Universitá degli Studi di Perugia
Facoltá di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Magistrale in

Informatica

Sistemi con Vincoli

Relazione Libreria SoftLib v5

Studenti:
Marco Bizzarri
Andrea Nardinocchi
Cristiano Santoni

Professore:
Prof. Stefano Bistarelli

Anno Accademico 2011 - 2012

1 Weights handling

The weights are one of the basic factors of the soft CSP and it is obvious that they need supporting structures that incorporate these features and give the opportunity to the programmer to specify the criteria for the calculation of the weights and to assign them to individual variables / couple of variables.

1.1 PrimitiveWeight

The heart of the library is curiously given, at least at the conceptual level, by this completely empty class whose unique purpose is "bring together" two other abstract classes: *PrimitiveBinaryWeight* and *PrimitiveUnaryWeight*.

```
public abstract class PrimitiveBinaryWeight extends PrimitiveWeight{
   public abstract Constraint setWeight(SoftStore store, SoftExtensionalSupport c);
   public abstract Constraint setWeight(SoftStore store, IntVar x, IntVar y);
   public abstract int evaluateWeight(int x, int y);
}
```

```
public abstract class PrimitiveUnaryWeight extends PrimitiveWeight{
       public abstract Constraint setWeight(SoftStore store, IntVar x);
       public abstract int evaluateWeight(int x);
}
```

These two classes define: the function prototype *setWeight* (illustrated below) and the function *evaluateWeight* that returns the weight in the form of integer for each value of the domain (in case of the unary) or for the couple of values of the respective domains (in case of the binary). Inheriting from these classes the programmer has the power to define his personal criteria to calculate weights and to implement them in the library or in his personal project.

1.1.1 the setWeight function

Because everything works properly with the library JaCoP, is obviously necessary that we have to transform the associated weight in a crisp constraint with the aim of leaving the difficult task of the searching of the solutions directly to the standard functions of the library created by Krzysztof Kuchcinski and Radoslaw Szymanek; the role is played by the function *setWeight* that receives as a parameters a *SoftStore* and one or two variables, and returns as output the constraint built according to the type of weight.

1.2 Implemented Weights

The library implements, however, the best known and useful methods for the calculation of weights unary or binary, listed below:

WeightDistance

It assigns the weight by calculating the distance between the values of the two domains. The assignment x = 3 and y = 7 will have as the weight the distance between the values 3 and 7.

WeightMax / WeightMin

It chooses as the weight respectively the larger / smaller value between the values of the two domains. The assignment x = 3 and y = 7 will have as the weight 3 (for *WeightMin*) or 7 (for *WeightMax*).

WeightExtensional

It is natural that exists the possibility that the weights could not be calculated dynamically via specially defined criteria but, instead, should be assigned by the user through lists that, for each combination of values of the domain, define a weight. The *WeightExtensional* does this retrieving the list of weights directly from the constructor.

WeightUnaryDistance

It assigns the weight calculating the distance between the value of the domain and a constant specified at the time of creation.

WeightUnaryProportional

It calculates the final weight by multiplying the value of the domain for a constant passed as parameter to the constructor.

WeightUnaryExtensional

As for the binary *WeightExtensional*, is passed as parameter a list of weights that, sequentially, are assigned to the values of the domain.

2 Soft Constraints implementation

Of course, with the implementation of the weight functions, becomes necessary to extend the management of constraints defining a superclass that allows the developer to add to the variable/couple of variables a weight, given as a specialization of the class *PrimitiveUnaryWeight* /*PrimitiveBinaryWeight*.

2.1 SoftConstraint

This interface defines the only method needed to expand a crisp constraint to a soft constraint: *getWeight*. This function is necessary for the execution of the soft arc consistency procedure and simply returns the weight assigned from the impose of the constraint as *PrimitiveWeight*. Of course it is also necessary to add a new prototype of *impose* that, in addition to take a custom implementation of the *Store* called *SoftStore*, it takes a *PrimitiveUnaryWeight* or a *PrimitiveBinaryWeight* according to the specifications of the constraint. To properly understand what we mean and how a user could implement its own personal constraint show two specializations of *SoftContraint* widely used.

2.1.1 SoftXneqC

This *Constraint* allows the programmer to define as a constraint that the domain value assigned to the variable must be different from the constant c passed as a parameter to the constructor together to the variable x. Since this is a specification of *XneqC*, present into the JaCoP library, it is obviously that extends from the same and, as we suggested earlier, that implements the interface *SoftConstraint*. Moreover, being a soft constraint, any type of attempt to apply the constraint with the function *impose* passing by a normal *Store* is interrupted by an exception that indicates a "unsupported operation": it is necessary a *SoftStore*. The *impose* will simply attach the instance to a local variable weight class (necessary step to regain the weight through the function *getWeight*) to get back the instance of the crisp constraint generated by *setWeight* function, add it to store and recall the *impose* method of the super class (by calling the function *impose* by passing only the *SoftStore* will ensure that the class will automatically generate the weight through *WeightUnaryProportional* with constant 1).

```
public void impose(SoftStore store) {
        this.impose(store, new WeightUnaryProportional(1));
}

public void impose(SoftStore store, PrimitiveUnaryWeight w) {
        this.w = w;
        Constraint cost = w.setWeight(store, x);
        store.addToUnary(x, new CoupleConstraint(this, cost));
        super.impose(store);
}
```

2.1.2 SoftXneqY

This *Constraint* allows you to constraint the assignment to the x variable must be different from the value assigned to y. Even in this case the inheritance comes directly from the class *XneqY*, present into the JaCoP library, and the implementation of the interface *SoftContraint*.

2.2 SoftExtensonalSupport

Also with regard to the *ExtensionalSupport* of Jacop, has been implemented a version that adds the ability to associate some weights to the constraints imposed call *SoftExtensionalSupport*. The management is carried out using the original *ExtensionalSupport* and rebuilding it as a new instance for each call of the method *impose*. The *SoftExtensionalSupport* associate, to the list containing the tuples passed as a parameter to the constructor, a weight function of type *PrimitiveBinaryWeight* when the *impose* method is called.

```
public interface SoftExtensionalSupport {
    public void impose(SoftStore store);
    public boolean testTuplesFromConstructor();
    public int getTuplesFromConstructor(int x, int y);
    public int sizeTuplesFromConstructor();
    public int sizeTuplesFromConstructor(int x);
    public IntVar[] getList();
}
```

The JaCoP library implements three basic types of *ExtensionalSupport* which are different according to the types of algorithms used in their implementation:

- *ExtensionalSupportVA* that attempts to balance the occupied memory and efficiency in the execution;
- *ExtensionalSupportSTR* that implements the technique presented by Christopher Lecoutre;
- *ExtensionalSupportMDD* that implements the technique presented by professor Roland Yap. We decided to implement the two most important and most used of the three presented: *softExtensionalSupportVA* and *softExtensionalSupportSTR*. Both inherit from their crisp reference class and implement the interfaces *SoftConstraints* and *SoftExtensionalSupport*.

3 Constraint storing

To compute the soft arc consistency on a fuzzy semiring constraint problem (FCSP), it's necessary to know the weight for both binary and unary constraints for each variables and have access to the constraints information.
In the previous version of the *SoftLib* all the constrains were directly imposed and stored in the *SoftStore* after their creation, making impossible to retrieve them in the future.
To make the constraints available, it was necessary to introduce additional data structures in the *SoftStore* class to contain constraints information before the *impose* method call.
In the latest version of the library have been introduced two kind of data structures: one containing the weight constraints added to the *SoftStore* (representing both binary and unary variables constraints) and another one containing the same constraints but expressed in a more convenient numeric form (*WeightStore* object). This second data structures is filled after all the constraint are set and just before the *softArcConsistency* computation. If the user will not call the *softArcConsistency* method, the *WeightStore* object will not be used.

```
public class SoftStore extends Store {

    [...]
    public ArrayList<ExBinaryReference> binary;
    public HashMap<IntVar,ArrayList<CoupleConstraint>> unary;

    protected WeightStore wStore;
    [...]

}
```

First of all the first data structure is going to be described.
As one can see in the previous piece of code, the structure is composed by:

- An *arraylist* of *ExBinaryReference* objects which keep information about binary constrains;
- An *hashmap* containing an *IntVar* (see JaCoP documentation) variable as key and an *arraylist* of *CoupleContraint* objects which keep informations about unary constraints.

Where:

CoupleContraint is a class created to match a *SoftConstraint* and the corresponding weight classical constraint by storing them as attributes;

```
public class CoupleConstraint {

    private SoftConstraint constraint;
    private Constraint cost;
    [...]

}
```

ExBinaryReference is a class which contains a couple of *IntVar* variables and the list of all the constraints they are involved in.

```
public class ExBinaryReference {

    private IntVar nodex, nodey;
    public ArrayList<CoupleConstraint> constraints;
    [...]

}
```

3.1 addToBinary and addToUnary methods

The unary and binary constrains data structures are filled when a new constraint is imposed to the *SoftStore* by using two separate methods of the *SoftStore* class:

- *public void* ***addToBinary*** *(IntVar x, IntVar y, CoupleConstraint c)*:
  This method is used to store constraints in which two variables are involved. It iterates between the variables that already exist in the *ExBinaryReference* list to check if the couple passed as parameter is already involved in some constraints. If so the *CoupleConstraint* parameter is added to the corresponding list of constraints, else a new *ExBinaryReference* object will be created and added to the list of *ExBinaryReference* objects.

- *public void* ***addToUnary*** *(IntVar x, CoupleConstraint c)*:
  This method is used to store single variable constraints. If the variable passed as parameter is already in the hashmap, the new constraint is added to it's constraint list, otherwise a new hashmap entry will be created.

3.2 WeightStore

As mentioned in the first part of the chapter, another data structure has been introduced to manipulate constrains.
To compute the *softArcConsistency* on the constraints graph is necessary to update the weights according to the values of the variables domain. To allow this operation the *WeightStore* class has been created.

```
public class WeightStore {

    public ArrayList<WeightBinaryDomains> wBinary;
    public HashMap<IntVar,WeightUnaryDomain> wUnary;
    [...]

}
```

The class is composed by an array of *WeightBinaryDomains* which stores binary constraints informations, and an *hashmap* of *IntVar* and *WeightUnaryDomain* used to store unary constraints.
This class does not need any method: it is simply used as data container and it is filled by some method of the *SoftStore* class which will be described in the next chapter.

Where:

WeightBinaryDomains is a class which contains the couples of IntVar variables, a list of hashmap used to match all the variables domain values with the corresponding weight as integer value (the *intCouple* class object matches variables domain values) and an *ExtensionalSupportSTR* constraint which will stores current filtered constraints' informations after the execution of the *softArcConsistency*. This constraints will be used in the final *impose*.

```
public class WeightBinaryDomains {

    public IntVar x, y;
    public ArrayList<HashMap<IntCouple, Integer>> w;
    public ExtensionalSupportSTR weightCon;
    [...]

}
```

WeightUnaryDomain is a class which contains a list of integer which stores the variable domain values, a list of hashmap used to match each domain value with the corresponding weight and an *ExtensionalSupportSTR* constraint which will stores current constraints filtered informations after the *softArcConsistency* will be executed. This constraints will be used in the final *impose*.

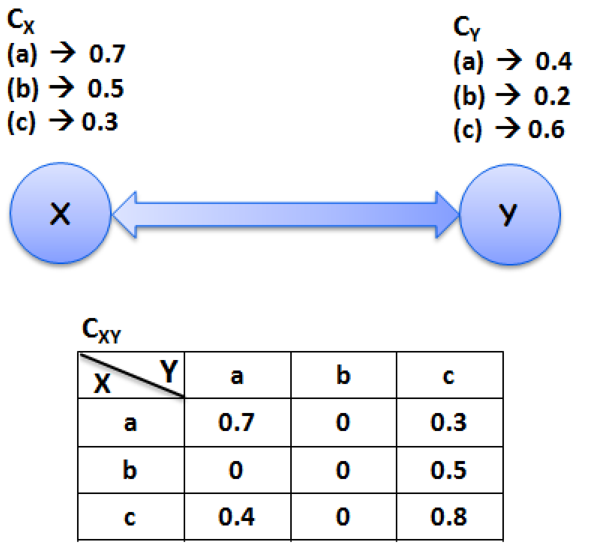
```
public class WeightUnaryDomain {

    public ArrayList<Integer> xDom;
    public ArrayList<HashMap<Integer, Integer>> w;
    public ExtensionalSupportSTR weightCon;
    [...]

}
```

4 Soft Arc Consistency

In this chapter we will illustrate the Soft Arc Consistency method and discuss about its implementation and the auxiliary data structured used.
In the framework of classical (crisp) CSPs, local consistency is a property of the CSP which permit to make assignments to the variables one by one without incur in inconsistent assignments. Usually this property has not to be true for a generic CSP, so what we can do is to adjust the variables domain to ensure local consistency. This process can be done as a preprocessing step or as a propagation step after every new assignment during the search algorithm.

4.1 Soft arc consistency algorithm

To extend the idea of local consistency to the framework of soft CSPs can be done by decreasing the preference value (of a variable assignment) instead of reducing variables domain. In the soft CSP framework we represent the cost/preference value associated to a soft constraint with an element from a semiring $(A, +, \cdot)$ where $\cdot$ is used to combine constraints and $+$ is used for the induced pseudo order. An additional condition to ensure the soft arc consistency is that the $\cdot$ operation has to be idempotent. The semring ([0,1], *MAX*, *MIN*) on which the Fuzzy CSPs are based is an example of a valid semiring on which the soft local consistency can be ensured. From now on we will assume to work on Fuzzy CSPs. Let's see an example to better understand the operations that compose the soft arc consistency algorithm: in figure 4.1 we can see a constraint graph of a soft CSP, the graph is composed by two nodes that represent the variables X and Y, each one with its own constraint (C_X and C_Y), and an arc connecting the two nodes which represent the C_{XY} constraint. To ensure soft arc consistency to the CSP in figure 4.1 we have



X \ Y	a	b	c
a	0.7	0	0.3
b	0	0	0.5
c	0.4	0	0.8

Figura 4.1: graph constraint example for a soft CSP example

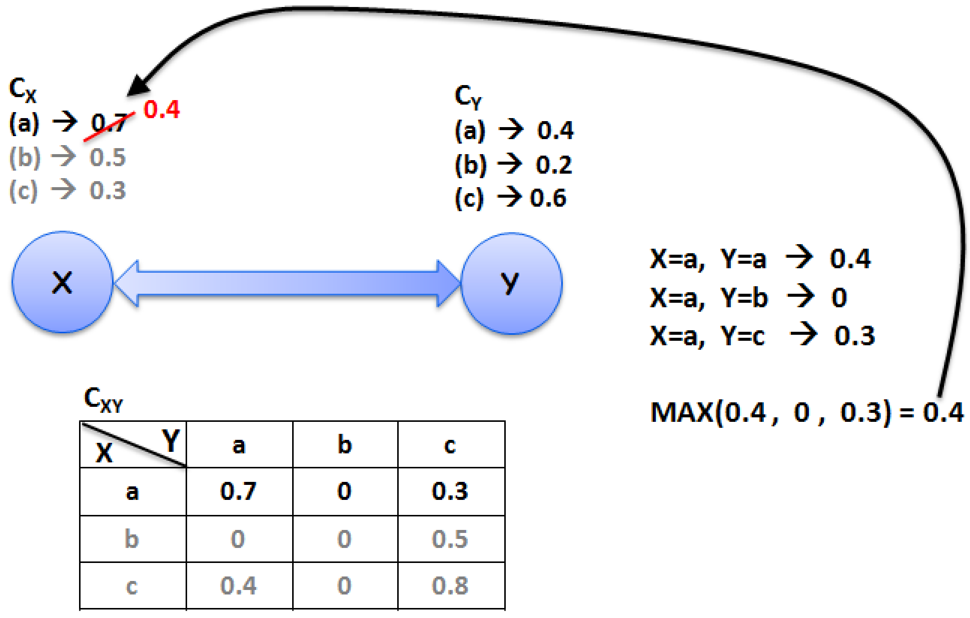
to:

- choose a variable (let's say X)

- choose a value in the X domain (let's say a)
- for every domain value of the other variable:
  - get the assignment preference value by combining the preference values of the constraints with $\cdot$ operation
  - choose the maximum of them as the new preference value for the choosen domain value of the choosen variable (using the relation induced by the $+$ operation)

These operations have to be repeated until the soft arc consistency is reached. In our CSP (Fuzzy CSP) we combine the preference values of the constraints using *MIN* and choose the maximum using the relation induced by *MAX*. In figure 4.2 we can see the operations of the Soft Arc Consistency algorithm for $X = \text{a}$.

Using soft CSP it is possible to express undesired value for a variable even if that value is in



X \ Y	a	b	c
a	0.7	0	0.3
b	0	0	0.5
c	0.4	0	0.8

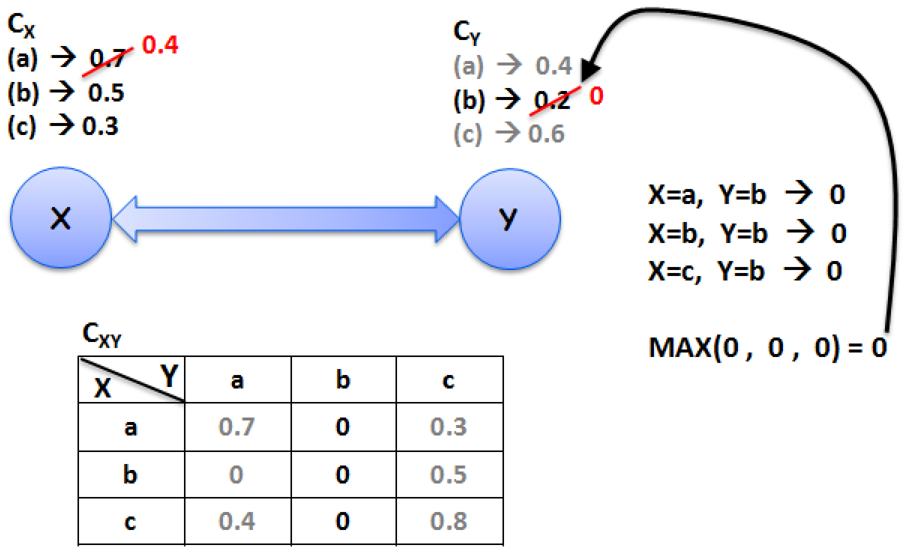
Figura 4.2: A STEP OF THE SOFT ARC CONSISTENCY WITH X = A FIXED

the variable domain. Using the absorbing element *0** fo the $\cdot$ operation (in Fuzzy CSPs it's the 0 for the *MIN* operation) as preference value for the undesired domain value will bring to a solution with a preference value of *0** that will be discarded from the set of valid solutions. An example of propagation of the *0** during the soft arc consistency algorithm is show in figure 4.3.

Once the Soft Arc Consistency algorithm finishes all its operations, in order to speed up the search process, it is possible to remove from the variables domains the values with a preference value of *0**.

4.2 WeightStore filling

In order to operate in a more efficent way on the constraints regarding the preference value, we store them in the auxiliary data structure (*wUnary* and *wBinary* fields of the class *WeightStore*)



X \ Y	a	b	c
a	0.7	0	0.3
b	0	0	0.5
c	0.4	0	0.8

Figura 4.3: Propation of 0 during Soft Arc Consistency

explained in the previous chapter. The methods used to fill this two fields are *private void formatUnaryWeight(WeightStore wstore)* and *private void formatBinaryWeight(WeightStore wstore)*. The method *private void formatUnaryWeight(WeightStore wstore)* is used to fill the *wUnary* field and it works iterating first on all the *IntVar* involved in at least one unary constraint and, for each of these variables, creating a new *WeightUnaryDomain* which represent the list of couple <domain value, preference value>. Fixed the variable, *formatUnaryWeight* iterates on all domain values to fill the list of <domain value, preference value>. To get the preference value associated to any domain value, the method uses a function common to all the *PrimitiveUnaryWeight* subclasses that is *public int evaluateWeight(int x)* that takes a value from the domain and returns the related preference value.
For the binary constraint the method used to get their tabular form is *private void formatBinaryWeight(*WeightStore *wstore)* that operates in a similar way to the previous one. It takes all the binary constraints from the field *binary* of the *SoftStore* and for each of them it creates a new *WeightBinaryDomains*. Fixed the two variables, it iterates on every combination of a value from a variable domain with a value from the domain of the other variable and, using *evaluateWeight(int x, int y)* declared in all subclasses of *PrimitiveBinaryWeight*, fill the HashMap that represent the list of triple <X domain value, Y domain Value, preference value>.

4.3 the softArcConsistency() method

Our implementation of soft arc consistency follows the scheme provided by AC-1 so what we do is to iterate soft arc consistency algorithm until no preference value changes. The condition previously explained is the condition of the main loop in the method, for each iteration we take a WeightBinaryDomains from the list wBinary in the and iterate as many times as the number of elements of the longest list among the list of binary constraints related to X and Y, the list of unary constraints related to X and the list of unary constraints related to Y. At this point we have fixed a binary constraint and two unary constraints (one for each variable involved in the binary constraint) and we can check if the subproblem is consistent: a first couple of nested for loops iterates on the domain of the first variable combining each value with all the domain values of the second variable and checking if the arc consistency is satisfied (assigning the new preference value if it is not). The second couple of nested for loops does the same operations inverting the role of the first and the second variable.In the last lines of the method we can find the call of two methods used to produce the new preference value constraints from the tabular value just processed (*weightUnaryFilter(wStore)* and *weightBinaryFilter(wStore)*) and the "dispose" operation for *binary* and *unary*, no more needed.

4.4 Constraint unification

Once the soft arc consistency algorithm has succesfully completed his preprocessing, we have to create new precerence value constraints based on the new data produced. This task is performed by *weightUnaryFilter(wStore)* and *weightBinaryFilter(wStore)*.

4.4.1 Unary Constraints

For the unary constraints *weightUnaryFilter(wStore)* iterates on *wUnary* (containing one *WeightUnaryDomain* per variable) and it scans the list of constraints represented by a list of *HashMap*<domain value, preference value>. Fixed the *HashMap* representing a constraint on the preference value, it stores the supported tuples with a preference value different from 0. During the first iteration of the just explained loop, we compute the real dimension for the supported tuples array (initially estimated equal to the variable domain size) and, before starting the second iteration, we compact the array. During this phase, when we find more than one preference value for a same domain value (tipically when there are two constraint for the preference value of a variable), we "combine" the value using the *MIN* operation getting the same result that would be generated during the search procedure from the *getCost()* function. In the last part of the function the domain for the new preference value variable is computed and the new constraint is created.

4.4.2 Binary Constraints

For the binary constraint, the method *weightBinaryFilter (wStore)* has the task to create the new preference value constraints. It has the same structure of the just explained method: the outer loop iterate on the *WeightBinaryDomains* list *wBinary* and for each of them it scans all the preference value constraints (represented as HashMap<(X domain value, Y domain value), preference value>) stored in a list. For each couple of variable represent by a *WeightBinaryDomains*, the method produces an array of supported tuples, combining with the *MIN* operation the preference values regarding the same domain value. As for *weightUnaryFilter()*, in the last lines the domain of the new preference value variable is computed and the new constraint created.

4.5 Preference value constraint imposition

Due to the need to perform operations on the constraint before their imposition, the impose operation for the constraints regarding the preference values has to be delayed after the call of *softArcConsistency()*. In order to standardize the operations to do, either the user want or want not to call the soft arc consistency method, the impose of the preference value constraints was separated and can be applied by calling the *imposeWeights()* method. To check if *softArcConsitency()* was called or not, *imposeWeight()* check the *unary* and *binary* fields and, if they are equal to *null*, the method jumps to the branch of the new constraint imposition, while if both the fields contains data it impose the constraint contained in the structures. At the end of the “after softArcConsistency()” branch the weightStore is “disposed”.

5 Example of SoftLib v5 usage

Now we are going to show an example of the current library version usage, with some explanations of the difference with the previous version.

A simple example of the library usage is the following:

```
SoftStore softStore = new SoftStore(SoftStore.Semiring.FUZZY);

int varNum = 5000;
int localDomainMax = 4;
int localDomainMin = 0;

IntVar[] list = new IntVar[varNum];
SoftXltC ucon;
for(int i=0; i< list.length; i++){
   list[i] = new IntVar(softStore, "X"+i, localDomainMin, localDomainMax);
   if(i%2==0){
      ucon = new SoftXltC(list[i], localDomainMax);
      ucon.impose(softStore, new WeightUnaryProportional(150));
   }else{
      ucon = new SoftXltC(list[i], localDomainMax);
      ucon.impose(softStore, new WeightUnaryProportional(1));
   }
}

softStore.softArcConsistency();

softStore.imposeWeights();

Search<IntVar> label = new DepthFirstSearch<IntVar>();
SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(list,
                                                            new SmallestDomain<IntVar>(),
                                                            new IndomainMin<IntVar>());

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

label.labeling(softStore, select, softStore.getCost());

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

In this example we build the variables contraints graph as a chain, the number of the nodes is specified by the "*varNum*" variable and their domain's bounds are specified by the "*localDomainMin*" and "*localDomainMax*" variables. The constraint used to link together the nodes is the *SoftXltC* and the weight contraint used is *WeightUnaryProportional*.
In this sample we want to perform the soft arc consistency operation so, as explained in the previous chapters, the semiring type parameter in the *SoftStore* object creation must be "FUZZY".

As the reader can see the only difference of the library usage with the previous version of the library is the *imposeWeight()* method call. It is needed either if the softArcConsistency is called or not, because the weight constraints would not be imposed until this method call.