# TP n°1: JaCoP

JaCoP is a Java constraint solver implementing various finite domain constraints (arithmetic constraints, set constraints, global constraints) and complete search strategies. The documentation is well-writen. It is worth read it!

## Installation

A jar file named JaCoP-3.2.jar can be downloaded from www.jacop.eu. This file has been installed in CIE. The path /usr/local/opt/eclipse_lib must be added in Eclipse.

## Packages

The solver is included in package JaCoP. For modeling and solving constraint-based problems, it is required to import JaCoP.core (domains, variables), JaCoP.constraints (constraints), and JaCoP.search (search algorithms).

```
import JaCoP.core.*;
import JaCoP.constraints.*;
import JaCoP.search.*;
```

## Modeling

In JaCoP, every component model (variables and constraints) must be inserted in a *store*.

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

A variable is an instance of a class extending the abstract class Var. IntVar is the subclass of integer variables. For instance, the following code creates variables for the $n$-queens problem. Each variable is inserted in the store. Each domain is equal to the interval of integers $0..n-1$.

```
final int n = 4;
IntVar[] queens = new IntVar[n];

for (int i=0; i<n; ++i) {
   queens[i] = new IntVar(store,"q"+i,0,n-1);
}
```

A constraint is an instance of a class extending the abstract class Constraint. There exist many concrete subclasses in JaCoP. For instance, an alldifferent constraint for the $n$-queens problem stating that there is no attack on columns is inserted in the store as follows.

```
store.impose(new Alldifferent(queens));
```

Arithmetic constraints are defined through primitive (simple) constraints. For instance, the constraint $|queens[j] - queens[i]| \neq j - i$ $(0 \leq i < j \leq n)$ stating that there is no attack on diagonals can be decomposed into three simple constraints

$$diffqjqi = queens[j] - queens[i],\ absdiffqjqi = |diffqjqi|,\ absdiffqjqi \neq j - i.$$

These constraints are inserted in the store as follows.

```
for (int i=0; i<n-1; ++i) {
   for (int j=i+1; j<n; ++j) {
      IntVar diffqjqi = new IntVar(store,1-n,n-1);
      store.impose(new XplusYeqZ(diffqjqi,queens[j],queens[i]));
      IntVar absdiffqjqi = new IntVar(store,0,n-1);
      store.impose(new AbsXeqY(diffqjqi,absdiffqjqi));
      store.impose(new XneqC(absdiffqjqi,j-i));
   }
}
```

## Consistency

The consistency of the store can be checked as follows. No constraint is violated if the result is *true*.

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

## Search

Once modeling is completed, the search strategy has to be defined, stating how to traverse the search tree. A search object is an instance of a class implementing the interface `Search<T>` where `T` is the type of variables. The `DepthFirstSearch<T>` class implements the DFS strategy.

```
DepthFirstSearch<IntVar> search = new DepthFirstSearch<IntVar>();
```

A `SelectChoicePoint<T>` object defines the algorithm processing a set of variables at a node of the search tree in order to create the children nodes. It takes as input the variable ordering strategy (the smallest domain in the code below) and the value selection strategy (the minimum value in the code below).

```
SelectChoicePoint<IntVar> select =
      new SimpleSelect<IntVar>(queens,
                               new SmallestDomain<IntVar>(),
                               new IndomainMin<IntVar>());
```

The search for a single solution is started by calling the `labelling` method, as follows.

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

The search of all the solutions is done by changing the solution listener before labeling.

```
search.getSolutionListener().searchAll(true);
search.getSolutionListener().recordSolutions(true);
```

The solutions can be displayed on the output screen.

```
for (int i=1; i<=search.getSolutionListener().solutionsNo(); i++){
   System.out.print("Solution " + i + ": [");
   for (int j=0; j<search.getSolution(i).length; j++) {
      if (j!=0) System.out.print(", ");
      System.out.print(search.getSolution(i)[j]);
   }
   System.out.println("]");
}

// Solution 1: [1, 3, 0, 2]
// Solution 2: [2, 0, 3, 1]
```

## Work of the day

1. Install `JaCoP`.
2. Experiment the aforementionned $n$-queens model. Is the solver able to find one solution in reasonable time for $n = 10$, $n = 20$, $n = 50$?
3. Try to change the value selection heuristics and compare the results (running time, number of nodes and backtracks).

```
Indomain<T>         IndomainMin<T>     IndomainMax<T>
IndomainMiddle<T>   IndomainMedian<T>  IndomainRandom<T>
```

4. Keep the best value selection heuristics. Now try to change the variable ordering heuristics and compare the results.

```
LargestDomain<T>          LargestMax<T>           LargestMin<T>
MaxRegret<T>              MinDomainOverDegree<T>  MostConstrainedDynamic<T>
MostConstrainedStatic<T>  SmallestDomain<T>       SmallestMax<T>
SmallestMin<T>            WeightedDegree<T>
```

5. Try to change the model as follows and compare the results. The arithmetic constraints preventing attacks on the diagonals

$$queens_i - queens_j \neq i - j,\ queens_i - queens_j \neq j - i$$

can be replaced with two `alldifferent` constraints

$$\text{alldifferent}(y_0, y_1, \ldots, y_{n-1}),\ \text{alldifferent}(z_0, z_1, \ldots, z_{n-1})$$

where for all $i$ and $j$

$$y_i = queens_i - i,\ z_j = queens_j + j.$$