## Contents

1 **Introduction**  
1.1 Getting started  7  

2 **Using JaCoP library**  11  
2.1 Finite Domain Variables  11  
2.2 Finite domains  12  
2.3 Constraints  12  
2.4 Search for solutions  13  

3 **Constraints**  17  
3.1 Primitive constraints  17  
3.2 Logical and conditional constraints  17  
3.3 Global Constraints  17  
3.3.1 Alldifferent constraints  17  
3.3.2 Circuit constraint  18  
3.3.3 Subcircuit constraint  19  
3.3.4 Element constraint  19  
3.3.5 Distance constraint  20  
3.3.6 Cumulative constraint  20  
3.3.7 Diffn constraint  22  
3.3.8 Min, Max and ArgMin, ArgMax constraints  22  
3.3.9 SumInt, SumBool and LinearInt constraints  23  
3.3.10 Table, ExtensionalSupport and ExtensionalConflict constraints  24  
3.3.11 Assignment constraint  25  
3.3.12 Count constraint  25  
3.3.13 AtLeast and AtMost constraints  26  
3.3.14 Values constraint  26  
3.3.15 Global cardinality constraint (GCC)  26  
3.3.16 Among and AmongVar  27  
3.3.17 Regular constraint  27  
3.3.18 Knapsack constraint  29  
3.3.19 Geost constraint  30  
3.3.20 NetworkFlow constraint  32  
3.3.21 Binpacking  36  
3.3.22 LexOrder  36  
3.3.23 ValuePrecede  37  
3.3.24 Member  38  
3.4 Decomposed constraints  39  

---

2.1 Finite Domain Variables  11  
2.2 Finite domains  12  
2.3 Constraints  12  
2.4 Search for solutions  13  

3.1 Primitive constraints  17  
3.2 Logical and conditional constraints  17  
3.3 Global Constraints  17  
3.3.1 Alldifferent constraints  17  
3.3.2 Circuit constraint  18  
3.3.3 Subcircuit constraint  19  
3.3.4 Element constraint  19  
3.3.5 Distance constraint  20  
3.3.6 Cumulative constraint  20  
3.3.7 Diffn constraint  22  
3.3.8 Min, Max and ArgMin, ArgMax constraints  22  
3.3.9 SumInt, SumBool and LinearInt constraints  23  
3.3.10 Table, ExtensionalSupport and ExtensionalConflict constraints  24  
3.3.11 Assignment constraint  25  
3.3.12 Count constraint  25  
3.3.13 AtLeast and AtMost constraints  26  
3.3.14 Values constraint  26  
3.3.15 Global cardinality constraint (GCC)  26  
3.3.16 Among and AmongVar  27  
3.3.17 Regular constraint  27  
3.3.18 Knapsack constraint  29  
3.3.19 Geost constraint  30  
3.3.20 NetworkFlow constraint  32  
3.3.21 Binpacking  36  
3.3.22 LexOrder  36  
3.3.23 ValuePrecede  37  
3.3.24 Member  38  
3.4 Decomposed constraints  39
## CONTENTS

3.4.1 Sequence constraint ......................................... 39  
3.4.2 Stretch constraint ............................................. 39  
3.4.3 Lex constraint .................................................. 40  
3.4.4 Soft-Alldifferent ................................................ 40  
3.4.5 Soft-GCC .......................................................... 41  

4  Set Solver .......................................................... 43  
4.1 Set Variables and Set Domains ................................. 43  
4.2 Set Constraints .................................................... 44  
4.3 Set Search .......................................................... 44  

5  Floating Point Solver ............................................. 45  
5.1 Floating Point Variables and Floating Point Domains .... 45  
5.2 Floating Point Constraints ..................................... 45  
5.3 Floating Point Search ............................................ 46  
5.4 Example ............................................................. 47  
5.5 Experimental Extensions ...................................... 47  
5.5.1 Derivatives ....................................................... 48  
5.5.2 Multivariate Interval Newton Method .................... 48  

6  Search ............................................................... 51  
6.1 Depth First Search ................................................. 51  
6.2 Restart search ..................................................... 52  
6.3 Priority search ..................................................... 53  
6.4 Search plug-ins .................................................... 55  
6.5 Credit search ....................................................... 56  
6.6 Limited discrepancy search .................................... 57  
6.7 Combining search ................................................ 57  

7  SAT solver .......................................................... 59  
7.1 Boolean expressions translations .............................. 59  
7.2 FDV encoding and constraint translations .................. 60  

8  Flatzinc .............................................................. 63  
8.1 Minizinc Installation ............................................. 63  
8.2 Minizinc Compilation and Execution .......................... 63  
8.3 Flatzinc Extensions .............................................. 65  
8.3.1 Flatzinc Loader ................................................ 65  
8.3.2 Floating-Point Minimization ............................... 65  

9  Scala Wrapper ........................................................ 67  
9.1 Example .............................................................. 67  
9.2 Scala Model ........................................................ 69  
9.3 Variables ............................................................ 69  
9.4 Constraints ........................................................ 69  
9.5 Search ............................................................... 70  
9.6 Other Methods ..................................................... 70
## CONTENTS

### A  JaCoP constraints
- A.1  Arithmetic constraints ................................................. 75
- A.2  Set constraints .......................................................... 76
- A.3  Floating point constraints ............................................. 77
- A.4  Logical, conditional and reified constraints ....................... 78
- A.5  Global constraints ...................................................... 79

### B  JaCoP search methods
- B.1  Variable and value selection for IntVar ............................ 87
- B.2  Variable and value selection for SetVar ........................... 88
- B.3  Variable selection for FloatVar ...................................... 89
- B.4  Search methods .......................................................... 89
- B.5  Important methods for search plug-ins ............................. 90

### C  JaCoP debugging facilities
- C.1  Simple trace facilities .................................................. 91
- C.2  CPviz interface .......................................................... 92

### D  JaCoP Execution Properties
- D.1  Java specific ............................................................ 95
- D.2  Flatzinc specific ....................................................... 95
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 [18].

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 [4, 25, 6, 22]. 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 org.jacop.core.*;
import org.jacop.constraints.*;
import org.jacop.search.*;
```

Obviously, different Java IDE (Eclipse, IntelliJ, NetBeans, etc.) and pure Java build tools (e.g., Ant, Maven) can be used for JaCoP based application development.
CHAPTER 1. INTRODUCTION

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.

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

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

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

Figure 1.1: An example graph.
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.

```java
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`). Clearly, 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.

```java
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:

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

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

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.

```java
IntVar x = new IntVar(store, "X", 1, 100);
x.addDom(120, 160);
```
CHAPTER 2. USING JACOP LIBRARY

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 (called 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.

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

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

```java
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 6).

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

```java
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.

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

or

```java
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 constraints is achieved calling the following method from class `Store`.

```java
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.

```java
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.

```java
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.

```java
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.

```java
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-search. For more details, see section 6.
Chapter 3

Constraints

IMPORTANT:
All constraints must be posed with `impose` method when domains of all their variables are fully defined.

3.1 Primitive constraints

JaCoP offers a set of primitive constraints that include basic arithmetic operations (+, −, *, /) as well as basic relations (=, ≠, <, ≤, >, ≥). Subtraction is not provided explicitly, but since constraints define relations between variables, it can be defined using addition. For detailed 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, half-reified (implication) and if-then-else (Conditional) constraints. For detailed list of these constraints see appendix A.4.

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

```java
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);
```
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 \([16]\). Finally, constraint Alldistinct implements a generalized arc consistency as proposed by Régis \([20]\).

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

```java
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.

```java
store.impose( new Alldifferent(v) );
a :: {2..3}, b :: {2..3}, c :: {1..3}
```

and

```java
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 \( c \).

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

The constraints will produce the following results after consistency enforcement.

```java
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 successor 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
3.3. GLOBAL CONSTRAINTS

```
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 Subcircuit constraint

Subcircuit constraint tries to enforce that FDVs which represent a directed graph will create a subcircuit. The graph is represented by the FDV domains in the same way as for the Circuit constraint. The result defines a subcircuit represented by values assigned to the FDVs. Nodes that do not belong to a subcircuit have the value pointing to itself.

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 Subcircuit(v);
store.impose(ctr);
```

can find a circuit [2, 1, 3], meaning that node 1 is connected to 2 and 2 to 1 while node 3 is not in the subcircuit and has a value 3.

All solutions to this constraint are [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1], where the first solution represents a solution with no subscircuits and subsequent solutions define found subscircuits.

3.3.4 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\textsuperscript{1}`. The initial domain of `v` will be pruned to `v :: {3, 10, 44}` after the initial 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}`.

\textsuperscript{1}one can also use form `store.impose(Element.choose(i, el, v));`
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.

There are also two other versions of element constraint that implement a version of bounds consistency, `ElementIntegerFast` and `ElementVariableFast`. This constraints make faster computations of consistency methods but may not remove all inconsistent values.

### 3.3.5 Distance constraint

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

The example below

```java
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.

### 3.3.6 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.

```java
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:

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

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 to \( \text{Limit} \), i.e.

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

There is a version of \text{Cumulative} that only requires that the sum of \( AR_k \leq \text{Limit} \) and it is defined with additional flag as follows. In this constraint, first two flags specify to make propagation based on edge finding algorithm and profile while the last flag informs the solver to not apply requirement (3.2).

\begin{verbatim}
import org.jacop.constraints.Cumulative;
Constraint ctr = Cumulative(o, d, r, Limit, true, true, false)
\end{verbatim}

There exist also a newer implementation of cumulative constraints based on methods presented in [27, 26, 23]. There are three following cumulative constraints.

- \text{CumulativeBasic} - does time-tabling propagation,
- \text{Cumulative} - does time-tabling and edge finding propagation, and
- \text{CumulativeUnary} - a specialized version for unary resources and it does in addition to time-tabling and edge finding (specialized for unary resource) propagations, not-first-not-last, detectable propagations.

All propagations, except time-tabling, are done on binary trees and have the following complexities, where \( n \) is the number of tasks and \( k \) is the number of distinct resource values for tasks. Edge finding \( O(k \cdot n \cdot \log n) \), not-first-not-last, detectable and overload \( O(n \cdot \log n) \).

An example of using this implementation for \text{Cumulative} constraint is as follows.

\begin{verbatim}
import org.jacop.constraints.cumulative.*;
Constraint ctr = Cumulative(o, d, r, Limit)
\end{verbatim}
3.3.7 Diffn constraint

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

```java
import org.jacop.constraints.diffn.*;
IntVar[][] rectangles = {{O11, O12, L11, L12}, ...,
                        {On1, On2, Ln1, Ln2}};
Constraint ctr = new Diffn(rectangles)
```

Alternatively, the constraint can be specified using four lists of origins and lengths of rectangles.

```java
import org.jacop.constraints.diffn.*;
IntVar[][] rectangles = {{O11, O21, ..., On1},
                        {O12, O22, ..., On2},
                        {L11, L21, ..., Ln1},
                        {L12, L22, ..., Ln2}};
Constraint ctr = new Diffn(rectangles)
```

The special case for the constraint is when one length becomes zero. In this case, the constraint places zero-length rectangles between other rectangles, that is the zero-length rectangle cannot be placed anywhere. To make possible, the so called, non-strict placement, the constraint has possibility to use the last Boolean parameter equal \false, meaning non-strict placement.

The Diffn 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.

Simplified version of the constraint that logically enforces no overlapping conditions on rectangles is available as well and is called Nooverlap. This constraint has weaker pruning but is faster for simple problems. DiffnDecomposed is the decomposed version of the constraint that combines no overlapping constraint with two cumulative constraints (in direction \(x\) ands \(y\)).

3.3.8 Min, Max and ArgMin, ArgMax 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

```java
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);
```
3.3. GLOBAL CONSTRAINTS

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).

The related constraints ArgMin ArgMax find an index of the minimal and maximal element respectively. In the standard version the first position of the minimal or maximal element is constrained. Indexing starts from 1 if the parameter offset is not used.

3.3.9 SumInt, SumBool and LinearInt constraints

These constraints enforce a (weighted) sum of elements of FDVs’ vector

SumInt and SumBool constraint is defined as follows \( x_1 + x_2 + \cdots + x_n \mathrel{\mathcal{R}} \text{sum} \), where all \( x_i \) and \( \text{sum} \) are FDVs and \( \mathcal{R} \in \{<,\leq,>,\geq,=,\neq\} \). SumBool has additional requirement that all FDVs must have domain 0..1.

The weighted sum is provided by LinearInt constraint defined as follows \( w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n \mathrel{\mathcal{R}} \text{sum} \), where all \( w_i \) and \( \text{sum} \) are integer constants, \( x_i \) are FDVs and \( \mathcal{R} \in \{<,\leq,>,\geq,=,\neq\} \). There exist also a constructor for LinearInt that makes it possible to give \( \text{sum} \) as a FDV. In this case the constraint is transformed to its normal form where FDV is added to the list of variables with weight \(-1\) and the result is 0. To define, for example, constraint \( w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = s \) where \( s \) is FDV, the constraint will be internally transformed to \( w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + (-1) \cdot s = 0 \).

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 SumInt(v, "==", sum);
Store.impose(ctr);

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

Constraints SumInt, SumBool and LinearInt are primitive constraints. It means that they can be an argument in other constraints, such as Reified, conditional and logical constraints.

For example, the linear constraint can be used in reification, as follows.

Store store = new Store();
IntVar a = new IntVar(Store, "a", 1, 3);
IntVar b = new IntVar(Store, "b", 1, 3);
IntVar c = new IntVar(Store, "c", 1, 3);
IntVar[] v = {a, b, c};
PrimitiveConstraint ctr =
    new LinearInt(store, v, new int[] {2, 3, -1}, "<", 10);
BooleanVar b = new BooleanVar(store);
store.impose(new Reified(ctr), b);
Primitive constraints LinearInt and SumInt implement pruning methods for large linear constraints presented in [10]. There exist also constraint LinearIntDom that implements domain consistency for linear constraint. This constraint has exponential complexity in worst case and need to be used carefully. This constraint can be useful when computing an index from two dimensional array to a vector for element constraint.

Constraints Sum, SumWeight and SumWeightDom extend class Constraint and are deprecated.

3.3.10 Table, ExtensionalSupport and ExtensionalConflict constraints

There exist several implementation of these constraints. Table and SimpleTable constraints use compact-table algorithms proposed in [5]. ExtensionalSupport and ExtensionalConflict are 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 [3] while ExtensionalSupportSTR uses simple tabular reduction (STR) and the method proposed in [15].

Extensional support, extensional conflict and table constraints define relations between $n$ FDVs. All 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 and table 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.

```java
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}}));
```
### 3.3.11 Assignment constraint

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

For example, the constraint

```java
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 $\text{Assignment}(x, y, 1)$ will index variables from 1 instead of default value 0.

### 3.3.12 Count constraint

Count constraint counts number of occurrences of value $\text{Val}$ on the list $\text{List}$ in FDV $\text{Var}$.

For example, the constraint

```java
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 $\text{Var} :: {1..2}$.

If variable $\text{Var}$ will be constrained to 1 then JaCoP will produce produces $\text{List}_1 :: {0..1}$.

NOTE! The position for parameters in constraint Count is changed comparing to previous versions.
CHAPTER 3. CONSTRAINTS

3.3.13 AtLeast and AtMost constraints

The two primitive constraints AtLeast and AtMost enforces number of occurrences of a give value on a list of variables.

AtLeast constraint requires that the number of values must be at least equal a given count value and constraint AtMost requires that the number of values must be at most equal a given count value. Constraints are primitive and can be used as parameters to other constraints, such as logical or reified.

3.3.14 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.

```java
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.

3.3.15 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.

```java
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);
```
The GCC constraint will allow only the following five combinations of $x$ variables:

- $[x_0=-1, x_1=0, x_2=2]$,
- $[x_0=-1, x_1=1, x_2=2]$,
- $[x_0=-1, x_1=2, x_2=0]$,
- $[x_0=-1, x_1=2, x_2=1]$, and
- $[x_0=-1, x_1=2, x_2=2]$.

### 3.3.16 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.

```java
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.

```java
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));
```

### 3.3.17 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.

![Automaton Diagram](image)

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

```java
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();
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));

3.3.18 Knapsack constraint

Knapsack constraint specifies knapsack problem. This implementation was inspired by the paper [12] and published in [17]. 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

\[2\]The constraint is based on Wadeck Follonier implementation carried out as his work on a student semester project.
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 a given profit. It is defined in JaCoP as follows.

\[
\text{Knapsack}(\text{int[]} \text{ profits, int[]} \text{ weights, IntVar[]} \text{ quantity, IntVar knapsackCapacity, IntVar knapsackProfit})
\]

It can be formalized using the following constraints.

\[
\sum_i \text{weights}[i] \cdot \text{quantity}[i] = \text{knapsackCapacity} \quad (3.3)
\]

\[
\sum_i \text{profits}[i] \cdot \text{quantity}[i] = \text{knapsackProfit} \quad (3.4)
\]

\[
\forall_i : \text{weights}[i] \in \mathbb{Z}_{>0} \land \forall_i : \text{profits}[i] \in \mathbb{Z}_{>0} \quad (3.5)
\]

### 3.3.19 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 [21]. 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.

**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.

![Shifted box in 2 dimensions. Reference origin is denoted by the black square.](image)

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.

\[
\text{DBox sbox = new DBox(new int[] \{0,0\}, new int[] \{2,1\});}
\]
Definition 2  A shape \( s \) is a set of shifted boxes. It has a unique identifier \( s.sid \).

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 sboxes, depicted in Figure 3.4.

```java
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.

```java
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.

```java
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.

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

### 3.3.20 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,
- \( l : A \to \mathbb{Z}_{\geq 0} \) is the lower capacity function on the arcs,
- \( u : A \to \mathbb{Z}_{\geq 0} \) is the upper capacity function on the arcs,
- \( c : A \to \mathbb{Z} \) is the flow cost-per-unit function on the arcs,
- \( b : N \to \mathbb{Z} \) 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:

$$\begin{align*}
\min z(x) &= \sum_{(i,j) \in A} c_{ij}x_{ij} \\
\text{s.t.} & \quad l_{ij} \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in A, \\
& \quad \sum_{j: (j,i) \in A} x_{ji} - \sum_{j: (i,j) \in A} x_{ij} = b_i \quad \forall i \in N
\end{align*}$$

The network is built with NetworkBuilder class using node, defined by class `org.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.

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

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

```java
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.

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

or

```java
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.

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

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

The constraint is finally posed using the following method.

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

For example, Figure 3.5 presents 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
3.3. **GLOBAL CONSTRAINTS**

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 \( x.\text{min}() \), denoted as \( x_{\text{min}} \). Moreover, the domain of variable \( sv \) is contained within domain \( d \) if and only if the actual flow on a given arc is maximal as specified by initial value \( x.\text{max}() \), denoted as \( x_{\text{max}} \). It is enforced by the following rules.

\[
\begin{align*}
sv.\text{dom}() \subseteq d & \Rightarrow x = x_{\text{max}}, \\
x < x_{\text{max}} & \Rightarrow sv.\text{dom}() \cap d = \emptyset, \\
sv.\text{dom}() \cap d = \emptyset & \Rightarrow x = x_{\text{min}}, \\
x > x_{\text{min}} & \Rightarrow sv.\text{dom}() \subseteq d,
\end{align*}
(3.9)
\]

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.

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

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)));
```

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.
CHAPTER 3. CONSTRAINTS

3.3.21 Binpacking

Binpacking constraint defines a problem of packing bins with the specified items of a given size. Each bin has a defined load. The constraint is defined as follows.

```
Binpacking(IntVar[] item, IntVar[] load, int[] size)
```

where `item` means the bin number assigned to item at position `i`, `load` defines load of bin `i` as finite domain variable (min and max load) and `size` defines items `i` size.

It can be formalize using the following formulation.

\[ \forall i \sum_{j \text{ where } stem[j]=i} size[j] = load[i] \] (3.11)

The binpacking constraint implements methods and algorithms proposed in [24].

3.3.22 LexOrder

The LexOrder constraint enforces ascending lexicographic order between two vectors that can be of different size. The constraints makes it possible to enforce strict ascending lexicographic order, that is the first vector must be always before the second one in the lexicographical order, or it can allow equality between the 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 two vectors of 0/1 variables.

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

IntVar[] y = new IntVar[N];
for (int i = 0; i < y.length; i++)
    y[i] = new IntVar(store, "y["+i+"]", 0, 1);

store.impose(new LexOrder(x, y, true));
```

The result is as follows.

\[
\begin{align*}
[0, 0, 0], [0, 0, 1] \\
[0, 0, 0], [0, 1, 0] \\
[0, 0, 0], [0, 1, 1] \\
[0, 0, 0], [1, 0, 0] \\
[0, 0, 0], [1, 0, 1] \\
[0, 0, 0], [1, 1, 0] \\
[0, 0, 0], [1, 1, 1] \\
[0, 0, 1], [0, 1, 0] \\
[0, 0, 1], [0, 1, 1] \\
[0, 0, 1], [1, 0, 0] \\
[0, 0, 1], [1, 0, 1] \\
[0, 0, 1], [1, 1, 0]
\end{align*}
\]
3.3. GLOBAL CONSTRAINTS

The algorithm of this constraint is based on the algorithm presented in [8].

3.3.23 ValuePrecede

It defines value precedence constraint for integers. Value precedence is defined for integer value $s$, integer value $t$ and an integer sequence $x = [x_0, ..., x_{n-1}]$ and means the following. If there exists $j$ such that $x_j = t$, then there must exist $i < j$ such that $x_i = s$. The algorithm is based on paper [14].

For example, the following code defines that 1 has to always proceed 4 in the sequence of four variables $x$ (ValuePrecede) and, in addition $x[2] = 4$ (constraint XeqC).

```java
int N = 4;

IntVar[] x = new IntVar[N];
for (int i = 0; i < N; i++) {
    x[i] = new IntVar(store, "x["+i+"]", 1, N);
}
store.impose(new XeqC(x[2], 4));
store.impose(new ValuePrecede(1, 4, x));
```

The resulting 24 solutions is as follows.

```plaintext
```
3.3.24 Member

Member constraint enforces that an element \( e \) is present on a list \( x \) and is defined as follows.

\[
\text{Member(IntVar[]} \ x, \text{ IntVar} \ e)\
\]

where \( x \) is a list of variables and \( e \) is the element. The constraint is primitive and can be used in different context as a parameter of other 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 \( \text{min} \) and \( \text{max} \) allowed values. Value \( q \), \( \text{min} \) and \( \text{max} \) 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.

```java
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 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 \( \text{min} \) and maximum \( \text{max} \) 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 these restrictions.

```java
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
));
```
CHAPTER 3. CONSTRAINTS

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.

```plaintext
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.

```plaintext
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);
```
3.4. **DECOMPOSED CONSTRAINTS**

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 `softCounters`.

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 Solver

JaCoP provides a library of set constraints in library org.jacop.set. This implementation is based on set intervals and related operations as originally presented in [9] and the first version has been implemented as a diploma project1. 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 org.jacop.core.IntervalDomain and a set domain as abstract class org.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) ⊆ lub(d). glb(d) can be considered as a set of all elements that are members of the set and lub(d) specifies the largest possible set.

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

```java
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

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

and an empty set domain as

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

4.2 Set Constraints

JaCoP implements a 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 as conditional and logical. Other constraints are treated as ordinary JaCoP constraints.

Consider the following code that uses union constraint.

```java
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\}..{4} \(\cup\) \{2\}..{5} = \{1\}..{10} and produces \{1\}..{4} \(\cup\) \{2\}..{5} = \{1..2\}..{1..5}. This represents 108 possible solutions.

4.3 Set 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 DepthFirstSearch 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 also work.

An example search can be specified as follows.

```java
Search<SetVar> search = new DepthFirstSearch<SetVar>();
SelectChoicePoint<SetVar> select = new SimpleSelect<SetVar>(
    vars,
    new MinLubCard<SetVar>(),
    new MaxGlbCard<SetVar>(),
    new IndomainSetMin<SetVar>());
solution.setSolutionListener(new SimpleSolutionListener<SetVar>());
boolean result = search.labeling(store, select);
```
Chapter 5

Floating Point Solver

JaCoP provides a library of floating point constraints in library org.jacop.floats. This implementation is based on floating point intervals and related constraints.

5.1 Floating Point Variables and Floating Point Domains

Floating point domain is defined in a similar way as integer variable domain as an ordered list of floating point intervals in class org.jacop.floats.core.FloatIntervalDomain. This class implements an abstract class org.jacop.floats.core.FloatDomain that in turn extends ordinary JaCoP domain defined in org.jacop.core.Domain. The floating point variable is defined using the floating point interval as follows.

```java
FloatVar f = new FloatVar(store, "f", 0.0, 10.0);
```

This java declaration defines floating point variable f with domain 0.0..10.0.

The float variables are used in floating point constraints, discussed in the next section. Special attention is paid to precision of floating point operations. Basically, JaCoP considers a floating point interval to represent a single value if a difference between max and min values of a variable is lower than an epsilon value (ε). Epsilon is calculated based on a given precision and ulp value (ulp is a floating point specific value know as “unit at last position”). The precision is defined in org.jacop.core.FloatDomain and can be set-up and obtained using following methods FloatDomain.setPrecision(double p) and FloatDomain.precision(). Epsilon is defined as a maximum of ulp and the precision. Class FloatDomain defines also min and max values of floating variables and values of π and e.

5.2 Floating Point Constraints

JaCoP implements a number of floating point constraints specified in appendix A.3. Constraints PeqC, PeqQ, PgtC, PgtQ, PgteqC, PgteqQ, PltC, PltQ, PlteqC, PlteqQ, PneqC, PneqQ, PplusCeqR, PplusQeqR, PminusCeqR, PminusQeqR and LinearFloat 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 division constraint.
CHAPTER 5. FLOATING POINT SOLVER

FloatDomain.setPrecision(1E-4);

FloatVar a = new FloatVar(store, "a", 0.1, 3.0);
FloatVar b = new FloatVar(store, "b", -0.2, 6.0);
FloatVar c = new FloatVar(store, "c", -1000, 1000);

store.impose(new PdivQeqR(a, b, c));

It performs operation \([0.100..3.000] / [-0.200..6.000]\) and produces a pruned domain for \(c::[-1000.0000..-0.5000, 0.0167..1000.0000]\), assuming precision \(10^{-4}\).

5.3 Floating Point Search

Floating point variables will require special search definition. Basically, a domain split search (bisection) is used together with consistency checking to find a solution.

JaCoP standard search for floating point variables still uses DepthFirstSearch but needs to use specific methods for selection of choice points as well as variable selection. org.jacop.floats.search.SplitSelectFloat is used to define how intervals of floating point variables will be split. Default behavior of this method is to split an interval in a middle and try the left part first. If variable leftFirst is set to false the right interval will be checked first. The variables will be selected based on the variable selection heuristic defined as a parameter for SplitSelectFloat class. There exist several methods to select variables specific for FloatVar. They are defined in org.jacop.floats.search. Other general methods that are not specific for a given domain can also be used here, for example org.jacop.search.MostConstrainedStatic.

An example search can be specified as follows.

```java
DepthFirstSearch<FloatVar> search = new DepthFirstSearch<FloatVar>();
SplitSelectFloat<FloatVar> select = new SplitSelectFloat<FloatVar>(
    store, x,
    new LargestDomainFloat<FloatVar>());
search.setSolutionListener(new PrintOutListener<FloatVar>());

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

Minimization, using DepthFirstSearch class, is done as for other variables by calling labeling method with the third parameter defining cost variable. Both IntVar and FloatVar can be used to defined cost criteria for minimization.

JaCoP offers also another minimization method. This method uses cost variable to lead the minimization. The method splits the domain of cost variable into two parts and then checks, using standard DepthFirstSearch labeling method if there is a solution in this part. If there is a solution in this part it continues to split the current interval and checks the left interval again. If there is no solution in this part (depth-first-search return false) it will try the right interval. The search finishes when the cost variable is ground (the difference between max and min is lower or equal epsilon). An example code is presented below.

```java
DepthFirstSearch<FloatVar> search = new DepthFirstSearch<FloatVar>();
SplitSelectFloat<FloatVar> select = new SplitSelectFloat<FloatVar>(
    store, x,
    new LargestDomainFloat<FloatVar>());
search.setSolutionListener(new PrintOutListener<FloatVar>());

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

The following code defines equation $x^2 + \sin(1/x^3) = 0$ together with the search that finds all 318 solutions.

```java
FloatDomain.setPrecision(1.0e-14);
FloatDomain.intervalPrint(false);
Store store = new Store();

FloatVar x = new FloatVar(store, "x", 0.1, 1.0);
FloatVar zero = new FloatVar(store, 0,0);
FloatVar one = new FloatVar(store, 1,1);

// $x^2 + \sin(1/(x^3)) = 0.0$
FloatVar[] temp = new FloatVar[4];
for (int i=0; i<temp.length; i++)
    temp[i] = new FloatVar(store, "temp["+i+"]", -1e150, 1e150);
store.impose(new PmulQeqR(x, x, temp[0]));
store.impose(new PmulQeqR(x, temp[0], temp[1]));
store.impose(new PdivQeqR(one, temp[1], temp[2]));
store.impose(new SinPeqR(temp[2], temp[3]));
store.impose(new PplusQeqR(temp[0], temp[3], zero));

DepthFirstSearch<FloatVar> search = new DepthFirstSearch<FloatVar>();
SplitSelectFloat<FloatVar> s = new SplitSelectFloat<FloatVar>(store,
    new FloatVar[] {x}, null);
search.setSolutionListener(new PrintOutListener<FloatVar>());
search.getSolutionListener().searchAll(true);
search.labeling(store, s);
```

The last solution is $x = 0.65351684593772$.

5.5 Experimental Extensions

JaCoP package org.jacop.floats offers also experimental extensions. They include a method to generate constraints that define the derivative of a function, represented by constraints, and a interval Newton method.
5.5.1 Derivatives

JaCoP can do symbolic (partial) derivatives of functions, that is using the defined constraints for a given function and a variable defining the result of the function, JaCoP will generate constraints that define a function for a derivative of the original function. There is a complication in this process since constraints are not functions and sometimes it is difficult or impossible to find out which constraint defines a function for a given variable. In such situations JaCoP uses a heuristic to find out this but it is also possible to define this explicitly. The following examples explains the process of computing derivatives of constraints.

```java
Set<FloatVar> vars = new HashSet<FloatVar>();
vars.add(x1);
vars.add(x2);
Derivative.init(store);
// Derivative.defineConstraint(x1x1, c0);
// Derivative.defineConstraint(x2x2, c1);
// Derivative.defineConstraint(x1x1x1x1, c2);
// Derivative.defineConstraint(x2x2x2x2, c3);

// ================== df/dx1 =================
FloatVar fx1 = Derivative.getDerivative(store, f, vars, x1);

// ================== df/dx2 =================
FloatVar fx2 = Derivative.getDerivative(store, f, vars, x2);
```

In the above example, JaCoP will generate two partial derivatives $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$. Set `vars` is used to keep all variables of function defined by variable `f`. The function defined by variable `f` is usually defined by several constraints using temporary variables, such as `x1x1`, `x2x2`, `x1x1x1x1` and `x2x2x2x2`. In such situations JaCoP tries, using heuristics, to find out what constraint defines a given temporary variable. Sometimes it is not possible and the programmer needs to define it, as indicated by the commented code and methods `Derivative.defineConstraint`. For example, constraint `c0` defines variable `x1x1`. Method `getDerivative` of class `Derivative` generates a variable that defines the result of the derivative.

Derivatives, can be used for setting additional constraints on solutions. For example, when looking for minimum or maximum of a function or in the interval Newton method.

5.5.2 Multivariate Interval Newton Method

Interval Newton methods combine classical Newton method with interval analysis [13] and are used to provide better bounds on variables.

Traditionally, univariate Newton method computes a solution for equation $f(x) = 0$ using the following approximation.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

This can be extended by using vector `x` and a Jacobian matrix instead of $f'(x)$. Furthermore one can extend it to interval arithmetic. This is implemented in JaCoP.
5.5. EXPERIMENTAL EXTENSIONS

using “constraint” EquationSystem. This constraint takes as parameters a vector of functions and variables of these functions. The constraints computes then all partial derivatives and form a set of equations that are later solved, if possible, by interval Gauss-Seidel method.

This can be formalized as system of linear equations that, when solved, provides bounds on values \((x_{n+1} - x_n)\) of variables of equations.

\[ J_F(x_n)(x_{n+1} - x_n) = -F(x_n) \]

where \(F : \mathbb{R}^k \rightarrow \mathbb{R}^k\) are continuously differentiable functions and \(J_F(x_n)\) is Jacobian matrix.

Below, constraint eqs specifies a set of two functions \(f0\) and \(f1\) of two variables \(x1\) and \(x2\). Constraint eqs will try to prune domains of variables \(x1\) and \(x2\).

```java
EquationSystem eqs = new EquationSystem(store,
    new FloatVar[] {f0, f1}, new FloatVar[] {x1, x2});
store.impose(eqs);
```
Chapter 6

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 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 6.4.

6.1 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.

```java
Search<T> search = new DepthFirstSearch<T>();
SelectChoicePoint<T> select = new SimpleSelect<T>(var,
    varSelect,
    tieBreakerVarSelect
    indomain);

boolean result = search.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 variables 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 choice 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.

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

```java
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 `search.getSolutionListener().solutionsNo()`.

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

```java
IntVar cost;
...
boolean result = search.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 be used. For example, 10s time-out can be set with the following statement.

```java
search.setTimeOut(10);
```

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

### 6.2 Restart search

In some situation classical depth first search algorithm is not best suited for finding a solution or finding an optimal solution. In such situation a so called restart search can be used. This search method runs an ordinary depth first search and then restarts the search from the beginning. Restarting of the search is forced when certain conditions
6.3. PRIORITY SEARCH

are met. In our case, we restart search if a given number of fails occurred. Therefore, restart search takes as parameters, the specification of the depth first search, the calculator for providing a limit on fails and possibly a cost variable. An example code for defining restart search is as follows.

```java
import org.jacop.search.restart.*;

DepthFirstSearch<IntVar> label = new DepthFirstSearch<>();
SelectChoicePoint<IntVar> select = new SplitSelect<>(vars,
        new ActivityMax<IntVar>(store),
        new IndomainMin<IntVar>());

RestartSearch<IntVar> rs = new RestartSearch<>(store, label,
        select,
        new LubyCalculator<IntVar>(2000),
        cost);

boolean result = rs.labeling();
```

Depth first search, label is defined as usually with select method that uses variable selection method ActivityMax, that is it will selects variables that are most often pruned. This makes possible to select variables in different order when restart happens. Restart search uses Luby calculator to give the number of allowed fails. It starts from 2000 fails and then uses a Luby number to multiply by 2000 on consecutive restarts to compute the limit on the number of fails. Other calculators are also available.

The restart search works also with the depth first search method defined as a sequence of several search methods.

It is a good strategy to add a solution listener to the last search or add default solution listener to restart search by calling RestartSearch method `rs.addReporter(new CustomReport(vars))`.

Finally, to make restart search effective one has to define it in such a way that it would use a different variable ordering and/or value assignment on different restarts. In the example above we have used ActivityMax variable selection heuristic that dynamically changes the order of selected variables based on their pruning statistics. The other similar methods are based on accumulated failure count (afc). Value selection can use, for example, random value assignment.

6.3 Priority search

Priority search specifies complex nested searches [7]. A vector of variables is used to select a search that will be explored next. An example code below defines a priority search that based on variables `p` selects one of `dfs[i]` depth-first-searches to execute. All searches are connected and can be considered as one depth-first-search.

```java
IntVar[] p = new IntVar[n];
p[0] = new IntVar(store, "p[0]", 2,2);
p[1] = new IntVar(store, "p[1]", 1,1);
p[2] = new IntVar(store, "p[2]", 3,3);

d = new IntVar[3][2];
```
```java
IntVar[] vs = new IntVar[6];
int k=0;
for (int i = 0; i < n; i++)
    for (int j = 0; j < 2; j++) {
        d[i][j] = new IntVar(store, \"d[{i}][{j}]\", 0, 5);
        vs[k++] = d[i][j];
    }
store.impose(new Alldiff(vs));

DepthFirstSearch[] search = new DepthFirstSearch[3];
for (int i = 0; i < n; i++) {
    DepthFirstSearch<IntVar> dfs = new DepthFirstSearch<>();
    dfs.setSelectChoicePoint(new InputOrderSelect<IntVar>(store, d[i], new IndomainMin<IntVar>()));
    dfs.setPrintInfo(false);
    search[i] = dfs;
}

PrioritySearch<IntVar> label = new PrioritySearch<IntVar>(p, new SmallestMin<IntVar>(), search);
lable.setPrintInfo(false);
lable.setSolutionListener(new Printer());
lable.setAssignSolution(true);
lable.getSolutionListener().recordSolutions(true);

cost = d[1][1];
boolean result = label.labeling(store,cost);

class Printer<T extends Var> extends SimpleSolutionListener<T> {
    public boolean executeAfterSolution(Search<T> search, SelectChoicePoint<T> select) {
        boolean returnType = super.executeAfterSolution(search, select);
        System.out.println(cost);
        System.out.print("d = [ ");
        for (int i = 0; i < d.length; i++)
            System.out.print(java.util.Arrays.asList(d[i]) + " ");
        System.out.println(" ]");
        System.out.println("-------");
        return returnType;
    }
}
```

PrioritySearch extends DepthFirstSearch and takes as arguments a vector of variables used for search selection, variables selection method and a vector of searches. During search it selects a search from a vector of searches and executes it. Selection is controlled by selection variables and specified variable selection method. For the exam-
6.4. SEARCH PLUG-INS

The search plug-ins are selected in the following order dfs[1], dfs[0] and dfs[2]. The resulting solutions, printed by Printer are following.

\[
\begin{align*}
d[1][1] &= 1 \\
\text{d} &= \{ [d[0][0] = 2, d[0][1] = 3] \ [d[1][0] = 0, d[1][1] = 1] \ [d[2][0] = 4, d[2][1] = 5] \ \\
\end{align*}
\]

PrioritySearch can be combined with other search combinators. It can be part of sequence of searches, each search in the vector of searches can also be a sequence of searches. Finally, restart search can use this search or combination of searches.

6.4 Search plug-ins

The search plug-in 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 6.1 lists the search-plugins available in JaCoP and their membership in a respective group.

<table>
<thead>
<tr>
<th>changing search</th>
<th>cannot change search (information sharing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>solution listener</td>
<td>exit listener</td>
</tr>
<tr>
<td>exit child listener</td>
<td>time-out listener</td>
</tr>
<tr>
<td>consistency listener</td>
<td>initialize listener</td>
</tr>
</tbody>
</table>

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 be 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. These 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.

### 6.5 Credit search

Credit search combines credit based exhaustive search at the beginning of the tree with local search in the rest of the tree [2]. 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 6.1 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.

![Credit search example](image)

**Figure 6.1**: Credit search example.

An example of the command which produces the search tree depicted in Fig. 6.1 is
6.6 Limited discrepancy search

Limited discrepancy search (LDS) uses the partial search method proposed in [11]. 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.

```java
Search<IntVar> search = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(var,
        new SmallestDomain<IntVar>(),
        new IndomainMiddle<IntVar>());
LDS<IntVar> lds = new LDS<IntVar>(2);
search.setExitChildListener(lds);
boolean result = search.labeling(store, select);
```

6.7 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.

```java
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);

When several search methods is combined to form a sequential search and the min-
imization is done one has to set for each sub-search a flag informing this search that it
is a part of minimization. It is achieved by the following instruction.

slave.setOptimize(true);
Chapter 7

SAT solver

JaCoP has a SAT solver that can be used together with JaCoP solver. It has been developed by Simon Cruanes and Radosław Szymanek. The SAT solver can be integrated in JaCoP as a SAT constraint and used with JaCoP search methods. This functionality is offered by the `org.jacop.satwrapper.SatWrapper` class. Typical translation of Boolean constraints, used in flatzinc compilations are defined in `org.jacop.satwrapper.SatTranslation`.

7.1 Boolean expressions translations

Typical Boolean formulas can be added to SAT solver using class `SatTranslation`. This class offers translation of typical operations as `and`, `or`, `not`, `xor`, implication, all equalities and inequalities as well as reified versions of these. The following code illustrates use of the translation.

```java
SatTranslation sat = new SatTranslation(store);
sat.impose(); // impose SAT-solver as constraint

BooleanVar a = new BooleanVar(store, "a");
BooleanVar b = new BooleanVar(store, "b");
BooleanVar c = new BooleanVar(store, "c");

// adding SAT clauses
sat.generate_implication(a, b);
sat.generate_and(new BooleanVar[]{a, b}, c);
```

The code generates Boolean clause of the form \((a \rightarrow b) \land (a \land b = c)\) that is translated to CNF Boolean clause supported by the SAT solver.

\[
(\pi \lor b) \land \\
(\pi \lor \pi \lor c) \land \\
(a \lor \pi) \land \\
(b \lor \pi)
\]

The SAT translation performs several tasks. First, it defines SAT literals and binds Boolean variables to these literals. In our example, Boolean variables a, b and c are translated to literals, denoted by the same letter in the above formulas but they are different representations. Second, it adds CNF clauses representing Boolean clauses to
the SAT solver. The SAT wrapper will then propagate changes between CP and SAT solver based on the evaluations of both solvers.

The search for solutions is identical with CP framework. In case of the example, we can make search on variables a, b and c. All solutions to the constraint are \{0, 0, 0\}, \{0, 1, 0\}, \{1, 1, 1\}

### 7.2 FDV encoding and constraint translations

SatWrapper offers a general way to define literals and clauses. The process is divided into two parts. First, each FDV and a inequality relation on it is bound to a literal. Then a clause is defined on the literals. We illustrate this process below.

Assume that we want to define a constraint \((a \leq 10) \rightarrow (b = 3 \land c > 7)\) with IntVar a, b and c. The literals for this constraint are defined as \([a \leq 10]\), \([b = 3]\) and \([c \leq 7]\) (negation of this literal is \(a > 7\)). This is encoded in JaCoP as follows.

```java
IntVar a = new IntVar(store, "a", 0, 100);
IntVar b = new IntVar(store, "b", 0, 100);
IntVar c = new IntVar(store, "c", 0, 100);

SatWrapper wrapper = new SatWrapper();
store.impose(wrapper);

wrapper.register(a);
wrapper.register(b);
wrapper.register(c);

int aLiteral = wrapper.cpVarToBoolVar(a, 10, false);
int bLiteral = wrapper.cpVarToBoolVar(b, 3, true);
int cLiteral = wrapper.cpVarToBoolVar(c, 7, false);
```

In the above method `cpVarToBoolVar` the first parameter is the FDV variable, the second is the rhs value and the third parameter defines type if relation (false for \(\leq\) and true for \(=\)).

When literals are defined we can generate clauses using CNF format, as presented below. First, we translate the clause to CNF form as \((a \leq 10) \lor (b = 3) \land (a \leq 10) \lor (c > 7)\). Then we encode this into JaCoP SAT solver.

```java
IntVec clause = new IntVec(wrapper.pool);

// first clause
clause.add(- aLiteral);
clause.add(bLiteral);
wrapper.addModelClause(clause.toArray());

clause = new IntVec(wrapper.pool);

// second clause
clause.add(- aLiteral);
clause.add(- cLiteral);
wrapper.addModelClause(clause.toArray());
```
In the above code minus in front of the literal in statement clause.add means that the negated literal is added to the clause.
A possible solution generated by JaCoP is \{0, 3, 8 \}.
Chapter 8

Flatzinc

JaCoP has a front-end to flatzinc [19]. It can parse flatzinc files and execute them. Flatzinc files are usually generated from minizinc [19] models and are “flattened” models that specify all variables, constraints and search defined in minizinc models. To fully use power of JaCoP, compilation of minizinc files should use JaCoP library definitions that define translation of minizinc global constraints into JaCoP global constraints.

8.1 Minizinc Installation

The minizinc to flatzinc compiler (available at http://www.minizinc.org/software.html) produces flatzinc files that can include global constraints provided by a certain solver. JaCoP specific definitions are available in src/main/minizinc/org/jacop/minizinc/ at GitHub or at sourceforge. This directory must be used by the minizinc compiler. It needs to be copied from JaCoP minizinc directory (src/jacop/minizinc/org/jacop/minizinc) into the minizinc directory of your minizinc installation and rename it to jacop. This directory is located at ‘PATH_TO_MINIZINC_INSTALLATION’/share/minizinc. You can later use -Gjacop or flag --solver jacop with command minizinc -c. Alternatively one can make JaCoP a default solver.

Installation of JaCoP in minizinc IDE requires additional files. JaCoP configuration file src/main/minizinc/org.jacop.msc (available at GitHub) needs to be copied to IDE installation at ‘PATH_TO_MINIZINC_INSTALLATION’/share/minizinc/solvers. The execution script fzn-jacop (presented later in section 8.2) must be copied to ‘PATH_TO_MINIZINC_INSTALLATION’/bin directory. The path to the jar file needs to be given explicitly or defined in CLASSPATH environment variable. For more details on how to install a new solver in minizinc IDE, please refer to “Minizinc Reference Manual” (section 4.3.5. Solver Configuration Files).

8.2 Minizinc Compilation and Execution

In this section we describe the compilation and execution of minizinc files using command line interface and JaCoP. JaCoP can also be added to minizinc IDE provided by NICTA http://www.minizinc.org/ide/index.html.

Minizinc models can be executed by minizinc command.
Different options can be used. For description of options, please check "Minizinc User Manual".

The typical compilation of a minizinc model is carried out by the `minizinc` command as well. Use option `-c` for that.

```
minizinc -c model.mzn
```

If the model requires data files specified in `dzn` files the command is as follows.

```
minizinc -c model.mzn -d data.dzn
```

The `minizinc` command generates two files. The file with extension `fzn` contains "flattened" model and file with extension `ozn` contains specifications on the output formatting. JaCoP uses only `fzn` file for solving and printing the output in the standard format. This output can be parsed by `minizinc` command and printed using the format specified in minizinc model. This is automatically used in MiniZincIDE but can also be achieved from the command line by using shell pipe ("|"). Typical execution of this model is provided by JaCoP in class `Fz2jacop` and can be done using command line skeleton specified below.

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

The execution with minizinc formatting, using `fzn-jacop` command defined at the end of this section, is done as follows.

```
fzn-jacop model.fzn | minizinc --ozn-file model.ozn
```

Currently the following options are supported (printed using `-h` option).

```
Usage: java org.jacop.fz.Fz2jacop [options] <file>.fzn
Options:
- h, --help
  Print this message.
- a, --all, --all-solutions
- v, --verbose
- t <value>, --time-out <value>
  <value> - time in milisecond.
- s, --statistics
- n <value>, --num-solutions <value>
  <value> - limit on solution number.
- b, --bound - use bounds consistency whenever possible;
  overrides annotation ":: domain" and selects constraints
  implementing bounds consistency (default false).
- sat use SAT solver for boolean constraints.
- cs, --complementary-search - gathers all model, non-defined
  variables to create the final search
- i, --interval print intervals instead of values for floating variables
- p <value>, --precision <value> defines precision for floating operations
  overrides precision definition in search annotation.
- f <value>, --format <value> defines format (number digits after decimal point)
  for floating variables.
- o, --outputfile defines file for solver output
```
It is practical to create a script (called for example flatzinc-to-jacop) that can execute JaCoP on flatzinc file. Below is a simple bash script (assuming jacop.jar is in the current directory), for example.

```bash
#!/bin/bash
exec java -cp jacop.jar org.jacop.fz.Fz2jacop "$@
```

8.3 Flatzinc Extensions

JaCoP offers possibility to extend flatzinc models execution with JaCoP specific methods. They are specified in this section.

8.3.1 Flatzinc Loader

In some situations it is useful to parse the model, create all variables and constraints and then use JaCoP to handle the search and possibly other processing. This functionality is provided by class FlatzincLoader. This class implements methods for loading a flatzinc model and methods for getting information about the model.

An example illustrating how to use this functionality to build a flatzinc solver is implemented in org.jacop.examples.flatzinc.FlatzincSolver.

8.3.2 Floating-Point Minimization

Standard minimization method is based on branch-and-bound method. This means that whenever a solution is found the next solution must have its cost lower than the cost of the current solution. This method work fine for IntVar cost functions but it does not work efficiently for FloatVar. Therefore JaCoP offers another method, called Optimization, described in section 5.3. This method tries first to find a solution for the model. If such solution exists it tries to narrow the interval for the cost variable to left half and searches for a solution. If such solution exists it goes again to the left sub-interval. If there is no solution it tries the right sub-interval. The process stops when the cost variable interval is considered as ground (within a given precision). This method is implemented, for flatzinc models, in org.jacop.examples.flatzinc.FloatMinimize.
Chapter 9

Scala Wrapper

JaCoP can also be used from Scala \[1\]. Scala wrapper (org.jacop.scala) is defined to overload basic operations, equalities and inequalities, define global constraints as well as make search specifications simpler.

9.1 Example

The following example presents a classical CP example SendMoreMoney written in Scala and using JaCoP library.

The example imports JaCoP wrapper in line 1 and use trait jacop. In this example, in lines 5-12 we define finite domain variables used in the model as well as a vector of these variables in line 14. AllDifferent constraint is defined in line 16. Then arithmetic equality between two sums is define in lines 18-19. Finally, constraint that s and m must be greater than zero are defined. Notice, that Scala wrapper overloads all arithmetical operations, such as + in the example, and inequalities and equalities (they are prefixed with #).

Finally, the search is defined for the model. It is build of the method that requests search for a single solution satisfying the model constraints and search specification (method search).

For more details, please check the next subsections of this chapter.

```scala
import org.jacop.scala._

object SendMoreMoney extends App with jacop {

  val s = new IntVar("s", 0, 9)
  val e = new IntVar("e", 0, 9)
  val n = new IntVar("n", 0, 9)
  val d = new IntVar("d", 0, 9)
  val m = new IntVar("m", 0, 9)
  val o = new IntVar("o", 0, 9)
  val r = new IntVar("r", 0, 9)
  val y = new IntVar("y", 0, 9)

  val fd = Array(s, e, n, d, m, o, r, y)
```

Our model uses multiplications and additions to form a weighted sum of variables in lines 18-19. This is decomposed to JaCoP constraints `XmulYeqZ`, `XplusYeqZ` and `XeqY` that results in 39 variables and 25 constraints. The solution found by this program is presented below.

Solution: s = 9 e = 5 n = 6 d = 7 m = 1 o = 0 r = 8 y = 2

Search statistics:
==================
Search nodes : 7
Propagations : 504
Search decisions : 4
Wrong search decisions : 3
Search backtracks : 0
Max search depth : 4
Number solutions : 1

The program can be further improved by using the global constraint intended for such situations. To use JaCoP `LinearInt` constraint one can make the following specification. It results in 10 variables and 6 constraints.

\[
\text{sum}(\text{Array}(s, e, n, d, m, o, r, e), \\
\text{Array}(1000, 100, 10, 1, 1000, 100, 10, 1)) \#=
\text{sum}(\text{Array}(m, o, n, e, y), \text{Array}(10000, 1000, 100, 10, 1))
\]

or to use a single `LinearInt` constraint we can use the following code. It has 9 variables and 5 constraints.

\[
\text{sum}(\text{Array}(s, e, n, d, m, o, r, e, m, o, n, e, y), \\
\text{Array}(1000, 100, 10, 1, 1000, 100, 10, 1,}
\]

```
val result = satisfy( search(fd, input_order, indomain_min),
  printSol() )

statistics

def printSol() = () => {
  print("Solution: ")
  for (v <- fd)
    print( v+" " )
  println
}
```
The solution to the last model is the same as before but uses only 3 search nodes and has 21 propagations. This is caused by more efficient linear constraint and its pre-solving (eliminating multiple variables in the constraint by summing up their weights).

9.2 Scala Model

All variables, constraints and other solver specific data are kept in the store, defined by class Store in JaCoP. Scala wrapper extends Store and defines class Model. Model is created implicitly. If needed one can access it using method getModel and set a new Model using method setModel(new Model()).

9.3 Variables

In Scala wrapper one can define four types of variables: IntVar, BoolVar, SetVar and FloatVar as well as constraints over these variables. Since the store is defined implicitly all variables can be created by specifying their names and domains only (e.g., lines 5-12 in the SendMoreMoney example). BoolVar do not have specification of domain since it is given as domain 0/1.

For convenience, a set of integers is defined as IntSet and can be used to build domains for variables. This class has operations, such as union (+), intersection (*), subtraction (\) and set complement (\^) defined. Therefore, domains can be defined as illustrated on the examples below.

```scala
// definition of x1::{1..10, 13}
val set1 = new IntSet(1,10) + 13
val x1 = new IntVar("x1", set1)

// definition of x2::{1..10, 13..15}
val set2 = new IntSet(1,10) + new IntSet(13, 15)
val x2 = new IntVar("x2", set2)
```

9.4 Constraints

One can use all JaCoP constraints, defined in Java, directly in a Scala program but for convenience some methods are overloaded to provide more intuitive way of specifying constraints.

For IntVar operators, +, -, *, div, mod and \^ are defined. Moreover, the following equality and inequality are defined #=, #\=, #<, #<=, #> and #>=. Constraint that requires a variable value to be in a set is defined as in operator.

Similar operators are defined for FloatVar. In this case, however, mod and in are not applicable. Additionally, there constraints defining mathematical functions ad defined in Table 9.1

Set variables have the following operators: + (union), * (intersection), \ (set subtraction), <= (disjoint sets), in (set inclusion) and method card (set cardinality). Set inequalities are the following: #= (equality), #< (lexicographically less or equal) and #>= (lexicographically greater or equal).
CHAPTER 9. SCALA WRAPPER

### Table 9.1: Floating point constraints.

<table>
<thead>
<tr>
<th>Function</th>
<th>Example</th>
<th>JaCoP constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sum_{i=0}^{N} x_i$</td>
<td>$v = \text{sum(x_vector)}$</td>
<td>$\text{LinearFloat}$</td>
</tr>
<tr>
<td>$\sum_{i=0}^{N} a_i \cdot x_i$</td>
<td>$v = \text{sum(x_vector, a_vector)}$</td>
<td>$\text{LinearFloat}$</td>
</tr>
<tr>
<td>$v =</td>
<td>x</td>
<td>$</td>
</tr>
<tr>
<td>$v = e^x$</td>
<td>$v = \text{exp(x)}$</td>
<td>$\text{ExpPeqR}$</td>
</tr>
<tr>
<td>$v = \ln(x)$</td>
<td>$v = \text{ln(x)}$</td>
<td>$\text{LnPeqR}$</td>
</tr>
<tr>
<td>$v = \sqrt{x}$</td>
<td>$v = \text{sqrt(x)}$</td>
<td>$\text{SqrtPeqR}$</td>
</tr>
<tr>
<td>$v = \sin(x)$</td>
<td>$v = \text{sin(x)}$</td>
<td>$\text{SinPeqR}$</td>
</tr>
<tr>
<td>$v = \cos(x)$</td>
<td>$v = \text{cos(x)}$</td>
<td>$\text{CosPeqR}$</td>
</tr>
<tr>
<td>$v = \cos(x)$</td>
<td>$v = \text{acos(x)}$</td>
<td>$\text{AcosPeqR}$</td>
</tr>
<tr>
<td>$v = \tan(x)$</td>
<td>$v = \text{tan(x)}$</td>
<td>$\text{TanPeqR}$</td>
</tr>
<tr>
<td>$v = \text{atan}(x)$</td>
<td>$v = \text{atan(x)}$</td>
<td>$\text{AtanPeqR}$</td>
</tr>
</tbody>
</table>

Boolean variables have the following operators: $\#=$ (equality), $\&\&$ (and), $\|\|$ (or), $\timesor$ (exclusive or), $\sim$ (negation), $\Rightarrow$ (implication) and $\Leftrightarrow$ (reification). Implication and reification take a Boolean variable and a primitive constraint as parameters. In reification an order of parameters does not matter.

In the object package of the JaCoP wrapper there are defined global constraints and therefore they can be directly used in programs. For example, in the SendMoreMoney program we use alldifferent constraint in line 16. Table 9.2 presents a list of global constraints available through the wrapper.

### 9.5 Search

Scala wrapper offers several search methods as well as possibility to select methods for variable selection and value selection. In the example in this section, search is defined to find a solution that satisfies constraints. Method `search` defines `input\_order` variable selection method and `indomain\_min` value selection method. Moreover, method `satisfy` passes method `printSol` that is used to print a solution when it is found.

The search methods defined in Scala wrapper are presented in Table 9.3.

### 9.6 Other Methods

Scala wrapper implements several methods that can be used in the model. For floating-point solver, there are defined two methods for checking and setting the current definition of precision in the solver.

```
precision() // get floating-point solver precision
setPrecision(1e-12) // set floating-point solver precision to 10^{-12}
```

Search statistics can be obtained using method

```
statistics
```

as used in the example in this chapter.
<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>JaCoP constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>alldifferent</td>
<td>alldifferent(vector)</td>
<td>Alldifferent</td>
</tr>
<tr>
<td>alldistinct</td>
<td>alldistinct(vector)</td>
<td>Alldistinct</td>
</tr>
<tr>
<td>gcc</td>
<td>gcc(vector1, vector2)</td>
<td>GCC</td>
</tr>
<tr>
<td>sum</td>
<td>sum(vector, result)</td>
<td>SumInt</td>
</tr>
<tr>
<td></td>
<td>result = sum(vector)</td>
<td>or SumBool</td>
</tr>
<tr>
<td></td>
<td>result = sum(vector, intVector)</td>
<td>LinearInt</td>
</tr>
<tr>
<td></td>
<td>result = sumDom(vector, intVector)</td>
<td>LinearIntDom</td>
</tr>
<tr>
<td>abs</td>
<td>result = abs(v)</td>
<td>AbsXeqY</td>
</tr>
<tr>
<td>min</td>
<td>result = min(v)</td>
<td>Min</td>
</tr>
<tr>
<td>max</td>
<td>result = max(vector)</td>
<td>Max</td>
</tr>
<tr>
<td>count</td>
<td>result = count(vector, value)</td>
<td>Count</td>
</tr>
<tr>
<td></td>
<td>count(vector, result, value)</td>
<td>Count</td>
</tr>
<tr>
<td>atleast</td>
<td>atleast(vector, count, value)</td>
<td>AtLeast</td>
</tr>
<tr>
<td>atmost</td>
<td>atmost(vector, count, value)</td>
<td>AtMost</td>
</tr>
<tr>
<td>values</td>
<td>result = values(vector)</td>
<td>Values</td>
</tr>
<tr>
<td></td>
<td>values(vector, result)</td>
<td>Values</td>
</tr>
<tr>
<td>element</td>
<td>element(index, vector, value)</td>
<td>ElementInteger</td>
</tr>
<tr>
<td></td>
<td>value = vectore(index)</td>
<td>or ElementVariable</td>
</tr>
<tr>
<td>diff2</td>
<td>diff2(rectangles)</td>
<td>Diff2</td>
</tr>
<tr>
<td>diffn</td>
<td>diffn(rectangles)</td>
<td>Diffn</td>
</tr>
<tr>
<td>cumulative</td>
<td>cumulative(t_vect, d_vect, r_vect, limit)</td>
<td>Cumulative</td>
</tr>
<tr>
<td>circuit</td>
<td>circuit(vector)</td>
<td>Circuit</td>
</tr>
<tr>
<td>subcircuit</td>
<td>subcircuit(vector)</td>
<td>Subcircuit</td>
</tr>
<tr>
<td>assignment</td>
<td>assignment(vector1, vector2)</td>
<td>Assignment</td>
</tr>
<tr>
<td>among</td>
<td>among(vector, set, result)</td>
<td>Among</td>
</tr>
<tr>
<td></td>
<td>among(vector1, vector2, result)</td>
<td>AmongVar</td>
</tr>
<tr>
<td>knapsack</td>
<td>knapsack(profits, weights, quantity)</td>
<td>Knapsack</td>
</tr>
<tr>
<td>regular</td>
<td>regular(dfa, vector)</td>
<td>Regular</td>
</tr>
<tr>
<td>clause</td>
<td>clause(vectorP, vectorN)</td>
<td>Clause</td>
</tr>
<tr>
<td>arg_min</td>
<td>result = arg_min(vector)</td>
<td>ArgMin</td>
</tr>
<tr>
<td>arg_max</td>
<td>result = arg_max(vector)</td>
<td>ArgMax</td>
</tr>
<tr>
<td>sequence</td>
<td>sequence(vector, set, q, min, max)</td>
<td>Sequence</td>
</tr>
<tr>
<td>stretch</td>
<td>stretch(vector, min_vect, max_vect, vector)</td>
<td>Stretch</td>
</tr>
<tr>
<td>lex</td>
<td>lex(vector1, vector2)</td>
<td>LexOrder</td>
</tr>
<tr>
<td></td>
<td>lex(matrix)</td>
<td>Lex</td>
</tr>
<tr>
<td>valueprecede</td>
<td>valueprecede(vector, s, t)</td>
<td>ValuePrecede</td>
</tr>
<tr>
<td>softAlldifferent</td>
<td>softAlldifferent(vector, cost)</td>
<td>SoftAlldifferent</td>
</tr>
<tr>
<td>softGCC</td>
<td>softGCC(vector, hardLB, hardUB, values, softCount))</td>
<td>softGCC</td>
</tr>
</tbody>
</table>

Table 9.2: Global constraints available from Scala wrapper.

*a deterministic finite automaton specified using methods defined in jacop.scala*
### Table 9.3: Search methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>satisfy</td>
<td>Find a single solution for vector of variables</td>
</tr>
<tr>
<td>satisfy_seq</td>
<td>Find a single solution for sequence of searches</td>
</tr>
<tr>
<td>satisfyAll</td>
<td>Find all solutions for vector of variables</td>
</tr>
<tr>
<td>satisfyAll_seq</td>
<td>Find all solutions for sequence of searches</td>
</tr>
<tr>
<td>minimize</td>
<td>Find a solution with minimal cost for vector of variables</td>
</tr>
<tr>
<td>maximize</td>
<td>Find a solution with maximal cost for vector of variables</td>
</tr>
<tr>
<td>minimize_seq</td>
<td>Find a solution with minimal cost for sequence of searches</td>
</tr>
<tr>
<td>maximize_seq</td>
<td>Find a solution with maximal cost for sequence of searches</td>
</tr>
</tbody>
</table>

### Table 9.4: Selection methods for search.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>Defines vector of variables, variable and value selection methods for IntVar</td>
</tr>
<tr>
<td>search_vector</td>
<td>Defines vector of vectors of variables, variable and value selection methods</td>
</tr>
<tr>
<td>search_split</td>
<td>Defines vector of variables and variable selection methods for split search</td>
</tr>
<tr>
<td>search_float</td>
<td>Defines vector of variables, variable and value selection methods for FloatVar</td>
</tr>
</tbody>
</table>

*applicable only to IntVar

*b applicable only to FloatVar

### Table 9.5: Variable selection heuristics.

<table>
<thead>
<tr>
<th>Name</th>
<th>JaCoP method</th>
</tr>
</thead>
<tbody>
<tr>
<td>input_order</td>
<td>null</td>
</tr>
<tr>
<td>first_fail</td>
<td>SmallestDomain</td>
</tr>
<tr>
<td>anti_first_fail</td>
<td>LargestDomain</td>
</tr>
<tr>
<td>most_constrained</td>
<td>MostConstrainedStatic</td>
</tr>
<tr>
<td>smallest_min</td>
<td>SmallestMin</td>
</tr>
<tr>
<td>smallest</td>
<td>SmallestMin</td>
</tr>
<tr>
<td>smallest_max</td>
<td>SmallestMax</td>
</tr>
<tr>
<td>largest</td>
<td>LargestMax</td>
</tr>
<tr>
<td>max_regret</td>
<td>MaxRegret</td>
</tr>
</tbody>
</table>

### Table 9.6: Variable selection heuristics for SetVar.

<table>
<thead>
<tr>
<th>Name</th>
<th>JaCoP method</th>
</tr>
</thead>
<tbody>
<tr>
<td>first_fail_set</td>
<td>MinCardDiff</td>
</tr>
<tr>
<td>anti_first_fail_set</td>
<td>MaxCardDiff</td>
</tr>
<tr>
<td>most_constrained_set</td>
<td>MostConstrainedStatic</td>
</tr>
<tr>
<td>min_glb_card</td>
<td>MinGbCard</td>
</tr>
<tr>
<td>min_lub_card</td>
<td>MinLubCard</td>
</tr>
</tbody>
</table>
### 9.6. SCALA:OTHER

#### Table 9.7: Value selection heuristics.

<table>
<thead>
<tr>
<th>Name</th>
<th>JaCoP method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>indomain_min</code></td>
<td><code>IndomainMin</code></td>
</tr>
<tr>
<td><code>indomain_max</code></td>
<td><code>IndomainMax</code></td>
</tr>
<tr>
<td><code>indomain_middle</code></td>
<td><code>IndomainMiddle</code></td>
</tr>
<tr>
<td><code>indomain_median</code></td>
<td><code>IndomainMedian</code></td>
</tr>
<tr>
<td><code>indomain_random</code></td>
<td><code>IndomainRandom</code></td>
</tr>
</tbody>
</table>

#### Table 9.8: Value selection heuristics for SetVar.

<table>
<thead>
<tr>
<th>Name</th>
<th>JaCoP method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>indomain_min_set</code></td>
<td><code>IndomainSetMin</code></td>
</tr>
<tr>
<td><code>indomain_max_set</code></td>
<td><code>IndomainSetMax</code></td>
</tr>
<tr>
<td><code>indomain_random_set</code></td>
<td><code>IndomainSetRandom</code></td>
</tr>
</tbody>
</table>

Table 9.7: Value selection heuristics.

Table 9.8: Value selection heuristics for SetVar.
Appendix A

JaCoP constraints

A.1 Arithmetic constraints

<table>
<thead>
<tr>
<th>Primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = \text{Const}$</td>
<td>$X\text{eq}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X = Y$</td>
<td>$X\text{eq}(X, Y)$</td>
</tr>
<tr>
<td>$X \neq \text{Const}$</td>
<td>$X\text{eq}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X \neq Y$</td>
<td>$X\text{eq}(X, Y)$</td>
</tr>
<tr>
<td>$X &gt; \text{Const}$</td>
<td>$X\text{gt}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X &gt; Y$</td>
<td>$X\text{gt}(X, Y)$</td>
</tr>
<tr>
<td>$X \geq \text{Const}$</td>
<td>$X\text{gteq}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X \geq Y$</td>
<td>$X\text{gteq}(X, Y)$</td>
</tr>
<tr>
<td>$X &lt; \text{Const}$</td>
<td>$X\text{lt}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X &lt; Y$</td>
<td>$X\text{lt}(X, Y)$</td>
</tr>
<tr>
<td>$X \leq \text{Const}$</td>
<td>$X\text{lteq}(X, \text{Const})$</td>
</tr>
<tr>
<td>$X \leq Y$</td>
<td>$X\text{lteq}(X, Y)$</td>
</tr>
<tr>
<td>$X \cdot \text{Const} = Z$</td>
<td>$X\text{mul}(X, \text{Const}, Z)$</td>
</tr>
<tr>
<td>$X + \text{Const} = Z$</td>
<td>$X\text{plus}(X, \text{Const}, Z)$</td>
</tr>
<tr>
<td>$X + Y = Z$</td>
<td>$X\text{plus}(X, Y, Z)$</td>
</tr>
<tr>
<td>$X + Y + \text{Const} = Z$</td>
<td>$X\text{plus}(X, Y, \text{Const}, Z)$</td>
</tr>
<tr>
<td>$X + Y + Q = Z$</td>
<td>$X\text{plus}(X, Y, Q, Z)$</td>
</tr>
<tr>
<td>$X + \text{Const} \leq Z$</td>
<td>$X\text{plus}(X, \text{Const}, Z)$</td>
</tr>
<tr>
<td>$X + Y \leq Z$</td>
<td>$X\text{plus}(X, Y, Z)$</td>
</tr>
<tr>
<td>$X + Y &gt; \text{Const}$</td>
<td>$X\text{plus}(X, Y, \text{Const})$</td>
</tr>
<tr>
<td>$X + Y + Q &gt; \text{Const}$</td>
<td>$X\text{plus}(X, Y, Q, \text{Const})$</td>
</tr>
<tr>
<td>$</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \cdot Y = Z$</td>
<td>$X\text{mul}(X, Y, Z)$</td>
</tr>
<tr>
<td>$X \div Y = Z$</td>
<td>$X\text{div}(X, Y, Z)$</td>
</tr>
<tr>
<td>$X \mod Y = Z$</td>
<td>$X\text{mod}(X, Y, Z)$</td>
</tr>
<tr>
<td>$X^Y = Z$</td>
<td>$X\text{exp}(X, Y, Z)$</td>
</tr>
</tbody>
</table>
A.2 Set constraints

<table>
<thead>
<tr>
<th>Primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e \in A$</td>
<td>EinA(e, A)</td>
</tr>
<tr>
<td>$S_1 = S_2$</td>
<td>AeqB(S1, S2)</td>
</tr>
<tr>
<td>$S_1 \subseteq S_2$</td>
<td>AinB(S1, S2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1 \cup S_2 = S_3$</td>
<td>UnionBeqC(S1, S2, S3)</td>
</tr>
<tr>
<td>$S_1 \cap S_2 = S_3$</td>
<td>IntersectBeqC(S1, S2, S3)</td>
</tr>
<tr>
<td>$S_1 \setminus S_2 = S_3$</td>
<td>DiffBeqC(S1, S2, S3)</td>
</tr>
<tr>
<td>$S_1 \not= S_2$</td>
<td>DisjointB(S1, S2)</td>
</tr>
<tr>
<td>Match</td>
<td>Match(Set, VarArray)</td>
</tr>
<tr>
<td>$# S = C$</td>
<td>CardA(S, C)</td>
</tr>
<tr>
<td>$# S = X$</td>
<td>CardAeqX(S, X)</td>
</tr>
<tr>
<td>Weighted sum $&lt; S, W &gt; = X$</td>
<td>SumWeightedSet(S, W, X)</td>
</tr>
<tr>
<td>$Set[X] = Y$</td>
<td>ElementSet(X, Set, Y)</td>
</tr>
</tbody>
</table>
## A.3 Floating point constraints

<table>
<thead>
<tr>
<th>Primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P = Const$</td>
<td>PeqC($P$, Const)</td>
</tr>
<tr>
<td>$P = Q$</td>
<td>PeqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P \neq Const$</td>
<td>PneqC($P$, Const)</td>
</tr>
<tr>
<td>$P \neq Q$</td>
<td>PneqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P &gt; Const$</td>
<td>PgtC($P$, Const)</td>
</tr>
<tr>
<td>$P &gt; Q$</td>
<td>PgtQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P \geq Const$</td>
<td>PgteqC($P$, Const)</td>
</tr>
<tr>
<td>$P \geq Q$</td>
<td>PgteqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P &lt; Const$</td>
<td>PltC($P$, Const)</td>
</tr>
<tr>
<td>$P &lt; Q$</td>
<td>PltQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P \leq Const$</td>
<td>PlteqC($P$, Const)</td>
</tr>
<tr>
<td>$P \leq Q$</td>
<td>PlteqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$P + Const = R$</td>
<td>PplusCeqR($P$, Const, $R$)</td>
</tr>
<tr>
<td>$P + Q = R$</td>
<td>PplusQeqR($P$, $Q$, $R$)</td>
</tr>
<tr>
<td>$P - Const = R$</td>
<td>PminusCeqR($P$, Const, $R$)</td>
</tr>
<tr>
<td>$P - Q = R$</td>
<td>PminusQeqR($P$, $Q$, $R$)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \cdot Const = R$</td>
<td>PmulCeqR($P$, Const, $R$)</td>
</tr>
<tr>
<td>$P \cdot Q = R$</td>
<td>PmulQeqR($P$, $Q$, $R$)</td>
</tr>
<tr>
<td>$P/Q = R$</td>
<td>PdivQeqR($P$, $Q$, $R$)</td>
</tr>
<tr>
<td>$\sqrt{P} = R$</td>
<td>SqrtPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\sin(P) = R$</td>
<td>SinPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\cos(P) = R$</td>
<td>CosPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\tan(P) = R$</td>
<td>TanPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\arcsin(P) = R$</td>
<td>AsinPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\arccos(P) = R$</td>
<td>AcosPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\arctan(P) = R$</td>
<td>AtanPeqR($P$, $Q$)</td>
</tr>
<tr>
<td>$\ln(P) = Q$</td>
<td>LnPeqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$\exp(P) = Q$</td>
<td>ExpPeqQ($P$, $Q$)</td>
</tr>
<tr>
<td>$</td>
<td>P</td>
</tr>
<tr>
<td>$V[Index] = Value$</td>
<td>Element($Index$, $V$, $Value$)</td>
</tr>
<tr>
<td>$X = P$</td>
<td>XeqQ($X$, $P$) where $x$ is IntVar and $P$ is FloatVar</td>
</tr>
</tbody>
</table>
A.4 Logical, conditional and reified constraints

<table>
<thead>
<tr>
<th>Primitive Constraint</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬c</td>
<td>Not(c);</td>
</tr>
<tr>
<td>c1 ∨ c2 ∨ ⋯ ∨ cn</td>
<td>PrimitiveConstraint[] c = {c1, c2, ...cn}; Or(c);</td>
</tr>
<tr>
<td></td>
<td>ArrayList&lt;PrimitiveConstraint&gt; c =</td>
</tr>
<tr>
<td></td>
<td>new ArrayList&lt;PrimitiveConstraint&gt;();</td>
</tr>
<tr>
<td></td>
<td>c.add(c1); c.add(c2); ...c.add(cn);</td>
</tr>
<tr>
<td></td>
<td>Or(c);</td>
</tr>
<tr>
<td>c1 ∧ c2 ∧ ⋯ ∧ cn</td>
<td>PrimitiveConstraint[] c = {c1, c2, ...cn}; And(c);</td>
</tr>
<tr>
<td></td>
<td>ArrayList&lt;PrimitiveConstraint&gt; c =</td>
</tr>
<tr>
<td></td>
<td>new ArrayList&lt;PrimitiveConstraint&gt;();</td>
</tr>
<tr>
<td></td>
<td>c.add(c1); c.add(c2); ...c.add(cn);</td>
</tr>
<tr>
<td></td>
<td>And(c);</td>
</tr>
<tr>
<td>a1 ∨ ⋯ ∨ an ∨ b1 ∨ ⋯ ∨ bm</td>
<td>BooleanVar[] a = {a1, a2, ..., an};</td>
</tr>
<tr>
<td></td>
<td>BooleanVar[] b = {b1, b2, ..., bm};</td>
</tr>
<tr>
<td></td>
<td>BoolClause(a, b)</td>
</tr>
<tr>
<td></td>
<td>c1 ⇔ c2</td>
</tr>
<tr>
<td></td>
<td>Eq(c1, c2);</td>
</tr>
<tr>
<td></td>
<td>X in Dom</td>
</tr>
<tr>
<td></td>
<td>In(X, Dom);</td>
</tr>
<tr>
<td></td>
<td>c ⇔ b</td>
</tr>
<tr>
<td></td>
<td>Reified(c, b);</td>
</tr>
<tr>
<td></td>
<td>Xor(c, b);</td>
</tr>
<tr>
<td></td>
<td>b ⇒ c</td>
</tr>
<tr>
<td></td>
<td>Implies(b, c);</td>
</tr>
<tr>
<td>if b1 then c1 elseif b2 then c2... else cn</td>
<td>Conditional(B, c);</td>
</tr>
<tr>
<td></td>
<td>last bn must always be true, i.e, (bn = 1)</td>
</tr>
<tr>
<td>if c1 then c2</td>
<td>IfThen(c1, c2);</td>
</tr>
<tr>
<td>if c1 then c2 else c3</td>
<td>IfThenElse(c1, c2, c3);</td>
</tr>
<tr>
<td>∃ i xi = e</td>
<td>Member(x, e);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primitive constraints on Boolean variables</th>
<th>JaCoP specification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BooleanVar[] b = {b1, b2, ..., bn}; or</td>
</tr>
<tr>
<td></td>
<td>ArrayList&lt;BooleanVar&gt; b =</td>
</tr>
<tr>
<td></td>
<td>new ArrayList&lt;BooleanVar&gt;();</td>
</tr>
<tr>
<td></td>
<td>b.add(b1); b.add(b2); ...b.add(bn);</td>
</tr>
<tr>
<td></td>
<td>BooleanVar result =</td>
</tr>
<tr>
<td></td>
<td>new BooleanVar(store, &quot;result&quot;);</td>
</tr>
<tr>
<td>result = b1 ∧ b2 ∧ ⋯ ∧ bn</td>
<td>AndBool(b, result)</td>
</tr>
<tr>
<td>result = b1 ∨ b2 ∨ ⋯ ∨ bn</td>
<td>OrBool(b, result)</td>
</tr>
<tr>
<td>result = b1 ⊕ b2⋯ ⊕ bn</td>
<td>XorBool(b, result)</td>
</tr>
<tr>
<td>result = b1 → b2</td>
<td>IfThenBool(b1, b2, result)</td>
</tr>
<tr>
<td>result = b1 == b2 == ⋯ == bn</td>
<td>EqBool(b, result)</td>
</tr>
</tbody>
</table>
A.5. Global constraints

**SumInt, SumBool, LinearInt** used with ==, <, >, <=, >=, !=

(examples with == only)

\[ x_1 + x_2 + \cdots + x_n = \text{sum} \]

*Primitive constraint*

```java
IntVar[] x = {x1, x2, ..., xn};
IntVar sum = new IntVar(...)
SumInt(store, x, "==", sum);
```

or

```java
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
IntVar sum = new IntVar(...)
SumInt(store, x, "==", sum);
```

\[ b_1 + b_2 + \cdots + b_n = \text{sum} \]

*Primitive constraint*

```java
BooleanVar[] b = {b1, b2, ..., bn};
IntVar sum = new IntVar(...)
SumBool(store, b, "==", sum);
```

or

```java
ArrayList<BooleanVar> b = new ArrayList<BooleanVar>();
b.add(b1); b.add(b2); ...b.add(bn);
IntVar sum = new IntVar(...)
SumBool(store, b, "==", sum);
```

\[ w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n = \text{sum} \]

*Primitive constraint*

```java
IntVar[] x = {x1, x2, ..., xn};
IntVar sum = new IntVar(...)
int[] w = {w1, w2, ..., wn};
LinearInt(store, x, w, "==", sum);
```

or

```java
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(w2); ...w.add(wn);
LinearInt(store, x, w, "==", sum);
```

```java
LinearIntDom(store, x, w, "==", sum); domain consistency (== and !=)
```

**alldifferent**([x1, x2, ..., xn])

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

or

```java
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
```
APPENDIX A. JACOP CONSTRAINTS

\textbf{Alldifferent}(x);

\textbf{alldiff}([x_1, x_2, \ldots, x_n])
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    Alldiff(x);
    or
    ArrayList<IntVar> x = new ArrayList<IntVar>();
    x.add(x1); x.add(x2); \ldots x.add(xn);
    Alldiff(x);
\end{verbatim}

\textbf{alldistinct}([x_1, x_2, \ldots, x_n])
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    Alldistinct(x);
    or
    ArrayList<IntVar> x = new ArrayList<IntVar>();
    x.add(x1); x.add(x2); \ldots x.add(xn);
    Alldistinct(x);
\end{verbatim}

\textbf{among}([x_1, x_2, \ldots, x_n], \textit{val}, count)
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    IntervalDomain val = new IntervalDomain(k, l);
    IntVar count = new IntVar(...);
    Among(x, val, count);
\end{verbatim}

\textbf{amongVar}([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_m], count)
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    IntVar[] y = {y1, y2, \ldots, ym};
    IntVar count = new IntVar(...);
    Among(x, y, count);
\end{verbatim}

\textbf{assignment}([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n])
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    IntVar[] y = {y1, y2, \ldots, yn};
    Assignment(x, y);
    or
    ArrayList<IntVar> x = new ArrayList<IntVar>();
    x.add(x1); x.add(x2); \ldots x.add(xn);
    ArrayList<IntVar> y = new ArrayList<IntVar>();
    y.add(y1); y.add(y2); \ldots y.add(yn);
    Assignment(x, y);
\end{verbatim}

\textbf{circuit}([x_1, x_2, \ldots, x_n])
\begin{verbatim}
    IntVar[] x = {x1, x2, \ldots, xn};
    Circuit(x);
\end{verbatim}
A.5. GLOBAL CONSTRAINTS

or
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Circuit(x);

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

count(value, [x1, x2, ..., xn], 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);

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

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

cumulative([t1, t2, ..., tn]; [d1, d2, ..., dn]; [r1, r2, ..., rn], ResourceLimit)
APPENDIX A. JACOP CONSTRAINTS

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

cumulative_unary([t1, t2, ..., tn], [d1, d2, ..., dn], [r1, r2, ..., rn], ResourceLimit)

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

diffn([[x1, y1, dx1, dy1], ..., [xn, yn, dxn, dyn]])

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

diffn([x1, ..., xn], [y1, ..., yn], [dx1, ..., dxn], [dy1, ..., dyn])

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

distance(x, y, dist)

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

element(Index, [n1, n2, ..., nn], Value)

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

element(Index, [x1, x2, ..., xn], Value)

IntVar Index, Value;
IntVar[] x = {x1, x2, ..., xn};
```
A.5. GLOBAL CONSTRAINTS

Element(Index, x, Value);
or
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Element(Index, x, Value);

\[
\text{table}\left(\left[ x_1, x_2, \ldots, x_n \right], \left\{\{1, 2, \ldots, n\}, \ldots, \{\ldots\}\right\} \right)
\]

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

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

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

\[
\text{gcc}\left(\left[ x_1, x_2, \ldots, x_n \right], \left[ y_1, y_2, \ldots, y_m \right] \right)
\]

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

\[
\text{min}\left(\left[ x_1, x_2, \ldots, x_n \right], X_{min} \right)
\]

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);

\[
\text{max}\left(\left[ x_1, x_2, \ldots, x_n \right], X_{max} \right)
\]

IntVar[] x = {x1, x2, ..., xn};
Max(x, Xmax);
or
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); ...x.add(xn);
Max(x, Xmax);
\textbf{arg\_min}(\[x_1, x_2, \ldots, x_n\], indexMin)

\begin{verbatim}
IntVar[] x = {x1, x2, \ldots, xn};
ArgMin(x, indexMin);
\end{verbatim}

or

\begin{verbatim}
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); \ldots x.add(xn);
ArgMin(x, indexMin);
\end{verbatim}

\textbf{arg\_max}(\[x_1, x_2, \ldots, x_n\], indexMax)

\begin{verbatim}
IntVar[] x = {x1, x2, \ldots, xn};
Max(x, indexMax);
\end{verbatim}

or

\begin{verbatim}
ArrayList<IntVar> x = new ArrayList<IntVar>();
x.add(x1); x.add(x2); \ldots x.add(xn);
Max(x, indexMax);
\end{verbatim}

\textbf{knapsack}(profits, weights, quantity, knapsackCapacity, knapsackProfit)

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

\textbf{geost}(objects, constraints, shapes)

\begin{verbatim}
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);
\end{verbatim}
A.5. GLOBAL CONSTRAINTS

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

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

sequence([x1, x2, ..., xn], 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, [x1, x2, ..., xn])
int[] values, main, max;
IntVar[] x = {x0, x1 ..., xn};
Stretch(values, min, max, x);

values([x1, x2, ..., xn], count)
IntVar[] x = {x0, x1 ..., xn};
IntVar count = new IntVar(...);
Values(x, count);

lex([[x1, x12, ..., x1n], ..., [xk1, xk2, ..., xkm]])
IntVar[] x = {{x0, x1 ..., xn}, ...};
Lex(x);
or
Lex(x, true);

lex([x1, x2, ..., xn], [y1, y2, ..., ym])
IntVar[] x = {x0, x1 ..., xn};
IntVar[] y = {y0, y1 ..., yn};
LexOrder(x, y);
or
LexOrder(x, y, true);

value_precede(t, s, [x1, x2, ..., xn])
IntVar[] x = {x0, x1 ..., xn};
ValuePrecede(t, s, x);

soft-alldifferent([x1, x2, ..., xn], 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([x1, x2, ..., xn], 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 IntVar

• value selection methods

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

• variable selection methods
APPENDIX B. JACOP SEARCH METHODS

<table>
<thead>
<tr>
<th>Comparator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallestDomain</td>
<td>selects FDV which has the smallest domain size</td>
</tr>
<tr>
<td>MostConstrainedStatic</td>
<td>selects FDV which has the most constraints assign to it</td>
</tr>
<tr>
<td>MostConstrainedDynamic</td>
<td>selects FDV which has the most pending constraints assign to it</td>
</tr>
<tr>
<td>SmallestMin</td>
<td>selects FDV with the smallest value in its domain</td>
</tr>
<tr>
<td>LargestDomain</td>
<td>selects FDV with the largest domain size</td>
</tr>
<tr>
<td>LargestMin</td>
<td>selects FDV with the largest value in its domain</td>
</tr>
<tr>
<td>SmallestMax</td>
<td>selects FDV with the smallest maximal value in its domain</td>
</tr>
<tr>
<td>MaxRegret</td>
<td>selects FDV with the largest difference between the smallest</td>
</tr>
<tr>
<td>WeightedDegree</td>
<td>selects FDV with the highest weight divided by its size</td>
</tr>
<tr>
<td>AFCMax</td>
<td>selects FDV with the highest Accumulated Failure Count (afc)</td>
</tr>
<tr>
<td>AFCMaxDeg</td>
<td>selects FDV with the highest Accumulated Failure Count (afc) divided by its domain</td>
</tr>
<tr>
<td>AFCMin</td>
<td>selects FDV with the lowest Accumulated Failure Count (afc)</td>
</tr>
<tr>
<td>AFCMinDeg</td>
<td>selects FDV with the lowest Accumulated Failure Count (afc) divided by its domain</td>
</tr>
<tr>
<td>ActivityMax</td>
<td>selects FDV with the highest number of the accumulated prunings</td>
</tr>
<tr>
<td>ActivityMaxDeg</td>
<td>selects FDV with the highest number of the accumulated prunings divided by its domain</td>
</tr>
<tr>
<td>ActivityMin</td>
<td>selects FDV with the lowest number of the accumulated prunings</td>
</tr>
<tr>
<td>ActivityMinDeg</td>
<td>selects FDV with the lowest number of the accumulated prunings divided by its domain</td>
</tr>
<tr>
<td>RandomVar</td>
<td>selects random FDV</td>
</tr>
</tbody>
</table>

B.2 Variable and value selection for SetVar

• value selection methods

<table>
<thead>
<tr>
<th>Indomain method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndomainSetMin</td>
<td>selects a minimal value from not yet assigned values for set variable</td>
</tr>
<tr>
<td>IndomainSetMax</td>
<td>selects a maximal value from not yet assigned values for set variable</td>
</tr>
<tr>
<td>IndomainSetRandom</td>
<td>selects a random value from not yet assigned values for set variable</td>
</tr>
</tbody>
</table>

• variable selection methods

<table>
<thead>
<tr>
<th>Comparator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinCardDiff</td>
<td>selects set variable which has the smallest difference in cardinality between lub and glb.</td>
</tr>
<tr>
<td>MaxCardDiff</td>
<td>selects set variable which has the greatest difference in cardinality between lub and glb.</td>
</tr>
<tr>
<td>MinGlbCard</td>
<td>selects set variable which has the glb with the smallest cardinality</td>
</tr>
<tr>
<td>MaxGlbCard</td>
<td>selects set variable which has the glb with the greatest cardinality</td>
</tr>
<tr>
<td>MinLubCard</td>
<td>selects set variable which has the lub with the smallest cardinality</td>
</tr>
<tr>
<td>MaxLubCard</td>
<td>selects set variable which has the lub with the greatest cardinality</td>
</tr>
<tr>
<td>MostConstrainedStatic</td>
<td>selects set variable which has most constraints assign to it</td>
</tr>
<tr>
<td>MostConstrainedDynamic</td>
<td>selects set variable which has the most pending constraints assign to it</td>
</tr>
<tr>
<td>RandomVar</td>
<td>selects random set variable</td>
</tr>
</tbody>
</table>
### B.3 Variable selection for FloatVar

- **variable selection methods**

<table>
<thead>
<tr>
<th>Comparator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallestDomainFloat</td>
<td>selects floating point variable with the smallest domain size</td>
</tr>
<tr>
<td>LargestDomainFloat</td>
<td>selects floating point variable with the largest domain size</td>
</tr>
<tr>
<td>LargestMinFloat</td>
<td>selects floating point variable with the largest min value</td>
</tr>
<tr>
<td>LargestMaxFloat</td>
<td>selects floating point variable with the largest max value</td>
</tr>
<tr>
<td>SmallestMinFloat</td>
<td>selects floating point variable with the smallest min value</td>
</tr>
<tr>
<td>SmallestMaxFloat</td>
<td>selects floating point variable with the smallest max value</td>
</tr>
<tr>
<td>WeightedDegreeFloat</td>
<td>selects floating point variable with the highest weight divided by its size</td>
</tr>
<tr>
<td>MostConstrainedStatic</td>
<td>selects floating variable which has most constraints assign to it</td>
</tr>
<tr>
<td>MostConstrainedDynamic</td>
<td>selects floating variable which has the most pending constraints assign to it</td>
</tr>
<tr>
<td>RandomVar</td>
<td>selects random floating variable</td>
</tr>
</tbody>
</table>

### B.4 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**

```java
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**

```java
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.
APPENDIX B. JACOP SEARCH METHODS

```java
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.

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

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

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

- **Search for optimal solution**

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

### B.5 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 Simple trace facilities

JaCoP offers simple methods for tracing failed constraints and changes of variables. These methods can be used as they are or extended to provide special information.

Class FailConstraintsStatistics is the implementation of ConsistencyListener and collects statistics of failed constraints. Possible use is illustrated by the following code.

In this code, FailConstraintsStatistics is added to the search to collect statistics.

```java
Search<IntVar> search = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(
    vars.toArray(new IntVar[1]),
    null,
    new IndomainMin<IntVar>());

FailConstraintsStatistics fcs = new FailConstraintsStatistics(store);
search.setConsistencyListener(fcs);
```

The statistics can be printed after search using toString method of this class, for example as instruction System.out.println(fcs); in this case. The printout will report that for this example it was 10 XneqY constraint that failed during search. Of this ten there was 4 XneqY730 and 4 XneqY721.

```
*** Failed classes of constraints ***
class org.jacop.constraints.XneqY 10
*** Failed constraints ***
XneqY730 4
XneqY721 6
*** Fails not caused by constraints 0
```

Obviously this class can be extended and different kind of information can be gathered.

Class VariableTrace makes it possible to trace all prunings of the domain of given variables. It is defined as a constraint with parameters that specify all variables
that will be traced. For the above example constraints XneqY721 and XneqY730 contain variables f06, f07 and f16. Tracing of variable f07 is given by the following code.

```
store.impose(new VariableTrace(elements[0][7]));
```

where elements[0][7] defines variable f07.

Tracing pruning of variable f07 give the following output.

```
Var: f07::{2..9}, level: 0, constraint: XneqY14 : XneqY(f01 = 1, f07::{2..9} )
Var: f07::{2..3, 5..9}, level: 0, constraint: XneqY25 : XneqY(f04 = 2, f07::{2..3, 5..9} )
Var: f07::{3, 6..9}, level: 0, constraint: XneqY36 : XneqY(f07::{3, 6..9}, f08 = 5 )
Var: f07::{6..9}, level: 0, constraint: XneqY577 : XneqY(f07::{6..9}, f17 = 3 )
Var: f07::{6, 8..9}, level: 0, constraint: XneqY584 : XneqY(f07::{6, 8..9}, f07 = 7 )
Var: f07::{6, 8}, level: 0, constraint: XneqY732 : XneqY(f07::{6, 8}, f18 = 9 )
Var: f07 = 8, level: 4, constraint: XneqY32 : XneqY(f05 = 6, f07 = 8 )
Backtrack level: 4, vars: f07::{6, 8}
Var: f07 = 6, level: 5, constraint: XneqY32 : XneqY(f05 = 8, f07 = 6 )
Backtrack level: 5, vars: f07::{6, 8}
Var: f07 = 8, level: 6, constraint: XneqY721 : XneqY(f06 = 6, f07 = 8 )
Backtrack level: 6, vars: f07::{6, 8}
Backtrack level: 5, vars: f07::{6, 8}
Backtrack level: 4, vars: f07::{6, 8}
Backtrack level: 3, vars: f07::{6, 8}
Var: f07 = 8, level: 4, constraint: XneqY20 : XneqY(f02 = 6, f07 = 8 )
Backtrack level: 4, vars: f07::{6, 8}
Var: f07 = 6, level: 5, constraint: XneqY20 : XneqY(f02 = 8, f07 = 6 )
Backtrack level: 5, vars: f07::{6, 8}
Var: f07 = 8, level: 7, constraint: XneqY721 : XneqY(f06 = 6, f07 = 8 )
Backtrack level: 7, vars: f07::{6, 8}
Backtrack level: 6, vars: f07::{6, 8}
Backtrack level: 5, vars: f07::{6, 8}
Backtrack level: 4, vars: f07::{6, 8}
Backtrack level: 3, vars: f07::{6, 8}
Var: f07 = 8, level: 7, constraint: XneqY32 : XneqY(f05 = 6, f07 = 8 )
Backtrack level: 7, vars: f07::{6, 8}
Var: f07 = 6, level: 7, constraint: XneqY32 : XneqY(f05 = 8, f07 = 6 )
Backtrack level: 7, vars: f07::{6, 8}
Backtrack level: 6, vars: f07::{6, 8}
Backtrack level: 5, vars: f07::{6, 8}
Backtrack level: 4, vars: f07::{6, 8}
Backtrack level: 3, vars: f07::{6, 8}
Backtrack level: 2, vars: f07::{6, 8}
Var: f07 = 6, level: 3, constraint: XneqY77 : XneqY(f08 = 8, f07 = 6 )
Backtrack level: 4, vars: f07 = 6
Backtrack level: 3, vars: f07 = 6
Backtrack level: 2, vars: f07 = 6
Backtrack level: 1, vars: f07 = 6
```

Extending class VariableTrace and, in particular, overriding methods queueVariable and removeLevelLate can provide ways of specializing this printout.

### 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 org.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);
```
C.2. CPVIZ INTERFACE

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.
Appendix D

JaCoP Execution Properties

JaCoP properties that can be controlled by option -D in java command. For example, java -Dmax_edge_find_size=200 controls the maximal size of the task list in the cumulative constraint to use edge-finding algorithm.

D.1 Java specific

- max_edge_find_size: use edge finder in the cumulative constraints if it contains less tasks than this number; default 100.
- sub_circuit_scc_pruning: use pruning method based on strongly connected components; default true (one of sub_circuit_* must be true).
- sub_circuit_dominance_pruning: use pruning method based on node dominance in graphs; default false (one of sub_circuit_* must be true).

D.2 Flatzinc specific

- fz_cumulative_use_unary: during constraints generation in flatzinc compiler use CumulativeUnary for tasks that have resource capacity greater than half of the cumulative capacity bound; default=false
- fz_cumulative_use_disjunctions: during constraints generation in flatzinc compiler pairwise task disjunction constraints for all pairs of tasks that have sum of resource capacities greater than the cumulative capacity bound; default=false
Bibliography


