectChoicePoint(selectSlave);

Search<IntVar> master = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> selectMaster =
          new SimpleSelect<IntVar>(vars1,
                                  new SmallestMin<IntVar>(),
                                  new SmallestDomain<IntVar>(),
                                  new IndomainMin<IntVar>());
master.addChildSearch(slave);

boolean result = master.labeling(store, selectMaster);
```

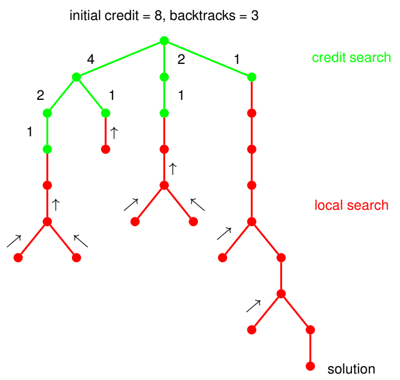
Credit search

- Credit search combines credit based exhaustive search at the beginning of the tree with local search in the rest of the tree.
- The search is controlled by three parameters:
  - number of credits,
  - credit distribution, and
  - number of backtracks during local search.
- Since we control the search it is possible to partially explore the whole tree and avoid situations when the search is stuck at one part of the tree which is a common problem of B&B algorithm when a depth first search strategy is used.

Credit search (cont'd)

Parameters

- credit = 8
- credit distribution = $\frac{1}{2}$
- number of backtracks = 3



Credit search (cont'd)

An example of the command which produces the search tree depicted in the previous slide is as follows.

```
SelectChoicePoint<IntVar> select = ...
int credits=8, backtracks=3, maxDepth=1000;
CreditCalculator<IntVar> credit =
                         new CreditCalculator<IntVar>(
                                         credits,
                                         backtracks,
                                         maxDepth);
Search<IntVar> search = new DepthFirstSearch<IntVar>();
search.setConsistencyListener(credit);
search.setExitChildListener(credit);
search.setTimeOutListener(credit);

boolean result = search.labeling(store, select);
```

Limited discrepancy search (LDS)

- it basically allows only a number of different decisions along a search path, called discrepancies.
- If the number of discrepancies is exhausted backtracking is initiated.
- The number of discrepancies- parameter for LDS.

```
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select =
                       new SimpleSelect<IntVar>(var,
                           new SmallestDomain<IntVar>(),
                           new IndomainMiddle<IntVar>());
LDS<IntVar> lds = new LDS<IntVar>(2);
label.getExitChildListener().setChildrenListeners(lds);

   boolean result = label.labeling(store, select);
```