About arc-consistency in semiring-based constraint problems

Stefano Bistarelli and Francesca Rossi
Dipartimento di Informatica, Corso Italia 40, 56125 Pisa, Italy
E-mail: {bista,rossi}@di.unipi.it

1 Introduction

Semiring-based constraint problems (SCSPs) [1, 2] extend classical constraint problems (CSPs) by allowing preferences, costs, priorities, probabilities, and other non-crisp features. They are based over the notion of a semiring, that is, a set plus two operations over the elements of such a set: $+$ and $\times$. Each combination of values for some variables connected by a constraint has a semiring value associated with it, which is to be interpreted as its level of preference. Then, the two operations are used, respectively, to project out some variables and to combine constraints.

It has been shown [1, 2] that classical local consistency techniques, which are very successful in CSPs in discovering inconsistencies and speeding up the search for a solution [9, 8, 11, 10], can be extended to the general SCSP framework if the $\times$ operation is idempotent. Here we study the properties and possible advantages of the extension to SCSPs of the most successful of such techniques, called *arc-consistency* [16, 9], although we claim that all our results can be extended to higher levels of consistency.

In particular, we study the relationship between classical arc-consistency (AC) and semiring-based arc-consistency (SAC) over SCSPs, showing that they discover the same inconsistencies. Thus sometimes it may be reasonable to apply just classical AC instead of SAC because AC is more efficient. Moreover, we show that, if an SCSP is SAC, then 1) some hidden variables can be removed without changing its set of optimal solutions and associated semiring values, 2) if its constraint graph is tree-shaped, an optimal solution can be found without backtracking, and 3) non-tree-shaped SCSPs can be solved by performing backtracking only on a subset of the variables (those involved in a cycle-cutset). These last three results are extensions of similar results already existing for the CSP case, which however here are particularly interesting both because of the generality of the SCSP framework, and also because SCSPs usually express constraint optimization, and not just satisfaction, problems.

The paper is organized as follows. Section 2 presents the basic notions about semiring-based CSPs and arc-consistency in such problems. Then, Section 3 compares the power of classical arc-consistency with that of semiring-based arc-consistency. Then, Section 4 studies the possibility of removing hidden variables in an arc-consistent SCSP, Section 5 shows that tree-shaped SCSPs can be solved easily, and Section 6 extends this result to SCSPs with cycles with bounded size. Finally, Section 7 concludes the paper summarizing its results and hinting at promising directions for future work.

This paper is in the form of an extended abstract, which means that most of its content (definitions, proofs of results, ...) is only informally described. The point of this paper is just to give the main ideas and notions underlying some possible uses of arc-consistency in SCSPs, and to describe some results which may be used in the future to improve existing algorithms for solving constraint problems.

2 Semiring-based CSPs

Here we give the basic notions about constraint solving over semirings, introduced in [1, 2].

A *semiring* is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$, called the additive operation, is a closed (i.e., $a, b \in A$ implies $a + b \in A$), commutative (i.e., $a + b = b + a$) and associative (i.e., $a + (b + c) = (a + b) + c$) operation such that $a + \mathbf{0} = a = \mathbf{0} + a$ (i.e., $\mathbf{0}$ is its unit element); $\times$, called the multiplicative operation, is a closed and associative operation such that $\mathbf{1}$ is its unit element and $a \times \mathbf{0} = \mathbf{0} = \mathbf{0} \times a$ (i.e., $\mathbf{0}$ is its absorbing element); $\times$ distributes over $+$ (i.e., $a \times (b + c) = (a \times b) + (a \times c)$).

Semirings are standard and well-know structures. However, for semiring-based SCSPs we need some more properties, leading to structures we call *c-semirings*. A c-semiring is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that $+$ is idempotent (i.e., $a \in A$ implies $a + a = a$); $\times$ is commutative; and $\mathbf{1}$ is the absorbing element of $+$. Intuitively, the idempotence of $+$ is needed to get a partial order over the elements of the semiring (otherwise we would not have the reflexive property), the commutativity of $\times$ allows us to consider sets of constraints (instead of ordered tuples), and the fact that $\mathbf{1}$ is the absorbing element of $+$ makes the element $\mathbf{1}$ the maximum element of the partial order, which is useful to have an upper bound to all the preference levels. We recall, in fact, that the elements of the chosen semiring can be interpreted in many ways: cost, level of preference, certainty, probability, etc.

The following is a list of useful properties about c-semirings which will be used in this paper. Their proof can be found in [2]. First, given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$ is a partial order. The intuitive meaning of this partial order is to state when an element is better than another one: if $a \leq_S b$ then we mean that b is better than

a. Also, since $\mathbf{0}$ is the unit element of the additive operation, it is the minimum element of the ordering. Thus, for any $a \in A$, we have $\mathbf{0} \leq_S a$. Moreover, both the additive and the multiplicative operation are monotone on the ordering $\leq_S$. Since $\mathbf{1}$ is also the absorbing element of the additive operation, then $a \leq_S \mathbf{1}$ for all a. Thus $\mathbf{1}$ is the maximum element of the partial ordering. This implies that the $\times$ operation is *intensive*, that is, that $a \times b \leq_S a$. This is important since it means that combining more constraints leads to a worse (w.r.t. the $\leq_S$ ordering) result.

Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and its partial order $\leq_S$, $\langle A, \leq_S \rangle$ is a complete lattice. Moreover, for any $a, b \in A$, we have $a \vee b = a + b$, where $\vee$ is the lub operation of the lattice. Furthermore, given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and the corresponding lattice $\langle A, \leq_S \rangle$, if $\times$ is idempotent, then we have that: $+$ distributes over $\times$; $\times = \wedge$, where $\wedge$ is the glb operation of the lattice; and $\langle A, \leq_S \rangle$ is a distributive lattice.

In the following we will assume to always refer to a certain semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, to a domain D and to a set of variables V.

A constraint is a pair $\langle def, con \rangle$, where $con \subseteq V$, and it is called the *type* of the constraint; and $def : D^k \rightarrow A$ (where k is the cardinality of con) and it is called the *value* of the constraint. In other words, each constraints, which connects a certain set of variables, is defined by associating an element of the semiring with each tuple of values of the domain for the involved variables.

A constraint problem (SCSP) is then just a set of constraints over a given constraint system, plus a selected set of variables (thus a *type*): $P = \langle C, con \rangle$. These are the variables of interest in the problem, i.e., the variables of which we want to know the possible assignments compatibly with all the constraints.

It is interesting to notice that classical CSPs are just SCSPs where the semiring has just two values: *true* and *false*, and the two operations are *logical and* and *logical or*. That is, the semiring is $S_{CSP} = \langle \{true, false\}, \vee, \wedge, false, true \rangle$. Since in this case we have just two semiring values, usually for each constraint we just specify the tuples which have value *true*, saying that these are the tuples which satisfy the constraint. Similarly, for each unary constraint, we just specify the values for that variables which have value *true*, saying that they constitute the domain of that variable.

In SCSPs, the values specified for the tuples of each constraint are used to compute corresponding values for the tuples of values of the variables in *con*, according to the semiring operations: the multiplicative operation is used to combine the values of the tuples of each constraint to get the value of a tuple for all the variables, and the additive operation is used to obtain the value of the tuples of the variables in the type of the problem. More precisely, we can define the operations of *combination* ($\otimes$) and *projection* ($\Downarrow$) over constraints.

Given two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$, their combination, written $c_1 \otimes c_2$, is the constraint $c = \langle def, con \rangle$ with $con = con_1 \cup con_2$ and $def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2})$. This operation can be easily extended to more than two arguments, say $C = \{c_1, \ldots, c_n\}$, by performing

$c_1 \otimes c_2 \otimes \ldots \otimes c_n$, which we will usually denote by $(\bigotimes C)$.

Given a constraint $c = \langle def, con \rangle$, and a set I of variables ($I \subseteq V$), the projection of c over I, written $c \Downarrow_I$, is the constraint $\langle def', con' \rangle$ over CS with $con' = I \cap con$ and $def'(t') = \Sigma_{\{t \mid t \downarrow^{con}_{I \cap con} = t'\}} def(t)$.

Using the operations of combination and projection, we can now define the notion of solution of an SCSP: given a constraint problem $P = \langle C, con \rangle$, the solution of P is a constraint defined as $Sol(P) = (\bigotimes C) \Downarrow_{con}$. In words, the solution of an SCSP is the constraint induced on the variables in con by the whole problem. Such constraint provides, for each tuple of values of D for the variables in con, a corresponding value of A.

Computing the best solution for an SCSP is an NP-hard problem. Thus it can be convenient in many cases to approximate such notion. In classical CSPs, this is done using the so-called local consistency techniques [10, 15]. The main idea is to choose some subproblems in which to eliminate local inconsistency, and then iterate such elimination in all the chosen subproblems until stability. The most widely know local consistency algorithms are arc-consistency (see [9] for its original formulation), where subproblems contain just one constraint, and path-consistency [11], where subproblems are triangles (that is, they are complete binary graphs over three variables). The concept of k-consistency [6], where subproblems contain all constraints among subsets of k chosen variables, generalizes them both: path-consistency is just 3-consistency and arc-consistency is just 2-consistency.

Since arc-consistency is in practice the most successful of all these techniques, and also the easiest to describe, in this paper we will focus on this notion. We claim however that all our results can be straightforwardly extended to any level of consistency.

Consider an SCSP problem $P = \langle C, con \rangle$ and assume that for each variable, say x, we have just one unary constraint connecting it, say c_x. Assume also, that for each pair of variables, say x and y, there is just one constraint connecting them, say c_{xy}. Then P is semiring-based arc-consistent (SAC) if, for any two variables x and y in V, we have the following:

$$c_x = (\otimes \{c_y, c_{xy}, c_x\}) \Downarrow_x$$

In words, this means that the information given by the constraints c_y, c_x and c_{xy} over x, that is, the semiring value associated to each value for x, is already contained in c_x.

We recall here that classical CSPs are just SCSPs over the c-semiring $S_{CSP} = \langle \{true, false\}, \vee, \wedge, false, true \rangle$. Thus the just defined notion of SAC becomes here the classical one of arc-consistency (AC) [9]: that for each pair of variables x and y, and for each value in the domain of x, say d_x, there exists a value in the domain of y, say d_y, such that the pair $\langle d_x, d_y \rangle$ satisfies the constraint between x and y.

For example, Figure 1 shows a CSP which is not AC, because for value b for x there is no value for y such that the constraint between x and y is satisfied. In this picture and also in the following one, we will use the graph-like representation of constraint problems, where variables are nodes and constraints are arcs, and where domains and constraint definitions are denoted as labels of the corresponding graph objects.

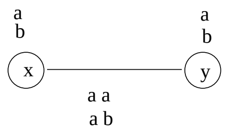

Figure 1: A CSP which is not AC.

If an SCSP is not SAC, we can make it so by basically enforcing the above definition over any unary constraint, possibly iterating more than once over the same pairs of variables, until we reach a fix-point. In other words, for each pair of variables x and y, we change the definition of c_x to make it coincide with $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_x$. This means that some values in the domain of x will possibly get a different value, which however is always smaller (that is, worse) w.r.t. the partial order $\leq_S$ because of the properties of $\times$.

In the special case of a CSP, the arc-consistency algorithm would possibly eliminate some domain elements for some variables, because they are not compatible with any element in the domain of an adjacent variable. In fact, elimination here means passing from *true* to *false*, that is, going to a worse value in the semiring.

It can be proved [2] that, if the $\times$ operation of the semiring is idempotent, then 1) a fix-point is always reached, no matter the order in which we look at different pairs of variables, provided that the order is fair, 2) the resulting SCSP is always the same, and 3) it is equivalent to the given one, that is, it gives the same semiring values to the same solutions.

Besides classical CSPs, another useful instance of the SCSP framework, which we will use in this paper for our examples, is the so-called fuzzy CSP framework (FCSP) [5, 13], where each tuple has associated a level of preference which is a real number between 0 and 1, constraints are combined by performing the *min* operation, and then solving a problem means maximizing the value for the tuples involving all the variables. Thus the semiring to use in this case is $S_{FCSP} = \langle [0,1], max, min, 0, 1\rangle$.

For example, Figure 2 shows a fuzzy CSP. Semiring values are written to the right of the corresponding tuples. The solutions of this SCSP are triples (that is, a value for each of the three variables) and their semiring value is obtained by looking at the smallest value for all the subtuples (as many as the constraints) forming the triple. For example, for tuple $\langle a, a\rangle$ (that is, $x = y = a$), we have to compute the minimum between 0.9, which is the value for $x = a$, 0.8, which is the value for $\langle x = a, y = a\rangle$, and 0.9, which is the value for $y = a$. So the resulting value for this tuple is 0.8. Notice also that this FCSP is not SAC. In fact, consider the semiring value assigned to $x = a$ by the unary constraint over x: it is 0.9. However, if we compute $(\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_x$ we get 0.8. To see this, first we have to combine the three constraints, obtaining a constraint over x and y. This will assign a semiring value to each pair for x and y, to be computed as we did before for tuple $x = y = a$. As another example, for tuple $\langle a, b\rangle$ we get $min(0.9, 0.2, 0.5) = 0.2$. Then we have to project over x. This will generate a constraint over x which assigns a semiring value to $x = a$ and another one for $x = b$. For example, since we know that $\langle a, a\rangle$ has value 0.8 and $\langle a, b\rangle$ has value 0.2, the projection will give value 0.8 (that is, the maximum between the two) to $x = a$. By comparing this value to the value given to $x = a$ by the unary constraint over x, which is 0.9, we can already say that this FCSP is not SAC, because the unary constraint over x and the one obtained by combining the three constraints of the problem should coincide.

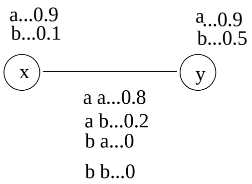

Figure 2: A fuzzy CSP.

3 Classical arc-consistency vs. semiring-based arc-consistency

Consider an SCSP $P = \langle C, con\rangle$. Let us now construct a corresponding CSP $cl(P)$ by simply forgetting the semiring values associated to all the tuples in all constraints of P, except for those values which are the minimum of the semiring. That is, if a tuple has a semiring value different from the minimum we say that such tuple satisfies the constraint in $cl(P)$, otherwise we say that it does not satisfy it. For example, the CSP in Figure 1 is the CSP corresponding to the FCSP of Figure 2 according to function cl.

Then, let us apply SAC over P and AC over $cl(P)$. As a result of that, some element in the domain of some variables in P will get a smaller semiring value, possibly the minimum of the semiring, and some domain elements of some variables in $cl(P)$ will be eliminated. The result of this section is the following one:

> in an SAC-consistent SCSP P, if the order $\leq_S$ induced by the $+$ operation is total, then the domain elements which will get the minimum of the semiring in P by applying an SAC algorithm are exactly those that will be eliminated in $cl(P)$ by applying an AC algorithm.

In fact, consider what happens during the SAC algorithm: for each pair of variables x and y, to see which value has to be given to the domain element t_x for x, that now has value a_x, we have to consider also all the pairs t_{xy} which extend t_x over y. Let us assume there are n of them, and let us call them $t^1_{xy}, \ldots, t^n_{xy}$. Assume also that each t^i_{xy} is given the semiring value a^i_{xy}. Finally, consider all domain elements of y, say $t^1_y, \ldots, t^n_y$, and assume that they now have the semiring values $a^1_y, \ldots, a^n_y$. What we have to do now to perform SAC over this constraint is to compute the following: $\sum_{i=1,\ldots,n}(a_x \times a^i_{xy} \times a^i_y)$. This is the semiring value to be associated to t_x. That is, if t_x has a different semiring value, the problem is not SAC-consistent.

Now the question is: when does this value coincide with the minimum of the semiring? In general we don't know. However, we know that, if the $\leq_S$ order is total, then $a \times b$ is always equal to either a or b, for any a, b in the semiring, and the same is for $a + b$. In fact, the glb and the lub of two elements in a total order is always one of the two elements.

Thus the only way that the above formula coincides with the minimum of the semiring is that such a minimum appears somewhere in the formula. That is, either a_x, or a^i_{xy} for some i, or a^i_y for some i, is already the minimum of the semiring. Said differently, if none of these values coincides with the minimum of the semiring, then we can be sure that the result of the above formula does not coincide with the minimum of the semiring.

Now, by the way we have constructed $cl(P)$, tuples which have the minimum of the semiring do not appear in $cl(P)$. That is, they do not satisfy the corresponding constraint. Therefore, by the way AC works, they will create a local inconsistency and they will cause the deletion of an element from the domain of a variable. More precisely, in $cl(P)$, to see whether element t_x of the domain of x has to be eliminated, we have to look at all pairs t_{xy} which extend t_x over y and which satisfy the constraint between x and y, and for each of such pairs we have to see whether its second element (that is, its projection over y) is in the domain of y or not. If all such elements are not in the domain of y, then t_x has to be eliminated by the domain of x. This means that for all pairs t_{xy} which extend t_x over y, either t_{xy} does not satisfy the constraint between x and y, or the projection of t_{xy} onto y is not in the domain of y. But if this is so, then, by the way $cl(P)$ was constructed, the element t_x in P will get the minimum of the semiring by SAC. The vice versa is also easily provable: if t_x gets the minimum of the semiring in P via SAC, then it will be eliminated in $cl(P)$ via AC.

Figure 3 shows the FCSP obtained by applying SAC onto the FCSP of Figure 2 and the CSP obtained by applying AC onto the CSP of Figure 1.

Why is this result interesting? Because applying AC over $cl(P)$ is more efficient, both in space and in time, than applying SAC over P. In fact, domains could have a much more compact representation, and each step of the algorithm could deal with a much smaller set of tuples (only those that have semiring value *true*).

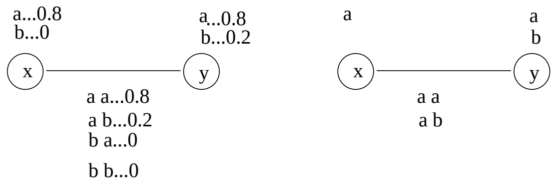

Figure 3: An SAC-consistent FCSP and the corresponding AC-consistent CSP.

Therefore one could use AC instead of SAC if he/she is interested only in discovering local inconsistencies. Of course some information will be lost, because the semiring values which are present in the problem after AC and which are different from the minimum are not as accurate, but in some cases this could be a reasonable trade-off between pruning power and space/time efficiency.

In fact, this approach could also be seen as the extreme case of a flexible abstraction technique, where, given an SCSP over a certain semiring S, we replace SAC over such semiring with SAC over a simpler semiring, not necessarily S_{CSP} (as we proposed in this section) which is the simplest semiring of all. In our framework, the usual Galois connection of abstract interpretation [3] would be function *cl*, which allows to pass from an SCSP to a CSP.

4 Removing Hidden Variables in SCSPs

An SCSP problem, as defined above, is a pair $\langle C, con\rangle$, where *con* is a subset of the variables, and specifies the variables of interest as far as solutions are concerned. Variables in *con* can then be called the *visible* variables, while all the others are *hidden*.

Since the solution set of such problems is defined as $(\otimes C) \Downarrow_{con}$, this means that a solution is an assignment of the visible variables, with an associated semiring value, such that it can be extended to the hidden variables in a way that the overall assignment (to all the variables) has the same semiring value. However, since we don't really care to know the values for the hidden variables, but just that there is a way to instantiate them, in some cases some of these variables can be found to be redundant. This means that their elimination would not change the solutions of the overall problem. This of course can be very convenient in terms of the search for one or the best solution, because the depth of the search tree coincides with the number of variables. In particular, this section shows the following:

> if we have an SCSP which is SAC-consistent, and if the order $\leq_S$ is total, then all hidden variables with degree 1 (or 0) are redundant and thus can be eliminated.

This is an extension to SCSP of a result which has been shown already for CSPs [12]. However, here is more interesting, since we are dealing with optimization and not satisfaction problems.

Consider an SCSP which is SAC-consistent, and assume that there is a hidden variable, say z, with degree 1. Let us now consider an instantiation of the visible variables with a corresponding semiring value which is given by combining all constraints involving subsets of the visible variables: say it is $\langle d_1, \ldots, d_n \rangle$ (there are n visible variables) with semiring value v. To show that z is redundant, we have to show that such a tuple can be extended to z without making its semiring value worse. In other words, there is a value for z, say d_z, such that the tuple $\langle d_1, \ldots, d_n, d_z \rangle$ has semiring value v.

To compute the semiring value for this tuple, once we have the semiring value v for the shorter tuple, we need to multiply (by using the $\times$ operator of the semiring) v by the semiring value given by c_{xz} (we assume that x is the i-th visible variable) to $\langle x = d_i, z = v \rangle$, say v_{xz}, and also by the semiring value given by c_z to $z = v$, say v_z. Notice that if z is indeed connected to x, this is the only constraint connected to z, otherwise we will just consider a null constraint which assigns the maximum of the semiring to each tuple. Thus we get $v \times v_{xz} \times v_z$, and we have to prove that this coincides with v. Now, we know that any SAC algorithm enforces the formula $c_x = (\otimes\{c_y, c_{xy}, c_x\}) \Downarrow_x$ for each pair of variables x and y. Thus this also applies to the pair of variables x and z of our example. Therefore the semiring values associated to elements of the domain of x, such as d_i, have been computed during the SAC algorithm by considering also c_{xz} and c_z. In other words, the information that we are adding now to v, that is, $v_{xz} \times v_z$, is already in v, because of SAC. Of course there are many ways to instantiate z, but if the order $\leq_S$ is total, one of them, that is, the best one, has been inserted in v. By choosing that value for z, by the idempotence of the $\times$ operation, which has to be assumed if we want to be sure that local consistency has the desired properties (see Section 2), we get that $v = v \times v_{xz} \times v_z$. Therefore, there exists a way to extend $\langle d_1, \ldots, d_n \rangle$ to z such that its semiring value is not changed. What is important to notice here is that all other ways of extending this tuple to z would worsen its semiring value, thus will not lead to an optimal solution.

Notice also that once some variables are eliminated, due to the result of this section, the degree of other variables may be reduced and thus satisfy the redundancy condition. Thus these other variables can be eliminated as well, until no variable has degree 1.

Notice also that it is not necessary to achieve SAC all over the problem to eliminate some hidden variable, but that it is enough to achieve it on the neighborhood (just two variables) of the variable to eliminate. Also, in such a neighborhood SAC could be achieved just in one direction, because we have to make sure that a solution involving the visible variables can be extended to the hidden variables, but we don't care about the opposite direction.

Again, we claim that the result of this section can easily be generalized to higher level of consistency, allowing the elimination of variables of higher degree. That is, the generalization should say that, if an SCSP is k-consistent, then all hidden variables with degree up to $k-1$ are redundant.

5 Tree-shaped SCSPs

The result of the previous section says that, if we have an SCSP with a total order over the semiring values, and if the SCSP is SAC-consistent, then any tuple involving the visible variables can be extended to any hidden variable with degree 1 without changing the semiring value of the tuple.

Consider now the same situation, except that the variable with degree 1 is not hidden, but visible. Still, the extension works just the same, and the only thing that is different here is that we have to constructively find that extension, and we cannot be satisfied with just knowing that it exists. Let us see what we have to do to find the (or a) domain element for z (the variable with degree 1) which does not change the semiring value of the given tuple. Since the difference between the semiring value of the new tuple and that of the old one depends only on the constraint connecting z, say c_{xz}, and on the domain of z, say c_z, we just have to look at all pairs of values for x and z such that $x = d_i$ (that is, the value that x has in the short tuple), and for each of such pairs we have to multiply the value given by c_{xz} by that given by c_z. Now, the best of such values is the one to choose. That is, we choose $z = d_z$ if $c_{xz}(d_i, d_z) \times c_z(d_z)$ is the best among all $c_{xz}(d_i, d'_z) \times c_z(d'_z)$ for all d'_z other values for z.

This is an operation that is linear in the domain size, since for each element of the domain of z, say d_z, we just need to compute $c_{xz}(d_i, d_z) \times c_z(d_z)$. More important, if the tuple we started with (that is, $\langle d_1, \ldots, d_n \rangle$) was the best one for the first n variables, then the new tuple, $\langle d_1, \ldots, d_n, d_z \rangle$ is still the best one for the n initial variables plus z. In fact, it is impossible to find a tuple which has a better semiring value for such $n+1$ variables. Otherwise the assumption that the initial tuple was the best one for the n variables will be contradicted.

This discussion leads to the main result of this section, which again is an extension of a result already known for classical CSPs [7]:

> if we have an SCSP with a total order over the semiring values, and such a SCSP is SAC-consistent and has the shape of a tree, then the optimal solution can be found in time linear in the number of its variables.

In fact, in a tree-shaped SCSP, all variables have either degree 1 (the leaves) or can get degree 1 when those with degree 1 are eliminated. Therefore the extension result just discussed can be applied at every step of the search. That is, at every step during the search, we have a partial solution involving only the previous variables, and we have to extend it to the next variable. By what we said before, this extension can be done without changing the semiring value of the solution, and also without the need to come back to check whether there is a better solution. Therefore a complete best solution can be found without any need to backtrack.

An interesting remark is that the semiring value for the best solution can be found without even finding the best solution. In fact, since each step does not change the semiring value of the current incomplete solution, the best value of a complete solution coincides with the best value for the first variable in the order. In fact, by choosing the best value for the first variable, we know that we can complete the solution maintaining that value.

Even for this result we can easily see that SAC is not really necessary, we just need a directional version of SAC which goes bottom-up in the tree.

6 Cycle-cutsets in SCSPs

By stretching the discussion of the previous section even more, we can consider SCSPs which in general do not have a tree shape, where however we identify in some way the part of the SCSP which creates the cycles. In such problems, we can first find the best solution for the difficult part (the one with cycles), and after that we can complete the solution on the rest of the variables, which now is tree-shaped. Therefore, the overall search for the best solution will possibly backtrack over the first part of the problem, to find the best solution for that part, but then will not backtrack at all on the rest of the problem.

In classical constraint solving this techniques is called *cycle-cutset* [4], since the first part of the problem (that possibly involving some backtracking) is identified by a subset of the variables whose elimination allows to remove all cycles of the problem. Here we can use the same technique and we get similar results, which however, as noted before, deal with optimization instead of satisfaction. Therefore here we can say something about the time complexity to find not *a* solution, but the *best* solution. Moreover, as in the previous section, we have an additional result which allows us to know the semiring value associated to the best solution before we actually compute the solution itself. In fact, once all variables in the cycle-cutset have been instantiated and we have found the best solution for them, the semiring value for this partial solution will coincide with the semiring value for the best complete solution.

More interesting results can be obtained if one applies the SAC algorithm not only before the search but also *during* the search, after each instantiation. In fact, instantiating a variable is like eliminating that variable for the problem to be solved, and thus new variables could become of degree 1 with respect to the set considered at the beginning of the search. By maintaining SAC at each step during the search, we could therefore exploit the presence of this variables to dynamically augment the set of variables over which the search does not need any backtracking. Again, this idea has been generated by recent results on classical CSPs [14] which however we claim can be extended to SCSPs as well.

7 Conclusions

We have studied some possible uses of SAC over SCSPs, which could make the search for the best solution faster. In the future we plan to extend formally these results to higher levels of consistency, although we are aware that AC is the most convenient pruning algorithm and thus in practice we will most probably use just the results of this paper. We also plan to assess the practical value of the results described here via experiments on randomly generated SCSPs.

References

[1] S. Bistarelli, U. Montanari, and F. Rossi. Constraint Solving over Semirings. In *Proc. IJCAI95*. Morgan Kaufman, 1995.

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

[3] P. Cousot and R. Cousot. Abstract interpretation and application to logic programming. *The Journal of Logic Programming*, 13 (2 and 3), 1992.

[4] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34(1), 1988.

[5] D. Dubois, H. Fargier, and H. Prade. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. IEEE International Conference on Fuzzy Systems*, pages 1131–1136. IEEE, 1993.

[6] E. C. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11), 1978.

[7] E. C. Freuder. Backtrack-free and backtrack-bounded search. In Kanal and Kumar, editors, *Search in Artificial Intelligence*, pages 343–369. Springer-Verlag, 1988.

[8] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1984.

[9] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[10] A.K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of AI (second edition)*, volume 1, pages 285–293. John Wiley & Sons, 1992.

[11] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7:95–132, 1974. Also Technical Report, Carnegie Mellon University, 1970.

[12] F. Rossi. Redundant hidden variables in finite domain constraint problems. In M.Meyer, editor, *Constraint Processing*. Springer-Verlag, LNCS 923, 1995.

[13] Zs. Ruttkay. Fuzzy constraint satisfaction. In *Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[14] D. Sabin and E.C. Freuder. Understanding and improving the mac algorithm. In *Proc. CP98, Springer-Verlag, LNCS 1330*, pages 167–181, 1997.

[15] Edward P. K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[16] D. L. Waltz. Understanding line drawings of scenes with shadows. In Winston, editor, *The Psycology of Computer Vision*. McGraw-Hill, 1975.