
# Choco Solver Documentation

***Release 4.0.1***

**Charles Prud'homme, Jean-Guillaume Fages, Xavier Lorca**

**Apr 12, 2017**

# Contents

CHAPTER 1

# Preliminaries

## About Choco Solver

Choco is a Free and Open-Source Software dedicated to Constraint Programming. It is written in Java, under BSD license. It aims at describing real combinatorial problems in the form of Constraint Satisfaction Problems and solving them with Constraint Programming techniques. Choco is used for:

- teaching : easy to use
- research : easy to extend
- real-life applications : easy to integrate

Choco is among the fastest CP solvers on the market. In 2013 and 2014, Choco has been awarded two silver medals and three bronze medals at the MiniZinc challenge that is the world-wide competition of constraint-programming solvers. In addition to these performance results, Choco benefits from academic contributors, who provide support and long term improvements, and the consulting company COSLING, which provides services ranging from training to the development and the integration of CP models into larger applications.

Choco official website is: http://www.choco-solver.org

### Technical overview

Choco Solver 4 is a Java 8 library including:

- various type of variables (integer, boolean, set and real),
- many constraints (alldifferent, count, nvalues, etc.),
- a configurable search (custom search, activity-based search, large neighborhood search, etc.),
- conflict explanations (conflict-based back jumping, dynamic backtracking, path repair, etc.).

It also includes facilities to interact with the search process, factories to help modeling, many samples, an interface to Ibex, etc. Choco Solver has also many extensions, including a FlatZinc parser to solve minizinc instances and a graph module to better solve graph problems such as the TSP. An overview of the features of Choco 4 may also be found

in the presentation made in the "CP Solvers: Modeling, Applications, Integration, and Standardization" workshop of CP2013. The source code of choco-solver-4 is hosted on GitHub.

## History

The first version of Choco dates from the early 2000s. A few years later, Choco 2 has encountered a great success in both the academic and the industrial world. For maintenance issue, Choco has been completely rewritten in 2011, leading to Choco 3. The first beta version of Choco 3 has been released in 2012 and it has shown significant performance improvement. Choco 4 comes with a simpler modeling API. The latest version is Choco 4.0.1.

## Main contributors

| | |
|---|---|
| **Core developers** | Charles Prud'homme and Jean-Guillaume Fages |
| | |
| **Main contributors** | Xavier Lorca, Narendra Jussien, Fabien Hermenier, Jimmy Liang. |
| | |
| **Previous versions contributors** | François Laburthe, Hadrien Cambazard, Guillaume Rochart, Arnaud Malapert, Sophie Demassey, Nicolas Beldiceanu, Julien Menana, Guillaume Richaud, Thierry Petit, Julien Vion, Stéphane Zampelli. |

If you want to contribute, let us know.

Choco is developed with Intellij IDEA and JProfiler, that are kindly provided for free.

## How to get Support ?

The company COSLING can provide you with professional support and specific software development related to Choco Solver. Feel free to contact them at *contact@cosling.com* to discuss your upcoming projects.

A forum is also available on the website of Choco. It is dedicated to technical questions about the Choco solver and basic modeling helps. If you encounter any bug or would like some features to be added, please feel free to open a discussion on the forum.

## How to cite Choco ?

A reference to this manual, or more generally to Choco 4, is made like this:

```
@manual{chocoSolver,
  author        = {Charles Prud'homme and Jean-Guillaume Fages and Xavier Lorca},
  title         = {Choco Solver Documentation},
  year          = {2016},
  organization  = {TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.},
  timestamp     = {Tue, 9 Feb 2016},
  url           = {http://www.choco-solver.org },
}
```

# Using Choco Solver

## Structure of the main package

You can download Choco Solver on the website of Choco Solver. You will get a zip file which contains the following files:

**choco-solver-4.0.1-with-dependencies.jar** A ready-to-use jar file including dependencies; it provides tools to declare a Model, the variables, the constraints, the search strategies, etc. In a few words, it enables modeling and solving CP problems.

**choco-solver-4.0.1.jar** A jar file excluding all dependencies and configuration file; Enable using choco-solver as a dependency of an application. Otherwise, it provides the same code as the jar with dependencies.

**choco-solver-4.0.1-sources.jar** The source of the core library, to be loaded in your favourite IDE to benefit from the javadoc while coding.

**choco-samples-4.0.1-sources.jar** The source of the artifact *choco-samples* made of problems modeled with Choco. It is a good start point to see what it is possible to do with Choco.

**apidocs-4.0.1.zip** Javadoc of Choco-4.0.1

**logback.xml** The logback configuration file; may be needed when choco-solver is used as a library.

Please, refer to *README.md* for more details.

## Adding Choco Solver to your project

### Directly

Simply add choco-solver-4.0.1-with-dependencies.jar to the classpath of your project (in a terminal or in your favorite IDE).

### With Maven

Choco Solver is available on the Maven Central Repository. To declare Choco as a dependency of your project, simply update the `pom.xml` of your project by adding the following instruction:

```
<dependency>
 <groupId>org.choco-solver</groupId>
 <artifactId>choco-solver</artifactId>
 <version>X.Y.Z</version>
</dependency>
```

where `X.Y.Z` is replaced by 4.0.1. Note that the artifact does not include any dependencies or *logback.xml*. Please, refer to *README.md* for the list of required dependencies.

### With SBT

To declare Choco as a dependency of your project, simply update the `build.sbt` of your project by adding the following instruction:

```
libraryDependencies ++= Seq(
  "org.choco-solver" % "choco-solver" % "X.Y.Z",
)
```

where X.Y.Z is replaced by 4.0.1.

## Compiling sources

As a Maven-based project, Choco Solver can be installed in a few instructions. Once you have downloaded the source (from the zip file or GitHub, simply run the following command:

```
mvn clean install -DskipTests
```

This instruction downloads the dependencies required for Choco Solver (such as the trove4j and logback) then compiles the sources. The instruction `-DskipTests` avoids running the tests after compilation (and saves you a couple of hours). Regression tests are run on a private continuous integration server.

Maven provides commands to generate files needed for an IDE project setup. For example, to create the project files for your favorite IDE:

**IntelliJ Idea**

```
mvn idea:idea
```

**Eclipse**

```
mvn eclipse:configure-workspace eclipse:eclipse
```

## Example

Simple example showing how to use Choco Solver

```
import org.chocosolver.solver.Model;
import org.chocosolver.solver.variables.IntVar;

/**
 * Trivial example showing how to use Choco Solver
 * to solve the equation system
 * x + y < 5
 * x * y = 4
 * with x in [0,5] and y in {2, 3, 8}
 *
 * @author Charles Prud'homme, Jean-Guillaume Fages
 * @since 9/02/2016
 */
public class Overview {

    public static void main(String[] args) {
        // 1. Create a Model
        Model model = new Model("my first problem");
        // 2. Create variables
        IntVar x = model.intVar("X", 0, 5);                  // x in [0,5]
        IntVar y = model.intVar("Y", new int[]{2, 3, 8});    // y in {2, 3, 8}
        // 3. Post constraints
        model.arithm(x, "+", y, "<", 5).post(); // x + y < 5
        model.times(x,y,4).post();              // x * y = 4
        // 4. Solve the problem
        model.getSolver().solve();
        // 5. Print the solution
        System.out.println(x); // Prints X = 2
```

```
        System.out.println(y); // Prints Y = 2
    }
}
```

## Change history

Changes to the library are logged into the CHANGES.md file.

CHAPTER 2

# Modeling

## The Model

The object `Model` is the key component. It is built as follows:

```
Model model = new Model();
```

or:

```
Model model = new Model("my problem");
```

This should be the first instruction, prior to any other modeling instructions, as it is needed to declare variables and constraints.

## Variables

### Principle

A variable is an *unknown*, mathematically speaking. The goal of a resolution is to *assign* a *value* to each variable. The *domain* of a variable –(super)set of values it may take– must be defined in the model.

Choco 4 includes several types of variables: `BoolVar`, `IntVar`, `SetVar` and `RealVar`. Variables are created using the `Model` object. When creating a variable, the user can specify a name to help reading the output.

### Integer variables

An integer variable an unknown whose value should be an integer. Therefore, the domain of an integer variable is a set of integers (representing possible values). To create an integer variable, the `Model` should be used:

```
// Create a constant variable equal to 42
IntVar v0 = model.intVar("v0", 42);
// Create a variable taking its value in [1, 3] (the value is 1, 2 or 3)
IntVar v1 = model.intVar("v1", 1, 3);
// Create a variable taking its value in {1, 3} (the value is 1 or 3)
IntVar v2 = model.intVar("v2", new int[]{1, 3});
```

It is then possible to build directly arrays and matrices of variables having the same initial domain with:

```
// Create an array of 5 variables taking their value in [-1, 1]
IntVar[] vs = model.intVarArray("vs", 5, -1, 1);
// Create a matrix of 5x6 variables taking their value in [-1, 1]
IntVar[][] vs = model.intVarMatrix("vs", 5, 6, -1, 1);
```

---

**Important:** It is strongly recommended to define an initial domain that is close to expected values instead of defining unbounded domains like `[Integer.MIN_VALUE, Integer.MAX_VALUE]` that may lead to :

- incorrect domain size (`Integer.MAX_VALUE - Integer.MIN_VALUE +1 = 0`)
- numeric overflow/underflow operations during propagation.

If *undefined* domain is really required, the following range should be considered: `[IntVar.MIN_INT_BOUND, IntVar.MAX_INT_BOUND]`. Such an interval defines *42949673* values, from *-21474836* to *21474836*.

---

There exists different ways to encode the domain of an integer variable.

### Bounded domain

When the domain of an integer variable is said to be *bounded*, it is represented through an interval of the form $[\![a, b]\!]$ where $a$ and $b$ are integers such that $a <= b$. This representation is pretty light in memory (it requires only two integers) but it cannot represent *holes* in the domain. For instance, if we have a variable whose domain is $[\![0, 10]\!]$ and a constraint enables to detect that values 2, 3, 7 and 8 are infeasible, then this learning will be lost as it cannot be encoded in the domain (which remains the same).

To specify you want to use bounded domains, set the `boundedDomain` argument to `true` when creating variables:

```
IntVar v = model.intVar("v", 1, 12, true);
```

**When using bounded domains, branching decisions must either be domain splits or bound assignments/removals.**
Indeed, assigning a bounded variable to a value strictly comprised between its bounds may results in infinite loop because such branching decisions will not be refutable.

### Enumerated domains

When the domain of an integer variable is said to be *enumerated*, it is represented through the set of possible values, in the form:

- $[\![a, b]\!]$ where $a$ and $b$ are integers such that $a <= b$
- $\{a, b, c, .., z\}$, where $a < b < c... < z$.

Enumerated domains provide more information than bounded domains but are heavier in memory (the domain usually requires a bitset).

To specify you want to use enumerated domains, either set the `boundedDomain` argument to `false` when creating variables by specifying two bounds or use the signature that specifies the array of possible values:

```
IntVar v = model.intVar("v", 1, 4, false);
IntVar v = model.intVar("v", new int[]{1,2,3,4});
```

**Modelling: Bounded or Enumerated?**

The choice of domain types may have strong impact on performance. Not only the memory consumption should be considered but also the used constraints. Indeed, some constraints only update bounds of integer variables, using them with bounded domains is enough. Others make holes in variables' domain, using them with enumerated domains takes advantage of the *power* of their filtering algorithm. Most of the time, variables are associated with propagators of various *power*. The choice of domain representation should then be done on a case by case basis.

## Boolean variable

Boolean variables, `BoolVar`, are specific `IntVar` that take their value in $[\![0, 1]\!]$. The avantage of `BoolVar` is twofold:

- They can be used to say whether or not constraint should be satisfied (reification)
- Their domain, and some filtering algorithms, are optimized

To create a new boolean variable:

```
BoolVar b = model.boolVar("b");
```

## Set variables

A set variable, `SetVar`, represents a set of integers, i.e. its value is a set of integers. Its domain is defined by a set interval `[LB,UB]` where:

- the lower bound, `LB`, is an `ISet` object which contains integers that figure in every solution.
- the upper bound, `UB`, is an `ISet` object which contains integers that potentially figure in at least one solution,

Initial values for both `LB` and `UB` should be such that `LB` is a subset of `UB`. Then, decisions and filtering algorithms will remove integers from `UB` and add some others to `LB`. A set variable is instantiated if and only if `LB = UB`.

A set variable can be created as follows:

```
// Constant SetVar equal to {2,3,12}
SetVar x = model.setVar("x", new int[]{2,3,12});

// SetVar representing a subset of {1,2,3,5,12}
SetVar y = model.setVar("y", new int[]{}, new int[]{1,2,3,5,12});
// possible values: {}, {2}, {1,3,5} ...

// SetVar representing a superset of {2,3} and a subset of {1,2,3,5,12}
SetVar z = model.setVar("z", new int[]{2,3}, new int[]{1,2,3,5,12});
// possible values: {2,3}, {2,3,5}, {1,2,3,5} ...
```

## Real variables

The domain of a real variable is an interval of doubles. Conceptually, the value of a real variable is a double. However, it uses a precision parameter for floating number computation, so its actual value is generally an interval of doubles,

whose size is constrained by the precision parameter. Real variables have a specific status in Choco 4, which uses Ibex solver to define constraints.

A real variable is declared with three doubles defining its bound and a precision:

```
RealVar x = model.realVar("x", 0.2d, 3.4d, 0.001d);
```

### Views: Creating variables from constraints

When a variable is defined as a function of another variable, views can be used to make the model shorter. In some cases, the view has a specific (optimized) domain representation. Otherwise, it is simply a modeling shortcut to create a variable and post a constraint at the same time. Few examples:

`x = y + 2`:

```
IntVar x = model.intOffsetView(y, 2);
```

`x = -y`:

```
IntVar x = model.intMinusView(y);
```

`x = 3*y`:

```
IntVar x = model.intScaleView(y, 3);
```

Views can be combined together, e.g. `x = 2*y + 5` is:

```
IntVar x = model.intOffsetView(model.intScaleView(y,2),5);
```

We can also use a view mecanism to link an integer variable with a real variable.

```
IntVar ivar = model.intVar("i", 0, 4);
double precision = 0.001d;
RealVar rvar = model.realIntView(ivar, precision);
```

This code enables to embed an integer variable in a constraint that is defined over real variables.

## Constraints

### Constraints and propagators

#### Main principles

A constraint is a logic formula defining allowed combinations of values for a set of variables, i.e., restrictions over variables that must be respected in order to get a feasible solution. A constraint is equipped with a (set of) filtering algorithm(s), named *propagator(s)*. A propagator **removes**, from the domains of the target variables, values that cannot correspond to a valid combination of values. A solution of a problem is variable-value assignment verifying all the constraints.

Constraint can be declared in *extension*, by defining the valid/invalid tuples, or in *intension*, by defining a relation between the variables. For a given requirement, there can be several constraints/propagators available. A widely used example is the *AllDifferent* constraint which ensures that all its variables take a distinct value in a solution. Such a rule can be formulated using :

- a clique of basic inequality constraints,
- a generic table constraint (an extension constraint that lists the valid tuples),
- **a dedicated global constraint analysing :**
  - instantiated variables (*Forward checking propagator*),
  - variable domain bounds (*Bound consistency propagator*),
  - variable domains (*Arc consistency propagator*).

Depending on the problem to solve, the efficiency of each option may be dramatically different. In general, we tend to use global constraints, that capture a good part of the problem structure. However, these constraints often model problems that are inherently NP-complete so only a partial filtering is performed in general, to keep polynomial time algorithms. This is for example the case of *NValue* constraint that one aspect relates to the problem of "minimum hitting set."

### Design choices

### Class organization

In Choco Solver 4, constraints are generally not associated with a specific java class. Instead, each constraint is associated with a specific method in `Model` that will build a generic `Constraint` with the right list of propagators. Each propagator is associated with a unique java class.

Note that it is not required to manipulate propagators, but only constraints. However, one can define specific constraints by defining combinations of existing and/or its own propagators.

### Solution checking

The satisfaction of the constraints is done on each solution by calling the `isSatisfied()` method of every constraint. By default, this method checks the `isEntailed()` method of each of its propagators.

---

**Note:** Additional checks (Java assertions) can be performed by adding the `-ea` instruction in the JVM arguments. This is useful when debugging a program.

---

## List of available constraints

Please refer to the javadoc of `Model` to have the list of available constraints.

## Posting constraints

To be effective, a constraint must be posted to the solver. This is achieved using the `post()` method:

```
model.allDifferent(vars).post();
```

Otherwise, if the `post()` method is not called, the constraint will not be taken into account during the solution process : it may not be satisfied in solutions.

## Reifying constraints

In Choco 4, it is possible to reify any constraint. Reifying a constraint means associating it with a `BoolVar` to represent whether or not the constraint is satisfied :

```
BoolVar b = constraint.reify();
```

Or:

```
BoolVar b = model.boolVar();
constraint.reifyWith(b);
```

Reifying a constraint means that we allow the constraint not to be satisfied. Therefore, the reified constraint **should not** be posted. For instance, let us consider "if `x<0` then `y>42`":

```
model.ifThen(
   model.arithm(x,"<",0),
   model.arithm(y,">",42)
);
```

**Note:** Reification is a specific process which does not rely on classical constraints. This is why `ifThen`, `ifThenElse`, `ifOnlyIf` and `reification` return void and do not need to be posted.

**Note:** A constraint is reified with only one boolean variable. Multiple calls to `constraint.reify()` will return the same variable. However, the following call will associate `b1` with the constraint and then post `b1 = b2`:

```
BoolVar b1 = model.boolVar();
BoolVar b2 = model.boolVar();
constraint.reifyWith(b1);
constraint.reifyWith(b2);
```

## Some specific constraints

### SAT constraints

A SAT solver is embedded in Choco. It is not designed to be accessed directly. The SAT solver is internally managed as a constraint (and a propagator), that's why it is referred to as SAT constraint in the following.

**Important:** The SAT solver is directly inspired by MiniSat[EenS03]. However, it only propagates clauses. Neither learning nor search is implemented.

Clauses can be added with the `SatFactory` (refer to javadoc for details). On any call to a method of `SatFactory`, the SAT constraint (and its propagator) is created and automatically posted to the solver. To declare complex clauses, you can call `SatFactory.addClauses(...)` by specifying a `LogOp` that represents a clause expression:

```
SatFactory.addClauses(LogOp.and(LogOp.nand(LogOp.nor(a, b), LogOp.or(c, d)), e),␣
↪model);
// with static import of LogOp
SatFactory.addClauses(and(nand(nor(a, b), or(c, d)), e), model);
```

### Automaton-based Constraints

`regular`, `costRegular` and `multiCostRegular` rely on an automaton, declared either implicitly or explicitly. There are two kinds of `IAutomaton` :

- `FiniteAutomaton`, needed for `regular`,
- `CostAutomaton`, required for `costRegular` and `multiCostRegular`.

`FiniteAutomaton` embeds an `Automaton` object provided by the `dk.brics.automaton` library. Such an automaton accepts fixed-size words made of multiple `char`, but the regular constraints rely on `IntVar`, so a mapping between `char` (needed by the underlying library) and `int` (declared in `IntVar`) has been made. The mapping enables declaring regular expressions where a symbol is not only a digit between *0* and *9* but any **positive** number. Then to distinct, in the word *101*, the symbols *0*, *1*, *10* and *101*, two additional `char` are allowed in a regexp: < and > which delimits numbers.

In summary, a valid regexp for the automaton-based constraints is a combination of **digits** and Java Regexp special characters.

---

**Examples of allowed RegExp**

`"0*11111110+10+10+11111110*"`, `"11(0|1|2)*00"`, `"(0|<10> |<20>)*(0|<10>)"`.

---

**Example of forbidden RegExp**

`"abc(a|b|c)*"`.

---

`CostAutomaton` is an extension of `FiniteAutomaton` where costs can be declared for each transition.

## Designing your own constraint

You can create your own constraint by creating a generic `Constraint` object with the appropriate propagators:

```
Constraint c = new Constraint("MyConstraint", new MyPropagator(vars));
```

---

**Important:** The array of variables given in parameter of a `Propagator` constructor is not cloned but referenced. That is, if a permutation occurs in the array of variables, all propagators referencing the array will be incorrect.

---

The only tricky part lies in the propagator implementation. Your propagator must extend the `Propagator` class but not all methods have to be overwritten. We will see two ways to implement a propagator ensuring that `X >= Y`.

### Basic propagator

You must at least call the super constructor to specifies the scope (set of variables) of the propagator. Then you must implement the two following methods:

`void propagate(int evtmask)`

> This method applies the global filtering algorithm of the propagator, that is, from *scratch*. It is called once during initial propagation (to propagate initial domains) and then during the solving process if the propagator is not incremental. It is the most important method of the propagator.

`isEntailed()`

This method checks the current state of the propagator. It is used for constraint reification. It checks whether the propagator will be always satisfied (`ESat.TRUE`), never satisfied (`ESat.FALSE`) or undefined (`ESat.UNDEFINED`) according to the current state of its domain variables. For instance,

- $A \neq B$ will always be satisfied when $A={0,1,2}$ and $B = \{4, 5\}$.
- $A = B$ will never be satisfied when $A = \{0, 1, 2\}$ and $B = \{4, 5\}$.
- The entailment of $A \neq B$ cannot be defined when $A = \{0, 1, 2\}$ and $B = \{1, 2, 3\}$.

`ESat isEntailed()` implementation may be approximate but should at least cover the case where all variables are instantiated, in order to check solutions.

Here is an example of how to implement a propagator for `X >= Y`:

```
// Propagator to apply X >= Y
public class MySimplePropagator extends Propagator<IntVar> {

    IntVar x, y;

    public MySimplePropagator(IntVar x, IntVar y) {
        super(new IntVar[]{x,y});
        this.x = x;
        this.y = y;
    }

    @Override
    public void propagate(int evtmask) throws ContradictionException {
        x.updateLowerBound(y.getLB(), this);
        y.updateUpperBound(x.getUB(), this);
    }

    @Override
    public ESat isEntailed() {
        if (x.getUB() < y.getLB())
            return ESat.FALSE;
        else if (x.getLB() >= y.getUB())
            return ESat.TRUE;
        else
            return ESat.UNDEFINED;
    }
}
```

### Elaborated propagator

The super constructor `super(Variable[], PropagatorPriority, boolean);` brings more information. `PropagatorPriority` enables to optimize the propagation engine (low arity for fast propagators is better). The boolean argument allows to specifies the propagator is incremental. When set to `true`, the method `propagate(int varIdx, int mask)` must be implemented.

---

**Note:** Note that if many variables are modified between two calls, a non-incremental filtering may be faster (and simpler).

---

The method `propagate(int varIdx, int mask)` defines the incremental filtering. It is called for every variable `vars[varIdx]` whose domain has changed since the last call.

The method `getPropagationConditions(int vIdx)` enables not to react on every kind of domain modification.

The method `setPassive()` enables to desactivate the propagator when it is entailed, to save time. The propagator is automatically reactivated upon backtrack.

The method `why(...)` explains the filtering, to allow learning.

Here is an example of how to implement a propagator for `X >= Y`:

```
// Propagator to apply X >= Y
public final class MyIncrementalPropagator extends Propagator<IntVar> {

    IntVar x, y;

    public MyIncrementalPropagator(IntVar x, IntVar y) {
        super(new IntVar[]{x,y}, PropagatorPriority.BINARY, true);
        this.x = x;
        this.y = y;
    }

    @Override
    public int getPropagationConditions(int vIdx) {
        if (vIdx == 0) {
            // awakes if x gets instantiated or if its upper bound decreases
            return IntEventType.combine(IntEventType.INSTANTIATE, IntEventType.
↪DECUPP);
        } else {
            // awakes if y gets instantiated or if its lower bound increases
            return IntEventType.combine(IntEventType.INSTANTIATE, IntEventType.
↪INCLOW);
        }
    }

    @Override
    public void propagate(int evtmask) throws ContradictionException {
        x.updateLowerBound(y.getLB(), this);
        y.updateUpperBound(x.getUB(), this);
        if (x.getLB() >= y.getUB()) {
            this.setPassive();
        }
    }

    @Override
    public void propagate(int varIdx, int mask) throws ContradictionException {
        if (varIdx == 0) {
            y.updateUpperBound(x.getUB(), this);
        } else {
            x.updateLowerBound(y.getLB(), this);
        }
        if (x.getLB() >= y.getUB()) {
            this.setPassive();
        }
    }

    @Override
    public ESat isEntailed() {
        if (x.getUB() < y.getLB())
            return ESat.FALSE;
        else if (x.getLB() >= y.getUB())
```

```
            return ESat.TRUE;
        else
            return ESat.UNDEFINED;
    }

    @Override
    public boolean why(RuleStore ruleStore, IntVar var, IEventType evt, int value) {
        boolean newrules = ruleStore.addPropagatorActivationRule(this);
        if (var.equals(x)) {
            newrules |=ruleStore.addLowerBoundRule(y);
        } else if (var.equals(y)) {
            newrules |=ruleStore.addUpperBoundRule(x);
        } else {
            newrules |=super.why(ruleStore, var, evt, value);
        }
        return newrules;
    }

    @Override
    public String toString() {
        return "prop(" + vars[0].getName() + ".GEQ." + vars[1].getName() + ")";
    }
}
```

## Idempotency

We distinguish two kinds of propagators:

> *Necessary* propagators, which ensure constraints to be satisfied.
>
> *Redundant* (or *Implied*) propagators that come in addition to some necessary propagators in order to get a stronger filtering.

Some propagators cannot be idempotent (Lagrangian relaxation, use of randomness, etc.). For some others, forcing idempotency may be very time consuming. A redundant propagator does not have to be idempotent but **a necessary propagator should be idempotent**[1] .

with its output domains returns its output domains. In that case, it has reached a fix point.

Trying to make a propagator idempotent directly may not be straightforward. We provide three implementation possibilities.

The *decomposed* (recommended) option:

> Split the original propagator into (partial) propagators so that the fix point is performed through the propagation engine. For instance, a channeling propagator $A \Leftrightarrow B$ can be decomposed into two propagators $A \Rightarrow B$ and $B \Rightarrow A$. The propagators can (but do not have to) react on fine events.

The *lazy* option:

> Simply post the propagator twice. Thus, the fix point is performed through the propagation engine.

The *coarse* option:

> the propagator will perform its fix point by itself. The propagator does not react to fine events. The coarse filtering algorithm should be surrounded like this:

[1] **idempotent**: calling a propagator twice has no effect, i.e. calling it

```
// In the case of ``SetVar``, replace ``getDomSize()`` by ``getEnvSize()-
↪getKerSize()``.
long size;
do{
  size = 0;
  for(IntVar v:vars){
    size+=v.getDomSize();
  }
  // really update domain variables here
  for(IntVar v:vars){
    size-=v.getDomSize();
  }
}while(size>0);
```

**Note:** Domain variable modifier returns a boolean valued to `true` if the domain variable has been modified.

CHAPTER 3

# Solving

Up to here, we have seen how to model a problem with the `Model` object. To solve it, we need to use the `Solver` object that is obtained from the model as follows:

```
Solver solver = model.getSolver();
```

The `Solver` is in charge of alternating constraint-propagation with search, and possibly learning, in order to compute solutions. This object may be configured in various ways.

## Core solving methods

### Solution computation

#### Finding one solution

A call to `solver.solve()` launches a resolution which stops on the first solution found, if any:

```
if(solver.solve()){
    // do something, e.g. print out variable values
}else if(solver.hasReachedLimit()){
    System.out.println("The could not find a solution
                        nor prove that none exists in the given limits");
}else {
    System.out.println("The solver has proved the problem has no solution");
}
```

If `solver.solve()` returns `true`, then a solution has been found and each variable is instantiated to a value. Otherwise, two cases must be considered:

- A limit has been declared and reached (`solver.hasReachedLimit()` returns true). There may be a solution, but the solver has not been able to find it in the given limit or there is no solution but the solver has not been able to prove it (i.e., to close to search tree) in the given limit. The resolution process stops in no particular place in the search tree.

- No limit has been declared or reached: The problem has no solution and the solver has proved it.

### Enumerating all solutions

You can enumerate all solutions of a problem with a simple while loop as follows:

```
while(solver.solve()){
    // do something, e.g. print out variable values
}
```

After the enumeration, the solver closes the search tree and variables are no longer instantiated to a value.

---

**Tip:** On a solution, one can get the value assigned to each variable by calling

```
ivar.getValue();     // instantiation value of an IntVar, return a int
svar.getValue();     // instantiation values of a SerVar, return a int[]
rvar.getLB();        // lower bound of a RealVar, return a double
rvar.getUB();        // upper bound of a RealVar, return a double
```

---

## Optimization

In Constraint-Programming, optimization is done by computing improving solutions, until reaching an optimum. Therefore, it can be seen as solving multiple times the model while adding constraints on the fly to prevent the solver from computing dominated solutions.

### Mono-objective optimization

The optimization process is the following: anytime a solution is found, the value of the objective variable is stored and a *cut* is posted. The cut is an additional constraint which states that the next solution must be (strictly) better than the current one. To solve an optimization problem, you must specify which variable to optimize and in which direction:

```
// to maximize X
model.setObjectives(Model.MAXIMIZE, X);
// or model.setObjectives(Model.MINIMIZE, X); to minimize X
while(solver.solve()){
    // an improving solution has been found
}
// the last solution found was optimal (if search completed)
```

You can use custom cuts by overriding the default cut behavior. The *cut computer* function defines how the cut should bound the objective variable. The input *number* is the best solution value found so far, the output *number* define the new bound. When maximizing (resp. minimizing) a problem, the cut limits the lower bound (resp. upper bound) of the objective variable. For instance, one may want to indicate that the value of the objective variable is the next solution should be

greater than or equal to the best value + 10

```
ObjectiveManager<IntVar, Integer> oman = solver.getObjectiveManager();
oman.setCutComputer(n -> n - 10);
```

---

**Tip:** When the objective is a function over multiple variables, you need to model it through one objective variable and additional constraints:

```
// Model objective function 3X + 4Y
IntVar OBJ = model.intVar("objective", 0, 999);
model.scalar(new IntVar[]{X,Y}, new int[]{3,4}, OBJ)).post();
// Specify objective
model.setObjectives(Model.MAXIMIZE, OBJ);
// Compute optimum
model.getSolver().solve();
```

### Multi-objective optimization

If you have multiple objective to optimize, you have several options. First, you may aggregate them in a function so that you end up with only one objective variable. Second, you can solve the problem multiple times, each one optimizing one variable and possibly fixing some bounds on the other. Third, you can enumerate solutions (without defining any objective) and add constraints on the fly to prevent search from finding dominated solutions. This is done by the *ParetoOptimizer* object which does the following: Anytime a solution is found, a constraint is posted which states that at least one of the objective variables must be strictly better: Such as $(X_0 < b_0 \vee X_1 < b_1 \vee \ldots \vee X_n < b_n)$ where $X_i$ is the ith objective variable and $b_i$ its value.

Here is a simple example on how to use the *ParetoOptimizer* to optimize two variables (a and b):

```
// simple model
Model model = new Model();
IntVar a = model.intVar("a", 0, 2, false);
IntVar b = model.intVar("b", 0, 2, false);
IntVar c = model.intVar("c", 0, 2, false);
model.arithm(a, "+", b, "=", c).post();

// create an object that will store the best solutions and remove dominated ones
ParetoOptimizer po = new ParetoOptimizer(Model.MAXIMIZE,new IntVar[]{a,b});
Solver solver = model.getSolver();
solver.plugMonitor(po);

// optimization
while(solver.solve());

// retrieve the pareto front
List<Solution> paretoFront = po.getParetoFront();
System.out.println("The pareto front has "+paretoFront.size()+" solutions : ");
for(Solution s:paretoFront){
        System.out.println("a = "+s.getIntVal(a)+" and b = "+s.getIntVal(b));
}
```

**Note:** All objectives must be optimized on the same direction (either minimization or maximization).

## Constraint propagation

One may want to propagate all constraints without search for a solution. This can be achieved by calling `solver.propagate()`. This method runs, in turn, the domain reduction algorithms of the constraints until it reaches a fix point. It may throw a `ContradictionException` if a contradiction occurs. In that case, the propagation engine must be flushed calling `solver.getEngine().flush()` to ensure there is no pending events.

**Warning:** If there are still pending events in the propagation engine, the propagation may results in unexpected results.

## Accessing variable values

The value of a variable can be accessed directly through the `getValue()` method only once the variable is instantiated, i.e. the value has been computed (call `isInstantiated()` to check it). Otherwise, the lower bound is returned (and an exception is thrown if `-ea` is on).

For instance, the following code may throw an exception because the solution has not been computed yet:

```
int v = variable.getValue();
solver.solve();
```

The following code may throw an exception in case no solution could be found (unsat problem or time limit reached):

```
solver.solve();
int v = variable.getValue();
```

The correct approach is the following :

```
if(solver.solve()){
    int v = variable.getValue();
}
```

In optimization, you can print every solution as follows:

```
while(solver.solve()){
    System.out.println(variable.getValue());
}
```

The last print correspond to the best solution found.

However, the following code does *NOT* display the best solution found:

```
while(solver.solve()){
    System.out.println(variable.getValue());
}
System.out.println("best solution found: "+variable.getValue());
```

Because it is outside the while loop, this code is reached once the search tree has been closed. It does not correspond to a solution state and therefore variable is no longer instantiated at this stage. To use solutions afterward, you need to record them using `Solution` objects.

## Recording solutions

A solution can be stored through a `Solution` object which maps every variable with its current value. It can be created as follows:

```
Solution solution = new Solution(model());
```

By default, a solution records the value of every variable, but you can specify a smaller scope in the `Solution` constructor.

Let `X` be the set of decision variables and `Y` another variable set that you need to store. To record other variables (e.g. an objective variables) you have two options:

- Declare them in the search strategy using a complementary strategy

```
solver.set(strategy(X),strategy(Y)).
```

- Specify which variables to store in the solution constructor

```
Solution solution = new Solution(model(), ArrayUtils.append(X,Y));
```

You can record the last solution found as follows :

```
Solution solution = new Solution(model());
while (solver.solve()) {
    solution.record();
}
```

You can also use a monitor as follows:

```
Solution solution = new Solution(model());
solver.plugMonitor(new IMonitorSolution() {
      @Override
      public void onSolution() {
          s.record();
      }
});
```

Or with lambdas:

```
Solution solution = new Solution(model());
solver.plugMonitor((IMonitorSolution) () -> s.record());
```

Note that the solution is erased on each new recording. To store all solutions, you need to create one new solution object for each solution.

You can then access the value of a variable in a solution as follows:

```
int val = s.getIntVal(Y[0])
```

The solution object can be used to store all variables in Choco Solver (binaries, integers, sets and reals)

## Search monitors

### Principle

A search monitor is an observer of the resolver. It gives user access before and after executing each main step of the solving process:

- *initialize*: when the solving process starts and the initial propagation is run,
- *open node*: when a decision is computed,
- *down branch*: on going down in the tree search applying or refuting a decision,
- *up branch*: on going up in the tree search to reconsider a decision,
- *solution*: when a solution is got,

- *restart search*: when the search is restarted to a previous node, commonly the root node,
- *close*: when the solving process ends,
- *contradiction*: on a failure,

With the accurate search monitor, one can easily observe with the resolver, from pretty printing of a solution to learning nogoods from restart, or many other actions.

The interfaces to implement are:

- `IMonitorInitialize`,
- `IMonitorOpenNode`,
- `IMonitorDownBranch`,
- `IMonitorUpBranch`,
- `IMonitorSolution`,
- `IMonitorRestart`,
- `IMonitorContradiction`,
- `IMonitorClose`.

Most of them gives the opportunity to do something before and after a step. The other ones are called after a step.

---

**Important:** A search monitor should not modify the resolver behavior (forcing restart and interrupting the search, for instance). This is the goal of the Move component of a resolver *Search loop*.

---

Simple example to print every solution:

```
Solver s = model.getSolver();
s.plugMonitor(new IMonitorSolution() {
    @Override
    public void onSolution() {
        System.out.println("x = "+x.getValue());
    }
});
```

In Java 8 style:

```
Solver s = model.getSolver();
s.plugMonitor((IMonitorSolution) () -> {System.out.println("x = "+x.getValue());});
```

## Search limits

### Built-in search limits

Search can be limited in various ways using the `Solver` (from `model.getSolver()`).

- `limitTime` stops the search when the given time limit has been reached. This is the most common limit, as many applications have a limited available runtime.

---

**Note:** The potential search interruption occurs at the end of a propagation, i.e. it will not interrupt a propagation algorithm, so the overall runtime of the solver might exceed the time limit.

---

- `limitSolution` stops the search when the given solution limit has been reached.
- `limitNode` stops the search when the given search node limit has been reached.
- `limitFail` stops the search when the given fail limit has been reached.
- `limitBacktrack` stops the search when the given backtrack limit has been reached.

For instance, to interrupt search after 10 seconds:

```
Solver s = model.getSolver();
s.limitTime("10s");
model.getSolver().solve();
```

### Custom search limits

You can design you own search limit by implementing a `Criterion` and using `resolver.limitSearch(Criterion c)`:

```
Solver s = model.getSolver();
s.limitSearch(new Criterion() {
    @Override
    public boolean isMet() {
        // todo return true if you want to stop search
    }
});
```

In Java 8, this can be shortened using lambda expressions:

```
Solver s = model.getSolver();
s.limitSearch(() -> { /*todo return true if you want to stop search*/ });
```

## Using resolution statistics

Resolution data are available in the `Solver` object, whose default output is `System.out`. It centralises widely used methods to have comprehensive feedback about the resolution process. There are two types of methods: those who need to be called **before** the resolution, with a prefix *show*, and those who need to called **after** the resolution, with a prefix *print*.

For instance, one can indicate to print the solutions all resolution long:

```
solver.showSolutions();
solver.findAllSolutions();
```

Or to print the search statistics once the search ends:

```
solver.solve();
solver.printStatistics();
```

On a call to `solver.printVersion()`, the following message will be printed:

```
** Choco 4.0.3 (2017-03) : Constraint Programming Solver, Copyleft (c) 2010-2017
```

On a call to `solver.printVersion()`, the following message will be printed:

```
- [ Search complete - [ No solution | {0} solution(s) found ]
  | Incomplete search - [ Limit reached | Unexpected interruption ] ].
   Solutions: {0}
[  Maximize = {1}  ]
[  Minimize = {2}  ]
   Building time : {3}s
   Resolution : {6}s
   Nodes: {7} ({7}/{6} n/s)
   Backtracks: {8}
   Fails: {9}
   Restarts: {10}
   Max depth: {11}
   Variables: {12}
   Constraints: {13}
```

Curly brackets *{instruction | }* indicate alternative instructions

Brackets *[instruction]* indicate an optional instruction.

If the search terminates, the message "Search complete" appears on the first line, followed with either the number of solutions found or the message "No solution". `Maximize` –resp. `Minimize`– indicates the best known value for the objective variable before exiting when an (single) objective has been defined.

Curly braces *{value}* indicate search statistics:

0. number of solutions found
1. objective value in maximization
2. objective value in minimization
3. building time in second (from `new Model()` to `solve()` or equivalent)
4. initialisation time in second (before initial propagation)
5. initial propagation time in second
6. resolution time in second (from `new Model()` till now)
7. number of nodes in the binary tree search : one for the root node and between one and two for each decision (two when the decision has been refuted)
8. number of backtracks achieved
9. number of failures that occurred (conflict number)
10. number of restarts operated
11. maximum depth reached in the binary tree search
12. number of variables in the model
13. number of constraints in the model

If the resolution process reached a limit before ending *naturally*, the title of the message is set to :

```
- Incomplete search - Limit reached.
```

The body of the message remains the same. The message is formatted thanks to the `IMeasureRecorder` interface which is a search monitor.

On a call to `solver.showSolutions()`, on each solution the following message will be printed:

```
{0} Solutions, [Maximize = {1}][Minimize = {2}], Resolution {6}s, {7} Nodes, \\
                                    {8} Backtracks, {9} Fails, {10} Restarts
```

followed by one line exposing the value of each decision variables (those involved in the search strategy).

On a call to `solver.showDecisions()`, on each node of the search tree a message will be printed indicating which decision is applied. The message is prefixed by as many "." as nodes in the current branch of the search tree. A decision is prefixed with `[R]` and a refutation is prefixed by `[L]`.

```
..[L]x  ==  1 (0) //X = [0,5] Y = [0,6] ...
```

**Warning:** `solver.printDecisions()` prints the tree search during the resolution. Printing the decisions slows down the search process.

# Search Strategies

The search space induced by variable domains is equal to $S = |d_1| * |d_2| * ... * |d_n|$ where $d_i$ is the domain of the $i^{th}$ variable. Most of the time (not to say always), constraint propagation is not sufficient to build a solution, that is, to remove all values but one from variable domains. Thus, the search space needs to be explored using one or more *search strategies*. A search strategy defines how to explore the search space by computing *decisions*. A decision involves a variables, a value and an operator, e.g. $x = 5$, and triggers new constraint propagation. Decisions are computed and applied until all the variables are instantiated, that is, a solution has been found, or a failure has been detected (backtrack occurs). Choco 4.0.1 builds a binary search tree: each decision can be refuted (if $x = 5$ leads to no solution, then $x! = 5$ is applied). The classical search is based on Depth First Search.

**Note:** There are many ways to explore the search space and this steps should not be overlooked. Search strategies or heuristics have a strong impact on resolution performances. Thus, it is strongly recommended to adapt the search space exploration to the problem treated.

## Default search strategy

If no search strategy is specified to the resolver, Choco 4 will rely on the default one (defined by a `defaultSearch` in `Search`). In many cases, this strategy will not be sufficient to produce satisfying performances and it will be necessary to specify a dedicated strategy, using `solver.setSearch(...)`. The default search strategy splits variables according to their type and defines specific search strategies for each type that are sequentially applied:

1. integer variables and boolean variables : `intVarSearch(ivars)` (calls `domOverWDegSearch`)
2. set variables: `setVarSearch(svars)`
3. real variables `realVarSearch(rvars)`
4. objective variable, if any: lower bound or upper bound, depending on the optimization direction

Note that *lastConflict* is also plugged-in.

## Specifying a search strategy

You may specify a search strategy to the resolver by using `solver.setSearch(...)` method as follows:

```
import static org.chocosolver.solver.search.strategy.Search.*;

// to use the default SetVar search on mySetVars
Solver s = model.getSolver();
s.setSearch(setVarSearch(mySetVars));

// to use activity based search on myIntVars
Solver s = model.getSolver();
s.setSearch(activityBasedSearch(myIntVars));

// to use activity based search on myIntVars
// then the default SetValSelectorFactoryVar search on mySetVars
Solver s = model.getSolver();
s.setSearch(activityBasedSearch(myIntVars), setVarSearch(mySetVars));
```

**Note:** Search strategies generally hold on some particular variable kinds only (e.g. integers, sets, etc.).

### Example

Let us consider we have two integer variables `x` and `y` and we want our strategy to select the variable of smallest domain and assign it to its lower bound. There are several ways to achieve this:

```
// 1) verbose approach using usual imports

import org.chocosolver.solver.search.strategy.Search;
import org.chocosolver.solver.search.strategy.assignments.DecisionOperator;
import org.chocosolver.solver.search.strategy.selectors.values.*;
import org.chocosolver.solver.search.strategy.selectors.variables.*;


    Solver s = model.getSolver();
    s.setSearch(Search.intVarSearch(
                    // selects the variable of smallest domain size
                    new FirstFail(model),
                    // selects the smallest domain value (lower bound)
                    new IntDomainMin(),
                    // apply equality (var = val)
                    DecisionOperator.int_eq,
                    // variables to branch on
                    x, y
    ));

// 2) Shorter approach : Use a static import for Search
// and do not specify the operator (equality by default)

import static org.chocosolver.solver.search.strategy.Search.*;

import org.chocosolver.solver.search.strategy.assignments.DecisionOperator;
import org.chocosolver.solver.search.strategy.selectors.values.*;
import org.chocosolver.solver.search.strategy.selectors.variables.*;


    Solver s = model.getSolver();
    s.setSearch(intVarSearch(
                    // selects the variable of smallest domain size
```

```
                    new FirstFail(model),
                    // selects the smallest domain value (lower bound)
                    new IntDomainMin(),
                    // variables to branch on
                    x, y
    ));


// 3) Shortest approach using built-in strategies imports

import static org.chocosolver.solver.search.strategy.Search.*;

    Solver s = model.getSolver();
    s.setSearch(minDomLBSearch(x, y));
```

**Important:** Black-box search strategies

There are many ways of choosing a variable and computing a decision on it. Designing specific search strategies can be a very tough task. The concept of *Black-box search heuristic* has naturally emerged from this statement. Most common black-box search strategies observe aspects of the CSP resolution in order to drive the variable selection, and eventually the decision computation (presumably, a value assignment). Three main families of heuristic, stemming from the concepts of variable conflict, activity and impact may be found in Chocolreleasel. Black-box strategies can be augmented with restarts.

## List of available search strategy

Most available search strategies are listed in `Search`. This factory enables you to create search strategies using static methods. Most search strategies rely on :

- variable selectors (see package `org.chocosolver.solver.search.strategy.selectors.values`)
- value selectors (see package `org.chocosolver.solver.search.strategy.selectors.variables`)
- operators (see `DecisionOperator`)

`Search` is not exhaustive, look at the selectors package to see learn more search possibilities.

## Designing your own search strategy

### Using selectors

To design your own strategy using Search.intVarSearch, you simply have to implement your own variable and value selectors:

```
public static IntStrategy intVarSearch(VariableSelector<IntVar> varSelector,
                                       IntValueSelector valSelector,
                                       IntVar... vars)
```

For instance, to select the first non instantiated variable and assign it to its lower bound:

```
Solver s = model.getSolver();
s.setSearch(intVarSearch(
        // variable selector
        (VariableSelector<IntVar>) variables -> {
            for(IntVar v:variables){
                if(!v.isInstantiated()){
                    return v;
                }
            }
            return null;
        },
        // value selector
        (IntValueSelector) var -> var.getLB(),
        // variables to branch on
        x, y
));
```

**Note:** When all variables are instantiated, a `VariableSelector` must return `null`.

### From scratch

You can design your own strategy by creating `Decision` objects directly as follows:

```
s.setSearch(new AbstractStrategy<IntVar>(x,y) {
    // enables to recycle decision objects (good practice)
    PoolManager<IntDecision> pool = new PoolManager();
    @Override
    public Decision getDecision() {
        IntDecision d = pool.getE();
        if(d==null) d = new IntDecision(pool);
        IntVar next = null;
        for(IntVar v:vars){
            if(!v.isInstantiated()){
                next = v; break;
            }
        }
        if(next == null){
            return null;
        }else {
            // next decision is assigning nextVar to its lower bound
            d.set(next,next.getLB(), DecisionOperator.int_eq);
            return d;
        }
    }
});
```

**Attention:** A particular attention should be made while using `IntVar` and their type of domain. Indeed, bounded domains do not support making holes in their domain. Thus, removing a value which is not a current bound will be missed, and can lead to an infinite loop.

## Making a decision greedy

You can make a decision non-refutable by using `decision.setRefutable(false)`

To make an entire search strategy greedy, use:

```
Solver s = model.getSolver();
s.setSearch(greedySearch(inputOrderLBSearch(x,y,z)));
```

## Restarts

Restart means stopping the current tree search, then starting a new tree search from the root node. Restarting makes sense only when coupled with randomized dynamic branching strategies ensuring that the same enumeration tree is not constructed twice. The branching strategies based on the past experience of the search, such as adaptive search strategies, are more accurate in combination with a restart approach.

Unless the number of allowed restarts is limited, a tree search with restarts is not complete anymore. It is a good strategy, though, when optimizing an NP-hard problem in a limited time.

Some adaptive search strategies resolutions are improved by sometimes restarting the search exploration from the root node. Thus, the statistics computed on the bottom of the tree search can be applied on the top of it.

Several restart strategies are available in `Solver`:

```
// Restarts after after each new solution.
solver.setRestartOnSolutions()
```

Geometrical restarts perform a search with restarts controlled by the resolution event **[#f1]_** `counter` which counts events occurring during the search. Parameter `base` indicates the maximal number of events allowed in the first search tree. Once this limit is reached, a restart occurs and the search continues until `base``*``grow` events are done, and so on. After each restart, the limit number of events is increased by the geometric factor `grow`. `limit` states the maximum number of restarts.

```
solver.setGeometricalRestart(int base, double grow, ICounter counter, int limit)
```

The Luby 's restart policy is an alternative to the geometric restart policy. It performs a search with restarts controlled by the number of resolution events **[#f1]_** counted by `counter`. The maximum number of events allowed at a given restart iteration is given by base multiplied by the Las Vegas coefficient at this iteration. The sequence of these coefficients is defined recursively on its prefix subsequences: starting from the first prefix 1, the $(k+1)^t h$ prefix is the $k^t h$ prefix repeated `grow` times and immediately followed by coefficient $\texttt{grow}^k$.

- the first coefficients for `grow` =2: [1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,1,...]
- the first coefficients for `grow` =3 : [1, 1, 1, 3, 1, 1, 1, 3, 1, 1, 1, 3, 9,...]

```
solver.setLubyRestart(int base, int grow, ICounter counter, int limit)
```

You can design your own restart strategies using:

```
solver.setRestarts( LongCriterion restartCriterion,
                    IRestartStrategy restartStrategy,
                    int restartsLimit);
```

# Moves

## Large Neighborhood Search (LNS)

Local search techniques are very effective to solve hard optimization problems. Most of them are, by nature, incomplete. In the context of constraint programming (CP) for optimization problems, one of the most well-known and widely used local search techniques is the Large Neighborhood Search (LNS) algorithm[2]. The basic idea is to iteratively relax a part of the problem, then to use constraint programming to evaluate and bound the new solution.

### Principle

LNS is a two-phase algorithm which partially relaxes a given solution and repairs it. Given a solution as input, the relaxation phase builds a partial solution (or neighborhood) by choosing a set of variables to reset to their initial domain; The remaining ones are assigned to their value in the solution. This phase is directly inspired from the classical Local Search techniques. Even though there are various ways to repair the partial solution, we focus on the technique in which Constraint Programming is used to bound the objective variable and to assign a value to variables not yet instantiated. These two phases are repeated until the search stops (optimality proven or limit reached).

The `LNSFactory` provides pre-defined configurations. Here is the way to declare LNS to solve a problem:

```
LNSFactory.rlns(solver, ivars, 30, 20140909L, new FailCounter(solver, 100));
solver.findOptimalSolution(Model.MINIMIZE, objective);
```

It declares a *random* LNS which, on a solution, computes a partial solution based on `ivars`. If no solution are found within 100 fails (`FailCounter(solver, 100)`), a restart is forced. Then, every `30` calls to this neighborhood, the number of fixed variables is randomly picked. `20140909L` is the seed for the `java.util.Random`.

The instruction `LNSFactory.rlns(solver, vars, level, seed, frcounter)` runs:

The factory provides other LNS configurations together with built-in neighbors.

### Neighbors

While the implementation of LNS is straightforward, the main difficulty lies in the design of neighborhoods able to move the search further. Indeed, the balance between diversification (i.e., evaluating unexplored sub-tree) and intensification (i.e., exploring them exhaustively) should be well-distributed.

### Generic neighbors

One drawback of LNS is that the relaxation process is quite often problem dependent. Some works have been dedicated to the selection of variables to relax through general concept not related to the class of the problem treated [5,24]. However, in conjunction with CP, only one generic approach, namely Propagation-Guided LNS [24], has been shown to be very competitive with dedicated ones on a variation of the Car Sequencing Problem. Nevertheless, such generic approaches have been evaluated on a single class of problem and need to be thoroughly parametrized at the instance level, which may be a tedious task to do. It must, in a way, automatically detect the problem structure in order to be efficient.

[2] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming, CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998.

### Combining neighborhoods

There are two ways to combine neighbors.

### Sequential

Declare an instance of `SequenceNeighborhood(n1, n2, ..., nm)`. Each neighbor ni is applied in a sequence until one of them leads to a solution. At step k, the $(k \mod m)^{th}$ neighbor is selected. The sequence stops if at least one of the neighbor is complete.

### Adaptive

Declare an instance of `AdaptiveNeighborhood(1L, n1, n2, ..., nm)`. At the beginning a weight of 1 at assigned to each neighbor ni. Then, if a neighbor leads to solution, its weight $w_i$ is increased by 1. Any time a partial solution has to be computed, a value `W` between 1 and $w_1 + w_2 + ... + w_n$ is randomly picked (`1L` is the seed). Then the weight of each neighbor is subtracted from `W`, as soon as `W`$\leq 0$, the corresponding neighbor is selected. For instance, let's consider three neighbors n1, n2 and n3, their respective weights w1=2, w2=4, w3=1. `W` = 3 is randomly picked between 1 and 7. Then, the weight of n1 is subtracted, `W``2-=1; the weight of n2 is subtracted, ``W`-4 = -3, `W` is less than 0 and n2 is selected.

### Defining its own neighborhoods

One can define its own neighbor by extending the abstract class `INeighbor`. It forces to implements the following methods:

| Method | Definition |
|---|---|
| | |
| `void recordSolution()` | Action to perform on a solution (typicallu, storing the current variables' value). |
| | |
| `Decision fixSomeVariables()` | Fix some variables to their value in the last solution, computing a partial solution and returns it as a decision. |
| | |
| `void restrictLess()` | Relax the number of variables fixed. Called when no solution was found during a LNS run (trapped into a local optimum). |
| | |
| `boolean isSearchComplete()` | Indicates whether the neighbor is complete, that is, can end. |

### Restarts

A generic and common way to reinforce diversification of LNS is to introduce restart during the search process. This technique has proven to be very flexible and to be easily integrated within standard backtracking procedures[3].

[3] Laurent Perron. Fast restart policies and large neighborhood search. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming at CP 2003*, volume 2833 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003.

### Walking

A complementary technique that appear to be efficient in practice is named *Walking* and consists in accepting equivalent intermediate solutions in a search iteration instead of requiring a strictly better one. This can be achieved by defining an `ObjectiveManager` like this:

```
solver.setObjectiveManager(new ObjectiveManager(objective, ResolutionPolicy.MAXIMIZE,␣
↪false));
```

Where the last parameter, named `strict` must be set to false to accept equivalent intermediate solutions.

Other optimization policies may be encoded by using either search monitors or a custom `ObjectiveManager`.

# Learning

## Explanations

Choco 4 natively support explanations[4]. However, no explanation engine is plugged-in by default.

### Principle

Nogoods and explanations have long been used in various paradigms for improving search. An explanation records some sufficient information to justify an inference made by the solver (domain reduction, contradiction, etc.). It is made of a subset of the original propagators of the problem and a subset of decisions applied during search. Explanations represent the logical chain of inferences made by the solver during propagation in an efficient and usable manner. In a way, they provide some kind of a trace of the behavior of the solver as any operation needs to be explained.

Explanations have been successfully used for improving constraint programming search process. Both complete (as the mac-dbt algorithm) and incomplete (as the decision-repair algorithm) techniques have been proposed. Those techniques follow a similar pattern: learning from failures by recording each domain modification with its associated explanation (provided by the solver) and taking advantage of the information gathered to be able to react upon failure by directly pointing to relevant decisions to be undone. Complete techniques follow a most-recent based pattern while incomplete technique design heuristics to be used to focus on decisions more prone to allow a fast recovery upon failure.

The current explanation engine is coded to be *Asynchronous, Reverse, Low-intrusive and Lazy*:

**Asynchronous:** Explanations are not computed during the propagation.

**Reverse:** Explanations are computed in a bottom-up way, from the conflict to the first event generated, *keeping* only relevant events to compute the explanation of the conflict.

**Low-intrusive:** Basically, propagators need to implement only one method to furnish a convenient explanation schema.

**Lazy:** Explanations are computed on request.

To do so, all events are stored during the descent to a conflict/solution, and are then evaluated and kept if relevant, to get the explanation.

[4] Narendra Jussien. The versatility of using explanations within constraint programming. Technical Report 03-04-INFO, 2003.

### In practice

Consider the following example:

The problem has no solution since the two constraints cannot be satisfied together. A naive strategy such as `inputOrderLB(bvars)` (which selects the variables in lexicographical order) will detect lately and many times the failure. By plugging-in an explanation engine, on each failure, the reasons of the conflict will be explained.

The explanation engine records *deductions* and *causes* in order to compute explanations. In that small example, when an explanation engine is plugged-in, the two first failures will enable to conclude that the problem has no solution. Only three nodes are created to close the search, seven are required without explanations.

---

**Note:** Only unary, binary, ternary and limited number of nary propagators over integer variables have a dedicated explanation algorithm. Although global constraints over integer variables are compatible with explanations, they should be either accurately explained or reformulated to fully benefit from explanations.

---

### Cause

A cause implements `ICause` and must defined the `boolean why(RuleStore ruleStore, IntVar var, IEventType evt, int value)` method. Such a method add new *event filtering* rules to the ruleStore in parameter in order to *filter* relevant events among all generated during the search. Every time a variable is modified, the cause is specified in order to compute explanations afterwards. For instance, when a propagator updates the bound of an integer variable, the cause is the propagator itself. So do decisions, objective manager, etc.

### Computing explanations

When a contradiction occurs during propagation, it can only be thrown by:

- a propagator which detects unsatisfiability, based on the current domain of its variables;
- or a variable whom domain became empty.

Consequently, in addition to causes, variables can also explain the current state of their domain. Computing the explanation of a failure consists in going up in the stack of all events generated in the current branch of the search tree and filtering the one relative to the conflict. The entry point is either a the unsatisfiabable propagator or the empty variable.

---

**Note:** Explanations can be computed without failure. The entry point is a variable, and only removed values can be explained.

---

Each propagator embeds its own explanation algorithm which relies on the relation it defines over variables.

**Warning:** Even if a naive (and weak) explanation algorithm could be provided by all constraints, we made the choice to throw an *SolverException* whenever a propagator does not defined its own explanation algorithm. This is restrictive, but almost all non-global constraints support explanation, which enables reformulation. The missing explanation schemas will be integrated all needs long.

For instance, here is the algorithm of `PropGreaterOrEqualX_YC` ($x \geq y + c$, `x` and `y` are integer variables, `c` is a constant):

The first lines indicates that the deduction is due to the application of the propagator (l.2), maybe through reification. Then, depending on the variable touched by the deduction, either the lower bound of `y` (l.4) or the upper bound of `x` (l.6) explains the deduction. Indeed, such a propagator only updates lower bound of `y` based on the upper bound of `x` and *vice versa*.

Let consider that the deduction involves `x` and is explained by the lower bound of `y`. The lower bound `y` needs to be explained. A new rule is added to the ruleStore to specify that events on the lower bound of `y` needs to be kept during the event stack analyse (only events generated before the current are relevant). When such events are found, the ruleStore can be updated, until the first event is analyzed.

The results is a set of branching decisions, and a set a propagators, which applied altogether leads the conflict and thus, explained it.

### Explanations for the system

Explanations for the system, which try to reduce the search space, differ from the ones giving feedback to a user about the unsatisfiability of its model. Both rely on the capacity of the explanation engine to motivate a failure, during the search form system explanations and once the search is complete for user ones.

**Important:** Most of the time, explanations are raw and need to be processed to be easily interpreted by users.

### Conflict-based backjumping

When Conflict-based Backjumping (CBJ) is plugged-in, the search is hacked in the following way. On a failure, explanations are retrieved. From all left branch decisions explaining the failure, the last taken, *return decision*, is stored to jump back to it. Decisions from the current one to the return decision (excluded) are erased. Then, the return decision is refuted and the search goes on. If the explanation is made of no left branch decision, the problem is proven to have no solution and search stops.

**Factory**: `solver.explanations.ExplanationFactory`

**API**:

```
CBJ.plugin(Solver solver, boolean nogoodsOn, boolean userFeedbackOn)
```

- *solver*: the solver to explain.
- *nogoodsOn*: set to *true* to extract nogood from each conflict,. Extracting nogoods slows down the overall resolution but can reduce the search space.
- *userFeedbackOn*: set to *true* to store the very last explanation of the search (recommended value: *false*).

### Dynamic backtracking

This strategy, Dynamic backtracking (DBT) corrects a lack of deduction of Conflict-based backjumping. On a failure, explanations are retrieved. From all left branch decisions explaining the failure, the last taken, *return decision*, is stored to jump back to it. Decisions from the current one to the return decision (excluded) are maintained, only the return decision is refuted and the search goes on. If the explanation is made of no left branch decision, the problem is proven to have no solution and search stops.

**Factory**: `solver.explanations.ExplanationFactory`

**API**:

```
DBT.plugin(Solver solver, boolean nogoodsOn, boolean userFeedbackOn)
```

- *solver*: the solver to explain.
- *nogoodsOn*: set to *true* to extract nogood from each conflict,. Extracting nogoods slows down the overall resolution but can reduce the search space.
- *userFeedbackOn*: set to *true* to store the very last explanation of the search (recommended value: *false*).

### Explanations for the end-user

Explaining the last failure of a complete search without solution provides information about the reasons why a problem has no solution. For the moment, there is no simplified way to get such explanations. CBJ and DBT enable retrieving an explanation of the last conflict.

```
// .. problem definition ..
// First manually plug CBJ, or DBT
ExplanationEngine ee = new ExplanationEngine(solver, userFeedbackOn);
ConflictBackJumping cbj = new ConflictBackJumping(ee, solver, nogoodsOn);
solver.plugMonitor(cbj);
if(!solver.solve()){
    // If the problem has no solution, the end-user explanation can be retrieved
    System.out.println(cbj.getLastExplanation());
}
```

Incomplete search leads to incomplete explanations: as far as at least one decision is part of the explanation, there is no guarantee the failure does not come from that decision. On the other hand, when there is no decision, the explanation is complete.

## Search loop

The search loop whichs drives the search is a freely-adapted version PLM[5]. PLM stands for: Propagate, Learn and Move. Indeed, the search loop is composed of three parts, each of them with a specific goal.

- Propagate: it aims at propagating information throughout the constraint network when a decision is made,
- Learn: it aims at ensuring that the search mechanism will avoid (as much as possible) to get back to states that have been explored and proved to be solution-less,
- Move: it aims at, unlike the former ones, not pruning the search space but rather exploring it.

Any component can be freely implemented and attached to the search loop in order to customize its behavior. There exists some pre-defined *Move* and *Learn* implementations, avaiable in 550_slf.

**Move**:

550_slfdfs, 550_slflds, 550_slfdds, 550_slfhbfs, 550_slfseq, 550_slfrestart, 550_slfrestartonsol, 550_slflns.

**Learn**:

550_slfcbj, 550_slfdbt,

One can also define its own *Move* or *Learn* implementation, more details are given in 48_plm.

[5] Narendra Jussien and Olivier Lhomme. Unifying search algorithms for CSP. Technical report 02-3-INFO, EMN.

## Implementing a search loop component

A search loop is made of three components, each of them dealing with a specific aspect of the search. Even if many *Move* and *Learn* implementation are already provided, it may be relevant to define its own component.

---

**Note:** The *Propagate* component is less prone to be modified, it will not be described here. However, its interface is minimalist and can be easily implemented. A look to *org.chocosolver.solver.search.loop.propagate.PropagateBasic.java* is a good starting point.

---

The two components can be easily set in the *Solver* search loop:

**`void setMove(Move m)`** The current *Move* component is replaced by *m*.

**`Move getMove()`** The current *Move* component is returned.

*void setLearn(Learn l)* and *Learn getLearn()* are also avaiable.

Having access to the current *Move* (resp. *Learn*) component can be useful to combined it with another one. For instance, the *MoveLNS* is activated on a solution and creates a partial solution. It needs another *Move* to find the first solution and to complete the partial solution.

### Move

Here is the API of *Move*:

**`boolean extend(SearchLoop searchLoop)`** Perform a move when the CSP associated to the current node of the search space is not proven to be not consistent. It returns *true* if an extension can be done, *false* when no more extension is possible. It has to maintain the correctness of the reversibility of the action by pushing a backup world when needed. An extension is commonly based on a decision, which may be made on one or many variables. If a decision is created (thanks to the search strategy), it has to be linked to the previous one.

**`boolean repair(SearchLoop searchLoop)`** Perform a move when the CSP associated to the current node of the search space is proven to be not consistent. It returns *true* if a reparation can be done, *false* when no more reparation is possible. It has to backtracking backup worlds when needed, and unlinked useless decisions. The depth and number of backtracks have to be updated too, and "up branch" search monitors of the search loop have to called (be careful, when many *Move* are combined).

**`Move getChildMove()`** It returns the child *Move* or *null*.

**`void setChildMove(Move aMove)`** It defined the child *Move* and erases the previously defined one, if any.

**`boolean init()`** Called before the search starts, it should initialize the search strategy, if any, and its child *Move*. It should return *false* if something goes wrong (the problem has trivially no solution), *true* otherwise.

**`AbstractStrategy<V> getStrategy()`** It returns the search strategy in use, which may be *null* if none has been defined.

**`void setStrategy(AbstractStrategy<V> aStrategy)`** It defines a search strategy and erases the previously defined one, that is, a service which computes and returns decisions.

`org.chocosolver.solver.search.loop.move.MoveBinaryDFS.java` is good starting point to see how a *Move* is implemented. It defines a Depth-First Search with binary decisions.

### Learn

The aim of the component is to make sure that the search mechanism will avoid (as much as possible) to get back to states that have been explored and proved to be solution-less. Here is the API of *Learn*

**`void record(SearchLoop searchLoop)`** It validates and records a new piece of knowledge, that is, the current position is a dead-end. This is alwasy called *before* calling *Move.repair(SearchLoop)*.

**`void forget(SearchLoop searchLoop)`** It forgets some pieces of knowledge. This is alwasy called *after* calling *Move.repair(SearchLoop)*.

`org.chocosolver.solver.search.loop.learn.LearnCBJ` is good, yet not trivial, example of *Learn*.

## Multi-thread resolution

Choco 4 provides a simple way to use several threads to treat a problem. The main idea of that driver is to solve the *same* model with different search strategies and to share few information to make these threads help each others.

To use a portfolio of solvers in parallel, use `ParallelPortfolio` as follows:

```
ParallelPortfolio portfolio = new ParallelPortfolio();
int nbModels = 5;
for(int s=0;s<nbModels;s++){
    portfolio.addModel(makeModel());
}
portfolio.solve();
```

In this example, `makeModel()` is a method you have to implement to create a `Model` of the problem. Here all models are the same and the portfolio will change the search heuristics of all models but the first one. This means that the first thread will run according to your settings whereas the others will have a different configuration.

In order to specify yourself the configuration of each thread, you need to create the portfolio by setting the optional boolean argument `searchAutoConf` to false as follows:

```
ParallelPortfolio portfolio = new ParallelPortfolio(false);
int nbModels = 5;
for(int s=0;s<nbModels;s++){
    portfolio.addModel(makeModel(s));
}
portfolio.solve();
```

In this second example, the parameter `s` enables you to change the search strategy within the `makeModel` method (e.g. using a `switch(s)`).

When dealing with multithreading resolution, very few data is shared between threads: everytime a solution has been found its value is shared among solvers. Moreover, when a solver ends, it communicates an interruption instruction to the others. This enables to explore the search space in various way, using different model settings such as search strategies (this should be done in the dedicated method which builds the model, though).

# CHAPTER 4

# Miscellaneous

## Settings

A `Settings` object is attached to each `Solver`. It declares default behavior for various purposes: from general purpose (such as the welcome message), modelling purpose (such as enabling views) or solving purpose (such as the search binder).

The API is:

**`String getWelcomeMessage()`** Return the welcome message.

**`Idem getIdempotencyStrategy()`** Define how to react when a propagator is not ensured to be idempotent.

**`boolean enableViews()`** Set to 'true' to allow the creation of views in the `VariableFactory`. Creates new variables with channeling constraints otherwise.

**`int getMaxDomSizeForEnumerated()`** Define the maximum domain size threshold to force integer variable to be enumerated instead of bounded while calling `VariableFactory#integer(String, int, int, Solver)`.

**`boolean enableTableSubstitution()`** Set to true to replace intension constraints by extension constraints.

**`int getMaxTupleSizeForSubstitution()`** Define the maximum domain size threshold to replace intension constraints by extension constraints. Only checked when `enableTableSubstitution()` is set to true.

**`boolean plugExplanationIn()`** Set to true to plug explanation engine in.

**`boolean enablePropagatorInExplanation()`** Set to true to add propagators in explanations

**`double getMCRPrecision()`** Define the rounding precision for 51_icstr_mcreg. MUST BE < 13 as java messes up the precisions starting from 10E-12 (34.0*0.05 == 1.70000000000005).

**`double getMCRDecimalPrecision()`** Defines the smallest used double for 51_icstr_mcreg.

**`short[] getFineEventPriority()`** Defines, for fine events, for each priority, the queue in which a propagator of such a priority should be scheduled in.

**short[] getCoarseEventPriority()** Defines, for coarse events, for each priority, the queue in which a propagator of such a priority should be scheduled in

**ISearchBinder getSearchBinder()** Return the default 31_searchbinder.

**ICondition getEnvironmentHistorySimulationCondition()** Return the condition to satisfy when rebuilding history of backtrackable objects is needed.

**boolean warnUser()** Return true if one wants to be informed of warnings detected during modeling/solving (default value is false).

**boolean enableIncrementalityOnBoolSum(int nbvars)** Return true if the incrementality is enabled on boolean sum, based on the number of variables involved. Default condition is : nbvars > 10.

**boolean outputWithANSIColors()** If your terminal support ANSI colors (Windows terminals don't), you can set this to true and decisions and solutions will be output with colors.

**boolean debugPropagation()** When this setting returns true, a complete trace of the events is output. This can be quite big, though, and it slows down the overall process.

**boolean cloneVariableArrayInPropagator()** If this setting is set to true (default value), a clone of the input variable array is made in any propagator constructors. This prevents, for instance, wrong behavior when permutations occurred on the input array (e.g., sorting variables). Setting this to false may limit the memory consumption during modelling.

# Extensions of Choco

## choco-parsers

choco-parsers is an extension of Choco 4. It provides a parser for the FlatZinc language, a low-level solver input language that is the target language for MiniZinc. This module follows the flatzinc standards that are used for the annual MiniZinc challenge. It only supports integer variables. You will find it at https://github.com/chocoteam/choco-parsers

## choco-graph

choco-graph is a Choco 4 module which allows to search for a graph, which may be subject to graph constraints. The domain of a graph variable G is a graph interval in the form [G_lb,G_ub]. G_lb is the graph representing vertices and edges which must belong to any single solution whereas G_ub is the graph representing vertices and edges which may belong to one solution. Therefore, any value G_v must satisfy the graph inclusion "G_lb subgraph of G_v subgraph of G_ub". One may see a strong connection with set variables. A graph variable can be subject to graph constraints to ensure global graph properties (e.g. connectedness, acyclicity) and channeling constraints to link the graph variable with some other binary, integer or set variables. The solving process consists of removing nodes and edges from G_ub and adding some others to G_lb until having G_lb = G_ub, i.e. until G gets instantiated. These operations stem from both constraint propagation and search. The benefits of graph variables stem from modeling convenience and performance.

This extension has documentation. You will find it at https://github.com/chocoteam/choco-graph

## choco-gui

choco-gui is an extension of Choco 4. It provides a Graphical User Interface with various views which can be simply plugged on any Choco Model object. You will find it at https://github.com/chocoteam/choco-gui

# Ibex Solver

To manage continuous constraints with Choco, an interface with Ibex has been done. It needs Ibex to be installed on your system.

> "IBEX is a C++ library for constraint processing over real numbers.
>
> It provides reliable algorithms for handling non-linear constraints. In particular, round off errors are also taken into account. It is based on interval arithmetic and affine arithmetic." – http://www.ibex-lib.org/

## Installing Ibex

See the installation instructions of Ibex to complied Ibex on your system. More specially, take a look at Installation as a dynamic library Do not forget to add the `--with-java-package=org.chocosolver.solver.constraints.real` configuration option.

## Using Ibex

Once the installation is completed, the JVM needs to know where Ibex is installed to fully benefit from the Choco-Ibex bridge and declare real variables and constraints. This can be done either with an environment variable of by adding `-Djava.library.path=path/to/ibex/lib` to the JVM arguments. The path */path/to/ibex/lib* points to the *lib* directory of the Ibex installation directory.