Università degli Studi "G. D'Annunzio"
Dipartimento di Scienze

Solving Weighted Argumentation Frameworks with Soft Constraints

Stefano Bistarelli Daniele Pirolandi
Francesco Santini

December 2, 2009

Technical Report no. R-2009-003 *Research Series*

Solving Weighted Argumentation Frameworks with Soft Constraints

Stefano Bistarelli [1,2,3] Daniele Pirolandi [1] Francesco Santini [2,3]

[1] *Dipartimento di Matematica e Informatica, Università di Perugia*
Via Vanvitelli, 1 Italy
bista@dmi.unipg.it, pirolandi@dmi.unipg.it

[2] *Diparitmento Di Scienze, Università "G. d'Annunzio"*
Viale Pindaro, 42 Italy
bista@sci.unich.it, santini@sci.unich.it

[3] *Istituto di Informatica e Telematica*
Via Moruzzi, 1 Italy
stefano.bistarelli@iit.cnr.it,francesco.santini@iit.cnr.it

December 2, 2009

Abstract. We suggest soft constraints as a mean to parametrically represent and solve "weighted" Argumentation problems: different kinds of preference levels related to arguments, e.g. a score representing a "fuzziness", a "cost" or a probability level of each argument, can be represented by choosing different semiring algebraic structures. The novel idea is to provide a common computational and quantitative framework where the computation of the classical Dung's extensions, e.g. the admissible extension, has an associated score representing "how much good" the set is. Preference values associated to arguments are clearly more informative and can be used to prefer a given set of arguments over others with the same characteristics (e.g. admissibility). Moreover, we propose a mapping from weighted Argumentation Frameworks to *Soft Constraint Satisfaction Problems* (*SCSPs*); with this mapping we can compute Dung semantics (e.g. admissible and stable) by solving the related SCSP. To implement this mapping we use JaCoP, a Java constraint solver.

Keywords: *Soft Constraints, Argumentation Frameworks*

Contents

1 Introduction

Argumentation [12] is based on the exchange and the evaluation of interacting arguments which may represent information of various kinds, especially beliefs or goals. Argumentation can be used for modeling some aspects of reasoning, decision making, and dialogue [10]. For instance, when an agent has conflicting beliefs (viewed as arguments), a (nontrivial) set of plausible consequences can be derived through argumentation from the most acceptable arguments for the agent. Argumentation has become an important subject of research in Artificial Intelligence and it is also of interest in several disciplines, such as Logic, Philosophy and Communication Theory [20].

Many theoretical and practical developments build on Dung's seminal theory of argumentation. A *Dung Argumentation Framework* (*AF*) is a directed graph consisting of a set of arguments and a binary conflict based *attack relation* among them. The sets of arguments to be considered are then defined under different semantics, where the choice of semantics equates with varying degrees of scepticism or credulousness.

The other ingredient in our research is Constraint Programming [21], which is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. The idea of the semiring-based formalism [7, 5] was to further extend the classical constraint notion by adding the concept of a structure representing the levels of satisfiability of the constraints. Such a structure (see Sec. 3 for further details) is a set with two operations: one $+$ is used to generate an ordering over the preference levels, while $\times$ is used to combine these levels. Because of the properties required on such operations, this structure is similar to a semiring (see Sec. 3). Problems defined according to this semiring-based framework are called *Soft Constraint Satisfaction Problems* (SCSPs).

In this paper we show that different weighted AFs based on fuzziness, probability or a preference in general (and already studied in literature, e.g. in [20, 3]), can be modeled and solved with the same soft constraint framework by only changing the related semiring in order to optimize the different criteria. Also classical AFs can be represented inside the soft framework by adopting the *Boolean* semiring. We provide a mapping from AFs to (S)CSPs in a way that the solution of the SCSP consists in the "best" desired extension, where "best" is computed by aggregating (with $\times$) the preference scores of all the chosen arguments, and comparing the final values (with $+$). The classical extensions of Dung can be found with our mapping, i.e. admissible, preferred, complete, stable and grounded ones. At last, we show an implementation of a CSP with *JaCoP* [19], a Java Constraint Programming solver.

Clearly, the classical attack relationship is not enough informative to deal with problems where we however need to take a decision: suppose a judge must decide between the arguments of two parties, and often no conclusive demonstration of the rightness of one side is possible. The arguments will not have equal value for the judge and the case will be decided by the judge preferring one argument over the other [20]. Moreover, having a quantitative framework permits us to quantify the aggregation of chosen arguments and to prefer a set of arguments over another. Examples in the real world are represented by scores given to comments in *Youtube* or news in *Slashdot*, or topics in Discussion Fora in general [16]. As the set of arguments gets wider, the search of the best solutions becomes a demanding task, and constraint-based frameworks come with many and powerful solving techniques: notice that deciding if a set is a preferred extension is a CO-NP-complete problem [4]. Moreover, preference score can be used to cut not promising solutions during the search and, however, to refine it by finding the only the best solutions. In this paper we start from qualitative argumentation [20, 3, 2] and we move towards a quantitative solution.

The remainder of this paper is organized as follows. In Sec. 2 we report the theory behind Dung Argumentation, while in Sec. 3 we summarize the background about soft constraints. Section 4 shows the basic idea of weighted AF based on semirings; in Sec. 5 we propose the mapping from AFs to SCSPs, the proofs of their solution equivalence and we show a practical encoding in JaCoP. A comparison with related work is given in Sec. 6. Finally, Sec. 7 presents our conclusions.

2 Dung Argumentation

In [12], the author has proposed an abstract framework for argumentation in which he focuses on the definition of the status of arguments. For that purpose, it can be assumed that a set of arguments is given, as well as the different conflicts among them. An argument is an abstract entity whose role is solely determined by its relations to other arguments.

Definition 1 *An Argumentation Framework (AF) is a pair $\langle \mathcal{A}_{rgs}, R \rangle$ of a set $\mathcal{A}_{rgs}$ of arguments and a binary relation R on $\mathcal{A}_{rgs}$ called the attack relation. $\forall a_i, a_j \in A$, $a_i R\, a_j$ means that a_i attacks a_j. An AF may be represented by a directed graph (the interaction graph) whose nodes are arguments and edges represent the attack relation. A set of arguments $\mathcal{B}$ attacks an argument a if a is attacked by an argument of $\mathcal{B}$. A set of arguments $\mathcal{B}$ attacks a set of arguments $\mathcal{C}$ if there is an argument $b \in \mathcal{B}$ which attacks an argument $c \in \mathcal{C}$.*

In Fig. 1 we show an example of AF represented as an *interaction graph*: the nodes represent the arguments and the directed arrow from c to d represents the attack of c towards d, that is $c\, R\, d$. Dung [12] gave several semantics to acceptability. These various semantics produce none, one or several acceptable sets of arguments,

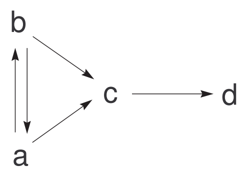

Figure 1. *An example of Dung Argumentation Framework; e.g. c attacks d.*

called extensions. One of these semantics, the stable semantics, is only defined via the notion of attacks:

Definition 2 *A set $\mathcal{B} \subseteq \mathcal{A}_{rgs}$ is conflict-free iff it does not exist two arguments a and b in $\mathcal{B}$ such that a attacks b. A conflict-free set $\mathcal{B} \subseteq \mathcal{A}_{rgs}$ is a stable extension iff for each argument which is not in $\mathcal{B}$, there exists an argument in $\mathcal{B}$ that attacks it.*

The other semantics for acceptability rely upon the concept of defense:

Definition 3 *An argument b is defended by a set $\mathcal{B} \subseteq \mathcal{A}_{rgs}$ (or $\mathcal{B}$ defends b) iff for any argument $a \in \mathcal{A}_{rgs}$, if a attacks b then $\mathcal{B}$ attacks a.*

An admissible set of arguments according to Dung must be a conflict-free set which defends all its elements. Formally:

Definition 4 *A conflict-free set $\mathcal{B} \subseteq \mathcal{A}_{rgs}$ is admissible iff each argument in $\mathcal{B}$ is defended by $\mathcal{B}$.*

Besides the stable semantics, three semantics refining admissibility have been introduced by Dung [12]:

Definition 5 *A preferred extension is a maximal (w.r.t. set inclusion) admissible subset of $\mathcal{A}_{rgs}$. An admissible $\mathcal{B} \subseteq \mathcal{A}_{rgs}$ is a complete extension iff each argument which is defended by $\mathcal{B}$ is in $\mathcal{B}$. The least (w.r.t. set inclusion) complete extension is the grounded extension.*

Notice that deciding if a set is a stable extension or an admissible set can be computed in polynomial time, but deciding if a set is a preferred extension is a CO-NP-complete problem [4].

3 Soft Constraints

A c-semiring [7, 5] S (or simply semiring in the following) is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ where A is a set with two special elements ($\mathbf{0}, \mathbf{1} \in A$) and with two operations $+$ and $\times$ that satisfy certain properties: $+$ is defined over (possibly infinite) sets of elements of A and thus is commutative, associative, idempotent, it is closed and $\mathbf{0}$ is its unit element and $\mathbf{1}$ is its absorbing element; $\times$ is closed, associative, commutative, distributes over $+$, $\mathbf{1}$ is its unit element, and $\mathbf{0}$ is its absorbing element (for the exhaustive definition, please refer to [7]). The $+$ operation defines a partial order $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$; we say that $a \leq_S b$ if b represents a value *better* than a. Other properties related to the two operations are that $+$ and $\times$ are monotone on $\leq_S$, $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum, $\langle A, \leq_S \rangle$ is a complete lattice and $+$ is its lub. Finally, if $\times$ is idempotent, then $+$ distributes over $\times$, $\langle A, \leq_S \rangle$ is a complete distributive lattice and $\times$ its glb.

A *soft constraint* [7, 5] may be seen as a constraint where each instantiation of its variables has an associated preference. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V over a finite domain D, a soft constraint is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring. Using this notation $\mathcal{C} = \eta \to A$ is the set of all possible constraints that can be built starting from S, D and V. Any function in $\mathcal{C}$ involves all the variables in V, but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : V \to D \to A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$ (the *support* of the constraint, or *scope*). Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$. Note also that $c\eta$ is the application of a constraint function $c : V \to D \to A$ to a function $\eta : V \to D$; what we obtain, is a semiring value $c\eta = a$. $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ respectively represent the constraint functions associating $\mathbf{0}$ and $\mathbf{1}$ to all assignments of domain values (i.e. the $\bar{a}$ function returns the semiring value a).

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ (see also [7, 5]). Informally, performing the $\otimes$ or between two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples.

Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* [7, 5, 6] of c over $V - \{v\}$, written $c \Downarrow_{(V \setminus \{v\})}$ is the constraint c' such that $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support.

A SCSP [5] defined as $P = \langle C, con \rangle$ (C is the set of constraints and $con \subseteq V$, i.e. a subset the problem variables). A problem P is α-consistent if $blevel(P) = \alpha$ [5]; P is instead simply "consistent" iff there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent [5].

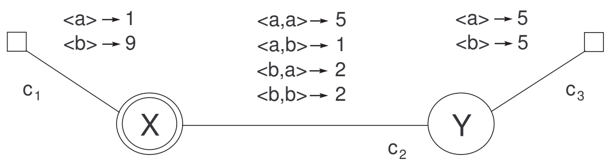

Figure 2. *A soft CSP based on a Weighted semiring.*

P is inconsistent if it is not consistent. The *best level of consistency* notion defined as $blevel(P) = Sol(P) \Downarrow_{\emptyset}$, where $Sol(P) = (\bigotimes C) \Downarrow_{con}$ [5].

A SCSP Example. Figure 2 shows a weighted CSP as a graph: the semiring used for this problem is the *Weighted* semiring, i.e. $\langle \mathbb{R}^+, \min, \hat{+}, \infty, 0\rangle$ ($\hat{+}$ is the arithmetic plus operation). Variables and constraints are represented respectively by nodes and by undirected arcs (unary for c_1 and c_3, and binary for c_2), and semiring values are written to the right of each tuple. The variables of interest (that is the set *con*) are represented with a double circle (i.e. variable X). Here we assume that the domain of the variables contains only elements a and b. For example, the solution of the weighted CSP of Fig. 2 associates a semiring element to every domain value of variable X. Such an element is obtained by first combining all the constraints together. For instance, for the tuple $\langle a, a\rangle$ (that is, $X = Y = a$), we have to compute the sum of 1 (which is the value assigned to $X = a$ in constraint c_1), 5 (which is the value assigned to $\langle X = a, Y = a\rangle$ in c_2) and 5 (which is the value for $Y = a$ in c_3). Hence, the resulting value for this tuple is 11. We can do the same work for tuple $\langle a, b\rangle \rightarrow 7$, $\langle b, a\rangle \rightarrow 16$ and $\langle b, b\rangle \rightarrow 16$. The obtained tuples are then projected over variable x, obtaining the solution $\langle a\rangle \rightarrow 7$ and $\langle b\rangle \rightarrow 16$. The *blevel* for the example in Fig. 2 is 7 (related to the solution $X = a$, $Y = b$).

4 Weighted Argumentation

To illustrate the need to extend the classical AF with preferences, we consider two individuals P and Q exchanging arguments A and B about the weather forecast (the example is taken from [20]):

P: Today will be dry in London since BBC forecast sunshine = A

Q: Today will be wet in London since CNN forecast rain = B

A and B claim contradictory conclusions and so attack each other. Under Dung's preferred semantics, there are two different admissible extensions represented by the sets $\{A\}$ and $\{B\}$, but neither argument is sceptically justified. One solution is to provide some means for preferring one argument to another in order to find a more informative answer, for example, the most trustworthy extension. For example, one might reason that A is preferred to B because the *BBC* are deemed more trustworthy than *CNN*. Suppose to have a fuzzy trust score associated with each argument, as shown in Fig. 3. This score, (between 0 and 1 that is between low and high trustworthiness) can be then used to prefer $\{A\}$ with a score of 0.9 over $\{B\}$ with a score of 0.7, i.e. forecast from *BBC* than from *CCN*.

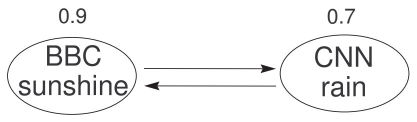

Figure 3. *The* CNN/BBC *example with trust scores.*

In some works [15] the preference score is associated with the attack relationship instead of with the argument itself and, thus, it models the "strength" of the attack, e.g. a fuzzy attack. This model can be cast in ours by composing these strengths in a value representing the preference of the argument, as in Fig. 4, where the trustworthiness of argument *CNN-rain* can be computed as the mathematical mean (or in general a function $\circ$, as defined also in [8] for computing the trust of a group of individuals) of the values associated with the attack towards it, i.e. $(0.9+0.5)\backslash 2 = 0.7$. Computing a trust evaluation of a node by considering a function of the links ending in it is a well-known solution, e.g. the *PageRank* of *Google* [16]. By composing attack and support values, it is also possible to quantitatively study bipolar argumentation frameworks [1].

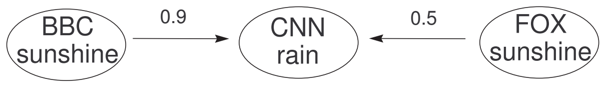

Figure 4. *A fuzzy Argumentation Framework with fuzzy scores modeling the attack strength.*

Notice that in [20, 3, 2] the preference among arguments is given in a qualitative way, that is argument a is better than argument b, which is better than argument c; in this section we study the problem from a quantitative point of view, with scores associated with arguments. We suggest the algebraic semiring structure (see Sec. 3) as a mean to parametrically represent and solve all the "weighted" AFs presented in literature (see Sec. 6), i.e. to represent the scores; in the following we provide some examples on how semirings fulfil these different tasks.

An argument can be seen as a chain of possible events that makes the hypothesis true. The credibility of a hypothesis can then be measured by the total probability that it is supported by arguments. The proper semiring to solve this problem consists in the *Probabilistic* semiring [5]: $\langle [0..1], max, \hat{\times}, 0, 1\rangle$, where the arithmetic multiplication (i.e. $\hat{\times}$) is used to compose the probability values together.

The Fuzzy Argumentation [23] approach enriches the expressive power of the classical argumentation model by allowing to represent the relative strength of the attack relationships between arguments, as well as the degree to which arguments are accepted. In this case, the *Fuzzy* semiring $\langle [0..1], min, max, 0, 1\rangle$ can be used.

In addition, the *Weighted* semiring $\langle \mathbb{R}^+, min, \hat{+}, 0, 1\rangle$, where $\hat{+}$ is the arithmetic plus, can model the (e.g. money) cost of the attack: for example, during an electoral campaign, a candidate could be interested in how many efforts or resources he should spend to counteract an argument of the opposing party.

At last, with the *Boolean* semiring $\langle \{true, false\}, \vee, \wedge, false, true\rangle$ we can cast the classic AFs originally defined by Dung [12] in the same semiring-based framework.

Moreover, notice that the cartesian product of two semirings is still a semiring [7, 5], and this can be fruitfully used to describe multi-criteria constraint satisfaction and optimization problems. For example, we can have both a probability and a fuzzy score given by a couple $\langle t, f\rangle$; we can optimize both costs at the same time.

We can extend the definitions provided in Sec. 3 in order to express all these weights of the attack relations with a semiring based environment. The following definitions model the semiring-based problem.

Definition 6 *A semiring-based Argumentation Framework (AF_S) is a quadruple $\langle \mathcal{A}_{rgs}, R, W, S\rangle$ of a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, a set $\mathcal{A}_{rgs}$ of arguments, the attack binary relation R on $\mathcal{A}_{rgs}$, and a unary function $R : \mathcal{A}_{rgs} \longrightarrow A$ called the* weight *function. $\forall a \in \mathcal{A}_{rgs}$, $W(a) = s$ means that a has a preference level $s \in A$.*

Therefore, the weight function W associates each argument with a semiring value ($s \in A$) that represents the preference expressed for that argument in terms of cost, fuzziness and so on. For example, using the *Fuzzy* semiring $\langle [0..1], min, max, 0, 1\rangle$ semiring for the problem represented in Fig. 3 allows us to state that the admissible extension $\{A\}$ (with a score of 0.9) is better than the other admissible extension $\{B\}$ (with a s.core of 0.7) since $0.9 > 0.7$. Therefore, with an AF_S our goal is to find the extensions proposed by Dung (e.g. the admissible extensions), but with an associated preference value.

Example *Concerning the interaction graph in Fig. 5, it represents the Weighted AF_S $W = (\mathcal{A}_{rgs}, R)$ with $S = \langle \mathbb{R}^+, \min, \hat{+}, \infty, 0\rangle$ and such that $\mathcal{A}_{rgs} = \{a, b, c, d, e\}$, $R(a, b) = 0.7, R(c, b) = 0.8, R(c, d) = 0.9, R(d, c) = 0.8, R(d, e) = 0.5, R(e, e) = 0.6$ and $W(a) = 7, W(b) = 20, W(c) = 6, W(d) = 10, W(e) = 12$. Notice that e attacks*

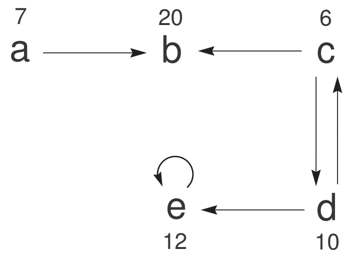

Figure 5. *An example of a weighted interaction graph.*

itself, that is in contrast with itself, e.g. "We have sunshine and it's raining" (it may be possible).

5 Mapping AFs to SCSPs

Our second result is a mapping from AF (and AF_S) to (S)CSPs. Given an $AF_S = \langle \mathcal{A}_{rgs}, R, W, S \rangle$, we define a variable for each argument $a_i \in \mathcal{A}_{rgs}$, i.e. $V = \{a_1, a_2, \ldots, a_n\}$ and each of these argument can be taken or not, i.e. the domain of each variable is $D = \{1, 0\}$, and if it is taken, a cost in the semiring can be assigned, mapping the level of preference of this argument.

To represent the quantitative preference over arguments, in this mapping we need only unary soft constraints on each variable, while the other constraints modeling, for example, the conflict-free relationship (see Sec. 2) are crisp even if represented in the soft framework. We plan to extend also these constraints to properly-said soft ones as suggested in Sec. 7. In the following explanation, notice that b attacks a meas that b is a father of a in the interaction graph, and c attacks b attacks a means that c is a grandfather of a. To compute the (weighted) extensions of Dung we need to define specific sets of constraints:

1. **Preference constraints.** The weight function $W(a_i) = s$ ($s \in A$) of an AF_S can be modeled with the unary constraints $c_{a_i}(a_i = 1) = s$, otherwise, when a_i is assigned to 0), the argument is not taken in the considered extension an so its cost must not be computed.

2. **Conflict-free constraints.** Since we want to find the conflict-free sets, if $R(a_i, a_j)$ is in the graph we need to prevent the solution to include both a_i and a_j in the considered extension: $c_{a_i,a_j}(a_i = 1, a_j = 1) = \mathbf{0}$. For the other possible assignment of the variables $((a = 0, b = 1)(a = 1, b = 0)$ and $(a = 0, b = 0))$, $c_{a_i,a_j} = \mathbf{1}$, since these assignments are permitted: in these

cases we are choosing only one argument between the two (or none of the two) and thus, we have no conflict.

3. **Admissible constraints.** For the admissibility, we need that, if son argument a_i has a father node a_f but a_i has no grandfather node a_g, then we must avoid to take a_f in the extension because it is attacked and cannot be defended by any ancestor: expressed with a binary constraint, $c_{a_i,a_f}(a_i = 1, a_f = 0) = \mathbf{0}$.

   Moreover, if a_i has several grandfathers $a_{g1}, a_{g2}, \ldots, a_{gk}$ and only one fathers a_f, we need to add a $k+1$-ary constraint $c_{a_i,a_{g1},\ldots,a_{gk}}(a_i = 1, a_{g1} = 0, \ldots, a_{gk} = 0) = \mathbf{0}$. The explanation is that at least a a grandfather must be taken in the admissible set, in order to defend a_i from one of his fathers a_f. Notice that, if a node is not attacked (i.e. he has no fathers), he can be taken or not in the admissible set.

4. **Complete constraints.** To compute a complete extension $\mathcal{B}$, we need that each argument a_i which is defended by $\mathcal{B}$ is in $\mathcal{B}$ (see Sec. 2). This is already enforced with the admissible constraints, except for the nodes without a father. For these a_i nodes we need to add the constraint $c_{a_i}(a_i = 0) = \mathbf{0}$.

5. **Stable constraints.** If we have a son node a_i with multiple fathers $a_{f1}, a_{f2}, \ldots, a_{fk}$, we need to add the constraint $c_{a_i,a_{f1},\ldots,a_{fk}}(a_i = 0, a_{f1} = 0, \ldots, a_{fk} = 0) = \mathbf{0}$. In words, if a node is not taken in the extension (i.e. $a_i = 0$), then it must be attacked by at least one of the taken nodes, that is at least a father of a_i needs to be taken in the stable extension (that is, $a_{fj} = 1$).

   Moreover, if a node a_i has no father in the graph, it has to be included in the stable extension (notice a_i cannot be attacked by nodes inside the extension, since he has no father). The corresponding unary constraint is $c_{a_i}(a_i = 0) = \mathbf{0}$.

Notice that by using the *Boolean* semiring, also the class of preference constraints becomes crisp and we can consequently model classical Dung AFs, that is not weighted frameworks. The following proposition states the equivalence between solving an AF_S and its related SCSP.

Proposition 1 (Solution equivalence) *Given an $AF_S = \langle \mathcal{A}_{rgs}, R, W, S \rangle$ and $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, the solutions of the related SCSP obtained with the mapping corresponds to find over AF_S the best(according to $+$)*

- *admissible extensions by using preference and conflict constraint classes.*
- *complete extensions by using preference, conflict and admissible constraint classes.*
- *stable extensions by using preference, conflict and stable constraint classes.*

By using the Boolean *semiring the solutions of the (S)CSP respectively correspond to all the classical admissible, complete and stable extensions of Dung [12].*

Moreover, to find the preferred extension (see Sec. 2) we simply need to find all the maximal (w.r.t. set inclusion) admissible extensions of $\mathcal{A}_{rgs}$, that is to find all the admissible sets (using the first three classes of constraints) and then returning only those subsets with the highest number of variables assigned to 1. Similar considerations hold for the grounded extension (see Sec. 2), that is we need to find all the complete extensions (the first four classes of constraints) and then to return only those subsets with the lowest number of variables assigned to 1 [1].

As suggested in Sec. 4, an AF_S can be represented as a weighted interaction graph as in Fig. 5, where we instead suppose to use a *Weighted* semiring, i.e. $\langle \mathbb{R}^+, \min, \hat{+}, \infty, 0 \rangle$, e.g. the argument a has received 7 negative comments. The goal in this case is to choose the extensions of Dung and to minimize the sum of the negative comments at the same time.

Notice that the presented soft constraint framework can be easily used to solve argumentation problems with additional constraints, as proposed in [11] only for boolean constraints. We can find further requirements on the sets of arguments which are expected as extensions, like "extensions must contain argument a when they contain b" or "extensions must not contain one of c or d when they contain a but do not contain b".

The Java Constraint Programming solver [19], JaCoP in short, is a Java library, which provides Java user with Finite Domain Constraint Programming paradigm. It provides different type of constraints: most commonly used primitive constraints, such as arithmetical constraints, equalities and inequalities, logical, reified and conditional constraints, combinatorial (global) constraints. The last version of JaCoP proposes many features, such as pruning events, multiple constraint queues, special data structures to handle efficiently backtracking, iterative constraint processing, and many more [19]. Moreover, it can run also large examples, e.g. ca. 180000 constraints.

In Fig. 6 we show the definition in JaCoP of all the constraints used to solve the AF_S example in Fig. 5. The full description of the code can be found in Appendix A. Considering for example the first conflict-free constraint in Fig. 5, $v[0], v[1]$, means that the constraint is between a and b and $(1,1)$ that the the constraint is not satisfied if both variables are taken in the set.

Considering the example in Fig. 5 the admissible sets are: $\{a\}, \{c\}, \{d\}, \{a, c\}, \{a, d\}$. Dung's semantics induce the following acceptable sets: stable one extension $\{a, d\}$, two preferred extensions $PE_1 = \{a, c\}, PE_2 = \{a, d\}$, three complete extensions $CE_1 = \{a, c\}, CE_2 = \{a, d\}, CE_3 = \{a\}$ and grounded extension $\equiv \{a\}$.

[1]Different interpretations of grounded/preferred extensions can be given by considering their cost instead of their the cardinality.

```
    // Defining the Variables of the SCSP
v[0] = new BooleanVariable(store, "a");
v[1] = new BooleanVariable(store, "b");
v[2] = new BooleanVariable(store, "c");
v[3] = new BooleanVariable(store, "d");
v[4] = new BooleanVariable(store, "e");

    // conflict-free constraints
public static void imposeConstraintConflictFree(Store store, BooleanVariable[] v) {
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]}, new int[][]{{1, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[1]}, new int[][]{{1, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]}, new int[][]{{1, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[2]}, new int[][]{{1, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]}, new int[][]{{1, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[4], v[4]}, new int[][]{{1, 1}}));  }

    // admissible constraints
public static void imposeConstraintAdmissibleSet(Store store, BooleanVariable[] v) {
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]}, new int[][]{{0, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[1]}, new int[][]{{0, 1}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[4]}, new int[][]{{0, 1}}));  }

   // stable constraints
public static void imposeConstraintStableExtensions(Store store, BooleanVariable[] v) {
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]}, new int[][]{{0}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[2], v[1]}, new int[][]{{0, 0, 0}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]}, new int[][]{{0, 0}}));
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]}, new int[][]{{0, 0}}));  }

    // complete constraints
public static void imposeConstraintCompleteExtensions(Store store, BooleanVariable[] v) {
store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]}, new int[][]{{0}}));     }
```

Figure 6. *The constraint in JaCoP for the mapping of Fig. 5.*

With our quantitative interpretation of AFs with preferences and considering the *Fuzzy* semiring $\langle \mathbb{R}^+, \min, \hat{+}, \infty, 0 \rangle$, we can prefer PE_1 over PE_2 $(W(a)\hat{+}W(c)) = 13$, $W(a)\hat{+}W(d) = 17$ and CE_3 over CE_1 and CE_2, since $W(a) = 7$. All these best solutions are obtained by using JaCoP.

6 Related Work

In [23], the authors have developed the notion of fuzzy unification and incorporated it into a novel fuzzy argumentation framework for extended logic programming: the attacks are associated to a fuzzy strength value, i.e. a V-attack. As well, a V-argument A is V-acceptable w.r.t. the set *Args* of V-arguments if each argument V-attacked A is V-attacked by an argument in *Args*.

In [3], AFs have been also extended to *Value Based Argumentation Frameworks* (*VAF*) where V is a generic nonempty set of values and *Val* is a function which maps from elements of *Args* to elements of V.

The work in [2] concerns the acceptability of arguments in preference-based argumentation frameworks. Preferences are represented with a preordering relationships (partial or total) that resembles the ordering defined by the + operator of semirings (see Sec. 3).

Probabilistic Argumentation [14, 18]. This theory is an alternative approach for non-monotonic reasoning under uncertainty. It allows to judge open questions (hypotheses) about the unknown or future world in the light of the given knowledge. From a qualitative point of view, the problem is to derive arguments in favor and against the hypothesis of interest.

In [20] the author has extended Dung's theory of argumentation to integrate metalevel argumentation about preferences. Dung's level of abstraction is preserved, so that arguments expressing preferences are distinguished by being the source of a second attack relation that abstractly characterizes application of preferences by attacking attacks between the arguments that are the subject of the preference claims.

Comparison. The framework proposed in this paper is able to solve all the above reported AFs (including the classical Dung framework [12]), both from the qualitative and (main novelty) quantitative point of view. Since in this paper we mainly propose a solving framework, we compare it with other related works.

In [17] crisp constraint have been used to model argumentation as constraint propagation in *Distributed Constraint Satisfaction Problem* (*DSCP*). Different agents represent the distributed points in the problem. The paper shows the appropriateness of constraints in solving large-scale argumentation systems. However, it seems to only solve classical problems, (i.e. no qualitative or quantitative extensions).

The are some frameworks based on Logic Programming-like languages. For example, the system *ASPARTIX* [13] is a tool for computing acceptable extensions for a broad range of formalizations of Dung's argumentation framework and generalizations thereof, e.g. value-based AFs [3] or preference-based [2]. *ASPARTIX* relies on a fixed disjunctive datalog program which takes an instance of an argumentation framework as input, and uses the answer-set solver DLV for computing the type of extension specified by the user. However, *ASPARTIX* does not solve any quantitative argumentation case, as well as other Answer Set Programming systems [22].

Other papers [9] work directly with algorithms over the interaction graph, and thus they do not provide a general expressive system as constraints formulation or logic programming do instead.

7 Conclusions and Future Work

In the paper we have revised the notions provided by Dung [12] in order to associate the argument preference with a weight (taken from a semiring structure) that represents the "goodness" of the argument in terms of cost, fuzziness, probability or else. Further on, we have suggested the Dung's semantics in their soft version. Moreover, we have presented a mapping from SCSPs to AFs and solved the obtained SCSP with JaCoP, a Java Constraint Programming solver, thus finding the solution of the related AF. We have proposed an unifying computational framework with strong mathematical foundations and solving techniques, where by only parametrically changing the semiring we can deal with different weighted (or not) AFs.

In the future, we would like to cluster arguments according to their (for example) coherence, still using soft constraints as the framework to obtain the solution. This can be useful to check the discrepancies/likeness during a negotiation process, inside different interviews to the same political candidate or during discussions in general. As an example, *"We do not want immigrants with the right to vote"* is clearly closer to *"Immigration must be stopped"*, than to *"We need a multicultural and open society in order to enrich the life of everyone and boost our economy"*, and should belong to the same cluster.

Moreover, we would like to solve over-constrained weighted AF problems, where weights are associated with arcs and represent the cost of the attack between two arguments. We want to relax the notion of admissibility to α-admissibility (and also the other notions and semantics of Dung), in order to include in the same admissible set also attacking arguments, whose attack costs are not worse than a threshold α; this in case classical admissible sets cannot be found in the given AF.

Acknowledgements

We would like to thank Massimiliano Giacomin for the important suggestions.

References

[1] L. Amgoud, C. C., M.-C. Lagasquie-Schiex, and P. Livet. On bipolarity in argumentation frameworks. *Int. J. Intell. Syst.*, 23(10):1062–1093, 2008.

[2] L. Amgoud and C. Cayrol. Inferring from inconsistency in preference-based argumentation frameworks. *J. Autom. Reasoning*, 29(2):125–169, 2002.

[3] T. J. M. Bench-Capon. Persuasion in practical argument using value-based argumentation frameworks. *J. Log. Comput.*, 13(3):429–448, 2003.

[4] P. Besnard and S. Doutre. Checking the acceptability of a set of arguments. In *Workshop on Non-Monotonic Reasoning*, pages 59–64, 2004.

[5] S. Bistarelli. *Semirings for Soft Constraint Solving and Programming*, volume 2962 of *LNCS*. Springer, 2004.

[6] S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.

[7] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based Constraint Solving and Optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[8] S. Bistarelli and F. Santini. Propagating multitrust within trust networks. In *ACM Symposium on Applied Computing*, pages 1990–1994. ACM, 2008.

[9] C. Cayrol, S. Doutre, and J. Mengin. On decision problems related to the preferred semantics for argumentation frameworks. *J. Log. Comput.*, 13(3):377–403, 2003.

[10] S. Coste-Marquis, C. Devred, S. Konieczny, M. Lagasquie-Schiex, and P. Marquis. On the merging of dung's argumentation systems. *Artif. Intell.*, 171(10-15):730–753, 2007.

[11] S. Coste-Marquis, C. Devred, and P. Marquis. Constrained argumentation frameworks. In *Knowledge Representation and Reasoning (KR)*, pages 112–122. AAAI Press, 2006.

[12] P. M. Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–357, 1995.

[13] U. Egly, S. Alice Gaggl, and S. Woltran. ASPARTIX: Implementing argumentation frameworks using answer-set programming. In *International Conference on Logic Programming (ICLP)*, pages 734–738. LNCS, Springer, 2008.

[14] R. Haenni. Probabilistic argumentation. *J. Applied Logic*, 7(2):155–176, 2009.

[15] J. Janssen, M. De Cock, and D. Vermeir. Fuzzy argumentation frameworks. In *Information Processing and Management of Uncertainty in Knowledge-based Systems*, pages 513–520, 2008.

[16] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decis. Support Syst.*, 43(2):618–644, 2007.

[17] H. Jung, M. Tambe, and S. Kulkarni. Argumentation as distributed constraint satisfaction: applications and results. In *Conference on Autonomous agents (AGENTS)*, pages 324–331, New York, NY, USA, 2001. ACM.

[18] Jürg Kohlas. Probabilistic argumentation systems a new way to combine logic with probability. *J. of Applied Logic*, 1(3-4):225–253, 2003.

[19] K. Kuchcinski and R. Szymanek. Jacop - java constraint programming solver, 2001. http://jacop.osolpro.com/.

[20] S. Modgil. Reasoning about preferences in argumentation frameworks. *Artif. Intell.*, 173(9-10):901–934, 2009.

[21] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[22] J. C. Nieves, U. Cortés, and M. Osorio. Possibilistic-based argumentation: An answer set programming approach. In *Mexican International Conference on Computer Science(ENC)*, pages 249–260. IEEE Computer Society, 2008.

[23] M. Schroeder and R. Schweimeier. Fuzzy argumentation for negotiating agents. In *AAMAS*, pages 942–943. ACM, 2002.

Appendix A

The appendix shows all the JaCoP [19] code written to solve the AF_S proposed in Fig. 5.

```
package ExamplesJaCoP;

import JaCoP.constraints.ExtensionalConflictVA;
import JaCoP.core.*;
import JaCoP.search.*;
import java.util.ArrayList;
import java.util.Vector;

public class Argumentation {

    static Argumentation m = new Argumentation();
    static int size = 5;                          // number of variables
    static int[] pesi = {7, 20, 6, 10, 12};     // weights associated with arguments
    static String[] etichette = {"Conflict free", "Admissible sets", "Stable extensions",
            "Complete extensions", "Preferred Extensions", "Ground extensions"};
    static int insieme = 2;
    static Store store;             // store
    static BooleanVariable[] v;     // array of variables

    public static void main(String[] args) {
        // defining the store
        store = new Store();

        // defining the array of variables
        v = new BooleanVariable[size];

        // defining the single variable inside the store
        v[0] = new BooleanVariable(store, "a");
        v[1] = new BooleanVariable(store, "b");
        v[2] = new BooleanVariable(store, "c");
        v[3] = new BooleanVariable(store, "d");
    v[4] = new BooleanVariable(store, "e");

    /**
     * 0 = conflict free
     * 1 = admissible set
     * 2 = stable extensions
     * 3 = complete extensions
     * 4 = preferred extensions
     * 5 = ground extensions
     */
    switch (insieme) {

        case 0: // conflict free
            imposeConstraintConflictFree(store, v);
            break;

        case 1: // admissible set
            imposeConstraintConflictFree(store, v);
            imposeConstraintAdmissibleSet(store, v);
            break;

        case 2: // stable extensions
            imposeConstraintConflictFree(store, v);
            imposeConstraintStableExtensions(store, v);
            break;

        case 3: // complete extensions
            imposeConstraintConflictFree(store, v);
            imposeConstraintAdmissibleSet(store, v);
            imposeConstraintCompleteExtensions(store, v);
            break;

        case 4: // preferred extensions: admissible + largest set
            imposeConstraintConflictFree(store, v);
            imposeConstraintAdmissibleSet(store, v);
            break;

        case 5: // ground extensions: complete + smallest set
            imposeConstraintConflictFree(store, v);
            imposeConstraintAdmissibleSet(store, v);
            imposeConstraintCompleteExtensions(store, v);
            break;
    }

    /*
     * returning the solutions
     */
    getSolutions(store, v, insieme);
    System.out.println("");

}

//  conflict-free constraints
public static void imposeConstraintConflictFree(Store store, BooleanVariable[] v) {
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]},
            new int[][]{{1, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[1]},
            new int[][]{{1, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]},
            new int[][]{{1, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[2]},
            new int[][]{{1, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]},
            new int[][]{{1, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[4], v[4]},
            new int[][]{{1, 1}}));
}

// admissible set constraints
public static void imposeConstraintAdmissibleSet(Store store, BooleanVariable[] v) {
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[1]},
            new int[][]{{0, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[1]},
            new int[][]{{0, 1}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[4]},
            new int[][]{{0, 1}}));
}

// stable constraints
public static void imposeConstraintStableExtensions(Store store, BooleanVariable[] v) {
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]},
            new int[][]{{0}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0], v[2], v[1]},
            new int[][]{{0, 0, 0}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[2], v[3]},
            new int[][]{{0, 0}}));
    // the constraint below is redundant w.r.t. the one just above
    //store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[2]},
            new int[][]{{0, 0}}));
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[3], v[4]},
            new int[][]{{0, 0}}));
}

// complete constraints
public static void imposeConstraintCompleteExtensions(Store store, BooleanVariable[] v) {
    store.impose(new ExtensionalConflictVA(new BooleanVariable[]{v[0]},
        new int[][]{{0}}));
}

public static void getSolutions(Store store, BooleanVariable[] v, int insieme) {

    // search for a solution and print results
    Search label = new DepthFirstSearch();
    // ordering the solutions
    SelectChoicePoint select = new InputOrderSelect(store, v, new IndomainMax());
    label.getSolutionListener().searchAll(true);
    // record solutions; if not set false
    label.getSolutionListener().recordSolutions(true);
    boolean result = label.labeling(store, select);
    int[][] soluzioni = label.getSolutionListener().getSolutions();


    if (insieme == 0 || insieme == 1 || insieme == 2 || insieme == 3) {
        // printing the solutions
        System.out.print(etichette[insieme] + ": ");
        for (int i = 0; i < label.getSolutionListener().solutionsNo(); i++) {
            System.out.print("(");
            for (int j = 0; j < size; j++) {
                if (soluzioni[i][j] == 1) {
                    System.out.print(v[j].id);
                }
            }
            System.out.print(")");
        }

        // obtaining the best solutions
        Vector<Integer> miglioriSoluzioni = getBestSolutions(soluzioni,
                label.getSolutionListener().solutionsNo());
        // printing the best solutions
        printBestSolutions(miglioriSoluzioni, soluzioni);


    } // preferred extensions
    else if (insieme == 4) {
        // computing the preferred extensions, and then printing
        ArrayList<Integer> indice = new ArrayList<Integer>();
        int max = 0;
        int temp = 0;
        for (int i = 0; i < label.getSolutionListener().solutionsNo(); i++) {
            temp = 0;
            for (int j = 0; j < size; j++) {
                if (soluzioni[i][j] == 1) {
                    temp++;
                }
            }
            if (temp == max) {
                indice.add(i);
            } else if (temp > max) {
                indice.clear();
                indice.add(i);
                max = temp;
            }
        }

        // printing the solutions
        System.out.print(etichette[insieme] + ": ");
        for (int i = 0; i < indice.size(); i++) {

            System.out.print("(");
            for (int j = 0; j < size; j++) {
                if (soluzioni[indice.get(i)][j] == 1) {
                    System.out.print(v[j].id);
                }
            }
            System.out.print(")");
        }


        int[][] soluzioniBuone = new int[indice.size()][size];
        for (int i = 0; i < indice.size(); i++) {
            soluzioniBuone[i] = soluzioni[indice.get(i)];
        }

        // obtaining the best solutions
        Vector<Integer> miglioriSoluzioni = getBestSolutions(soluzioniBuone,
                indice.size());
        // printing the best solutions
        printBestSolutions(miglioriSoluzioni, soluzioni);


    } // ground extensions
    else if (insieme == 5) {
        // computing ground extensions
        ArrayList<Integer> indice = new ArrayList<Integer>();
        int min = Integer.MAX_VALUE;
        int temp = 0;
        for (int i = 0; i < label.getSolutionListener().solutionsNo(); i++) {
            temp = 0;
            for (int j = 0; j < size; j++) {
                if (soluzioni[i][j] == 1) {
                    temp++;
                }
            }
            if (temp == min) {
                indice.add(i);
            } else if (temp < min) {
                indice.clear();
                indice.add(i);
                min = temp;
                //indice = i;
            }
        }

        // printing the solutions
        System.out.print(etichette[insieme] + ": ");
        for (int i = 0; i < indice.size(); i++) {

            System.out.print("(");
            for (int j = 0; j < size; j++) {
                if (soluzioni[indice.get(i)][j] == 1) {
                    System.out.print(v[j].id);
                }
            }
            System.out.print(")");
        }

        int[][] soluzioniBuone = new int[indice.size()][size];
        for (int i = 0; i < indice.size(); i++) {
            soluzioniBuone[i] = soluzioni[indice.get(i)];
        }

        // obtaining the best solutions
        Vector<Integer> miglioriSoluzioni = getBestSolutions(soluzioniBuone,
                indice.size());
        // printing the best solutions
        printBestSolutions(miglioriSoluzioni, soluzioniBuone);
    }
}


// array as in input and returns the indexes of the best elements
// computing solutions with the best (i.e. lowest) cost
```

```
    public static Vector<Integer> getBestSolutions(int[][] soluzioni, int numerosoluzioni) {
        Vector<Integer> indiciSoluzioni = new Vector<Integer>();
        Integer[] soluzioneConPeso = new Integer[2];

        int min = Integer.MAX_VALUE;
        int pesoSoluzione = 0;

        for (int j = 0; j < numerosoluzioni; j++) {
            pesoSoluzione = 0;
            soluzioneConPeso[0] = 0;
            soluzioneConPeso[1] = 0;

            for (int c = 0; c < size; c++) {
                if (soluzioni[j][c] == 1) {
                    pesoSoluzione = pesoSoluzione + pesi[j];
                }
            }

            if (pesoSoluzione < min) {
                indiciSoluzioni.removeAllElements();
                min = pesoSoluzione;
                indiciSoluzioni.add(j);
                //System.out.println("index of the added solution " + j);
            } else if (pesoSoluzione == min) {
                indiciSoluzioni.add(j);
            }

        }

        //System.out.println("solution index: " + indiciSoluzioni.get(0));
        return indiciSoluzioni;
    }

    public static int getWeigthSolution(int[] soluzione) {
        int pesoSoluzione = 0;
        for (int i = 0; i < size; i++) {
            if (soluzione[i] == 1) {
                pesoSoluzione = pesoSoluzione + pesi[i];
            }
        }
        return pesoSoluzione;
    }

    public static void printBestSolutions(Vector<Integer> miglioriSoluzioni, int[][] soluzioni) {
        System.out.println("");
        System.out.print("Bests " + etichette[insieme] + ": ");
        for (int i = 0; i < miglioriSoluzioni.size(); i++) {
            System.out.print("(");
            for (int j = 0; j < size; j++) {
                if (soluzioni[miglioriSoluzioni.get(i)][j] == 1) {
                    System.out.print(v[j].id);
                }
            }
            System.out.print(") = " + getWeigthSolution(soluzioni[miglioriSoluzioni.get(i)]));
        }
    }
}
```