import JaCoP.core.\*;   
import JaCoP.constraints.\*;   
import JaCoP.search.\*;   
   
public class Main {   
   
    static Main m = new Main ();   
   
    public static void main (String[] args) {   
        Store store = new Store();  // define FD store   
        int size = 4;   
        // define finite domain variables   
        IntVar[] v = new IntVar[size];   
        for (int i=0; i<size; i++)   
            v[i] = new IntVar(store, "v"+i, 1, size);   
        // define constraints   
        store.impose( new XneqY(v[0], v[1]) );   
        store.impose( new XneqY(v[0], v[2]) );   
        store.impose( new XneqY(v[1], v[2]) );   
        store.impose( new XneqY(v[1], v[3]) );   
        store.impose( new XneqY(v[2], v[3]) );   
   
        // search for a solution and print results   
        Search<IntVar> search = new DepthFirstSearch<IntVar>();   
        SelectChoicePoint<IntVar> select =   
            new InputOrderSelect<IntVar>(store, v,   
                                         new IndomainMin<IntVar>());   
        boolean result = search.labeling(store, select);   
   
        if ( result )   
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+   
                                              v[2] +", "+v[3]);   
        else   
            System.out.println("\*\*\* No");   
    }   
}

Solution: v0=1, v1=2, v2=3, v3=1

The problem is specified with the help of variables (FDVs) and constraints over these variables.

   Store store = new Store();

# 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).

   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:

   x.addDom(120, 160);

which represents a variable *X*with the domain 1*..*100 ∨ 120*..*160.

   System.out.println(x);

   X::{1..2, 14..16}

One special variable class is a BooleanVariable.

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

# 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) );

# Constraints

In JaCoP, there are three major types of constraints:

* primitive constraints,
* global constraints, and
* decomposable constraints.

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

   store.impose( new XeqY(x1, x2));

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

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

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().**

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

   IntVar x = new IntVar(store, "X", 1, 100);   
   IntVar y = new IntVar(store, "Y", 1, 100);   
   IntVar b = new IntVar(store, "B", 0, 1);   
   store.impose( new Refified( new XeqY(x, y), b);

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

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

# Search for solutions

It makes it possible to search for a single solution or to try to find a solution which minimizes/maximizes given cost function. This is achieved by using the depth-first-search together with constraint consistency enforcement.

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

   boolean result = store.consistency();

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

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

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

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

* how to assign values for each FDV from its domain; this is defined by IndomainMin class that starts assignments from the minimal value in the domain first and later assigns successive values.
* how to select FDV (**SelectChoicePoint**) 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 (**Search**); 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](http://jacopguide.osolpro.com/guideJaCoP.html#x1-54000B). The following example uses SimpleSelect that selects variables using the size of their domains, i.e., variable with the smallest domain is selected first.

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

In some situations it is better to group FDVs and assign the values to them one after the other. JaCoP supports this by another variable selection method called SimpleMatrixSelect. An example of its use is shown below. This choice point selection heuristic works on two-dimensional lists of FDVs.

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

The optimization requires specification of a cost function. The cost function is defined by a FDV that, with the help of attached (imposed) constraints, gets correct cost value.

   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);

Constraints

# Primitive constraints

JaCoP offers a set of primitive constraints that include basic arithmetic operations (+*,*-*,*\**,∕*) as well as basic relations (=*,≠,<,*≤*,>,*≥). Subtraction and division are not provided explicitly, but since constraints define relations between variables, they are defined using addition and multiplication. For detail list of primitive constraint see appendix [A.1](http://jacopguide.osolpro.com/guideJaCoP.html#x1-500001).

# Global Constraints

#### Alldifferent 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 Alldifferent(v);   
   store.impose(ctr);

Alldifferent constraint is provided as three different implementations. Constraint  **Alldifferent** uses a simple implementation described above, i.e., if the domain of a finite domain variable gets assigned a value, the propagation algorithm will remove this value from the other variables. Constraint  **Alldiff** implements this basic pruning method and, in addition, bounds consistency [[11](http://jacopguide.osolpro.com/guideJaCoP.html#Xpuget:1998)]. Finally, constraint  **Alldistinct** implements a generalized arc consistency as proposed by Régin [[12](http://jacopguide.osolpro.com/guideJaCoP.html#XRegin94)].

#### Circuit constraint

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

For example, the constraint

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

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

#### Element constraint

#### Distance constraint

#### Cumulative constraint

At any time instant the total use of these resources for the tasks does not exceed a given limit.

#### Diff2 constraint

Diff2 constraint takes as an argument a list of 2-dimensional rectangles and assures that for each pair *i,j*(*i≠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 *Oi* and *Li* are respectively called the origin and the length in *i*-th dimension. The diff2 constraint is specified as follows.

#### Min and Max constraints

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

#### Sum and SumWeight constraints

These constraints enforce that a sum of elements of FDVs’ vector is equal to a given FDV sum, that is *x*1 + *x*2 + ⋅⋅⋅ + *xn* = *sum*. The weighted sum is provided by the constraint SumWeight and imposes the following constraint *w*1 ⋅*x*1 + *w*2 ⋅*x*2 + ⋅⋅⋅ + *wn* ⋅*xn* = *sum*.

#### ExtensionalSupport and ExtensionalConflict constraints

There exist several implementation of these constraint distinguished by their suffixes. The base implementation with suffix VA tries to balance the usage of memory versus time efficiency. ExtensionalSupportMDD uses multi-valued decision diagram (MDD) as internal representation and algorithms proposed in [[2](http://jacopguide.osolpro.com/guideJaCoP.html#Xyap:2008)]while ExtensionalSupportSTR uses simple tabular reduction (STR) and the method proposed in [[8](http://jacopguide.osolpro.com/guideJaCoP.html#Xlecoutre:2008)].

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

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

The example below specifies the XOR logical relation of the form *a*⊕ *b*= *c*using both constraints.

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

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

#### Assignment constraint

Assignment constraint implements the following relation between two vectors of FDVs *Xi* = *j*∧ *Y* *j* = *i*.

#### Count constraint

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

#### 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.

#### Global cardinality constraint (GCC)

#### Among and AmongVar

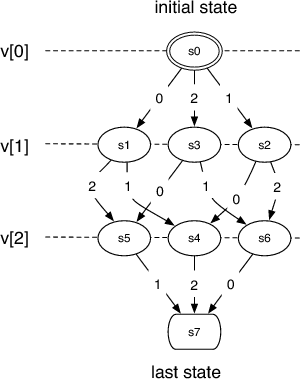
#### Regular constraint

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](http://jacopguide.osolpro.com/guideJaCoP.html#x1-270012)). 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](http://jacopguide.osolpro.com/guideJaCoP.html#x1-270012). This automaton defines condition for three variables to be different values 0, 1 or 2.



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

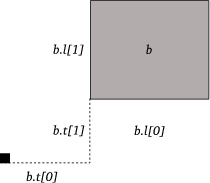
#### Knapsack constraint

#### Geost constraint

Geost is a geometrical constraint, which means that it applies to geometrical objects. It models placement problems under geometrical constraints, such as non overlapping constraints. Geost consistency algorithm was proposed by Beldiceanu et al [[13](http://jacopguide.osolpro.com/guideJaCoP.html#Xgeost_description)]. 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.*

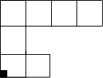


**Figure 3.3:**Shifted box in 2 dimensions. Reference origin is denoted by the black square.

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

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

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



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

In JaCoP, we can define n shapes as collection of shifted boxes. Shape identifiers start at 0 and must be assigned consecutive integers. Therefore we have shapes with identifiers in interval 0..n-1. The following JaCoP example defines a shape with identifier 0, consisting of three sboxs, depicted in Figure [3.4](http://jacopguide.osolpro.com/guideJaCoP.html#x1-290054).

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

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

An object in JaCoP is defined by class GeostObject. It specifies basically all parameters of an object. An example below specifies object 0 that can take shapes 0, 1, 2 and 3. The object can be placed using coordinates (X\_o1, Y\_o1). The object is present during time 2 to 14.

    ArrayList<GeostObject> objects = new ArrayList<GeostObject>();   
   
    IntVar X\_o1 = new IntVar(store, "x1", 0, 5);   
    IntVar Y\_o1 = new IntVar(store, "y1", 0, 5);   
    IntVar[] coords\_o1 = {X\_o1, Y\_o1};   
    IntVar shape\_o1 = new IntVar(store, "shape\_o1", 0, 3);   
    IntVar start\_o1 = new IntVar(store, "start\_o1", 2, 2);   
    IntVar duration\_o1 = new IntVar(store, "duration\_o1", 12, 12);   
    IntVar end\_o1 = new IntVar(store, "end\_o1", 14, 14);   
    GeostObject o1 = new GeostObject(0, coords\_o1, shape\_o1,   
                                     start\_o1, duration\_o1, end\_o1);   
    objects.add(o1);

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

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

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

    ArrayList<ExternalConstraint> constraints =   
                    new ArrayList<ExternalConstraint>();   
    int[] dimensions = {0, 1, 2};   
    NonOverlapping constraint1 =   
                    new NonOverlapping(objects, dimensions);   
    constraints.add(constraint1);   
    InArea constraint2 = new InArea(   
                  new DBox(new int[] {0,0}, new int[] {5,4}), null);   
    constraints.add(constraint2);

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

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

#### NetworkFlow constraint

#### Binpacking

# Decomposed constraints

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

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

**3.4.1 Sequence constraint**

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

**3.4.2 Stretch constraint**

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

**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.

**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.

**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.

Set Constraints

**4.1 Set Variables and Set Domains**

Set is defined as an ordered collection of integers using class JaCoP.core.IntervalDomain and a set domain as abstract clss JaCoP.set.core.SetDomain. Currently, there exists 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.

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

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

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

and an empty set domain as

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

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

**4.2 Set Constraints**

JaCoP implements number of set constraints specified in appendix [A.2](http://jacopguide.osolpro.com/guideJaCoP.html#x1-510002). Constraints AinS, AeqB and AinB are primitive constraints and can be reified and used in other constraints, such conditional and logical. Other constraints are treated as ordinary JaCoP constraints.

Consider the following code that uses union constraint.

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

It performs operation {{1}*..*{1*..*4}}⋃ {{2}*..*{2*..*5}} = {{}*..*{1*..*10}} and produces {{1}*..*{1*..*4}}⋃ {{2}*..*{2*..*5}} = {{1*..*2}*..*{1*..*5}}. This represents 108 possible solutions.

**4.3 Search**

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

JaCoP still uses DepthFirsysearch but needs different methods for set variable selection implementing ComparatorVariable and a method for value selection implementing Indomain. The special methods are specified in appendix [B.2](http://jacopguide.osolpro.com/guideJaCoP.html#x1-560002). In addition, variable selection methods MostConstrainedStaticand MostConstrainedDynamic will work also.

An example search can be specified as follows.

   Search<SetVar> search = new DepthFirstSearch<SetVar>();   
   
   SelectChoicePoint<SetVar> select = new SimpleSelect<SetVar>(   
                                vars,   
                                new MinLubCard<SetVar>(),   
                                new MaxGlbCard<SetVar>(),   
                                new IndomainsetMin<SetVar>());   
   search.setSolutionListener(new SimpleSolutionListener<SetVar>());   
   
   boolean result = search.labeling(store, select);

Search

JaCoP offers methods for finding a single solution, all solutions and a solution that minimizes a given cost function.

**5.1 Depth First Search**

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

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

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

where *T*is type of variables we are using for this search (usually IntVar or SetVar), *var*is a list of variables, *varSelect*is a comparator method for selecting variable and *tieBreakerV arSelect*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](http://jacopguide.osolpro.com/guideJaCoP.html#x1-54000B). This search, for varibales of type IntVar creates choice points *xi* = *val*and *xi≠val*where *xi* is variable identified by variable selection comparators and *val*is the value determined by indomain method. For variables of type SetVar the coice is made between *val*∈ *xi* or *val*∈∕*xi*.

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

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

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

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

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

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

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

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

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

   label.setTimeOut(10);

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

**5.1.1 Restart search**

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

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

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

   public class CostListener<T extends Var> extends   
                                         SimpleSolutionListener<T> {   
   
      public boolean executeAfterSolution(Search<T> search,   
                                      SelectChoicePoint<T> select) {   
         boolean returnCode = super.executeAfterSolution(search,   
                                                         select);   
   
         CostValue = cost.value();   
         return returnCode;   
      }   
   }

**5.2 Search plug-ins**

The search-plugin is an object, which is informed about the current state of the search and may influence the behavior of the search. They are divided into search-plugins that change the search behavior and plugins used for collecting and sharing information. Table [5.1](http://jacopguide.osolpro.com/guideJaCoP.html#x1-450011) lists the search-plugins available in JaCoP and their membership in a respective group.

**Table 5.1:**Search plug-ins available in JaCoP.

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

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

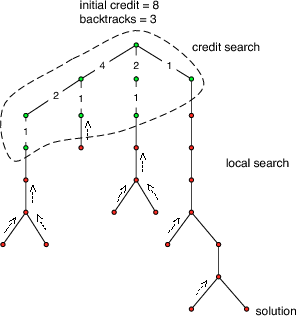
* *Solution listener*plug-in is called by search when a solution is found
* *Exit child listener*plug-in is called every time the search exits the search subtree (it has four different methods; two methods, which are called when the search has exited the left subtree and two methods for the right subtree).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

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

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

**5.3 Credit search**

Credit search combines credit based exhaustive search at the beginning of the tree with local search in the rest of the tree [[1](http://jacopguide.osolpro.com/guideJaCoP.html#XCreditSearch)]. In JaCoP, the credit search is controlled by three parameters: number of credits, number of backtracks during local search and maximum depth of search. In Figure [5.1](http://jacopguide.osolpro.com/guideJaCoP.html#x1-460011) 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.



**Figure 5.1:**Credit search example.

An example of the command which produces the search tree depicted in Fig. [5.1](http://jacopguide.osolpro.com/guideJaCoP.html#x1-460011) is as follows.

   SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(vars,   
                                       new SmallestDomain<IntVar>(),   
                                       new IndomainMin<IntVar>());   
   
   int credits=8, backtracks=3, maxDepth=1000;   
   CreditCalculator<IntVar> credit = new CreditCalculator<IntVar>(   
                                                  credits,   
                                                  backtracks,   
                                                  maxDepth);   
   Search<IntVar> search = new DepthFirstSearch<IntVar>();   
   search.setConsistencyListener(credit);   
   search.setExitChildListener(credit);   
   search.setTimeOutListener(credit);   
   
   boolean result = search.labeling(store, select);

**5.4 Limited discrepancy search**

Limited discrepancy search (LDS) uses the partial search method proposed in [[6](http://jacopguide.osolpro.com/guideJaCoP.html#XHaGi95)]. It basically allows only a number of different decisions along a search path, called discrepancies. If the number of discrepancies is exhausted backtracking is initiated. The number of discrepancies is specified as a parameter for LDS.

An example of LDS with one discrepancy is as follows.

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

**5.5 Combining search**

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

   Search<IntVar> slave = new DepthFirstSearch<IntVar>();   
   SelectChoicePoint<IntVar> selectSlave =   
          new SimpleSelect<IntVar>(vars2,   
                                   new SmallestMin<IntVar>(),   
                                   new SmallestDomain<IntVar>(),   
                                   new IndomainMin<IntVar>());   
   slave.setSelectChoicePoint(selectSlave);   
   
   Search<IntVar> master = new DepthFirstSearch<IntVar>();   
   SelectChoicePoint<IntVar> master =   
          new SimpleSelect<IntVar>(vars1,   
                                   new SmallestMin<IntVar>(),   
                                   new SmallestDomain<IntVar>(),   
                                   new IndomainMin<IntVar>());   
   master.addChildSearch(slave);   
   
   boolean result = master.labeling(store, selectMaster);

**Appendix A  
JaCoP constraints**

**A.1 Primitive constraints**

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| *X*= *Const* | XeqC(X, Const) |
| *X*= *Y* | XeqY(X, Y) |
| *X≠Const* | XneqC(X, Const) |
| *X≠Y* | XneqY(X, Y) |
| *X > Const* | XgtC(X, Const) |
| *X > Y* | XgtY(X, Y) |
| *X*≥ *Const* | XgteqC(X, Const) |
| *X*≥ *Y* | XgteqY(X, Y) |
| *X < Const* | XltC(X, Const) |
| *X < Y* | XltY(X, Y) |
| *X*≤ *Const* | XlteqC(X, Const) |
| *X*≤ *Y* | XlteqY(X, Y) |
| *X*⋅ *Const*= *Z* | XmulCeqZ(X, Const, Z) |
| *X*⋅ *Y*= *Z* | XmulYeqZ(X, Y, Z) |
| *X*÷ *Y*= *Z* | XdivYeqZ(X, Y, Z) |
| *X* mod*Y*= *Z* | XmodYeqZ(X, Y, Z) |
| *X*+ *Const*= *Z* | XplusCeqZ(X, Const, Z) |
| *X*+ *Y*= *Z* | XplusYeqZ(X, Y, Z) |
| *X*+ *Y*+ *Const*= *Z* | XplusYplusCeqZ(X, Y, Const, Z) |
| *X*+ *Y*+ *Q*= *Z* | XplusYplusQeqZ(X, Y, Q, Z) |
| *X*+ *Const*≤ *Z* | XplusClteqZ(X, Const, Z) |
| *X*+ *Y*≤ *Z* | XplusYlteqZ(X, Y, Z) |
| *X*+ *Y > Const* | XplusYgtC(X, Y, Const) |
| *X*+ *Y*+ *Q > Const* | XplusYplusQgtC(X, Y, Q, Const) |
| *XY*= *Z* | XexpYeqZ(X, Y, Z) |
|  |  |

**A.2 Set constraints**

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| *e*∈ *A* | EinA(e, A) |
| *S*1 = 2 | AeqB(S1, S2) |
| *S*1 ⊆ *S*2 | AinB(S1, S2) |
| *S*1 ⋃ *S*2 = *S*3 | AunionBeqC(S1, S2, S3) |
| *S*1 ⋂ *S*2 = *S*3 | AintersectBeqC(S1, S2, S3) |
| *S*1 \ *S*2 = *S*3 | AdiffBeqC(S1, S2, S3) |
| *S*1 *<> S*2 | AdisjointB(S1, S2) |
| Match | Match(Set, VarArray) |
| #*S*1 = *X* | CardAeqX(S, X) |
| Weighted sum < S, W > = X | SumWeightedSet(S, W, X) |
| *Set*[*X*] = *Y* | ElementSet(X, Set, Y) |
|  |  |

**A.3 Logical, conditional and reified constraints**

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| ¬*c* | Not(c); |
| *c*1 ⇔ *c*2 | Eq(c1, c2); |
| *c*1 ∧ *c*2 ∧⋅⋅⋅∧ *cn* | PrimitiveConstraint[] c = {c1, c2, …cn}; |
|  | And(c); |
|  | or |
|  | ArrayList c = |
|  | new ArrayList(); |
|  | c.add(c1); c.add(c2); …c.add(cn); |
|  | And(c); |
| *c*1 ∨ *c*2 ∨⋅⋅⋅∨ *cn* | PrimitiveConstraint[] c = {c1, c2, …cn}; |
|  | Or(c); |
|  | or |
|  | ArrayList c = |
|  | new ArrayList(); |
|  | c.add(c1); c.add(c2); …c.add(cn); |
|  | Or(c); |
| *X*in*Dom* | In(X, Dom); |
| *c*⇔ *B* | Reified(c, B); |
| *c*⇔¬*B* | Xor(c, B); |
| if *c*1then*c*2 | IfThen(c1, c2); |
| if *c*1then*c*2else*c*3 | IfThenElse(c1, c2, c3); |
|  |  |
| **Boolean operations on variables** |  |
|  | BooleanVar[] b = {b1, b2, …, bn}; |
|  | or |
|  | ArrayList b = new ArrayList<="" span=""> |
|  | b.add(b1); b.add(b2); …b.add(bn); |
|  | BoolanVariable result = new BooleanVar(store, "result"); |
| *result*= *b*1 ∧ *b*2 ∧⋅⋅⋅∧ *bn* | AndBool(b, result) |
| *result*= *b*1 ∨ *b*2 ∨⋅⋅⋅∨ *bn* | OrBool(b, result) |
| *result*= *b*1 ⊕ *b*2 | XorBool(b1, b2, result) |
| *result*= *b*1 → *b*2 | IfThenBool(b1, b2, result) |
| *result*= *b*1 == *b*2 == ⋅⋅⋅ == *bn* | EqBool(b, result) |
|  |  |

**A.4 Global constraints**

*x*1 + *x*2 + ⋅⋅⋅ + *xn* = *sum*

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

*w*1 ⋅ *x*1 + *w*2 ⋅ *x*2 + ⋅⋅⋅ + *wn* ⋅ *xn* = *sum*

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

**alldifferent**([*x*1*,x*2*,…,xn*])

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

**alldiff**([*x*1*,x*2*,…,xn*])

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

**alldistinct**([*x*1*,x*2*,…,xn*])

IntVar[] x = {x1, x2, …, xn};  
Alldistinct(x);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Alldistinct(x);

**among**([*x*1*,x*2*,…,xn*]*,val,count*)

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

**amongVar**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,ym*]*,count*)

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

**assignment**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,yn*])

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

**circuit**([*x*1*,x*2*,…,xn*])

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

**count**(*value,*[*x*1*,x*2*,…,xn*]*,var*)

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

**cumulative**([*t*1*,t*2*,…,tn*]*,*[*d*1*,d*2*,…,dn*]*,*[*r*1*,r*2*,…,rn*]*,ResourceLimit*)

IntVar[] t ={t1, t2, …, tn};  
IntVar[] d ={d1, d2, …, dn};  
IntVar[] r = {r1, r2, …, rn};  
IntVar Limit = new IntVar(…);  
Cumulative(t, d, r, Limit);[2](http://jacopguide.osolpro.com/guideJaCoP5.html#fn2x8)  
or using ArrayList

**diff2**([[*x*1*,y*1*,dx*1*,dy*1]*,…,*[*xn,yn,dxn,dyn*]])

IntVar[][] r = {{x1,y1,dx1,dy1}, …,  
{xn,yn,dxn,dyn}};  
Diff(r); or Diff2(Store, r);[1](http://jacopguide.osolpro.com/guideJaCoP4.html#fn1x8)  
or using ArrayList<arraylist></arraylist

**or**

**diff2**([*x*1*,…,xn*]*,*[*y*1*,…,yn*]*,*[*dx*1*,…,dxn*]*,*[*dy*1*,…,dyn*])

IntVar[] x = {x1, …, xn};  
IntVar[] y = {y1, …, yn};  
IntVar[] dx = {dx1, …, dxn};  
IntVar[] dy = {dy1, …, dyn};  
Diff(x, y, dx, dy); or Diff2(Store, x, y, dx, dy);[1](http://jacopguide.osolpro.com/guideJaCoP4.html#fn1x8)  
or using ArrayList

**distance**(*x,y,dist*)

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

**element**(*Index,*[*n*1*,n*2*,…,nn*]*,V alue*)

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

**element**(*Index,*[*x*1*,x*2*,…,xn*]*,V alue*)

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

**extensionalSupport**([*x*1*,x*2*,…,xn*]*,*{{1*,*2*,…,n*}*,*{*…*}*,…,*{*…*}})

**extensionalConflict**([*x*1*,x*2*,…,xn*]*,*{{1*,*2*,…,n*}*,*{*…*}*,…,*{*…*}})

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

**gcc**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,ym*])

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

**min**([*x*1*,x*2*,…,xn*]*,Xmin*)

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

**max**([*x*1*,x*2*,…,xn*]*,Xmax*)

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

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

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

**geost**(*objects,constraints,shapes*)

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

**regular**(*fsm,*[*x*1*,x*2*,…,xn*])

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

**sequence**([*x*1*,x*2*,…,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,*[*x*1*,x*2*,…,xn*])

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

**values**([*x*1*,x*2*,…,xn*]*,count*)

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

**lex**([[*x*11*,x*12*,…,x*1*n*]*,…,*[*xk*1*,xk*2*,…,xkm*]])

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

**soft-alldifferent**([*x*1*,x*2*,…,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**([*x*1*,x*2*,…,xn*]*,hardCounters,countedV alues,softCounters,cost,violation***\_***measure*)

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

**Appendix B  
JaCoP search methods**

**B.1 Variable and value selection for FDVs**

* **value selection methods**

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

* **variable selection methods**

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

**B.2 Variable and value selection for set variables**

* **value selection methods**

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

* **variable selection methods**

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

**B.3 Search methods**

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

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

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

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

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

* **Search for all solutions**

additional switches for search for all solutions.

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

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

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

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

   label.printAllSolutions();

* **Search for optimal solution**

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

**B.4 Important methods for search plug-ins**

* **solution listener**– SimpleSolutionListener

important methods

* + printAllSolutions()
  + getSolutions()
  + solutionsNo()
  + recordSolutions(boolean status)
  + searchAll(boolean status)
  + executeAfterSolution(Search search, SelectChoicePoint select)
  + setChildrenListeners(SolutionListener child)
* **time-out listener**– one can set customized time-out listener that implements TimeOutListener interface to perform specific actions at time-out (e.g., print information). Method executedAtTimeOut(int solutionsNo) will be executed at time-out.

**Appendix C  
JaCoP debugging facilities**

**C.1 Available switches**

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

**C.2 CPviz interface**

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

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

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

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

The next steps requires installation of CPviz software.

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

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

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

xml version="1.0" encoding="UTF-8"?>   
   
<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.

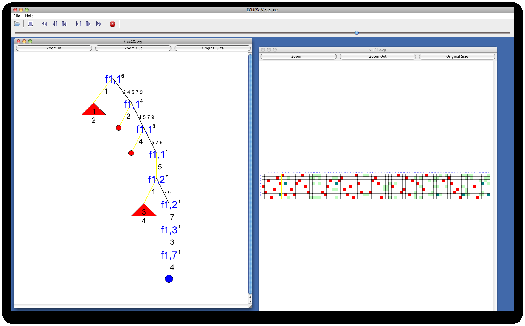
The visualization is provided by issuing the following command.

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](http://jacopguide.osolpro.com/guideJaCoP.html#x1-610141).



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