JaCoP Library
User's Guide

Krzysztof Kuchcinski and Radosław Szymanek
Version 3.0, November 29, 2010

Copyright ©2002 and 2010 Krzysztof Kuchcinski and Radosław Szymanek. All rights reserved.

Contents

1 Introduction **5**
1.1 Getting started 6

2 Using JaCoP library **9**
2.1 Finite Domain Variables 9
2.2 Finite domains 10
2.3 Constraints 10
2.4 Search for solutions 11

3 Constraints **15**
3.1 Primitive constraints 15
3.2 Logical and conditional constraints 15
3.3 Global Constraints 15
3.3.1 Alldifferent constraints 15
3.3.2 Circuit constraint 16
3.3.3 Element constraint 17
3.3.4 Distance constraint 17
3.3.5 Cumulative constraint 17
3.3.6 Diff2 constraint 18
3.3.7 Min and Max constraints 19
3.3.8 Sum and SumWeight constraints 19
3.3.9 ExtensionalSupport and ExtensionalConflict constraints . . . 20
3.3.10 Assignment constraint 20
3.3.11 Count constraint 21
3.3.12 Values constraint 21
3.3.13 Global cardinality constraint (GCC) 22
3.3.14 Among and AmongVar 22
3.3.15 Regular constraint 23
3.3.16 Knapsack constraint 25
3.3.17 Geost constraint 25
3.3.18 NetworkFlow constraint 28
3.4 Decomposed constraints 32
3.4.1 Sequence constraint 32
3.4.2 Stretch constraint 32
3.4.3 Lex constraint 33
3.4.4 Soft-Alldifferent 33
3.4.5 Soft-GCC 34

4 Set Constraints **35**
4.1 Set Variables and Set Domains 35
4.2 Set Constraints 36
4.3 Search 36

5 Search **37**
5.1 Depth First Search 37
5.1.1 Restart search 38
5.2 Search plug-ins 39
5.3 Credit search 40
5.4 Limited discrepancy search 41
5.5 Combining search 42

A JaCoP constraints **43**
A.1 Primitive constraints 43
A.2 Set constraints 44
A.3 Logical, conditional and reified constraints 45
A.4 Global constraints 46

B JaCoP search methods **53**
B.1 Variable and value selection for FDVs 53
B.2 Variable and value selection for set variables 53
B.3 Search methods 54
B.4 Important methods for search plug-ins 55

C JaCoP debugging facilities **57**
C.1 Available switches 57
C.2 CPviz interface 57

Chapter 1

Introduction

JaCoP library provides constraint programming paradigm implemented in Java. It provides primitives to define finite domain (FD) variables, constraints and search methods. The user should be familiar with constraint (logic) programming (CLP) to be able to use this library. A good introduction to CLP can be found, for example, in [10].

JaCoP library provides most commonly used *primitive constraints*, such as equality, inequality as well as *logical*, *reified* and *conditional constraints*. It contains also number of *global constraints*. These constraints can be found in most commercial CP systems [3, 15, 4, 14]. Finally, JaCoP defines also *decomposable constraints*, i.e., constraints that are defined using other constraints and possibly auxiliary variables.

JaCoP library can be used by providing it as a JAR file or by specifying access to a directory containing all JaCoP classes. An example how program Main.java, which uses JaCoP, can be compiled and executed in the Linux like environment is provided below.

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

or

```
javac -classpath .:JaCoP.jar Main.java
java -classpath .:JaCoP.jar Main
```

Alternatively one can specify the class path variable.

In Java application which uses JaCoP it is required to specify import statements for all used classes from JaCoP library. An example of the import statements that import the whole subpackages of JaCoP at once is shown below.

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

Obviously, different Java IDE (Eclipse, NetBeans, etc.) and pure Java build tools (e.g., Ant) can be used for JaCoP based application development.

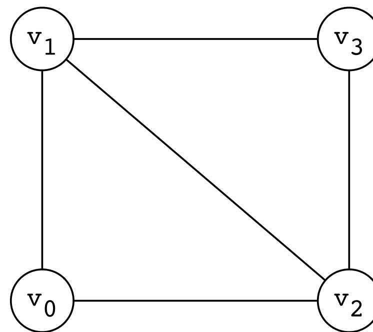

Figure 1.1: An example graph.

1.1 Getting started

Consider the problem of graph coloring as depicted in Fig. 1.1. Below, we provide a simplistic program with hard-coded constraints and specification of the search method to solve this particular graph coloring problem.

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

public class Main {

    static Main m = new Main ();

    public static void main (String[] args) {
        Store store = new Store();  // define FD 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");
    }
}
```

This program produces the following output indicating that vertices v0, v1 and v3 get different colors (1, 2 and 3 respectively), while vertex v3 is assigned color number 1.

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

Chapter 2

Using JaCoP library

The problem is specified with the help of variables (FDVs) and constraints over these variables. JaCoP support both finite domain variables (integer variables) and set variables. Both variables and constraints are stored in the store (`Store`). The store has to be created before variables and constraints. Typically, it is created using the following statement.

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

The store has large number of responsibilities. In short, it knits together all components required to model and solve the problem using JaCoP (CP methodology). One often abused functionality is printing method. The store has redefined the method `toString()`, but use it with care as printing large stores can be a very slow/memory consuming process.

In the next sections we will describe how to define FDVs and constraints.

2.1 Finite Domain Variables

Variable $X :: 1..100$ is specified in JaCoP using the following general statement (assuming that we have defined store with name `store`). Clerly, we required to have created store before we can create variables as any constructor for variable will require providing the reference to store in which the variable is created.

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

One can access the actual domain of FDV using the method `dom()`. The minimal and maximal values in the domain can be accessed using `min()` and `max()` methods respectively. The domain can contain "holes". This type of the domain can be obtained by adding intervals to FDV domain, as provided below:

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

which represents a variable X with the domain $1..100 \vee 120..160$.

FDVs can be defined without specifying their identifiers. In this case, a system will use an identifier that starts with "_" followed by a generated unique sequential number of this variable, for example "_123". This is illustrated by the following code snippet.

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

FDVs can be printed using Java primitives since the method `toString()` is redefined for class Variable. The following code snippet will first create a variable with the domain containing values 1, 2, 14, 15, and 16.

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

The last line of code prints a variable, which produces the following output.

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

One special variable class is a BooleanVariable. They have been added to JaCoP as they can be handled more efficiently than FDVs with multiple elements in their domain. They can be used as any other variable. However, some constraints may require BooleanVariables as parameters. An example of Boolean variable definition is shown below.

```
BooleanVar bv = new BooleanVar(s, "bv");
```

2.2 Finite domains

In the previous section, we have defined FDVs with domains without considering domain representation. JaCoP default domain (cllaed `IntervalDomain`) is represented as an ordered list of intervals. Each interval is represented by a pair of integers denoting the minimal and the maximal value. This representation makes it possible to define all possible finite domains of integers but it is not always computationally efficient. For some problems other representations might be more computationally efficient. Therefore, JaCoP also offers domain that is restricted to represent only one interval with its minimal and maximal value. This domain is called `BoundDomain` and can be used by a finite domain variable in a same way as interval domain. The only difference is that any attempt to remove values from inside the interval of this domain will have no effect.

The following code creates variable v with bound domain `1..10`.

```
IntVar v = new IntVar(s, "v", new BoundDomain(1, 10) );
```

2.3 Constraints

In JaCoP, there are three major types of constraints:

- primitive constraints,
- global constraints, and
- decomposable constraints.

Primitive constraints and global constraints can be imposed using `impose` method, while decomposable constraints are imposed using `imposeDecomposition` method. An example that imposes a primitive constraint `XneqY` is defined below. Again, in order to impose a constraint a store object must be available.

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

Alternatively, one can define first a constraint and then impose it, as shown below.

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

Both methods are equivalent.

The methods `impose(constraint)` and `constraint.impose(store)` often create additional data structures within the constraint store as well as constraint itself. Do note that constraint imposition does not involve checking if the constraint is consistent. Both methods of constraint imposition does not check whether the store remains consistent. If checking consistency is needed, the method `imposeWithConsistency(constraint)` should be used instead. This method throws `FailException` if the store is inconsistent. Note, that similar functionality can be achieved by calling the procedure `store.consistency()` explicitly (see section 2.4).

Constraints can have another constraints as their arguments. For example, *reified constraints* of the form $X = Y \Leftrightarrow B$ can be defined in JaCoP as follows.

```
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);
```

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

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

or

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

Note, that disjunction and other similar constraints accept only primitive constraints as parameters.

2.4 Search for solutions

After specification of the model consisting of variables and constraints, a search for a solution can be started. JaCoP offers a number of methods for doing this. It makes it possible to search for a single solution or to try to find a solution which minimizes/maximizes given cost function. This is achieved by using the depth-first-search together with constraint consistency enforcement.

The consistency check of all imposed constrains is achieved calling the following method from class `Store`.

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

When the procedure returns false then the `store` is in inconsistent state and no solution exists. The result `true` only indicates that inconsistency cannot be found. In other words, since the finite domain solver is not complete it does not automatically mean that the store is consistent.

To find a single solution the `DepthFirstSearch` method can be used. Since the search method is used both for finite domain variables and set variables it is recommended to specify the type of variables that are used in search. For finite domain variables, this type is usually `<IntVar>`. It is possible to not specify these information but it will generate compilation warnings if compilation option `-Xlint:unchecked` is used. A simple use of this method is shown below.

```
IntVar[] vars;
...
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select =
        new InputOrderSelect<IntVar>(store,
                                     vars, new IndomainMin<IntVar>());
boolean result = label.labeling(store, select);
```

The depth-first-search method requires the following information:

- how to assign values for each FDV from its domain; this is defined by `IndomainMin` class that starts assignments from the minimal value in the domain first and later assigns successive values.
- how to select FDV for an assignment from the array of FDVs (`vars`); this is decided explicitly here by `InputOrderSelect` class that selects FDVs using the specified order present in `vars`.
- how to perform labeling; this is specified by `DepthFirstSearch` class that is an ordinary depth-first-search.

Different classes can be used to implement `SelectChoicePoint` interface. They are summarized in Appendix B. The following example uses `SimpleSelect` that selects variables using the size of their domains, i.e., variable with the smallest domain is selected first.

```
IntVar[] vars;
...
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select =
           new SimpleSelect<IntVar>(vars,
                                        new SmallestDomain<IntVar>(),
                                        new IndomainMin<IntVar>());
boolean result = label.labeling(store, select);
```

In some situations it is better to group FDVs and assign the values to them one after the other. JaCoP supports this by another variable selection method called `SimpleMatrixSelect`.

An example of its use is shown below. This choice point selection heuristic works on two-dimensional lists of FDVs.

```
IntVar[][] varArray;
...
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select =
              new SimpleMatrixSelect<IntVar>(
                                    varArray,
                                    new SmallestMax<IntVar>(),
                                    new IndomainMin<IntVar>());
boolean result = label.labeling(store, select);
```

The optimization requires specification of a cost function. The cost function is defined by a FDV that, with the help of attached (imposed) constraints, gets correct cost value. A typical minimization for defined constraints and a cost FDV is specified below.

```
IntVar[] vars;
IntVar cost;
...
Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(vars,
                                    new SmallestDomain<IntVar>(),
                                    new IndomainMin<IntVar>());
boolean result = label.labeling(store, select, cost);
```

JaCoP offers number of different search heuristics based on depth-first-search. For example, credit search and limited discrepancy search. They are implemented using plug-in listeners that modify the standard depth-first-serch. For more details, see section 5.

Chapter 3

Constraints

3.1 Primitive constraints

JaCoP offers a set of primitive constraints that include basic arithmetic operations ($+, -, *, /$) as well as basic relations ($=, \neq, <, \leq, >, \geq$). Subtraction and division are not provided explicitly, but since constraints define relations between variables, they are defined using addition and multiplication. For detail list of primitive constraint see appendix A.1.

Primitive constraints can be used as arguments in logical and conditional constraints.

3.2 Logical and conditional constraints

Logical and conditional constraints use primitive constraints as arguments. In addition, JaCoP allows specification of the reified constraints. For detailed list of these constraints see appendix A.3

3.3 Global Constraints

3.3.1 Alldifferent constraints

Alldifferent constraint ensures that all FDVs from a given list have different values assigned. This constraint uses a simple consistency technique that removes a value, which is assigned to a given FDV from the domains of the other 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[] v = {a, b, c};
Constraint ctr = new Alldifferent(v);
store.impose(ctr);
```

enforces that the FDVs a, b, and c have different values.

Alldifferent constraint is provided as three different implementations. Constraint `Alldifferent` uses a simple implementation described above, i.e., if the domain of a finite domain variable gets assigned a value, the propagation algorithm will remove this value from the other variables. Constraint `Alldiff` implements this basic pruning method and, in addition, bounds consistency [11]. Finally, constraint `Alldistinct` implements a generalized arc consistency as proposed by Régin [12].

The example below illustrates the difference in constraints pruning power for `Alldifferent` and `Alldiff` constraints. Assume the following variables:

```
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};
```

The constraints will produce the following results after consistency enforcement.

```
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 will prune domains of variables a, b and c in the same way as `Alldiff` constraints but, in addition, it can remove single inconsistent values as illustrated below. Assume the following domains for a, b and c.

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

The constraints will produce the following results after consistency enforcement.

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

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

3.3.2 Circuit constraint

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 in the following way. 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.

For example, the 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[] v = {a, b, c};
```

```
Constraint ctr = new Circuit(store, 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.

3.3.3 Element constraint

Element constraint of the form `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

```
IntVar i = new IntVar(store, "i", 1, 3);
IntVar v = new IntVar(store, "v", 1, 50);
int[] el = {3, 44, 10};
Constraint ctr = new Element(i, el, v) ;
store.impose(ctr);
```

imposes the relation on the index variable $i :: \{1..3\}$, and the value variable v. The initial domain of v will be pruned to $v :: \{3, 10, 44\}$ after the intial consistency execution of this Element constraint. The change of one FDV propagates to another FDV. Imposing the constraint $V < 44$ results in change of $I :: \{1, 3\}$.

This constraint can be used, for example, to define discrete cost functions of one variable or a relation between task delay and its implementation resources. The constraint is simply implemented as a program which finds values allowed both for the first FDV and third FDV and updating them respectively.

3.3.4 Distance constraint

Distance constraint computes the absolute value between two FDVs. The result is another FDV, i.e., $d = |x - y|$.

The example below

```
IntVar a = new IntVar(store, "a", 1, 10);
IntVar b = new IntVar(store, "b", 2, 4);
IntVar c = new IntVar(store, "c", 0, 2);
Constraint ctr = new Distance(a, b, c);
store.impose(ctr);
```

produces result `a::1..6, b::2..4, c::0..2` since `a` must be pruned to have distance lower than three.

3.3.5 Cumulative constraint

Cumulative constraint was originally introduced to specify the requirements on tasks which needed to be scheduled on a number of resources. It 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, in our implementation, 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 used resources $Limit$. All parameters can be either domain variables or integers. The constraint is specified as follows.

```
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)
```

Formally, it enforces the following constraint:

$$\forall t \in [\min_{1 \le i \le n}(O_i), \max_{1 \le i \le n}(O_i + D_i)] : \sum_{k: O_k \le t \le O_k + D_k} AR_k \le Limit \tag{3.1}$$

In the above formulation, min and max stand for the minimum and the maximum values in the domain of each FDV respectively. The constraints ensures that at each time point, t, between the start of the first task (task selected by $min(O_i)$) and the end of the last task (task selected by $max(O_i + D_i)$) the cumulative resource use by all tasks, k, running at this time point is not greater than the available resource limit. This is shown in Fig 3.1.

JaCoP additionally requires that there exist at least one time point where the number of used resources is equal $Limit$, i.e.

$$\exists t \in [\min_{1 \le i \le n}(O_i), \max_{1 \le i \le n}(O_i + D_i)] : \sum_{k: O_k \le t \le O_k + D_k} AR_k = Limit \tag{3.2}$$

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 $T_3 :: \{5..10\}$

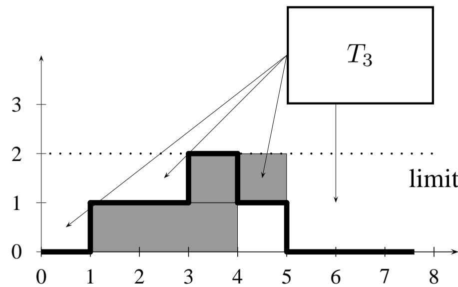

Figure 3.1: An example of the profile creation for the `cumulative` constraint.

3.3.6 Diff2 constraint

Diff2 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. The `diff2` constraint is specified as follows.

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

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

This constraint uses two different propagators. The first one tries to narrow O_i and L_i FDV's of each rectangle so that rectangles do not overlap. The second one is similar to the cumulative profile propagator but it is applied in both directions (in 2-dimensional space) for all rectangles. In addition, the constraint checks whether there is enough space to place all rectangles in the limits defined by each rectangle FDV's domains.

3.3.7 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);
```

will constraint FDV `min` to a minimal value of variables `a`, `b` and `c`.

NOTE! The position for parameters in constraints `Min` and `Max` is changed comparing to previous versions (i.e., parameters are swapped).

3.3.8 Sum and SumWeight constraints

These constraints enforce that a sum of elements of FDVs' vector is equal to a given FDV `sum`, that is $x_1 + x_2 + \cdots + x_n = sum$. The weighted sum is provided by the constraint `SumWeight` and imposes the following constraint $w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = sum$.

For example, the 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};
Constraint ctr = new Sum(v, sum);
Store.impose(ctr);
```

will constraint FDV `sum` to the sum of `a`, `b` and `c`.

3.3.9 ExtensionalSupport and ExtensionalConflict constraints

There exist several implementation of these constraint distinguished by their suffixes. The base implementation with suffix `VA` tries to balance the usage of memory versus time efficiency. `ExtensionalSupportMDD` uses multi-valued decision diagram (MDD) as internal representation and algorithms proposed in [2] while `ExtensionalSupportSTR` uses simple tabular reduction (STR) and the method proposed in [8].

Extensional support and extensional conflict constraints define relations between n FDVs. Both constraints are defined by a vector of n FDVs and a vector of n-tuples of integer values. The n-tuples define the relation between variables defined in the first vector.

The tuples of extensional support constraint define all combinations of values that can be assigned to variables specified in the vector of FDVs. Extensional conflict, on the other hand specifies the combinations of values that are not allowed in any assignment to variables.

The example below specifies the XOR logical relation of the form $a \oplus b = c$ using both constraints.

```
IntVar a = new IntVar(store, "a", 0, 1);
IntVar b = new IntVar(store, "b", 0, 1);
IntVar c = new IntVar(store, "c", 0, 1);
IntVar[] v = {a, b, c};
// version with ExtensionalSupport constraint
store.impose(new ExtensionalSupportVA(v,
                                      new int[][] {{0, 0, 0},
                                                   {0, 1, 1},
                                                   {1, 0, 1},
                                                   {1, 1, 0}}));
```

```
// version with ExtensionalConflict constraint
store.impose(new ExtensionalConflictVA(v,
                                       new int[][] {{0, 0, 1},
                                                    {0, 1, 0},
                                                    {1, 0, 0},
                                                    {1, 1, 1}}));
```

3.3.10 Assignment constraint

Assignment constraint implements the following relation between two vectors of FDVs $X_i = j \wedge Y_j = i$.

For example, the constraint

```
IntVar[] x = new IntVar[4];
IntVar[] y = new IntVar[4];
for (int i=0; i<4; i++) {
  x[i] = new IntVar(store, "x"+i, 0, 3);
  y[i] = new IntVar(store, "y"+i, 0, 3);
}
store.impose(new Assignment(x, y));
```

produces the following assignment to FDVs when `x[1] = 3`.

```
x0::{0..2}
x1=3
x2::{0..2}
x3::{0..2}
y0::{0, 2..3}
y1::{0, 2..3}
y2::{0, 2..3}
y3=1
```

The constraint has possibility to define the minimal index for vectors of FDVs. Therefore constraint `Assignment(x, y, 1)` will index variables from 1 instead of default value 0.

3.3.11 Count constraint

Count constraint counts number of occurrences of value `Val` on the list `List` in FDV `Var`.

For example, the constraint

```
IntVar[] List = new IntVar[4];
List[0] = new IntVar(store, "List_"+0, 0, 1);
List[1] = new IntVar(store, "List_"+1, 0, 2);
List[2] = new IntVar(store, "List_"+2, 2, 2);
List[3] = new IntVar(store, "List_"+3, 3, 4);

IntVar Var = new IntVar(store, "Var", 0, 4):

store.impose(new Count(List, Var, 2));
```

produces `Var :: {1..2}`.

If variable `Var` will be constrained to 1 then JaCoP will produce produces `List_1 :: {0..1}` .

NOTE! The position for parameters in constraint `Count` is changed comparing to previous versions.

3.3.12 Values constraint

Values constraint takes as arguments a list of variables and a counting variable. It counts a number of different values on the list of variables in the counting variable. For example, consider the following code.

```
IntVar x0 = new IntVar(store, "x0", 1,1);
IntVar x1 = new IntVar(store, "x1", 1,1);
IntVar x2 = new IntVar(store, "x2", 3,3);
IntVar x3 = new IntVar(store, "x3", 1,3);
IntVar x4 = new IntVar(store, "x4", 1,1);
IntVar[] val = {x0, x1, x2, x3, x4};
IntVar count = new IntVar(store, "count", 2, 2);
store.impose( new Values(count, val) );
```

Constraint `Values` will remove value 2 from variable $x3$ to assure that are only two different values (1 and 3) on the list of variables as specified by variable `count`.

NOTE! The position for parameters in constraint `Values` is changed comparing to previous versions.

3.3.13 Global cardinality constraint (GCC)

Global cardinality constraint (GCC) is defined using two lists of variables. The first list is the *value list* and the second list is the *counter list*. The constraint counts number of occurrences of different values in the variables from the value list. The counter list is used to counter occurrences of a specific value. It can also specify the number of allowed occurrences of a specific value on the value list. Variables on the counter list are assigned to values as follows. The lowest value in the domain of all variables from the value list is counted by the first variable on the counters list. The next value (+1) is counted by the next variable and so on.

For example, the following code counts number of values -1, 0, 1 and 2 on value list `x`. The values are counted using counter list `y` using the following mapping. -1 is counted in `y0`, 0 is counted in `y1`, 1 is counted in `y2` and 2 is counted in `y3`.

```
IntVar x0 = new IntVar(store, "x0", -1, 2);
IntVar x1 = new IntVar(store, "x1", 0, 2);
IntVar x2 = new IntVar(store, "x2", 0, 2);
IntVar[] x = {x0, x1, x2};

IntVar y0 = new IntVar(store, "y0", 1, 1);
IntVar y1 = new IntVar(store, "y1", 0, 1);
IntVar y2 = new IntVar(store, "y2", 0, 1);
IntVar y3 = new IntVar(store, "y3", 1, 2);
IntVar[] y = {y0, y1, y2, y3};

store.impose(new GCC(x, y));
```

The GCC constraint will allow only the following five combinations of x variables `[x0=-1, x1=0, x2=2]`, `[x0=-1, x1=1, x2=2]`, `[x0=-1, x1=2, x2=0]`, `[x0=-1, x1=2, x2=1]`, and `[x0=-1, x1=2, x2=2]`.

3.3.14 Among and AmongVar

Among constraint is specified using three parameters. The first parameter is the *value list*, the second one is a *set of values* specified as `IntervalDomain`, and finally the third parameter, the *counter*, counts the number of variables from the value list that get assigned values from the set of values. The constraint assures that exactly the number of variables defined by count variable is equal to one value from the set of values.

The following example constraints that either 2 or 4 variables from value list numbers are equal either 1 or 3. There exist 2880 such assignments.

```
IntVar numbers[] = new IntVar[5];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = new IntVar(store, "n" + i, 0,5);

IntVar count = new IntVar(store, "count", 2,2);
```

```
count.addDom(4,4);
IntervalDomain val = new IntervalDomain(1,1);
val.addDom(3,3);
store.impose(new Among(numbers, val, count));
```

AmongVar constraint is a generalization of Among constraint. Instead of specifying a set of values it uses a list of variables as the second parameter. It counts how many variables from the value list are equal to at least one variable from list of variables (second parameter).

The example below specifies the same conditions as the Among constraint in the above example.

```
IntVar numbers[] = new IntVar[5];
for (int i = 0; i < numbers.length; i++)
   numbers[i] = new IntVar(store, "n" + i, 0,5);

IntVar count = new IntVar(store, "count", 2,2);
count.addDom(4,4);
IntVar[] values = new IntVar[2];
values[0] = new IntVar(store, 1,1);
values[1] = new IntVar(store, 3,3);
store.impose(new AmongVar(numbers, values, count));
```

3.3.15 Regular constraint

Regular constraint accepts only the assignment to variables that are accepted by an automaton. The automaton is specified as the first parameter of this constraint and a list of variable is the second parameter. This constraint implements a polynomial algorithm to establish GAC.

The automaton is specified by its *states* and *transitions*. There are three types of states: initial state, intermediate states, and final states. Each transition has associated domain containing all values which can trigger this transition. Values assigned to transitions must be present in the domains of assigned constraint variable. Each value may cause firing of the related transition. The automaton eventually reaches a final state after taking the last transition as specified by the value of the last variable.

Each state can be assigned a level by topologically sorting states of the automaton. The variables from the list (second parameters) are assigned to these levels. All states at the same level are assigned the same variable (see Figure 3.2). If necessary, the automaton, containing cycles, is unrolled to match a list of variables. Each transitions has assigned values that are allowed for a variable when the transition in the automaton is selected. This is specified as the interval domain.

The example below implements the automaton from Figure 3.2. This automaton defines condition for three variables to be different values 0, 1 or 2.

```
IntVar[] var = new IntVar[3];
var[0] = new IntVar(store, "v"+0, 0, 2);
var[1] = new IntVar(store, "v"+1, 0, 2);
var[2] = new IntVar(store, "v"+2, 0, 2);

FSM g = new FSM();
```

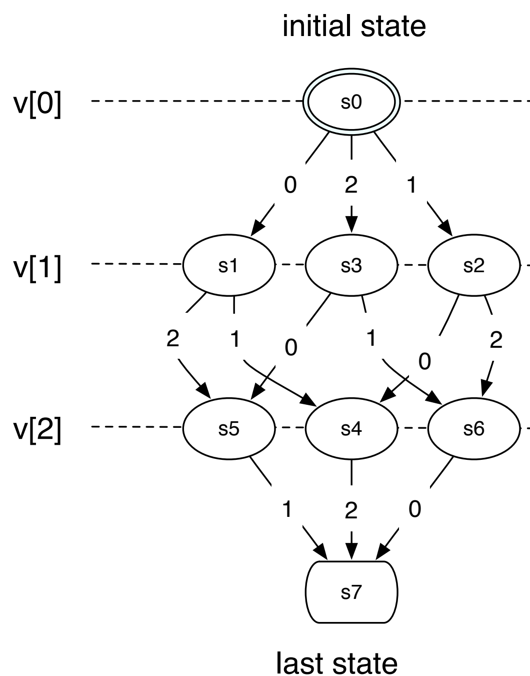

Figure 3.2: An example of the automaton for Regular constraint.

```
FSMState[] s = new FSMState[8];
for (int i=0; i<s.length; i++) {
   s[i] = new FSMState();
   g.allStates.add(s[i]);
}
g.initState = s[0];
g.finalStates.add(s[7]);

s[0].transitions.add(new FSMTransition(new IntervalDomain(0, 0),
                                       s[1]));
s[0].transitions.add(new FSMTransition(new IntervalDomain(1, 1),
                                       s[2]));
s[0].transitions.add(new FSMTransition(new IntervalDomain(2, 2),
                                       s[3]));

s[1].transitions.add(new FSMTransition(new IntervalDomain(1, 1),
                                       s[4]));
s[1].transitions.add(new FSMTransition(new IntervalDomain(2,2),
                                       s[5]));

s[2].transitions.add(new FSMTransition(new IntervalDomain(0, 0),
                                       s[4]));
s[2].transitions.add(new FSMTransition(new IntervalDomain(2,2),
                                       s[6]));
```

```
s[3].transitions.add(new FSMTransition(new IntervalDomain(0, 0),
                                        s[5]));
s[3].transitions.add(new FSMTransition(new IntervalDomain(1, 1),
                                        s[6]));

s[4].transitions.add(new FSMTransition(new IntervalDomain(2, 2),
                                        s[7]));
s[5].transitions.add(new FSMTransition(new IntervalDomain(1, 1),
                                        s[7]));
s[6].transitions.add(new FSMTransition(new IntervalDomain(0, 0),
                                        s[7]));

store.impose(new Regular(g, var));
```

3.3.16 Knapsack constraint

Knapsack constraint specifies knapsack problem. This implementation[1] was inspired by the paper [7] and published in [9]. The major extensions of that paper are the following. The quantity variables do not have to be binary. The profit and capacity of the knapsacks do not have to be integers. In both cases, the constraint accepts any finite domain variable.

The constraint specify number of categories of items. Each item has a given weight and profit. Both weight and profit are specified as positive integers. The problem is to select a number of items in each category to satisfy capacity constraint, i.e. the total weight must be in the limits specified by the capacity variable. Each such solution is then characterized by by a given profit. It is defined in JaCoP as follows.

```
Knapsack(int[] profits, int[] weights, IntVar[] quantity,
         IntVar knapsackCapacity, IntVar knapsackProfit)
```

It can be formalize using the following constraints.

$$\sum_i weights[i] \cdot quantity[i] = knapsackCapacity \tag{3.3}$$

$$\sum_i profits[i] \cdot quantity[i] = knapsackProfit \tag{3.4}$$

$$\forall_i : weights[i] \in \mathbb{Z}_{>0} \wedge \forall_i : profits[i] \in \mathbb{Z}_{>0} \tag{3.5}$$

3.3.17 Geost constraint

Geost is a geometrical constraint, which means that it applies to geometrical objects. It models placement problems under geometrical constraints, such as non overlapping constraints. Geost consistency algorithm was proposed by Beldiceanu et al [13]. The implementation of Geost in JaCoP is a result of a master thesis by Marc-Olivier Fleury.

In order to describe the constraint, we will introduce several definitions and relate them to JaCoP implementation.

[1]The constraint is based on Wadeck Follonier implementation carried our as his work on a student semester project.

Definition 1 *A* shifted box b *is a pair* $(b.t[], b.l[])$ *of vectors of integers of length* k*, where* k *is the number of dimensions of the problem. The origin of the box relative to a given reference is* $b.t[]$*, and* $b.l[]$ *contains the length of the box, for each dimension.*

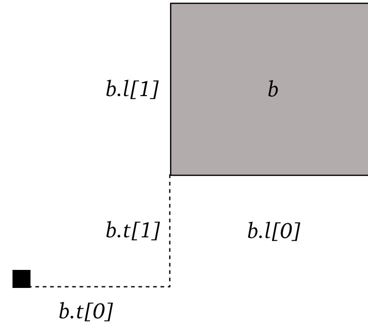

Figure 3.3: Shifted box in 2 dimensions. Reference origin is denoted by the black square.

Shifted box is defined in JaCoP using class DBox. For example, a two dimensional shifted box starting at coordinates (0,0) and having length 2 in first dimension and 1 in second direction is specified as follows.

```
DBox sbox = new DBox(new int[] {0,0}, new int[] {2,1});
```

Definition 2 *A* shape s *is a set of shifted boxes. It has a unique identifier* $s.sid$*.*

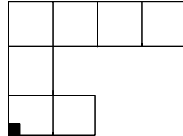

Figure 3.4: Example of a shape in 2 dimensions. Reference origin is denoted by the black square.

In JaCoP, we can define n shapes as collection of shifted boxes. Shape identifiers start at 0 and must be assigned consecutive integers. Therefore we have shapes with identifiers in interval `0..n-1`. The following JaCoP example defines a shape with identifier 0, consisting of three sboxs, depicted in Figure 3.4.

```
ArrayList<Shape> shapes = new ArrayList<Shape>();

ArrayList<DBox> shape1 = new ArrayList<DBox>();
shape1.add(new DBox(new int[] {0,0}, new int[] {2,1}));
shape1.add(new DBox(new int[] {0,1}, new int[] {1,2}));
shape1.add(new DBox(new int[] {1,2}, new int[] {3,1}));

shapes.add( new Shape(0, shape1) );
```

Definition 3 *An* object o *is a tuple* $(o.id, o.sid, o.x[], o.start, o.duration, o.end)$*.* $o.id$ *is a unique identifier,* $o.sid$ *is a variable that stores all shapes that* o *can take.* $o.x[]$ *is a* k*-dimensional vector of variables which represent the origin of* o*.* $o.start$*,* $o.duration$ *and* $o.end$ *define the interval of time in which* o *is present.*

An object in JaCoP is defined by class GeostObject. It specifies basically all parameters of an object. An example below specifies object 0 that can take shapes 0, 1, 2 and 3. The object can be placed using coordinates (`X_o1`, `Y_o1`). The object is present during time 2 to 14.

```
ArrayList<GeostObject> objects = new ArrayList<GeostObject>();

IntVar X_o1 = new IntVar(store, "x1", 0, 5);
IntVar Y_o1 = new IntVar(store, "y1", 0, 5);
IntVar[] coords_o1 = {X_o1, Y_o1};
IntVar shape_o1 = new IntVar(store, "shape_o1", 0, 3);
IntVar start_o1 = new IntVar(store, "start_o1", 2, 2);
IntVar duration_o1 = new IntVar(store, "duration_o1", 12, 12);
IntVar end_o1 = new IntVar(store, "end_o1", 14, 14);
GeostObject o1 = new GeostObject(0, coords_o1, shape_o1,
                                 start_o1, duration_o1, end_o1);
objects.add(o1);
```

Note that since object shapes are defined in terms of collections of shifted boxes, and since shifted boxes have a fixed size, Geost is not suited to solve problems in which object sizes can vary. Polymorphism provides some flexibility (shape variable having multiple values in their domain), but it is essentially intended to allow the modeling of objects that can take a small amount of different shapes. Typically objects that can be rotated. The duration of an object can be useful in cases where objects have variable sizes, because it is a variable, which means that some more flexibility is available. However, this feature is only available for one dimension. These restrictions are design choices made by the authors of Geost, probably because it fits well their primary field of application, which consists in packing goods in trucks. Using fixed sized shapes is also useful because it allows more deductions concerning possible placements.

When all shapes and objects are defined it is possible to specify geometrical constraints that must be fulfilled when placing these objects. Implemented geometrical constraints include *in-area* and *non-overlapping* constraints. In-area constraint enforces that objects have to lie inside a given *k*-dimensional sbox. Non-overlapping constraints require that no two objects can overlap.

The code below specifies two geometrical constraint, non-overlapping and in-area. They are specified by classes `NonOverlapping` and `InArea`. It **must** be noted that non-overlapping constraint in the code below specifies that all `objects` must not overlap in its two dimensions **and** time dimension (the time dimension is implemented as one additional dimension and therefore we specify dimensions 0, 1 and 2). In-area constraint requires that all object must be included in the sbox of dimensions 5x4.

```
ArrayList<ExternalConstraint> constraints =
                new ArrayList<ExternalConstraint>();
int[] dimensions = {0, 1, 2};
NonOverlapping constraint1 =
                new NonOverlapping(objects, dimensions);
```

```
constraints.add(constraint1);
InArea constraint2 = new InArea(
              new DBox(new int[] {0,0}, new int[] {5,4}), null);
constraints.add(constraint2);
```

Finally, the Geost constraint is imposed using the following code.

```
store.impose( new Geost(objects, constraints, shapes) );
```

3.3.18 NetworkFlow constraint

NetworkFlow constraint defines a minimum-cost network flow problem. An instance of this problem is defined on a directed graph by a tuple (N, A, l, u, c, b), where

- N is the set of nodes,
- A is the set of directed arcs,
- $l : A \rightarrow \mathbb{Z}_{\geq 0}$ is the lower capacity function on the arcs,
- $u : A \rightarrow \mathbb{Z}_{\geq 0}$ is the upper capacity function on the arcs,
- $c : A \rightarrow \mathbb{Z}$ is the flow cost-per-unit function on the arcs,
- $b : N \rightarrow \mathbb{Z}$ is the node mass balance function on the nodes.

A *flow* is a function $x : A \rightarrow \mathbb{Z}_{\geq 0}$. The minimum-cost flow problem asks to find a flow that satisfies all arc capacity and node balance conditions, while minimizing total cost. It can be stated as follows:

$$\min \quad z(x) = \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{3.6}$$

$$\text{s.t.} \quad l_{ij} \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in A, \tag{3.7}$$

$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i \quad \forall i \in N \tag{3.8}$$

The network is built with `NetworkBuilder` class using node, defined by class `JaCoP.net.Node` and arcs. Each node is defined by its name and node mass balance b. For example, node `A` with balance 0 is defined using network `net` as follows.

```
NetworkBuilder net = new NetworkBuilder();
Node A = net.addNode("A", 0);
```

Source node producing flow of capacity 5 and sin node consuming flow of capacity 5 are defined using similar constructs but different value of node mass balance, as indicated below.

```
Node source = net.addNode("source", 5);
Node sink = net.addNode("sink", -5);
```

Arc of the network are defined always between two nodes. They define connection between given nodes, the lower and upper capacity values assigned to the arc (values l and u) as well as the flow cost-per-unit function on the arc (value c). This can be defined using different methods with either integers or finite domain variables. For example, an arc from `source` to node `A` with $l = 0$ and $u = 5$ and $c = 3$ can be defined as follows.

```
net.addArc(A, B, 3, 0, 5);
```

or

```
x = new IntVar(store, "source->A", 0, 5);
net.addArc(A, B, 3, x);
```

It can be noted, that cost-per-unit value can also be defined as a finite domain variable.

The constraint has also the flow cost, defined as $z(x)$ in 3.6. It is defined as follows.

```
IntVar cost = new IntVar(store, "cost", 0, 1000);
net.setCostVariable(cost);
```

Note that the NetworkFlow only ensures that cost $z(x) \leq Z^{\max}$, where $z(x)$ is the total cost of the flow (see equation (3.6)). In our code it is defined as varibale `cost`.

The constraint is finally posed using the following method.

```
store.impose(new NetworkFlow(net));
```

For example, Figure 3.5 presnts the code for network flow problem depicted in Figure 3.6. The minimal flow, found by the solver, is 10 that is indicated in the figure.

Network builder has a special attribute `handlerList` that makes it possible to specify structural rules connected to the network. Each structural rule must implement `VarHandler` interface, which allows network flow constraint to cooperate with the structural rule. An important, already implemented rule, is available in the class `DomainStructure`. It specifies, for each structural variable `sv`, a list of arcs that the structural rule influence. This structural rule makes it possible to enforce minimum or maximum amount of flow on a given arc depending on the relationship between the domain of variable `sv` and domain `d`, specified within a structural rule. The domain of variable `sv` and domain `d` do not intersect if and only if the flow on a given arc is minimal as specified by initial value `x.min()`, denoted as x_{min}. Moreover, the domain of varibale `sv` is contained within domain `d` if and only if the actual flow on a given arc is maximal as specified by initial value `x.max()`, denoted as x_{max}. It is enforced by the following rules.

$$\begin{array}{rcl} sv.dom() \subseteq d & \Rightarrow & x = x_{max}, \\ x < x_{max} & \Rightarrow & sv.dom() \cap d = \varnothing, \end{array} \tag{3.9}$$

$$\begin{array}{rcl} sv.dom() \cap d = \varnothing & \Rightarrow & x = x_{min}, \\ x > x_{min} & \Rightarrow & sv.dom() \cap d \neq \varnothing, \end{array} \tag{3.10}$$

For example, creation of structural rule for arc between node `source` and `B` will enforce that this arc will have maximal flow if variable `s` is zero and minimal flow otherwise. The rule works also in the other direction, i.e. if the flow will be maximal variable `s=0` and if the flow will be minimal this variable will be 1.

```
Arc[] arcs = new Arc[1];
arcs[0] = net.addArc(source, B, 0, x[1]);
```

```
store = new Store();

IntVar[] x = new IntVar[8];

NetworkBuilder net = new NetworkBuilder();
Node source = net.addNode("source", 5);
Node sink = net.addNode("sink", -5);

Node A = net.addNode("A", 0);
Node B = net.addNode("B", 0);
Node C = net.addNode("C", 0);
Node D = net.addNode("D", 0);

x[0] = new IntVar(store, "x_0", 0, 5);
x[1] = new IntVar(store, "x_1", 0, 5);
net.addArc(source, A, 0, x[0]);
net.addArc(source, C, 0, x[1]);

x[2] = new IntVar(store, "a->b", 0, 5);
x[3] = new IntVar(store, "a->d", 0, 5);
x[4] = new IntVar(store, "c->b", 0, 5);
x[5] = new IntVar(store, "c->d", 0, 5);
net.addArc(A, B, 3, x[2]);
net.addArc(A, D, 2, x[3]);
net.addArc(C, B, 5, x[4]);
net.addArc(C, D, 6, x[5]);

x[6] = new IntVar(store, "x_6", 0, 5);
x[7] = new IntVar(store, "x_7", 0, 5);
net.addArc(B, sink, 0, x[6]);
net.addArc(D, sink, 0, x[7]);

IntVar cost = new IntVar(store, "cost", 0, 1000);
net.setCostVariable(cost);

store.impose(new NetworkFlow(net));
```

Figure 3.5: Constraint for network flow model from Figure 3.6

```
IntVar s = new IntVar(store, "s", 0, 1);
Domain[] domCond = new IntDomain[1];
domCond[0] = new IntervalDomain(0, 0);

net.handlerList.add(new DomainStructure(s,
                    Arrays.asList(domCond),
                    Arrays.asList(arcs)));
```

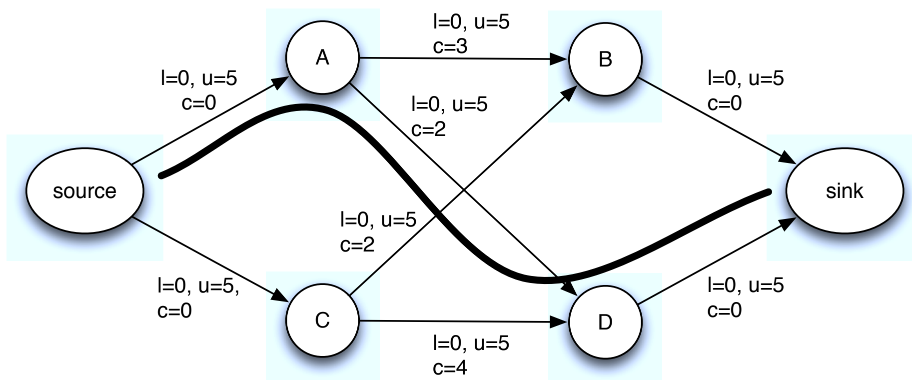

Figure 3.6: An example network and its minimal flow.

`SoftAlldifferent` constraint as well as `SoftGCC` constraint use `DomainStructure` rules to enforce flow from variable nodes to value nodes according the actual domain/-value of variables represented by variable nodes. It is crucial functionality for the implementation of those soft global constraints.

3.4 Decomposed constraints

Decomposed constraints do not define any new constraints and related pruning algorithms. They are translated into existing JaCoP constraints. `Sequence` and `Stretch` constraints are decomposed using `Regular` constraint.

Decomposed constraints are imposed using `imposeDecomposition` method instead of ordinary `impose` method.

3.4.1 Sequence constraint

Sequence constraint restricts values assigned to variables from a list of variables in such a way that any sub-sequence of length q contains N values from a specified set of values. Value N is further restricted by specifying min and max allowed values. Value q, min and max must be integer.

The following code defines restrictions for a list of five variables. Each sub-sequence of size 3 must contain 2 ($min = 2$ and $max = 2$) values 1.

```
IntVar[] var = new IntVar[5];
for (int i=0; i<var.length; i++)
   var[i] = new IntVar(store, "v"+i, 0, 2);

store.imposeDecomposition(new Sequence(var, //variable list
                      new IntervalDomain(1,1), //set of values
                      3, // q, sequence length
                      2, // min
                      2  // max
                    ));
```

There exist ten following solutions: [01101, 01121, 10110, 10112, 11011, 11211, 12110, 12112, 21101, 21121]

3.4.2 Stretch constraint

Stretch constraint defines what values can be taken by variables from a list and how sub-sequences of these values are formed. For each possible value it specifies a minimum (min) and maximum (max) length of the sub-sequence of these values.

For example, consider a list of five variables that can be assigned values 1 or 2. Moreover we constraint that the sub-sequence of value 1 must have length 1 or 2 and the sub-sequence of value 2 must have length either 2 or 3. The following code specifies this restrictions.

```
IntVar[] var = new IntVar[5];
for (int i=0; i<var.length; i++)
   var[i] = new IntVar(store, "v"+i, 1, 2);

store.imposeDecomposition(
                  new Stretch(new int[] {1, 2},  // values
                              new int[] {1, 2},  // min for 1 & 2
                              new int[] {2, 3},  // max for 1 & 2
                              var // variables
                        ));
```

This program produces six solutions: [11221, 11222, 12211, 12221, 22122, 22211]

3.4.3 Lex constraint

The Lex constraint enforces ascending lexicographic order between n vectors that can be of different size. The constraints makes it possible to enforce strict ascending lexicographic order, that is vector i must be always before vector $i+1$ in the lexicographical order, or it can allow equality between consecutive vectors. This is controlled by the last parameter of the constraint that is either `true` for strictly ascending lexicographic order or `false` otherwise. The default is non-strictly ascending lexicographic order when no second parameter is defined.

The following code specifies Lex constraint that holds when the strict ascending lexicographic order holds between four vectors of different sizes.

```
    IntVar[] d = new IntVar[6];
    for (int i = 0; i < d.length; i++) {
        d[i] = new IntVar(store, "d["+i+"]", 0, 2);
    }

    IntVar[][] x = { {d[0],d[1]}, {d[2]}, {d[3],d[4]}, {d[5]} };

    store.imposeDecomposition(new Lex(x, true));
```

This program produces nine following solutions.
[[0, 0], [1], [1, 0], [2]],
[[0, 0], [1], [1, 1], [2]],
[[0, 0], [1], [1, 2], [2]],
[[0, 1], [1], [1, 0], [2]],
[[0, 1], [1], [1, 1], [2]],
[[0, 1], [1], [1, 2], [2]],
[[0, 2], [1], [1, 0], [2]],
[[0, 2], [1], [1, 1], [2]],
[[0, 2], [1], [1, 2], [2]]

3.4.4 Soft-Alldifferent

Soft-alldifferent makes it possible to violate to some degree the alldifferent relation. The violations will come at a cost which is represented by cost variable. This constraint is decomposed with the help of network flow constraint.

There are two violation measures supported, *decomposition based*, where violation of any inequality relation between any pair contributes one unit of cost to the cost metric. The other violation measure is called *variable based*, which simply states how many times a variable takes value that is already taken by another variable.

The code below imposes a soft-alldifferent constraint over five variables with cost defined as being between 0 and 20.

```
IntVar[] x = new IntVar[5];
for (int i=0; i< x.length; i++)
   x[i] = new IntVar(store, "x"+i, 1, 4);
IntVar cost = new IntVar(store, "cost", 0, 20);
```

```
  store.imposeDecomposition(new SoftAlldifferent(x, cost,
                                ViolationMeasure.DECOMPOSITION_BASED );
```

3.4.5 Soft-GCC

Soft-GCC constraint makes it possible to violate to some degree GCC constraint. The Soft-GCC constraint requires number of arguments. In the code example below, `vars` specify the list of variables, which values are being counted, and the list of integers `countedValues` specifies the values that are counted. Values are counted in two counters specified by a programmer. The first list of counter variables, denoted by `hardCounters` in our example, specifies the hard limits that can not be violated. The second list, `softCounters`, specifies preferred values for counters and can be violated. Each position of a variable on these lists corresponds to the position of the value being counted on list `countedValues`. In practice, domains of variables on list `hardCounters` should be larger than domains of corresponding variables on list `hardCounters`.

Soft-GCC accepts only *value based* violation metric that, for each counted value, sums up the shortage or the excess of a given value among vars. There are other constructors of Soft-GCC constraint that allows to specify hard and soft counting constraints in multitude of different ways.

```
store.imposeDecomposition(new SoftGCC(vars, hardCounters,
                          countedValues, softCounters,
                          cost, ViolationMeasure.VALUE_BASED));
```

Chapter 4

Set Constraints

JaCoP provides a library of set constraints in library `JaCoP.set`. This implementation is based on set intervals and related operations as originally presented in [5] and the first version has been implemented as a diploma project[1]. JaCoP definition of set intervals, set domains and set variables is presented in section 4.1. Available constraints are discussed in section 4.2. Search, using set variables, is introduced in section 4.3.

4.1 Set Variables and Set Domains

Set is defined as an ordered collection of integers using class `JaCoP.core.IntervalDomain` and a set domain as abstract clss `JaCoP.set.core.SetDomain`. Currently, there exist only one implementation of set domain as a set interval, called `BoundSetDomain`. The set interval for BoundSetDomain `d` is defined by its greatest lower bound (*glb(d)*) and its least upper bound (*lub(d)*). For example, set domain $d = \{\{1\}..\{1..3\}\}$ is defined with $glb(d) = \{1\}$, set containing element 1, and $lub(d) = \{1..3\}$, set containing elements 1, 2 and 3. This set domain represent a set of sets $\{\{1\}, \{1..2\}, \{1, 3\}, \{1..3\}\}$. Each set domain to be correct must have $glb(d) \subseteq lub(d)$. $glb(d)$ can be considered as a set of all elements that are members of the set and $lub(d)$ specifes the largsest possible set.

The following statement defines set variable `s` for the set domain discussed above.

```
SetVar s = new SetVar(store, "s",
      new BoundSetDomain(new IntervalDomain(1,1),
                         new IntervalDomain(1,3)));
```

`BoundSetDomain` can specify a typical set domain, such as $d = \{\{\}..\{1..3\}\}$, in a simple way as

```
SetVar s = new SetVar(store, "s", 1, 3);
```

and an empty set domain as

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

[1] Robert Åkemalm, “Set theory in constraint programming”, Dept. of Computer Science, Lund University, 2009.

Set domain can be created using `IntervalDomain` and `BoundSetDomain` class methods. They make it possible to form different sets by adding elements to sets.

4.2 Set Constraints

JaCoP implements number of set constraints specified in appendix A.2. Constraints `AinS`, `AeqB` and `AinB` are primitive constraints and can be reified and used in other constraints, such conditional and logical. Other constraints are treated as ordinary JaCoP constraints.

Consider the following code that uses union constraint.

```
SetVar s1 = new SetVar(store, "s1",
                new BoundSetDomain(new IntervalDomain(1,1),
                                   new IntervalDomain(1,4)));
SetVar s2 = new SetVar(store, "s2",
                new BoundSetDomain(new IntervalDomain(2,2),
                                   new IntervalDomain(2,5)));
SetVar s = new SetVar(store, "s", 1,10);
Constraint c = new AunionBeqC(s1, s2, s);
```

It performs operation $\{\{1\}..\{1..4\}\} \bigcup \{\{2\}..\{2..5\}\} = \{\{\}..\{1..10\}\}$ and produces $\{\{1\}..\{1..4\}\} \bigcup \{\{2\}..\{2..5\}\} = \{\{1..2\}..\{1..5\}\}$. This represents 108 possible solutions.

4.3 Search

Set variables will require different search organization. Basically, during search the decisions will be made whether an element belongs to a set or it does not belong to this set.

JaCoP still uses `DepthFirsysearch` but needs different methods for set variable selection implementing `ComparatorVariable` and a method for value selection implementing `Indomain`. The special methods are specified in appendix B.2. In addition, variable selection methods `MostConstrainedStatic` and `MostConstrainedDynamic` will work also.

An example search can be specified as follows.

```
   Search<SetVar> search = new DepthFirstSearch<SetVar>();

   SelectChoicePoint<SetVar> select = new SimpleSelect<SetVar>(
                              vars,
                              new MinLubCard<SetVar>(),
                              new MaxGlbCard<SetVar>(),
                              new IndomainsetMin<SetVar>());
   search.setSolutionListener(new SimpleSolutionListener<SetVar>());

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

Chapter 5

Search

JaCoP offers methods for finding a single solution, all solutions and a solution that minimizes a given cost function. These methods are used together with methods defining variable selection and assignment of a selected value to variable. Both complete search methods and heuristic search methods can be defined in JaCoP.

JaCoP offers a powerful methods for search modification, called search plug-ins. Search plug-ins can be used both for collecting information about search as well as for changing search behaviour. For more information on search plug-ins see section 5.2.

5.1 Depth First Search

A solution satisfying all constraints can be found using a depth first search algorithm. This algorithm searches for a possible solution by organizing the search space as a search tree. In every node of this tree a value is assigned to a domain variable and a decision whether the node will be extended or the search will be cut in this node is made. The search is cut if the assignment to the selected domain variable does not fulfill all constraints. Since assignment of a value to a variable triggers the constraint propagation and possible adjustment of the domain variable representing the cost function, the decision can easily be made to continue or to cut the search at this node of the search tree.

Typical search method for a single solution, for a list of variables, is specified as follows.

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

where T is type of variables we are using for this search (usually `IntVar` or `SetVar`), var is a list of variables, $varSelect$ is a comparator method for selecting 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. Finally, $indomain$ method is used to select a value that will be assigned to a selected variable. Different variable selection and indomain methods are specified in appendix

B. This search, for varibales of type IntVar creates choice points $x_i = val$ and $x_i \neq val$ where x_i is variable identified by variable selection comparators and val is the value determined by indomain method. For variables of type SetVar the coice is made between $val \in x_i$ or $val \notin x_i$.

The standard method can be further modified to create search for all solutions. This is achieved by adopting the standard solution listener as specified below.

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

In the first line the flag that changes search to find all solutions is set. It is set in the default solution listener. In this example, we also set a flag that informs search to record all found solutions. If this flag is not set the search will only count solutions without storing them. The values for found solutions can be printed using `label.getSolutionListener().printAllSolutions()` method or the following piece of code.

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

Even if the solutions are not recorded, they are counted and number of found solutions can be retrieved using method `label.getSolutionListener().solutionsNo()`.

The minimization in JaCoP is achieved by defining variable for cost and using branch-and-bound (B&B) search, as specified below.

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

B&B search uses depth-first-search to find a solution. Each time a solution with cost $costValue_i$ is found a constraint $cost < costValue_i$ is imposed. Therefore the search finds solutions with lower cost until it eventually fails to find any solution that proves that the last found solution is optimal, i.e., there is no better solution.

Sometimes we want to interrupt search and report the best solution found in a given time. For this purpose, the search time-out functionality can used. For example, 10s time-out can be set with the following statement.

```
label.setTimeOut(10);
```

Moreover, one can define own time-out listener to perform specific actions.

5.1.1 Restart search

In some situation classical B&B algorithm is not best suited for optimization and so called *restart search* is used. This optimization search method finds a solution and then start search from the beginning but with additional constraint restricting the cost variable in the same way as B&B search. JaCoP does not support directly this kind of search but it can be easily implemented using the following code (use to maximize cost defined by variable `cost`).

```
label.setSolutionListener(new CostListener<IntVar>());
store.setLevel(store.level+1);
boolean Result = true, optimalResult = false;
while (Result) {
   Result = label.labeling(store, select);
   store.impose(new XgtC(cost, CostValue));
   optimalResult = optimalResult || Result;
}
store.removeLevel(store.level);
store.setLevel(store.level-1);
```

The search iteratively calls depth-first-search until no better solution is found. It also rises the store level before search and returns to "fresh" store to make it possible to operate on it later. This code requires access to value $CostValue$ that can be retrieved by providing a customized version of solution listener and its method `executeAfterSolution`. This method simply stores the value of the cost variable when a solution is found. See the code below for details.

```
public class CostListener<T extends Var> extends
                                      SimpleSolutionListener<T> {

   public boolean executeAfterSolution(Search<T> search,
                                   SelectChoicePoint<T> select) {
      boolean returnCode = super.executeAfterSolution(search,
                                                        select);

      CostValue = cost.value();
      return returnCode;
   }
}
```

5.2 Search plug-ins

The search-plugin is an object, which is informed about the current state of the search and may influence the behavior of the search. They are divided into search-plugins that change the search behavior and plugins used for collecting and sharing information. Table 5.1 lists the search-plugins available in JaCoP and their membership in a respective group.

Table 5.1: Search plug-ins available in JaCoP.

changing search	**cannot change search (information sharing)**
solution listener	exit listener
exit child listener	time-out listener
consistency listener	initialize listener

The search plug-ins are called during search when they reach a specific state, as specified below.

- *Solution listener* plug-in is called by search when a solution is found
- *Exit child listener* plug-in is called every time the search exits the search subtree (it has four different methods; two methods, which are called when the search has exited the left subtree and two methods for the right subtree).
- *Consistency listener* plug-in is called after consistency method at the current search node.
- *Exit listener* plug-in is called each time the search is about to exit (it can used to collect relevant search information).
- *Time-out listener* plug-in is called when the time-out occurs (if specified).
- *Initialize listener* plug-in is called at the beginning of the search.

Changing search plug-ins can override the status of the search by returning true or false status. For example, exit child listener method `leftChild` can override the status of the search by returning true or false status. If it returns true then the search continues and enters the right child to keep looking for a solution. Returning false instructs the search to skip exploring the right subtree

JaCoP makes it possible to combine several plugins hierarchically. Each listener may have multiple children listeners attached to it, which have potential to influence the behaviour of the parent. A very simple example of using this behavior, is using one listener to remember solution and another one to print it. This two different functionalities may be provided by two different listeners. In general, if search calls several children listeners the parent listener decides how to treat the results returned by them. The listeners already implemented in JaCoP use the following default rule to combine the return codes from different listeners. They combine their own return code with the return code of a child listener using conjunction of return codes. Several child listeners combine their return codes using disjunction of return codes.

5.3 Credit search

Credit search combines credit based exhaustive search at the beginning of the tree with local search in the rest of the tree [1]. In JaCoP, the credit search is controlled by three parameters: number of credits, number of backtracks during local search and maximum depth of search. In Figure 5.1 there is an example of the credit search tree. The search has initially 8 credits. The number of possible backtracks is three. During search half of the credits is distributed to the selected choice. The rest of the credits is distributed using the same principle for the next choice point. The first part of the search is based on the credits and makes it possible to investigate many possible assignments to domain variables while the other part is supposed to lead to a solution and can use a number of backtracks specified for this search. Moreover, the maximal depth of the search cannot be exceeded. 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.

An example of the command which produces the search tree depicted in Fig. 5.1 is as follows.

Figure 5.1: Credit search example.

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

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);
```

5.4 Limited discrepancy search

Limited discrepancy search (LDS) uses the partial search method proposed in [6]. 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 is specified as a parameter for LDS.

An example of LDS with one discrepancy is as follows.

```
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);
```

5.5 Combining search

JaCoP offers, through its plug-ins, possibility to combine several search methods into a single complex search. For example, the following code presents a search that is build as consecutive invocation of two search methods.

```
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> master =
       new SimpleSelect<IntVar>(vars1,
                                 new SmallestMin<IntVar>(),
                                 new SmallestDomain<IntVar>(),
                                 new IndomainMin<IntVar>());
master.addChildSearch(slave);

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

Appendix A

JaCoP constraints

A.1 Primitive constraints

Constraint	JaCoP specification
$X = Const$	`XeqC(X, Const)`
$X = Y$	`XeqY(X, Y)`
$X \neq Const$	`XneqC(X, Const)`
$X \neq Y$	`XneqY(X, Y)`
$X > Const$	`XgtC(X, Const)`
$X > Y$	`XgtY(X, Y)`
$X \geq Const$	`XgteqC(X, Const)`
$X \geq Y$	`XgteqY(X, Y)`
$X < Const$	`XltC(X, Const)`
$X < Y$	`XltY(X, Y)`
$X \leq Const$	`XlteqC(X, Const)`
$X \leq Y$	`XlteqY(X, Y)`
$X \cdot Const = Z$	`XmulCeqZ(X, Const, Z)`
$X \cdot Y = Z$	`XmulYeqZ(X, Y, Z)`
$X \div Y = Z$	`XdivYeqZ(X, Y, Z)`
$X \mod Y = Z$	`XmodYeqZ(X, Y, Z)`
$X + Const = Z$	`XplusCeqZ(X, Const, Z)`
$X + Y = Z$	`XplusYeqZ(X, Y, Z)`
$X + Y + Const = Z$	`XplusYplusCeqZ(X, Y, Const, Z)`
$X + Y + Q = Z$	`XplusYplusQeqZ(X, Y, Q, Z)`
$X + Const \leq Z$	`XplusClteqZ(X, Const, Z)`
$X + Y \leq Z$	`XplusYlteqZ(X, Y, Z)`
$X + Y > Const$	`XplusYgtC(X, Y, Const)`
$X + Y + Q > Const$	`XplusYplusQgtC(X, Y, Q, Const)`
$X^Y = Z$	`XexpYeqZ(X, Y, Z)`

A.2 Set constraints

Constraint	JaCoP specification
$e \in A$	`EinA(e, A)`
$S_1 =_2$	`AeqB(S1, S2)`
$S_1 \subseteq S_2$	`AinB(S1, S2)`
$S_1 \bigcup S_2 = S_3$	`AunionBeqC(S1, S2, S3)`
$S_1 \bigcap S_2 = S_3$	`AintersectBeqC(S1, S2, S3)`
$S_1 \setminus S_2 = S_3$	`AdiffBeqC(S1, S2, S3)`
$S_1 <> S_2$	`AdisjointB(S1, S2)`
Match	Match(Set, VarArray)
$\#S_1 = X$	`CardAeqX(S, X)`
Weighted sum < S, W > = X	`SumWeightedSet(S, W, X)`
$Set[X] = Y$	`ElementSet(X, Set, Y)`

A.3 Logical, conditional and reified constraints

Constraint	JaCoP specification
$\neg c$	Not(c);
$c1 \Leftrightarrow c2$	Eq(c1, c2);
$c1 \wedge c2 \wedge \cdots \wedge cn$	PrimitiveConstraint[] c = {c1, c2, ...cn}; And(c); or ArrayList<PrimitiveConstraint> c = new ArrayList<PrimitiveConstraint>(); c.add(c1); c.add(c2); ...c.add(cn); And(c);
$c1 \vee c2 \vee \cdots \vee cn$	PrimitiveConstraint[] c = {c1, c2, ...cn}; Or(c); or ArrayList<PrimitiveConstraint> c = new ArrayList<PrimitiveConstraint>(); c.add(c1); c.add(c2); ...c.add(cn); Or(c);
X in Dom	In(X, Dom);
$c \Leftrightarrow B$	Reified(c, B);
$c \Leftrightarrow \neg B$	Xor(c, B);
if $c1$ then $c2$	IfThen(c1, c2);
if $c1$ then $c2$ else $c3$	IfThenElse(c1, c2, c3);
Boolean operations on variables	
	BooleanVariable[] b = {b1, b2, ..., bn}; or ArrayList<BooleanVariable> b = new ArrayList<BooleanVariable(); b.add(b1); b.add(b2); ...b.add(bn); BoolanVariable result = new BooleanVariable(store, "result");
$result = b1 \wedge b2 \wedge \cdots \wedge bn$	AndBool(b, result)
$result = b1 \vee b2 \vee \cdots \vee bn$	OrBool(b, result)
$result = b1 \oplus b2$	XorBool(b1, b2, result)
$result = b1 \rightarrow b2$	IfThenBool(b1, b2, result)
$result = b1 == b2 == \cdots == bn$	EqBool(b, result)

A.4 Global constraints

$x_1 + x_2 + \cdots + x_n = sum$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar sum = new IntVar(...)
Sum(x, sum);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
IntVar sum = new IntVar(...)
Sum(x, sum);
```

$w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = sum$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar sum = new IntVar(...)
int[] w = {w1, w2, ..., wn};
SumWeight(x, w, sum);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
IntVar sum = new IntVar(...)
ArrayList<Integer> w=new ArrayList<Integer>();
w.add(w1); w.add(w1); ...w.add(wn);
SumWeight(x, w, sum);
```

alldifferent$([x_1, x_2, \ldots, x_n])$

```
IntVar[] x = {x1, x2, ..., xn};
Alldifferent(x);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Alldifferent(x);
```

alldiff$([x1, x2, \ldots, xn])$

```
IntVar[] x = {x1, x2, ..., xn};
Alldiff(x);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Alldiff(x);
```

alldistinct$([x_1, x_2, \ldots, x_n])$

```
IntVar[] x = {x1, x2, ..., xn};
Alldistinct(x);
```
or

```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Alldistinct(x);
```

among$([x_1, x_2, \ldots, x_n], val, count)$

```
IntVar[] x = {x1, x2, ..., xn};
IntervalDomain val = new IntervalDomain(k,l);
IntVar count = new IntVar(...);
Among(x, val, count);
```

amongVar$([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_m], count)$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar[] y = {y1, y2, ..., ym};
IntVar count = new IntVar(...);
Among(x, y, count);
```

assignment$([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n])$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar[] y = {y1, y2, ..., yn};
Assignment(x, y);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
ArrayList<IntVar> y = new ArrayList<IntVar>();
y.add(y1); y.add(y2); ...y.add(yn);
Assignment(x, y);
```

circuit$([x_1, x_2, \ldots, x_n])$

```
IntVar[] x = {x1, x2, ..., xn};
Circuit(Store, x);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Circuit(Store, x);
```

count$(value, [x_1, x_2, \ldots, x_n], var)$

```
int value = ...;
IntVar var = new IntVar(...);
IntVar[] x = {x1, x2, ..., xn};
Count(x, var, value);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Count(x, var, value);
```

cumulative$([t_1, t_2, \ldots, t_n], [d_1, d_2, \ldots, d_n], [r_1, r_2, \ldots, rn], ResourceLimit)$

```
IntVar[] t ={t1, t2, ..., tn};
IntVar[] d ={d1, d2, ..., dn};
IntVar[] r = {r1, r2, ..., rn};
IntVar Limit = new IntVar(...);
Cumulative(t, d, r, Limit);[2]
```
or using `ArrayList<IntVar>`

diff2$([[x_1, y_1, dx_1, dy_1], \ldots, [x_n, y_n, dx_n, dy_n]])$

```
IntVar[][] r = {{x1,y1,dx1,dy1}, ...,
{xn,yn,dxn,dyn}};
```
`Diff(r);` or `Diff2(Store, r);`[1]
or using `ArrayList<ArrayList<IntVar>>`

or

diff2$([x_1, \ldots, x_n], [y_1, \ldots, y_n], [dx_1, \ldots, dx_n], [dy_1, \ldots, dy_n])$

```
IntVar[] x = {x1, ..., xn};
IntVar[] y = {y1, ..., yn};
IntVar[] dx = {dx1, ..., dxn};
IntVar[] dy = {dy1, ..., dyn};
```
`Diff(x, y, dx, dy);` or `Diff2(Store, x, y, dx, dy);`[1]
or using `ArrayList<IntVar>`

distance$(x, y, dist)$

```
IntVar x, y, dist;
Distance(x, y, dist);
```

element$(Index, [n_1, n_2, \ldots, n_n], Value)$

```
IntVar Index, Value;
int[] i = {n1, n2, ..., nn };
Element(Index, i, Value);
```

element$(Index, [x_1, x_2, \ldots, x_n], Value)$

```
IntVar Index, Value;
IntVar[] x = {x1, x2, ..., xn };
Element(Index, x, Value);
```
or
```
ArrayList<IntVar> x = new ArrayList<IntVar>();
```

[1]Cumulative has two additional boolean parameters; the first one defines whether the edge finding algorithm is run and the second one defines whether the profile based algorithm is run. by default both algorithms are run.

[2]Diff and Diff2 have the same functionality but Diff recomputes most information when making consistency check while Diff2 stores this information. Therefore Diff is slower but uses less memory than Diff2. Diff and Diff2 have an additional boolean parameter specifying whether the profile based propagation algorithm will be run. By default it is run.

```
x.add(x1); x.add(x2); ...x.add(xn);
Element(Index, x, Value);
```

extensionalSupport$([x_1, x_2, \ldots, x_n], \{\{1, 2, \ldots, n\}, \{\ldots\}, \ldots, \{\ldots\}\})$

extensionalConflict$([x_1, x_2, \ldots, x_n], \{\{1, 2, \ldots, n\}, \{\ldots\}, \ldots, \{\ldots\}\})$

```
IntVar[] x = {x1, x2, ..., xn};
int[][] intTuple = {{...}, ...});
ExtensinalSupportVA(x,intTuple);
or
ExtensinalConflictVA(x,intTuple);
or
ExtensinalSupportSTR(x,intTuple);
or
ExtensinalSupportMDD(x,intTuple);
```

gcc$([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_m])$

```
IntVar[] x = {x1, x2, ..., xn};
IntVar[] y = {y1, y2, ..., ym};
GCC(x, y);
```

min$([x_1, x_2, \ldots, x_n], Xmin)$

```
IntVar[] x = {x1, x2, ..., xn};
Min(x, Xmin);
or
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Min(x, Xmin);
```

max$([x_1, x_2, \ldots, x_n], Xmax)$

```
IntVar[] x = {x1, x2, ..., xn};
Max(x, Xmin);
or
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Max(x, Xmax);
```

knapsack$(profits, weights, quantity, knapsackCapacity, knapsackProfit)$

```
int[] profits = {p1, p2, ..., pn};
int[] weights = {w1, w2, ..., wn};
IntVar[] quantity = {q1, q2, ..., qn};
IntVar knapsackCapacity = new IntVar(...);
IntVar knapsackProfit = new IntVar(...);
Knapsack(profits, weights, quantity, knapsackCapacity, knapsackProfit);
```

geost$(objects, constraints, shapes)$

```
IntVar xOrigin = new IntVar(store, "x1", 0, 20);
IntVar yOrigin = new IntVar(store, "y1", 0, 5);
IntVar shapeNo = new IntVar(store, "s1", 1, 1);
IntVar startGeost = new IntVar(store, "start"+1, 0, 0);
IntVar durationGeost = new IntVar(store, "duration"+1, 1, 1);
IntVar endGeost = new IntVar(store, "end"+1, 1, 1);
IntVar[] coords = {xOrigin, yOrigin};
int objectId = 1;
GeostObject o = new GeostObject(objectId, coords, shapeNo, startGeost,
durationGeost, endGeost);
ArrayList<GeostObject> objects = new ArrayList<GeostObject>();
objects.add(o);
int[] origin = {0, 0};
int[] length = {10, 2};
Shape shape = new Shape(j, new DBox(origin, length));
ArrayList<Shape> shapes = new ArrayList<Shape>();
shapes.add(shape);
int[] dimensions = {0, 1};
NonOverlapping constraint = new NonOverlapping(objects, dimensions);
ArrayList<ExternalConstraint> constraints = new ArrayList<ExternalConstraint>();
constraints.add(constraint);
store.impose(new Geost(objects, constraints, shapes));
```

regular$(fsm, [x_1, x_2, \ldots, x_n])$

```
FSM fsm = new FSM();
IntVar[] x = {x1, x2, ..., xn};
Regular(fsm, x);
```

sequence$([x_1, x_2, \ldots, x_n], set, q, min, max)$

```
IntVar[] x = {x0, x1 ...xn};
IntervalDomain set = new IntervalDomain(...);
int q, main, max;
Sequence(x, set, q, min, max);
```

stretch$(values, min, max, [x_1, x_2, \ldots, x_n])$

```
int[] values, main, max;
IntVar[] x = {x0, x1 ..., xn};
Stretch(values, min, max, x);
```

values$([x_1, x_2, \ldots, x_n], count)$

```
IntVar[] x = {x0, x1 ..., xn};
IntVar count = new IntVar(...);
Values(x, count);
```

lex$([[x_{11}, x_{12}, \ldots, x_{1n}], \ldots, [x_{k1}, x_{k2}, \ldots, x_{km}]])$

```
IntVar[][] x = {{x0, x1 ..., xn}, ...};
Lex(x);
```
or
```
Lex(x, true);
```

soft-alldifferent$([x_1, x_2, \ldots, x_n], cost, violation_measure)$

```
IntVar[] x = {x0, x1 ..., xn};
IntVar count = new IntVar(...);
SoftAlldifferent(x, cost, ViolationMeasure.DECOMPOSITION_BASED);
```
or
```
SoftAlldifferent(x, cost, ViolationMeasure.VARIABLE_BASED);
```

soft-GCC$([x_1, x_2, \ldots, x_n], hardCounters, countedValues, softCounters, cost, violation_measure)$

```
IntVar[] x = {x0, x1 ..., xn};
IntVar[] hardCounters = {h1, h2, ..., hn};
int[] countedValues = {v1, v2, ..., vn};
IntVar[] softCounters = {s1, s2, ..., sn};
SoftGCC(x, hardCounters, countedValues, softCounters, cost, ViolationMeasure.VALUE_BASED);
```
or
other constructors (see API specification).

Appendix B

JaCoP search methods

B.1 Variable and value selection for FDVs

- **value selection methods**

Indomain method	Description
`IndomainMin`	selects a minimal value from the current domain of FDV
`IndomainMax`	selects a maximal value from the current domain of FDV
`IndomainMiddle`	selects a middle value from the current domain of FDV and then left and right values
`IndomainRandom`	selects a random value from the current domain of FDV
`IndomainSimpleRandom`	faster than `IndomainRandom` but does not achieve uniform probability
`IndomainList`	uses values in an order provided by a programmer if values not specified uses default indomain method
`IndomainHierarchical`	uses indomain method based provided variable-indomain mapping

- **variable selection methods**

Comparator	Description
`SmallestDomain`	selects FDV which has the smallest domain size
`MostConstrainedStatic`	selects FDV which has most constraints assign to it
`MostConstrainedDynamic`	selects FDV which has the most pending constraints assign to it
`SmallestMin`	selects FDV with the smallest value in its domain
`LargestDomain`	selects FDV with the largest domain size
`LargestMin`	selects FDV with the largest value in its domain
`SmallestMax`	selects FDV with the smallest maximal value in its domain
`MaxRegret`	selects FDV with the largest difference between the smallest

B.2 Variable and value selection for set variables

- **value selection methods**

Indomain method	Description
IndomainSetMin	selects a minimal value from not yet assigned values for set variable
IndomainSetMax	selects a maximal value from not yet assigned values for set variable
IndomainSetRandom	selects a random value from not yet assigned values for set variable

- **variable selection methods**

Comparator	Description
MinCardDiff	selects set variable which has the smallest difference in cardinality between lub and glb.
MaxCardDiff	selects set variable which has the greatest difference in cardinality between lub and glb.
MinGlbCard	selects set variable which has the glb with the smallest cardinality.
MaxGlbCard	selects set variable which has the glb with the greatest cardinality.
MinLubCard	selects set variable which has the lub with the smallest cardinality.
MaxLubCard	selects set variable which has the lub with the greatest cardinality.
MostConstrainedStatic	selects set variable which has most constraints assign to it.
MostConstrainedDynamic	selects set variable which has the most pending constraints assign to it.

B.3 Search methods

We specify search methods for finite domain variables (`IntVar`). Similar methods can be defined for set variables (`SetVar`).

- **Search for a single solution with *list of variables***

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

- **Search for a single solution with *list of list of variables***

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

- **Search for all solutions**

  additional switches for search for all solutions.

```
label.getSolutionListener().searchAll(true);
// record solutions; if not set false
label.getSolutionListener().recordSolutions(true);
boolean result = label.labeling(store, select);
```

  To be able to print found solutions during search the following solution listener has to be added to the search.

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

  The found solutions can also be printed after search is completed using the following statement.

```
label.printAllSolutions();
```

- **Search for optimal solution**

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

B.4 Important methods for search plug-ins

- **solution listener** – `SimpleSolutionListener`

  important methods

  – `printAllSolutions()`
  – `getSolutions()`
  – `solutionsNo()`
  – `recordSolutions(boolean status)`
  – `searchAll(boolean status)`
  – `executeAfterSolution(Search search, SelectChoicePoint select)`
  – `setChildrenListeners(SolutionListener child)`

- **time-out listener** – one can set customized time-out listener that implements `TimeOutListener` interface to perform specific actions at time-out (e.g., print information). Method `executedAtTimeOut(int solutionsNo)` will be executed at time-out.

Appendix C

JaCoP debugging facilities

C.1 Available switches

Most important debugging facility is made available through the Boolean variables of the Switch class. In order to use this you have to have specially compiled JaCoP library where all master switches are turn on. This reduces the efficiency of the JaCoP, but gives debugging facilities. Assuming that you have turn on all master switches then you can use different switches to obtain the desired debugging information. Please, consult file `JaCoP/core/Switches.java` for more information.

C.2 CPviz interface

JaCoP can generate trace in a format accepted by CPviz, an open-source visualization toolkit for finite domain constraint programming (`http://sourceforge.net/projects/cpviz/`). This functionality is provided by class `JaCoP.search.TraceGenerator.java` and it is added to search. The simplest method to do it is to simply add the following line in your program.

```
TraceGenerator<IntVar> select =
    new TraceGenerator<IntVar>(search, varSelect, vars);
```

where `search` is the search method for your problem, `varSelect` is the variable and value selection method of your search and `vars` is an array of variables to be traced.

The program that extends search with `TraceGenerator` will generate two files (by default named `tree.xml` and `vis.xml`) that register all search decisions and variables and their domains or values. CPviz program can use these files and generate visualization for the search.

The next steps requires installation of CPviz software.

The generation of visual information is done by issuing the following commands.

```
mkdir viz.out
java -cp <path to CPviz>/viz/bin/ ie.ucc.cccc.viz.Viz config.xml \
              tree.xml vis.xml
```

File `config.xml` defines the configuration. An example file is presented below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated by KKU -->
<configuration version="1.0" directory="viz.out">
    <tool show="tree" fileroot="tree" repeat="all"/>
    <tool show="viz" fileroot="viz"/>
</configuration>
```

For more details refer to CPviz documentation.

In the case of this configuration, the files will be generated in directory `viz.out`. Please, note that this directory *must* exist for CPviz to work correctly.

The visualization is provided by issuing the following command.

```
java -cp batik.jar:jhall.jar:<path to cpviz>/viztool/src \
           components.InternalFrame
```

where `batik.jar` and `jhall.jar` are separate software packages that must be installed separately.

Once the visualization tool is started one has to open file `viz.out/aaa.idx`.

An example of a screen dump for sudoku puzzle model is depicted in Figure C.1.

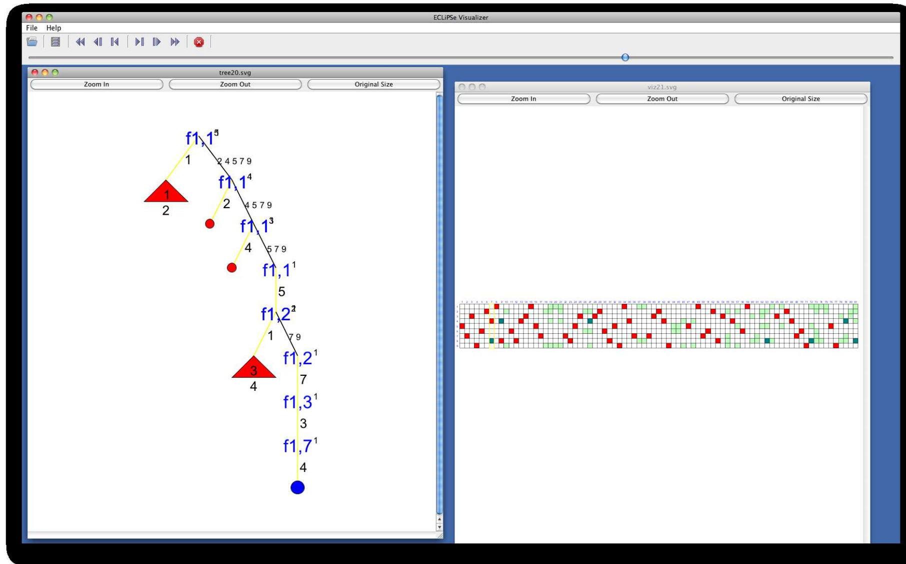

Figure C.1: An example screen dump for search visualization.

Bibliography

[1] N. Beldiceanu, E. Bourreau, H. Simonis, and P. Chan. Partial search strategy in CHIP. Presented at 2nd Metaheuristic International Conference MIC97, Sophia/Antipolis, France, July 21–24, 1997.

[2] K. C. Cheng and R. H. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 509–523, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] COSYTEC. *CHIP System Documentation*, 1996.

[4] D. Diaz. *GNU Prolog A Native Prolog Compiler with Constraint Solving over Finite Domains, Edition 1.7, for GNU Prolog version 1.2.16*, September 2002.

[5] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *An International Journal on Constraints*, 1(3):191–244, 1997.

[6] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *IJCAI-95*, Montreal, Canada, 1995.

[7] I. Katriel, M. Sellmann, E. Upfal, and P. Van Hentenryck. Propagating knapsack constraints in sublinear time. In *AAAI*, pages 231–236, 2007.

[8] Ch. Lecoutre. Optimization of simple tabular reduction for table constraints. In *CP '08: Proceedings of the 14th international conference on Principles and Practice of Constraint Programming*, pages 128–143, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Yuri Malitsky, Meinolf Sellmann, and Radoslaw Szymanek. Filtering bounded knapsack constraints in expected sublinear time. In *AAAI Conference on Artificial Intelligence*, 2010.

[10] K. Marriott and P. J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[11] J. F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artiffcial Intelligence (AAAI'98)*, pages 359–366. American Association for Artificial Intelligence, 1998.

[12] J-C. Régin. A filtering algorithm for constraint of difference in CSPs. In *12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, Menlo Park, California, USA, 1994.

[13] Rida Sadek, Emmanuel Poder, Mats Carlsson, Nicolas Beldiceanu, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proceedings of CP'2007: 13th International Conference on Principles and Practice of Constraint Programming*, page 14, Providence, Rhode Island, USA, 2007. Lecture Notes in Computer Science: Vol. 4741.

[14] Ch. Schulte and G. Smolka. Finite domain constraint programming in Oz. A tutorial, 2001. version 1.2.1.

[15] Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Version 3.11*, June 2004.