Semiring-Based Constraint Satisfaction and Optimization

STEFANO BISTARELLI, UGO MONTANARI,
AND FRANCESCA ROSSI

University of Pisa, Pisa, Italy

Abstract. We introduce a general framework for constraint satisfaction and optimization where classical CSPs, fuzzy CSPs, weighted CSPs, partial constraint satisfaction, and others can be easily cast. The framework is based on a semiring structure, where the set of the semiring specifies the values to be associated with each tuple of values of the variable domain, and the two semiring operations ($+$ and $\times$) model constraint projection and combination respectively. Local consistency algorithms, as usually used for classical CSPs, can be exploited in this general framework as well, provided that certain conditions on the semiring operations are satisfied. We then show how this framework can be used to model both old and new constraint solving and optimization schemes, thus allowing one to both formally justify many informally taken choices in existing schemes, and to prove that local consistency techniques can be used also in newly defined schemes.

Categories and Subject Descriptors: I.1.2 [**Algebraic Manipulation**]: Algorithms—*analysis of algorithms*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*uncertainty; "fuzzy", and probabilistic reasoning*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods.

General Terms: Algorithms, Theory.

Additional Key Words and Phrases: Constraint solving, dynamic programming, local consistency, non-crisp constraint reasoning.

1. *Introduction*

Classical constraint satisfaction problems (CSPs) [Montanari 1974; Mackworth 1992] are a very expressive and natural formalism to specify many kinds of real-life problems. In fact, problems ranging from map coloring, vision, robotics, job-shop scheduling, VLSI design, etc., can easily be cast as CSPs and solved using one of the many techniques that have been developed for such problems or subclasses of them.[1]

Authors' address: Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, e-mail: {bista,ugo,rossi}@di.unipi.it.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery (ACM), Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 1997 ACM 0004-5411/97/0300-0201 $03.50

[1] See, for example, Freuder [1978; 1988], Mackworth and Freuder [1985], Mackworth [1977], and Montanari [1974].

However, they also have evident limitations, mainly due to the fact that they do not appear to be very flexible when trying to represent real-life scenarios where the knowledge is not completely available nor crisp. In fact, in such situations, the ability of stating whether an instantiation of values to variables is allowed or not is not enough or sometimes not even possible. For these reasons, it is natural to try to extend the CSP formalism in this direction.

For example, in Rosenfeld et al. [1976], Dubois et al. [1993], and Ruttkay [1994], CSPs have been extended with the ability to associate with each tuple, or to each constraint, a level of preference, and with the possibility of combining constraints using min-max operations. This extended formalism has been called Fuzzy CSPs (FCSPs). Other extensions concern the ability to model incomplete knowledge of the real problem [Fargier and Lang 1993], to solve over-constrained problems [Freuder and Wallace 1992], and to represent cost optimization problems.

In this paper, we define a constraint solving framework where all such extensions, as well as classical CSPs, can be cast. However, we do not relax the assumption of a finite domain for the variables of the constraint problems. The main idea is based on the observation that a semiring (i.e., a domain plus two operations satisfying certain properties) is all that is needed to describe many constraint satisfaction schemes. In fact, the domain of the semiring provides the levels of consistency (which can be interpreted as cost, or degrees of preference, or probabilities, or others), and the two operations define a way to combine constraints together. More precisely, we define the notion of constraint solving over any semiring. Specific choices of the semiring will then give rise to different instances of the framework, which may correspond to known or new constraint solving schemes.

In classical CSPs, so-called local consistency techniques[2] have been proved to be very effective when approximating the solution of a problem. In this paper, we study how to generalize these techniques to our framework, and we provide sufficient conditions over the semiring operations which assure that they can also be fruitfully applied to the considered scheme. By "fruitfully applicable," we mean that (1) the algorithm terminates and (2) the resulting problem is equivalent to the given one and it does not depend on the nondeterministic choices made during the algorithm. In particular, such conditions rely mainly on having an idempotent operator (the $\times$ operator of the semiring).

The advantage of our framework, that we call *SCSP* (for *Semiring-based CSP*), is that one can hopefully see his own constraint solving paradigm as an instance of SCSP over a certain semiring, and can inherit the results obtained for the general scheme. In particular, one can immediately see whether a local consistency technique can be applied. In fact, our sufficient conditions, which are related to the chosen semiring, guarantee that the above basic properties of local consistency hold. Therefore, if they are satisfied, local consistency can safely be applied. Otherwise, it means that we cannot be sure that in general local consistency will be meaningful in the chosen instance.

In this paper, we consider several known and new constraint-solving frameworks, casting them as instances of SCSP, and we study the possibility of applying

[2] See, for example, Freuder [1978; 1988], Mackworth [1977; 1992], Montanari [1974], and Montanari and Rossi [1986].

local consistency algorithms. In particular, we confirm that CSPs enjoy all the properties needed to use such algorithms, and that they can also be applied to FCSPs. Moreover, we consider probabilistic CSPs [Fargier and Lang 1993], and we see that local consistency algorithms might be meaningless for such a framework, as well as for weighted CSPs (since the above-cited sufficient conditions do not hold).

We also define a suitable notion of dynamic programming over SCSPs, and we prove that it can be used over any instance of the framework, regardless of the properties of the semiring. Furthermore, we show that when it is possible to provide a whole class of SCSPs with parsings trees of bounded size, then the SCSPs of the class can be solved in linear time.

The notion of semiring for constraint solving has been used also in Giacobazzi et al. [1992]. However, the use of such a notion is completely different from ours. In fact, in Giacobazzi et al. [1992] the semiring domain (hereby denoted by A) is used to specify the domain of the variables, whereas here we always assume a finite domain (hereby denoted by D) and A is used to model the values associated with the tuples of values of D.

The work which is most related to ours is the one presented in Schiex et al. [1995]. However, the structure they use is not a semiring, but a totally ordered commutative monoid. This means that the order (to be used to compare different tuples) is always total, while in our framework it is in general partial, and this may be useful in some cases, like for example for the instances in Section 7. Also, they associate values with constraints instead of tuples. Moreover, while our aim is to study the properties of the semiring (i.e., mainly the idempotency of $\times$) that are sufficient to safely use the local consistency algorithms, they investigate the use of the values associated with the constraints for deriving useful lower bounds for branch-and-bound algorithms. Finally, they also study the generalization of the property of arc-consistency, and they show that strict monotonicity of the operator of the monoid guarantees that achieving this generalized notion is NP-complete. Note, however, that one can pass from one formulation to the other by using an appropriate transformation [Bistarelli et al. 1996], but only if a total order is assumed.

The paper is organized as follows: Section 2 defines c-semirings and their properties. Then Section 3 introduces constraint problems over any semirings and the associated notions of solution and consistency. Then, Section 4 introduces the concept of local consistency for SCSPs and gives sufficient conditions for the applicability of the local consistency algorithms. Section 5 describes a dynamic programming algorithm to solve SCSPs that can be applied to any instance of our framework, without any condition. Then, Section 6 studies several instances of the SCSP framework and for each of them proves the applicability, or not, of the local consistency algorithms. Section 7 shows that our framework allows also for a mixed form of reasoning, where problems are optimized according to several criteria.

This paper is a revised and extended version of Bistarelli et al. [1995]. The main changes with respect to that paper, apart from the greater level of detail and formalization throughout the whole paper, concern the definition of local consistency algorithms instead of the less general k-consistency algorithms, the more extensive use of typed locations both for local consistency and for dynamic programming, the description of dynamic programming as a particular local

consistency algorithm, and the proof that in some cases (of SCSPs with bounded parsing tree) dynamic programming has a linear complexity.

2. *C-Semirings and Their Properties*

We extend the classical notion of constraint satisfaction to allow also for (1) noncrisp statements and (2) a more general interpretation of the operations on such statements. This allows us to model both classical CSPs and several extensions of them (like fuzzy CSPs [Dubois et al. 1993; Ruttkay 1994], partial constraint satisfaction [Freuder and Wallace 1992], etc.), and also to possibly define new frameworks for constraint solving. To formally do that, we associate a semiring with the standard notion of constraint problem, so that different choices of the semiring represent different concrete constraint satisfaction schemes. In fact, such a semiring will give us both the domain for the noncrisp statements and also the allowed operations on them.

Definition 2.1 (*semiring*). A *semiring* is a tuple $\langle A, \text{sum}, \times, \mathbf{0}, \mathbf{1}\rangle$ such that

—A is a set and $\mathbf{0}, \mathbf{1} \in A$;

—sum, called the additive operation, is a commutative (i.e., $\text{sum}(a, b) = \text{sum}(b, a)$) and associative (i.e., $\text{sum}(a, \text{sum}(b, c)) = \text{sum}(\text{sum}(a, b), c)$) operation such that $\text{sum}(a, \mathbf{0}) = a = \text{sum}(\mathbf{0}, a)$ (i.e., $\mathbf{0}$ is its unit element);

—$\times$, called the multiplicative operation, is an 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 sum (i.e., for any $a, b, c \in A$, $a \times \text{sum}(b, c) = \text{sum}((a \times b), (a \times c))$).

In the following, we will consider semirings with additional properties of the two operations. Such semirings will be called *c-semiring*, where "c" stands for "constraint", meaning that they are the natural structures to be used when handling constraints.

Definition 2.2 (*c-semiring*). A *c-semiring* is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that

—A is a set and $\mathbf{0}, \mathbf{1} \in A$;

—$+$ is defined over (possibly infinite) sets of elements of A as follows:[3]

  —for all $a \in A$, $(\{a\}) = a$;

  — $(\emptyset) = \mathbf{0}$ and $(A) = \mathbf{1}$;

  — $(\cup A_i, i \in S) = (\{ (A_i), i \in S\})$ for all sets of indices S (flattening property).

—$\times$ is a binary, associative and commutative operation such that $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element;

—$\times$ distributes over $+$ (i.e., for any $a \in A$ and $B \subseteq A$, $a \times (B) = (\{a \times b, b \in B\})$).

Operation $+$ is defined over any set of elements of A, also over infinite sets. This will be useful later in proving Theorem 2.9. The fact that $+$ is defined over

[3] When $+$ is applied to a two-element set, we will use the symbol $+$ in infix notation, while in general we will use the symbol in prefix notation.

sets of elements, and not *pairs* or *tuples*, automatically makes such an operation commutative, associative, and idempotent. Moreover, it is possible to show that **0** is the unit element of $+$; in fact, by using the flattening property we get $(\{a, \mathbf{0}\}) = (\{a\} \cup \emptyset) = (\{a\}) = a$. This means that a c-semiring is a semiring (where the *sum* operation is $+$) with some additional properties.

It is also possible to prove that **1** is the absorbing element of $+$. In fact, by flattening and by the fact that we set $(A) = \mathbf{1}$, we get $(\{a, \mathbf{1}\}) = (\{a\} \cup A) = (A) = \mathbf{1}$.

Let us now consider the advantages of using c-semirings instead of semirings. First, the idempotence of the $+$ operation is needed in order to define a partial ordering $\leq_S$ over the set A, which will enable us to compare different elements of the semiring. Such partial order is defined as follows: $a \leq_S b$ iff $a + b = b$. Intuitively, $a \leq_S b$ means that b is "better" than a, or, from another point of view, that, between a and b, the $+$ operation chooses b. This ordering will be used later to choose the "best" solution in our constraint problems.

THEOREM 2.3 ($\leq_S$ IS A PARTIAL ORDER). *Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then $\leq_S$ is a partial order.*

PROOF. We have to prove that $\leq_S$ is reflexive, transitive, and antisymmetric:

—Since $+$ is idempotent, we have that $a + a = a$ for any $a \in A$. Thus, by definition of $\leq_S$, we have that $a \leq_S a$. Thus, $\leq_S$ is reflexive.

—Assume $a \leq_S b$ and $b \leq_S c$. This means that $a + b = b$ and $b + c = c$. Thus, $c = b + c = (a + b) + c = a + (b + c) = a + c$. Note that here we also used the associativity of $+$. Thus $\leq_S$ is transitive.

—Assume that $a \leq_S b$ and $b \leq_S a$. This means that $a + b = b$ and $b + a = a$. Thus $a = b + a = a + b = b$. Thus, $\leq_S$ is antisymmetric. □

The fact that **0** is the unit element of the additive operation implies that **0** is the minimum element of the ordering. Thus, for any $a \in A$, we have $\mathbf{0} \leq_S a$.

It is important to notice that both the additive and the multiplicative operations are monotone on such an ordering.

THEOREM 2.4 ($+$ AND $\times$ ARE MONOTONE OVER $\leq_S$). *Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, consider the relation $\leq_S$ over A. Then $+$ and $\times$ are monotone over $\leq_S$. That is, $a \leq_S a'$ implies $a + b \leq_S a' + b$ and $a \times b \leq_S a' \times b$.*

PROOF. Assume $a \leq_S a'$. Then, by definition of $\leq_S$, $a + a' = a'$.

Thus, for any b, $a' + b = a + a' + b$. By idempotence of $+$, we also have $a' + b = a + a' + b = a + a' + b + b$, which, by commutativity of $+$, becomes $a' + b = (a + b) + (a' + b)$. By definition of $\leq_S$, we have $a + b \leq_S a' + b$.

Also, from $a + a' = a'$ derives that, for any b, $a' \times b = (a' + a) \times b =$ (by distributiveness) $(a' \times b) + (a \times b)$. This means that $(a \times b) \leq_S (a' \times b)$. □

The commutativity of the $\times$ operation is desirable when such an operation is used to combine several constraints. In fact, were it not commutative, it would mean that different orders of the constraints give different results.

Since **1** is also the absorbing element of the additive operation, then $a \leq_S \mathbf{1}$ for all a. Thus, **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 (with respect to the $\leq_S$ ordering) result.

THEOREM 2.5 ($\times$ IS INTENSIVE). *Given any c-semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, *consider the relation* $\leq_S$ *over A. Then* $\times$ *is intensive, that is,* $a, b \in A$ *implies* $a \times b \leq_S a$.

PROOF. Since **1** is the unit element of $\times$, we have $a = a \times 1$. Also, since **1** is the absorbing element of $+$, we have $\mathbf{1} = \mathbf{1} + b$. Thus $a = a \times (\mathbf{1} + b)$. Now, $a \times (\mathbf{1} + b) = \{$by distributiveness of $\times$ over $+\}$ $(a \times \mathbf{1}) + (a \times b) = \{\mathbf{1}$ unit element of $\times\}$ $a + (a \times b)$. Thus, we have $a = a + (a \times b)$, which, by definition of $\leq_S$, means that $(a \times b) \leq_S a$. $\square$

In the following, we will sometimes need the $\times$ operation to be closed on a certain finite subset of the c-semiring.

Definition 2.6 (*AD-closed*). Given any c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, consider a finite set $AD \subseteq A$. Then $\times$ is AD-closed if, for any $a, b \in AD$, $(a \times b) \in AD$.

We will now show that c-semirings can be assimilated to complete lattices. Moreover, we will also sometimes need to consider c-semirings where $\times$ is idempotent, which we will show equivalent to distributive lattices. See Davey and Priestly [1990] for a deep treatment of lattices.

Definition 2.7 (*lub*, *glb*, *lattice*, *complete lattice* [*Davey and Priestly* 1990]). Consider a partially ordered set S and any subset I of S. Then we define the following:

—an upper bound (resp., lower bound) of I is any element x such that, for all $y \in I$, $y \leq x$ (resp., $x \leq y$);

—the least upper bound (lub) (resp., greatest lower bound (glb)) of I is an upper bound (resp., lower bound) x of I such that, for any other upper bound (resp., lower bound) x' of I, we have that $x \leq x'$ (resp., $x' \leq x$).

A lattice is a partially ordered set where every subset of two elements has a lub and a glb. A complete lattice is a partially ordered set where every subset has a lub and a glb.

We will now prove a property of partially ordered sets where every subset has the lub, which will be useful in proving that $\langle A, \leq_S\rangle$ is a complete lattice. Notice that when every subset has the lub, then also the empty set has the lub. Thus, in partial orders with this property, there is always a global minimum of the partial order (which is the lub of the empty set).

LEMMA 2.8 (LUB $\Rightarrow$ GLB). *Consider a partial order* $\langle A, \leq\rangle$ *where there is the lub of every subset* I *of* A. *Then there exists the glb of I as well.*

PROOF. Consider any set $I \subseteq A$, and let us call $LB(I)$ the set of all lower bounds of S. That is, $LB(I) = \{x \in A \mid \text{for all } y \in I, x \leq_S y\}$. Then consider $a = lub(LB(I))$. We will prove that a is the glb of I.

Consider any element $y \in I$. By definition of $LB(I)$, we have that all elements x in $LB(I)$ are smaller than y, thus y is an upper bound of $LB(I)$. Since a is by definition the smallest among the upper bounds of $LB(I)$, we also have that $a \leq_S y$. This is true for all elements y in I. Thus, a is a lower bound of I, which means that it must be contained in $LB(I)$. Thus, we have found an element of A which is the greatest among all lower bounds of I. □

THEOREM 2.9 ($\langle A, \leq_S \rangle$ IS A COMPLETE LATTICE). *Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the partial order $\leq_S$. Then $\langle A, \leq_S \rangle$ is a complete lattice.*

PROOF. To prove that $\langle A, \leq_S \rangle$ is a complete lattice it is enough to show that every subset of A has the lub. In fact, by Lemma 2.8, we would get that each subset of A has both the lub and the glb, which is exactly the definition of a complete lattice.

We already know by Theorem 2.3 that $\langle A, \leq_S \rangle$ is a partial order. Now, consider any set $I \subseteq A$, and let us set $m = sum(I)$ and $n = lub(I)$. Taken any element $x \in I$, we have that $x + m =$ (by flattening) $sum(\{x\} \cup I) = sum(I) = m$. Therefore, by definition of $\leq_S$, we have that $x \leq_S m$. Thus, also $n \leq_S m$, since m is an upper bound of I and n by definition is the least among the upper bounds.

On the other hand, we have that $m + n =$ (by definition of sum) $sum(\{m\} \cup \{n\}) =$ (by flattening and since $m = sum(I)$) $sum(I \cup \{n\}) =$ (by giving an alternative definition of the set $I \cup \{n\}$) $sum(\bigcup_{x \in I}(\{x\} \cup \{n\})) =$ (by flattening) $sum(\{sum(\{x\} \cup \{n\}), x \in I\}) =$ (since $x \leq_S n$ and thus $sum(\{x\} \cup \{n\}) = n$) $sum(\{n\}) = n$. Thus, we have proved that $m \leq_S n$, which, together with the previous result (that $n \leq_S m$) yields $m = n$. In other words, we proved that $sum(I) = lub(I)$ for any set $I \subseteq A$. Thus, every subset of A, say I, has a least upper bound (which coincides with $sum(I)$). Thus, $\langle A, \leq_S \rangle$ is a lub-complete partial order. □

Note that the proof of the previous theorem also says that the sum operation coincides with the lub of the lattice $\langle A, \leq_S \rangle$.

THEOREM 2.10 ($\times$ IDEMPOTENT IMPLIES $\langle A, \leq_S \rangle$ DISTRIBUTIVE AND $\times$ = GLB). *Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, consider the corresponding complete lattice $\langle A, \leq_S \rangle$. If $\times$ is idempotent, then we have that*:

(1) $+$ *distributes over* $\times$;

(2) $\times$ *coincides with the glb operation of the lattice*;

(3) $\langle A, \leq_S \rangle$ *is a distributive lattice.*

PROOF

(1) $(a + b) \times (a + c) =$ {since $\times$ distributes over $+$}
$((a + b) \times a) + ((a + b) \times c)) =$ {same as above}
$((a \times a) + (a \times b)) + ((a + b) \times c)) =$ {by idempotence of $\times$}
$(a + (a \times b)) + ((a + b) \times c)) =$ {by intensivity of $\times$ and definition of $\leq_S$}
$a + ((a + b) \times c)) =$ {since $\times$ distributes over $+$}
$a + ((c \times a) + (c \times b)) =$ {by intensivity of $\times$ and definition of $\leq_S$}
$a + (c \times b)$.

(2) Assume that $a \times b = c$. Then, by intensivity of $\times$ (see Theorem 2.5), we have that $c \leq_S a$ and $c \leq_S b$. Thus, c is a lower bound for a and b. To show that it is a glb, we need to show that there is no other lower bound c' such that $c \leq_s c'$. Assume that such c' exists. We now prove that it must be $c' = c$:

$c' = \{\text{since } c \leq_S c'\}$

$c' + c = \{\text{since } c = a \times b\}$

$c' + (a \times b) = \{\text{since } + \text{ distributes over } \times\text{, see previous point}\}$

$(c' + a) \times (c' + b) = \{\text{since we assumed } c' \leq_S a \text{ and } c' \leq_S b\text{, and by definition of } \times\}$

$a \times b = \{\text{by assumption}\}$

c.

(3) This comes from the fact that $+$ is the lub, $\times$ is the glb, and $\times$ distributes over $+$ by definition of semiring (the distributiveness in the other direction is given by Lemma 6.3 in Davey and Priestly [1990], or can be seen in the first point above). $\square$

Note that, in the particular case in which $\times$ is idempotent and $\leq_S$ is total, we have that $a + b = max(a, b)$ and $a \times b = min(a, b)$.

3. *Constraint Systems and Problems*

We will now define the notion of constraint system, constraint, and constraint problem, which will be parametric with respect to the notion of c-semiring just defined. Intuitively, a constraint system specifies the c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ to be used, the set of all variables and their domain D.

Definition 3.1 (*constraint system*). A constraint system is a tuple $CS = \langle S, D, V\rangle$, where S is a c-semiring, D is a finite set, and V is an ordered set of variables.

Now, a constraint over a given constraint system specifies the involved variables and the "allowed" values for them. More precisely, for each tuple of values (of D) for the involved variables, a corresponding element of A is given. This element can be interpreted as the tuple's weight, or cost, or level of confidence, or else.

Definition 3.2 (*constraint*). Given a constraint system $CS = \langle S, D, V\rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, a constraint over CS is a pair $\langle def, con\rangle$, where

—$con \subseteq V$, it is called the *type* of the constraint;

—def: $D^k \rightarrow A$ (where k is the cardinality of con) is called the *value* of the constraint.

A constraint problem is then just a set of constraints over a given constraint system, plus a selected set of variables (thus a *type*). These are the variables of interest in the problem, that is, the variables of which we want to know the possible assignments compatibly with all the constraints.

Definition 3.3 (*constraint problem*). Given a constraint system $CS = \langle S, D, V\rangle$, a constraint problem P over CS is a pair $P = \langle C, con\rangle$, where C is a set of constraints over CS and $con \subseteq V$. We also assume that $\langle def_1, con'\rangle \in C$ and

$\langle def_2, con' \rangle \in C$ implies $def_1 = def_2$. In the following we will write SCSP to refer to such constraint problems.

Note that the above condition (that there are no two constraints with the same type) is not restrictive. In fact, if in a problem we had two constraints $\langle def_1, con' \rangle$ and $\langle def_2, con' \rangle$, we could replace both of them with a single constraint $\langle def, con' \rangle$ with $def(t) = def_1(t) \times def_2(t)$ for any tuple t. Similarly, if C were a multiset, for example, if a constraint $\langle def, con' \rangle$ had n occurrences in C, we could replace all its occurrences with the single constraint $\langle def', con' \rangle$ with

$$def'(t) = \underbrace{def(t) \times \cdots \times def(t)}_{n \text{ times}}$$

for any tuple t. However, this assumption implies that the operation of union of constraint problems is not just set union, since it has to take into account the possible presence of constraints with the same type in the problems to be combined (at most one in each problem), and, in that case, it has to perform the just described constraint replacement operations.

When all variables are of interest, like in many approaches to classical CSP, con contains all the variables involved in any of the constraints of the given problem, say P. Such a set, called $V(P)$, is a subset of V that can be recovered by looking at the variables involved in each constraint. That is, $V(P) = \bigcup_{\langle def, con' \rangle \in C} con'$.

As for classical constraint solving, also SCSPs as defined above can be graphically represented via labeled hypergraphs where nodes are variables, hyperarcs are constraints, and each hyperarc label is the definition of the corresponding constraint (which can be seen as a set of pairs ⟨tuple, value⟩). The variables of interest can then be marked in the graph.

Note that the above definition is parametric with respect to the constraint system CS and thus with respect to the semiring S. In the following, we will present several instantiations of such a framework, and we will show them to coincide with known and also new constraint satisfaction systems.

In the *SCSP* framework, 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. Analogous operations have been originally defined for fuzzy relations in Zadeh [1965], and have then been used for fuzzy CSPs in Dubois et al. [1993]. Our definition is however more general since we do not consider a specific c-semiring (like that we will define for fuzzy CSPs later) but a general one.

Definition 3.4 (*tuple projection*). Given a constraint system $CS = \langle S, D, V \rangle$ where V is totally ordered via ordering $\prec$, consider any k-tuple[4] $t = \langle t_1, \ldots, t_k \rangle$

[4] Given any integer k, a k-tuple is just a tuple of length k. Also, given a set S, an S-tuple is a tuple with as many elements as the size of S.

of values of D and two sets $W = \{w_1, \ldots, w_k\}$ and $W' = \{w'_1, \ldots, w'_m\}$ such that $W' \subseteq W \subseteq V$ and $w_i \prec w_j$ if $i \leq j$ and $w'_i \prec w'_j$ if $i \leq j$. Then the projection of t from W to W', written $t \downarrow^{W}_{W'}$, is defined as the tuple $t' = \langle t'_1, \ldots, t'_m \rangle$ with $t'_i = t_j$ if $w'_i = w_j$.

Definition 3.5 (*combination*). Given a constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, and two constraints $c_1 = \langle def_1, con_1 \rangle$ and $c_2 = \langle def_2, con_2 \rangle$ over CS, their combination, written $c_1 \otimes c_2$, is the constraint $c = \langle def, con \rangle$ with

$$con = con_1 \cup con_2$$

and

$$def(t) = def_1(t \downarrow^{con}_{con_1}) \times def_2(t \downarrow^{con}_{con_2}).$$

Since $\times$ is both commutative and associative, also $\otimes$ is so. Thus, this operation can be easily extended to more than two arguments, say $C = \{c_1, \ldots, c_n\}$, by performing $c_1 \otimes c_2 \otimes \cdots \otimes c_n$, which we will sometimes denote by $(\bigotimes C)$.

Definition 3.6 (*projection*). Given a constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a constraint $c = \langle def, con \rangle$ over CS, 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

$$def'(t') = \sum_{\{t \mid t \downarrow^{con}_{I \cap con} = t'\}} def(t).$$

PROPOSITION 3.7. *Given a constraint $c = \langle def, con \rangle$ over a constraint system CS and a set I of variables ($I \subseteq V$), we have that $c \Downarrow_I = c \Downarrow_{I \cap con}$.*

PROOF. Follows trivially from Definition 3.6. □

A useful property of the projection operation is the following:

THEOREM 3.8. *Given a constraint c over a constraint system CS, we have that $c \Downarrow_{S_1} \Downarrow_{S_2} = c \Downarrow_{S_2}$ if $S_2 \subseteq S_1$.*

PROOF. To prove the theorem, we have to show that the two constraints $c_1 = c \Downarrow_{S_1} \Downarrow_{S_2}$ and $c_2 = c \Downarrow_{S_2}$ coincide, that is, they have the same con and the same def. Assume $c = \langle def, con \rangle$, $c_1 = \langle def_1, con_1 \rangle$, and $c_2 = \langle def_2, con_2 \rangle$. Now, $con_1 = S_2 \cap (S_1 \cap con)$. Since $S_2 \subseteq S_1$, we have that $con_1 = S_2 \cap con$. Also, $con_2 = S_2 \cap con$; thus, $con_1 = con_2$. Consider now def_1. By Definition 3.6, we have that $def_1(t_1) = \sum_{\{t' \mid t' \downarrow^{S_1}_{S_2} = t_1\}} \sum_{\{t \mid t \downarrow^{con}_{S_1} = t'\}} def(t)$, which, by associativity of $+$, is the same as $\sum_{\{t \mid t \downarrow^{con}_{S_2} = t_1\}} def(t)$, which coincides with $def_2(t_1)$ by Definition 3.6. □

We will now prove a property which can be seen as a generalization of the distributivity property in predicate logic, which we recall is $x.(p \wedge q) = (x.p) \wedge q$ if x not free in q (where p and q are two predicates). The extension we prove

for our framework is given by the fact that $\otimes$ can be seen as a generalized $\wedge$ and $\Downarrow$ as a variant[5] of . This property will be useful later in Section 5.

THEOREM 3.9. *Given two constraints* $c_1 = \langle def_1, con_1 \rangle$ *and* $c_2 = \langle def_2, con_2 \rangle$ *over a constraint system CS, we have that* $(c_1 \otimes c_2) \Downarrow_{(con_1 \cup con_2)-x} = c_1 \Downarrow_{con_1 - x} \otimes c_2$ *if* $x \cap con_2 = \emptyset$.

PROOF. Let us set $c = (c_1 \otimes c_2) \Downarrow_{(con_1 \cup con_2)-x} = \langle def, con \rangle$ and $c' = c_1 \Downarrow_{con_1 - x} \otimes c_2 = \langle def', con' \rangle$. We will first prove that $con = con'$: $con = (con_1 \cup con_2) - x$ and $con' = (con_1 - x) \cup con_2$, which is the same as con if $x \cap con_2 = \emptyset$, as we assumed. Now we prove that $def = def'$. By definition, $def(t) = \Sigma_{\{t' | t' \downarrow^{con_1 \cup con_2}_{con_1 \cup con_2 - x} = t\}} (def_1(t' \downarrow^{con_1 \cup con_2}_{con_1}) \times def_2(t' \downarrow^{con_1 \cup con_2}_{con_2}))$. Now, $def_2(t' \downarrow^{con_1 \cup con_2}_{con_2})$ is independent from the summation, since x is not involved in c_2. Thus, it can be taken out. Also, $t' \downarrow^{con_1 \cup con_2}_{con_2}$ can be substituted by $t \downarrow^{con_1 \cup con_2}_{con_2}$, since t' and t must coincide on the variables different from x. Thus, we have: $(\Sigma_{\{t' | t' \downarrow^{con_1 \cup con_2}_{con_1 \cup con_2 - x} = t\}} def_1(t' \downarrow^{con_1 \cup con_2}_{con_1})) \times def_2(t \downarrow^{con_1 \cup con_2 - x}_{con_2})$. Now, the summation is done over those tuples t' that involve all the variables and coincide with t on the variables different from x. By observing that the elements of the summation are given by $def_1(t' \downarrow^{con_1 \cup con_2}_{con_1})$, and thus they only contain variables in con_1, we can conclude that the result of the summation does not change if it is done over the tuples involving only the variables in con_1 and still coinciding with t over the variables in con_1 that are different from x. Thus, we get:

$$\left(\sum_{\{t_1 | t_1 \downarrow^{con_1}_{con_1 - x} = t \downarrow^{con_1 \cup con_2 - x}_{con_1 - x}\}} def_1(t_1) \right) \times def_2(t \downarrow^{con_1 \cup con_2 - x}_{con_2}).$$

It is easy to see that this formula is exactly the definition of def'. $\square$

Using the operations of combination and projection, we can now define the notion of solution of an SCSP. In the following, we will consider a fixed constraint system $CS = \langle S, D, V \rangle$, where $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$.

Definition 3.10 (*solution*). Given a constraint problem $P = \langle C, con \rangle$ over a constraint system CS, the solution of P is a constraint defined as $Sol(P) = (\otimes C) \Downarrow_{con}$.

In words, the solution of an SCSP is the constraint induced on the variables in con by the whole problem. Such a constraint provides, for each tuple of values of D for the variables in con, a corresponding value of A. Sometimes, it is enough to know just the best value associated with such tuples. In our framework, this is still a constraint (over an empty set of variables), and will be called the best level of consistency of the whole problem, where the meaning of "best" depends on the ordering $\leq_S$ defined by the additive operation.

Definition 3.11 (*best level of consistency*). Given an SCSP problem $P = \langle C, con \rangle$, we define $blevel(P) \in S$ such that $\langle blevel(P), \emptyset \rangle = (\otimes C) \Downarrow_{\emptyset}$. Moreover, we say that P is consistent if $\mathbf{0} <_S blevel(P)$, and that P is α-consistent if $blevel(P) = \alpha$.

[5] Note however that the operator corresponding to x is given by $\Downarrow^{W}_{W-\{x\}}$.

Informally, the best level of consistency gives us an idea of how much we can satisfy the constraints of the given problem. Note that *blevel*(P) does not depend on the choice of the distinguished variables, due to the associative property of the additive operation. Thus, since a constraint problem is just a set of constraints plus a set of distinguished variables, we can also apply function *blevel* to a set of constraints only. More precisely, *blevel*(C) will mean *blevel*$(\langle C, con\rangle)$ for any con.

Note that *blevel*(P) can also be obtained by first computing the solution and then projecting such a constraint over the empty set of variables, as the following proposition shows.

PROPOSITION 3.12. *Given an SCSP problem P, we have that $Sol(P) \Downarrow_\emptyset = \langle blevel(P), \emptyset\rangle$.*

PROOF. $Sol(P) \Downarrow_\emptyset$ coincides with $((\otimes\ C) \Downarrow_{con}) \Downarrow_\emptyset$ by definition of $Sol(P)$. This coincides with $(\otimes\ C) \Downarrow_\emptyset$ by Theorem 3.8, thus it is the same as $\langle blevel(P), \emptyset\rangle$ by Definition 3.11. □

Another interesting notion of solution, more abstract than the one defined above, but sufficient for many purposes, is the one that does not consider those tuples whose associated value is worse (with respect to $\leq_S$) than that of other tuples.

Definition 3.13 (*abstract solution*). Given an SCSP problem $P = \langle C, con\rangle$, consider $Sol(P) = \langle def, con\rangle$. Then the abstract solution of P is the set $ASol(P) = \{\langle t, v\rangle \mid def(t) = v$ and there is no t' such that $v \leq_S def(t')\}$.

Note that, when the $\leq_S$ ordering is a total order (i.e., when, for any a and b, $a + b$ is either a or b), then the abstract solution contains only those tuples whose associated value coincides with *blevel*(P). In general, instead, an incomparable set of tuples is obtained, and thus *blevel*(P) is just an upper bound of the values associated with the tuples.

By using the ordering $\leq_S$ over the semiring, we can also define a corresponding ordering on constraints with the same type.

Definition 3.14 (*constraint ordering*). Consider two constraints c_1, c_2 over CS, and assume that $con_1 = con_2$ and $|con_1| = k$. Then $c_1 \sqsubseteq_S c_2$ if and only if, for all k-tuples t of values from D, $def_1(t) \leq_S def_2(t)$.

Notice that, if $c_1 \sqsubseteq_S c_2$ and $c_2 \sqsubseteq_S c_1$, then $c_1 = c_2$.

THEOREM 3.15 ($\sqsubseteq_S$ IS A PARTIAL ORDER). *The relation $\sqsubseteq_S$ over the set of constraints over CS is a partial order.*

PROOF. By definition, such a relation is antisymmetric. Also, it is easy to see that it is reflexive and transitive. □

The notion of constraint ordering, and the fact that the solution is a constraint, can also be useful to define an ordering on problems.

Definition 3.16 (*problem ordering and equivalence*). Consider two SCSP problems $P_1 = \langle C_1, con\rangle$ and $P_2 = \langle C_2, con\rangle$ over CS. Then $P_1 \sqsubseteq_P P_2$ if $Sol(P_1) \sqsubseteq_S Sol(P_2)$. If $P_1 \sqsubseteq_P P_2$ and $P_2 \sqsubseteq_P P_1$, then they have the same solution. Thus, we say that P_1 and P_2 are equivalent and we write $P_1 \equiv P_2$.

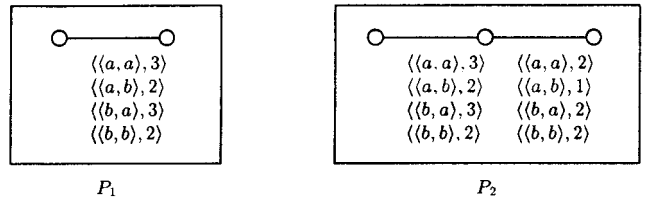

FIG. 1. Two SCSPs.

THEOREM 3.17 ($\sqsubseteq_P$ IS A PREORDER AND $\equiv$ IS AN EQUIVALENCE). *The relation $\sqsubseteq_P$ over the set of constraint problems over CS is a preorder. Moreover, $\equiv$ is an equivalence relation.*

PROOF. It is trivial to see that $\sqsubseteq_P$ is reflexive and transitive, due to the definition of constraint ordering $\sqsubseteq_S$. Thus, $\sqsubseteq_P$ is a preorder. From this, it derives that $\equiv$ is reflexive and transitive as well. Moreover, it is trivially symmetric. □

The ordering $\sqsubseteq_P$ can also be used for ordering sets of constraints, since a set of constraint is just a problem where *con* contains all the variables.

It is interesting now to note that, as in the classical CSP case, also the SCSP framework is monotone. That is, if some constraints are added, the solution of the new problem precedes that of the old one in the ordering $\sqsubseteq_S$. In other words, the new problem precedes the old one with respect to the preorder $\sqsubseteq_P$.

THEOREM 3.18 (MONOTONICITY). *Consider two SCSP problems $P_1 = \langle C_1, con\rangle$ and $P_2 = \langle C_1 \cup C_2, con\rangle$ over CS. Then $P_2 \sqsubseteq_P P_1$ and $blevel(P_2) \leq_S blevel(P_1)$.*

PROOF. $P_2 \sqsubseteq_P P_1$ follows from the intensivity of $\times$ and the monotonicity of $+$. The same holds also for $blevel(P_2) \leq_S blevel(P_1)$, since both *Sol* and $blevel(P)$ are obtained by projecting over a subset of the variables (which is always empty in the case of $blevel(P)$). □

When one is interested in the abstract solution rather than in the solution of an SCSP, then the above notion of monotonicity is lost. In fact, it is possible that by adding some constraints the best level of consistency gets worse, while the set of tuples in the abstract solution grows. See, for example, Figure 1, where there is a problem P_1 with one constraint and both variables as type, and a problem P_2 with the same constraint plus another one, and the two leftmost variables as type. Assume also that $\times$ is min and that $+$ is max. As we will see later, this amounts to considering the fuzzy constraint satisfaction framework, called FCSP. Now, it is easy to see that $blevel(P_1) = 3$ and $blevel(P_2) = 2$. The abstract solution of P_1 contains the tuples $\langle a, a\rangle$ and $\langle b, a\rangle$, while that of P_2 contains also $\langle a, b\rangle$ and $\langle b, b\rangle$. Thus, adding one constraint in this case makes the best level of consistency worse while enlarging the set of tuples in the abstract solution.

However, another notion of monotonicity holds, but only when the best level is not changed by the addition of some constraints and the order $\leq_S$ is total.

THEOREM 3.19 (SUBSET-MONOTONICITY VS. ASOL). *Consider two SCSP problems $P_1 = \langle C_1, con\rangle$ and $P_2 = \langle C_1 \cup C_2, con\rangle$ over CS, and assume that $\leq_S$ is total and that $blevel(P_1) = blevel(P_2)$. Then $Asol(P_2) \subseteq Asol(P_1)$.*

PROOF. If $\leq_S$ is total, then the abstract solution is a set of tuples with associated value exactly the best level. Take a tuple t in $ASol(P_2)$. This means that the value associated with t in P_2 is $blevel(P_2)$. Then, by Theorem 3.18, the value associated with t in P_1, say v, is such that $blevel(P_2) \leq_S v$. By assumption, $blevel(P_1) = blevel(P_2)$. If $v \neq blevel(P_2)$, then this assumption is violated. Thus, it must be $v = blevel(P_2) = blevel(P_1)$. Thus, t is also in $Asol(P_1)$. Therefore, $Asol(P_2) \subseteq Asol(P_1)$. □

4. *Local Consistency*

Computing any one of the previously defined notions (the best level of consistency, the solution, and the abstract solution) is an NP-hard problem. Thus, it can be convenient in many cases to approximate such notions. In classical CSPs, this is done using the so-called local consistency techniques.[6] 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 known local consistency algorithms are arc-consistency [Mackworth 1977], where subproblems contain just one constraint, and path-consistency [Montanari 1974], where subproblems are triangles (i.e., they are complete binary graphs over three variables). The concept of k-consistency [Freuder 1978], where subproblems contain all constraints among subsets of k chosen variables, generalizes them both. However, all subproblems are of the same size (which is k). In general, one may imagine several algorithms where instead the considered subparts of the SCSP have different sizes. This generalization of the k-consistency algorithms has been described in Montanari and Rossi [1991] for classical CSPs. Here we follow the same approach as in Montanari and Rossi [1991], but we extend the definitions and results presented there to the SCSP framework, and we show that all the properties still hold, provided that certain properties of the semiring are satisfied.

Applying a local consistency algorithm to a constraint problem means explicitating some implicit constraints, thus possibly discovering inconsistency at a local level. In classical CSPs, this is crucial, since local inconsistency implies global inconsistency. We will now show that such a property holds also for SCSPs.

DEFINITION 4.1 (LOCAL INCONSISTENCY). *Consider an SCSP problem $P = \langle C, con\rangle$. Then we say that P is locally inconsistent if there exists $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$.*

THEOREM 4.2 (NECESSITY OF LOCAL CONSISTENCY). Consider an SCSP problem P which is locally inconsistent. Then it is not consistent.

PROOF. We have to prove that $blevel(P) = \mathbf{0}$. We know that P is locally inconsistent. That is, there exists $C' \subseteq C$ such that $blevel(C') = \mathbf{0}$. By Theorem 3.18, we have that $blevel(P) \leq_S blevel(C')$; thus, $blevel(P) \leq_S \mathbf{0}$. Since $\mathbf{0}$ is the minimum in the ordering $\leq_S$, then we immediately have that $blevel(P)$ must be $\mathbf{0}$ as well. □

In the SCSP framework, we can be even more precise, and relate the best level of consistency of the whole problem (or, equivalently, of the set of all its

[6] See, for example, Freuder [1978; 1988], Kumar [1992], and Montanari and Rossi [1991].

constraints) to that of its subproblems, even though such level is not **0**. In fact, it is possible to prove that if a problem is α-consistent, then all its subproblems are β-consistent, where $\alpha \leq_S \beta$.

THEOREM 4.3. (LOCAL AND GLOBAL A-CONSISTENCY). *Consider a set of constraints C over CS, and any subset C' of C. If C is α-consistent, then C' is β-consistent, with $\alpha \leq_S \beta$.*

PROOF. The proof is similar to the previous one. If C is α-consistent, it means, by definition, that $blevel(C) = \alpha$. Now, if we take any subset C' of C, by Theorem 3.18 we have that $blevel(C) \leq_S blevel(C')$. Thus, $\alpha \leq_S blevel(C')$. □

In order to define the subproblems to be considered by a local consistency algorithm, we use the notion of local consistency rule. The application of such a rule consists of solving one of the chosen subproblems. To model this, we need the additional notion of *typed location*. Informally, a typed location is just a location l (as in ordinary store-based programming languages) which has a set of variables con as type, and thus can only be assigned a constraint $c = \langle def, con \rangle$ with the same type. In the following, we assume to have a location for every set of variables, and thus we identify a location with its type. Note that at any stage of the computation the store will contain a constraint problem with only a finite number of constraints, and thus the relevant locations will always be finitely many.

Definition 4.4 (*typed location*). A typed location l is a set of variables.

Definition 4.5 (*value of a location*). Given a CSP $P = \langle C, con \rangle$, the value $[l]_P$ of the location l in P is defined as the constraint $\langle def, l \rangle \in C$ if it exists, as $\langle \mathbf{1}, l \rangle$ otherwise. Given n locations $l_1, \ldots, l_n$, the value $[\{l_1, \ldots, l_n\}]_P$ of this set of locations in P is defined instead as the set of constraints $\{[l_1]_P, \ldots, [l_n]_P\}$.

Definition 4.6 (*assignment*). An assignment is a pair $l := c$ where $c = \langle def, l \rangle$. Given a CSP $P = \langle C, con \rangle$, the result of the assignment $l := c$ is the problem $[l := c](P)$ defined as:

$$[l := c](P) = \langle \{\langle def', con' \rangle \in C \mid con' \neq l\} \cup c, con \rangle.$$

Thus, an assignment $l := c$ is seen as a function from constraint problems to constraint problems, which modifies a given problem by changing just one constraint, the one with type l. The change consists in replacing such a constraint with c. If there is no constraint of type l, then constraint c is added to the given problem.

Definition 4.7 (*local consistency rule*). Given an $SCSP$ $P = \langle C, con \rangle$, a local consistency rule r for P is defined as $r = l \leftarrow L$, where L is a set of locations, l is a location, and $l \notin L$.

Applying r to P means assigning to location l the constraint obtained by solving the subproblem of P containing the constraints specified by the locations in $L \cup \{l\}$.

Definition 4.8 (*rule application*). Given a local consistency rule $r = l \leftarrow L$ and a constraint problem P, the result of applying r to P is

$$[l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l\rangle)](P).$$

Since a rule application is defined as a function from problems to problems, the application of a sequence S of rules to a problem is easily provided by function composition. Thus, we have that $[r_1; S](P) = [S]([r_1](P))$.

Notice for example that $[l \leftarrow \emptyset](P) = P$.

THEOREM 4.9 (EQUIVALENCE FOR RULES). Given an *SCSP* P and a rule r for P, we have that $P \equiv [r](P)$ if $\times$ is idempotent.

PROOF. Assume that $P = \langle C, con\rangle$, and let $r = l \leftarrow L$. Then we have $P' = [l \leftarrow L](P) = [l := Sol(\langle [L \cup \{l\}]_P, l\rangle)](P) = \langle \{\langle def', con'\rangle \in C \mid con' \neq l\} \cup Sol(\langle [L \cup \{l\}]_P, l\rangle), con\rangle$. Let now $C(r) = [L \cup \{l\}]_P$, and $C' = C - C(r)$. Then P contains the constraints in the set $C' \cup C(r)$, while P' contains the constraints in $C' \cup (C(r) - c) \cup ((\otimes C(r)) \Downarrow_l)$, where $c = \langle def, l\rangle = [l]_P$. In fact, by definition of solution, $Sol(\langle [L \cup \{l\}]_P, l\rangle)$ can be written as $(\otimes C(r)) \Downarrow_l$. Since the set C' is present in both P and P', we will not consider it, and we will instead prove that the constraint $c_{pre} = \otimes C(r)$ coincides with $c_{post} = (\otimes(C(r) - \{c\})) \otimes ((\otimes C(r)) \Downarrow_l)$. In fact, if this is so, then $P \equiv P'$.

First of all, it is easy to see that c_{pre} and c_{post} have the same type $\bigcup(L \cup \{l\})$. Let us now consider the definitions of such two constraints. Let us set $c_i = [l_i]_P = \langle def_i, l_i\rangle$ for all $l_i \in L$, and assume $L = \{l_1, \ldots, l_n\}$. Thus, $C(r) = \{c, c_1, \ldots, c_n\}$. We have that, taken any tuple t of length $|\bigcup(L \cup \{l\})|$, $def_{pre}(t) = def(t \downarrow_l) \times (\Pi_i\ def_i(t \downarrow_{l_i}))$, while $def_{post}(t) = (\Pi_i\ def_i(t \downarrow_{l_i})) \times {}_{t'|t' \downarrow_l = t \downarrow_l} (def(t' \downarrow_l) \times (\Pi_i\ def_i(t' \downarrow_{l_i})))$. Now, since the sum is done over all t' such that $t' \downarrow_l = t \downarrow_l$, we have that $def(t' \downarrow_l) = def(t \downarrow_l)$. Thus, $def(t' \downarrow_l)$ can be taken out from the sum since it appears the same in each factor. Thus, we have $def_{post}(t) = (\Pi_i\ def_i(t \downarrow_{l_i})) \times def(t \downarrow_l) \times {}_{t'|t' \downarrow_l = t \downarrow_l} (\Pi_i\ def_i(t' \downarrow_{l_i}))$. Consider now ${}_{t'|t' \downarrow_l = t \downarrow_l} (\Pi_i\ def_i(t' \downarrow_{l_i}))$: one of the t' such that $t' \downarrow_l = t \downarrow_l$ must be equal to t. Thus, the sum can be written also as $(\Pi_i\ def_i(t \downarrow_{l_i}) + {}_{t'|t' \downarrow_l = t \downarrow_l, t' \neq t} (\Pi_i\ def_i(t' \downarrow_{l_i}))$. Thus, we have $def_{post}(t) = (\Pi_i\ def_i(t \downarrow_{l_i})) \times def(t \downarrow_l) \times (\Pi_i\ def_i(t \downarrow_{l_i}) + {}_{t'|t' \downarrow_l = t \downarrow_l, t' \neq t} (\Pi_i\ def_i(t' \downarrow_{l_i})))$.

Let us now consider any two elements a and b of the semiring. Then it is easy to see that $a \times (a + b) = a$. In fact, by intensivity of $\times$ (see Theorem 2.5), we have that $a \times c \leq_S a$ for any c. Thus, if we choose $c = a + b$, $a \times (a + b) \leq_S a$. On the other hand, since $a + b$ is the lub of a and b, we have that $a + b \geq_S a$. Thus, by monotonicity of $\times$, we have $a \times (a + b) \geq_S a \times a$. Now, since we assume that $\times$ is idempotent, we have that $a \times a = a$, thus $a \times (a + b) \geq_S a$. Therefore, $a \times (a + b) = a$. This result can be used in our formula for $def_{post}(t)$, by setting $a = (\Pi_i\ def_i(t \downarrow_{l_i}))$ and $b = {}_{t'|t' \downarrow_l = t \downarrow_l, t' \neq t} (\Pi_i\ def_i(t' \downarrow_{l_i}))$. In fact, by replacing $a \times (a + b)$ with a, we get $def_{post}(t) = (\Pi_i\ def_i(t \downarrow_{l_i})) \times def(t \downarrow_l)$, which coincides with $def_{pre}(t)$. $\square$

Definition 4.10 (*stable problem*). Given an SCSP problem P and a set R of local consistency rules for P, P is said to be stable with respect to R if, for each $r \in R$, $[r](P) = P$.

A local consistency algorithm consists of the application of several rules to the same problem, until stability of the problem with respect to all the rules. At each

step of the algorithm, only one rule is applied. Thus, the rules will be applied in a certain order, which we call a strategy.

Definition 4.11 (*strategy*). Given a set R of local consistency rules for an SCSP, a strategy for R is an infinite sequence $S \in R^{\infty}$. A strategy S is fair if each rule of R occurs in S infinitely often.

Definition 4.12 (*local consistency algorithm*). Given an SCSP problem P, a set R of local consistency rules for P, and a fair strategy S for R, a local consistency algorithm applies to P the rules in R in the order given by S. Thus, if the strategy is $S = s_1 s_2 s_3 \cdots$, the resulting problem is

$$P' = [s_1; s_2; s_3; \cdots](P)$$

The algorithm stops when the current SCSP is stable with respect to R.

Note that this formulation of local consistency algorithms extends the usual one for k-consistency algorithms [Freuder 1978], which can be seen as local consistency algorithms where all rules in R are of the form $l \leftarrow L$, where $L = \{l_1, \ldots, l_n\}$ and $|\bigcup_{i=1,\ldots,n} l_i| = |l| = k - 1$. That is, exactly $k - 1$ variables are involved in the locations[7] in L.

When a local consistency algorithm terminates,[8] the result is a new problem which has the graph structure of the initial one plus possibly new arcs representing newly introduced constraints (we recall that constraints posing no restrictions on the involved variables are usually not represented in the graph structure), but where the definition of some of the constraints has been changed. More precisely, assume $R = \{r_1, \ldots, r_n\}$, and $r_i = l_i \leftarrow L_i$ for all $i = 1, \ldots, n$. Then the algorithm uses, among others, n typed locations l_i of type con_i. Assume also that each of such typed locations has value def_i when the algorithm terminates. Consider also the set of constraints $C(R)$ which have type con_i. That is, $C(R) = \{\langle def, con\rangle$ such that $con = con_i$ for some i between 1 and $n\}$. Informally, these are the constraints that may have been modified (via the typed location mechanism) by the algorithm. Then, if the initial SCSP is $P = \langle C, con\rangle$, the resulting SCSP is $P' = \text{local-cons}(P, R, S) = \langle C', con\rangle$, where $C' = C - C(R) \cup (\bigcup_{i=1,\ldots,n} \langle def_i, con_i\rangle)$.

In classical CSPs, any local consistency algorithm enjoys some important properties. We now will study these same properties in the SCSP framework, and point out the corresponding properties of the semiring operations which are sufficient for them to hold. The desired properties are as follows:

(1) any local consistency algorithm returns a problem which is equivalent to the given one;
(2) any local consistency algorithm terminates in a finite number of steps;
(3) the strategy, if fair, used in a local consistency algorithm does not influence the resulting problem.

THEOREM 4.13 (EQUIVALENCE). *Consider an SCSP P and the problem $P' = \text{local-cons}(P, R, S)$. Then $P \equiv P'$ if the multiplicative operation of the semiring* ($\times$) *is idempotent.*

[7] We recall that locations are just sets of variables.
[8] We will consider the termination issue later.

PROOF. By Definition 4.12, a local consistency algorithm is just a sequence of applications of local consistency rules. Since, by Theorem 4.9, we know that each rule application returns an equivalent problem, by induction we can conclude that the problem obtained at the end of a local consistency algorithm is equivalent to the given one. □

THEOREM 4.14 (TERMINATION). *Consider any SCSP* $P = \langle C, con\rangle$ *over the constraint system* $CS = \langle S, D, V\rangle$ *and the set* $AD = \bigcup_{\langle def,\ con\rangle \in C} R(def)$, *where* $R(def) = \{a \mid \quad t \text{ with } def(t) = a\}$. *Then the application of a local consistency algorithm to P terminates in a finite number of steps if AD is contained in a set I which is finite and such that* $+$ *and* $\times$ *are I-closed.*

PROOF. Each step of a local consistency algorithm may change the definition of one constraint by assigning a different value to some of its tuples. Such value is strictly worse (in terms of $\leq_S$) since $\times$ is intensive. Moreover, it can be a value which is not in AD but in $I - AD$. If the state of the computation consists of the definitions of all constraints, then at each step we get a strictly worse state (in terms of $\sqsubseteq_S$). The sequence of such computation states, until stability, has finite length, since by assumption I is finite and thus the value associated with each tuple of each constraint may be changed at most $|I|$ times. □

An interesting special case of the above theorem occurs when the chosen semiring has a finite domain A. In fact, in that case the hypotheses of the theorem hold with $I = A$.

COROLLARY 4.15. *Consider any SCSP problem P over the constraint system* $CS = \langle S, D, V\rangle$, *where* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, *and A is finite. Then the application of a local consistency algorithm to P terminates in a finite number of steps.*

We will now prove that, if $\times$ is idempotent, no matter which strategy is used during a local consistency algorithm, the result is always the same problem.

THEOREM 4.16 (ORDER-INDEPENDENCE). *Consider an SCSP problem P and two different applications of the same local consistency algorithm to P, producing respectively the SCSPs* $P' = \textit{local-cons}(P, R, S)$ *and* $P'' = \textit{local-cons}(P, R, S')$. *Then* $P' = P''$ *if the multiplicative operation of the semiring* $(\times)$ *is idempotent.*

PROOF. Each step of the local consistency algorithm, which applies one of the local consistency rules in R, say r, has been defined as the application of a function $[r]$ that takes an SCSP problem P and returns another problem $P' = [r](P)$, and that may change the definition of the constraint connecting the variables in l (if the applied rule is $r = l \leftarrow L$). Thus, the whole algorithm may be seen as the repetitive application of the functions $[r]$, for all $r \in R$, until no more change can be done. If we can prove that each $[r]$ is a closure operator [Davey and Priestly 1990], then classical results on chaotic iteration [Cousot 1977] allow us to conclude that the problem resulting from the whole algorithm does not depend on the order in which such functions are applied. Closure operators are just functions, which are idempotent, monotone, and intensive. We will now prove that each $[r]$ enjoys such properties. We remind that $[r]$ just combines a constraint with the combination of other constraints.

If $\times$ is idempotent, then $[r]$ is idempotent as well, that is, $[r]([r](P)) = [r](P)$ for any P. In fact, combining the same constraint more than once does not

change the problem by idempotency of $\times$. Also, the monotonicity of $\times$ (see Theorem 2.4) implies that of $[r]$, that is, $P \sqsubseteq P'$ implies $[r](P) \sqsubseteq [r](P')$. Note that, although $\sqsubseteq$ has been defined only between constraints, here we use it among constraint problems, meaning that such a relationship holds among each pair of corresponding constraints in P and P'. Finally, the intensivity of $\times$ implies that $[r]$ is intensive as well, that is, $[r](P) \sqsubseteq P$. In fact, P and $[r](P)$ just differ in one constraint, which, in $[r](P)$, is just the corresponding constraint of P combined with some other constraints. □

Note that the above theorems require that $\times$ is idempotent for a local consistency algorithm to be meaningful. However, there is no requirement over the nature of $\leq_S$. More precisely, $\leq_S$ can be partial. This means that the semiring operations + and $\times$ can be different from max and min respectively (see note at the end of Section 2).

Note also that, by definition of rule application, constraint definitions are changed by a local consistency algorithm in a very specific way. In fact, even in the general case in which the ordering $\leq_S$ is partial, the new values assigned to tuples are always smaller than or equal to the old ones in the partial order. In other words, local consistency algorithms do not "jump" in the partially ordered structure from one value to another one which is unrelated to the first one. More precisely, the following theorem holds.

THEOREM 4.17 (LOCAL CONSISTENCY AND PARTIAL ORDERS). *Given an SCSP problem P, consider any value v assigned to a tuple in a constraint of such problem. Also, given any set R of local consistency rules and strategy S, consider $P' =$ local-cons(P, R, S), and the value v' assigned to the same tuple of the same constraint in P'. Then we have that $v' \leq_S v$.*

PROOF. By definition of rule application (see Definition 4.8), the formula defining the new value associated to a tuple of a constraint can be assimilated to $(v \times a) + (v \times b)$, where v is the old value for that tuple and a and b are combinations of values for tuples of other constraints in the graph of the rule. Now, we have $(v \times a) + (v \times b) = v + (a \times b)$ by distributivity of + over $\times$ (see Theorem 2.10). Also, $(v \times c) \leq_S v$ for any c by intensivity of $\times$ (see Theorem 2.5), thus $v + (a \times b) \leq_S v$. □

One could imagine other generalizations with the same desired properties as the ones proved above (i.e., termination, equivalence, and order-independence). For example, one could design an algorithm that avoids (or compensates) the effect of cumulation coming from a nonidempotent $\times$. Or also, one could generalize the equivalence property, by allowing the algorithm to return a non-equivalent problem (but however being in a certain relationship with the original problem). Finally, the order-independence property states that any order is fine, as far as the resulting problem is concerned. Another desirable property, related to this, could be the existence of an ordering which makes the algorithm belong to a specific complexity class.

5. *Dynamic Programming*

Dynamic programming [Bellman 1997; Bellman and Dreyfus 1962] can be used to solve a problem by solving some subproblems of it and then combining their

solutions to obtain the solution of the whole problem (see, e.g., Gnesi et al. [1981] for its use in graphs and Montanari and Rossi [1986] for its adoption in classical CSPs). In the SCSP framework, a suitably instantiated version of dynamic programming can be fruitfully used as well: at each step, a subset of constraints is chosen and solved, and its solution (which, we recall, is a constraint) replaces the whole subset of constraints.

Before defining the algorithm, we need to describe in a tree-like way the SCSP to be solved, so that the algorithm can then follow the tree structure in a bottom-up way.

Definition 5.1 (*parsing tree*). Given an SCSP $P = \langle C, con \rangle$, a parsing tree of P is a sequence of local consistency rules $S = r_1; \ldots; r_n$, where $r_i = (l_i \leftarrow L_i)$. Let

—$L = \{l_i, i = 1, \ldots, n\} \cup \bigcup_{i=1,\ldots,n} L_i$ (i.e., L is the set of all locations occurring in all rules in S);

—$l_i \prec l$ iff $l \in L_i$ (i.e., rule r_i "generates" location l);

—$l_i \prec v$ iff $v \notin l_i$ and $v \in l'$ with $l' \in L_i$ (i.e., rule r_i "generates" variable v);

—for any x location or variable, $prec(x) = \{l | l \prec x\}$.

Then we require that

—(*tree-like*) for any x variable or location, if $x = l_n$ or $x \in l_n$, then $prec(x) = \emptyset$, else $prec(x)$ is a singleton;

—(*cover*) $con = l_n$, and $\langle def, con' \rangle \in C$ implies $con' \in L$;

—(*bottom-up parsing*) $l_i \prec l_j$ implies $j < i$.

In words, the above definition provides a sequence of rules that cover the whole problem and are connected among them as a tree. Moreover, the sequence gives a bottom-up visit of the tree. It is now easy to define a dynamic programming algorithm where, at each step, one rule is processed, according to the given sequence.

Definition 5.2 (*dynamic programming algorithm*). Given an *SCSP* P, consider a parsing tree $S = r_1; \cdots; r_n$ of P. Then compute $[S](P)$.

We will now prove that the value of location l_n when the algorithm terminates coincides with the solution of the given problem P.

THEOREM 5.3 (SOLUTION). *Given an SCSP $P = \langle C, l \rangle$ and a parsing tree S for P, we have that $Sol(P) = [l]_{[S](P)}$.*

PROOF. We will prove the statement of the theorem by induction on the length of the parsing tree. For the base case, consider a parsing tree with just one rule, that is, $l \leftarrow L$. It is easy to see that $[l]_{[r](P)}$ is the solution of P. In fact, applying this rule means exactly solving the whole problem. Assume now that the statement holds for all parsings with $n - 1$ rules, and consider a parsing S_1 with n rules $r_1, \ldots, r_n$, where $r_i = l_i \leftarrow L_i$ for all $i = 1, \ldots, n$. Consider now the first rule of this parsing, that is, $r_1 = l_1 \leftarrow L_1$, and take another rule r_i such that $l_1 = l_i$ or $l_1 \in L_i$, and there is no other rule r_j with $j < i$ and such that $l_1 = l_j$ or $l_1 \in L_j$. That is, r_i is the first rule after r_1 which has l_1 either in its left-hand side or in its right-hand side. Note that such rule must exist by definition of

parsing tree. In fact, either $l_1 = l_n$, and in this case l_1 must appear at least in the left hand side on r_n, or $l_1 \neq l_n$, in which case it must be generated by one rule (r_i). We will consider now the two separate cases: that $l_1 = l_i$ and that $l_1 \in L_i$.

Assume that $l_1 \in L_i$. Consider then the sequence $S_2 = r_2, \ldots, r_{i-1}, r_1, r_i, \ldots, r_n$. It is easy to see that this is still a parsing for the given problem. Moreover, the constraint of type l_n resulting after applying the rules in S_2 coincides with that obtained after applying the rules in S_1. In fact, since l_1 can be generated by just one rule (by definition of parsing), all rules in $\{r_2, \ldots, r_{i-1}\}$ cannot change the definition of l_1 and thus applying r_1 before or after them cannot change the resulting constraint. Consider now another sequence S_3 of $n - 1$ rules $r_2, \ldots, r_{i-1}, r'_i, r_{i+1}, \ldots, r_n$, where $r'_i = l_i \leftarrow L_i \cup L_1$. It is easy to see that this sequence of rules is again a parsing tree for P, thus by induction hypothesis the constraint with type l_n after the application of S_2 is the solution of the problem. Now we have to show that such a constraint coincides with that obtained at location l_n after applying sequence S_2 (which we already know to coincide with that obtained at location l_n after sequence S_1). What makes S_2 and S_3 differ is the fact that rule r_1 of S_2 has been "merged" with rule r_i to give rule r'_i in S_3. More precisely, the application of rule r'_i combines all constraints specified by $L_1 \cup L_i$, while the application of first rule r_1 and then rule r_i combines first the constraint specified by L_1 and then combines the resulting constraint with those constraints specified by L_i. We will show that these two methods yield the same final constraint.

In this proof, for ease of readability, we will write l instead of $[l]_P$. On one side (rule r'_i) we have the constraint $(\otimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i}$, and on the other side (rule r_1 and then r_i) we have the constraint $(\otimes((L_i - \{l_1\}) \cup \{l_i\} \cup \{(\otimes(L_1 \cup \{l_1\})) \Downarrow_{l_1}\})) \Downarrow_{l_i}$. Thus, we have to prove that:

$$(\otimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i} = (\otimes((L_i - \{l_1\}) \cup \{l_i\} \cup \{(\otimes L_1 \cup \{l_1\}) \Downarrow_{l_1}\})) \Downarrow_{l_i}.$$

We will start from the left-hand side of the formula and try to reach the right-hand side. We have:

$(\otimes(L_i \cup L_1 \cup \{l_i\})) \Downarrow_{l_i} =$

{by separating $(L_i \cup \{l_i\}) - \{l_1\}$ and $L_1 \cup \{l_1\}$ which are disjoint}

$((\otimes((L_i \cup \{l_i\}) - \{l_1\}))\otimes(\otimes(L_1 \cup \{l_1\}))) \Downarrow_{l_i} =$

{by Theorem 3.8, and letting $V_i = \bigcup(L_i \cup \{l_i\})$, that is the set of all variables involved in rule r_i}

$((\otimes((L_i \cup \{l_i\}) - \{l_1\}))\otimes(\otimes(L_1 \cup \{l_1\}))) \Downarrow_{V_i} \Downarrow_{l_i} =$

{by Theorem 3.9, since the variables not in V_i are not involved in $L_i \cup \{l_i\} - \{l_1\}$}

$((\otimes((L_i \cup \{l_i\}) - \{l_1\}))\otimes(\otimes(L_1 \cup \{l_1\}) \Downarrow_{V_i})) \Downarrow_{l_i} =$

{by Proposition 3.7, since $V_i \cap V_1 = l_1$, where $V_1 = (\bigcup L_1) \cup l_1$}

$((\otimes((L_i \cup \{l_i\}) - \{l_1\}))\otimes(\otimes(L_1 \cup \{l_1\})) \Downarrow_{l_1}) \Downarrow_{l_i}$

which coincides with the right hand side of the formula.

Let us now consider the second case, in which $l_1 = l_i$. Then consider the sequence $S_2 = r_2, \ldots, r_{i-1}, r_1, r_i, \ldots, r_n$. With a reasoning similar to above, it is easy to see that S_2 is still a parsing tree for P and that the constraint of type l_n obtained via S_2 is the same as the one obtained via S_1, since by assumption location l_1 does not appear in rules $r_2, \ldots, r_{i-1}$. Consider now $S_3 = r_2, \ldots, r_{i-1}, r'_i, r_{i+1}, \ldots, r_n$ where $r'_i = l_1 \leftarrow L_i \cup L_1$. Again, S_3 is a parsing tree for P. Moreover, it has $n - 1$ rules, thus the constraint of type l_n after the application of S_3 coincides with the solution of P. Now we will show that this is the same as that obtained after the application of S_2. More precisely, we have to show that

$$(\otimes (L_1 \cup L_i \cup \{l_1\})) \Downarrow_{l_1} = (\otimes (L_i \cup (\otimes (L_1 \cup \{l_1\})) \Downarrow_{l_1}) \Downarrow_{l_1}.$$

This can be easily proved by using a reasoning similar to that of the first case. □

Note that this dynamic programming algorithm can be applied to any instance of the SCSP framework, even when $\times$ is not idempotent (and thus the local consistency algorithms cannot safely be applied).

Obviously any SCSP has a parsing tree[9] but not all classes of SCSPs have *convenient* parsing trees for each SCSP of the class (see Martelli and Montanari [1972], where it is shown that the class of rectangular lattices does not have such a property). Here for convenience we mean that the size of each rule is smaller than some bound N, which is fixed and the same for the whole class of considered SCSPs. In such a case, each step of the algorithm solves an SCSP with bounded size. Thus the complexity of this step may be exponential in the size of such a SCSP, but constant with respect to the size of the overall problem. Therefore, the complexity of the algorithm is linear in the number of rules of the parsing tree, which is linear in the number of variables of the problem. Thus, the overall algorithm is linear in the number of variables of the problem. When applied to standard CSPs, this algorithm reduces to the *perfect relaxation* algorithm of Montanari and Rossi [1986; 1991].

Definition 5.4 (*N-bounded parsing tree*). Given an SCSP $P = \langle C, con \rangle$ and an integer N, consider a parsing tree $S = \{l_1 \leftarrow L_1; \cdots; l_n \leftarrow L_n\}$ for P. Then S is N-bounded for P if

(1) for all $i = 1, \ldots, n$, $|l_i \cup \bigcup_{l \in L_i} l| \leq N$ (i.e., the number of variables of a rule is bounded by N);
(2) for all $i = 1, \ldots, n$, there is a variable v such that $l_i \prec v$ (i.e., each rule generates at least a variable);
(3) $V_P = \bigcup_{\langle def, con' \rangle \in C} con' = \bigcup_{l \in L} l$ (i.e., all the variables occurring in the rules are present in the constraint problem P).

[9] Just take the parsing tree with just one rule.

THEOREM 5.5 (LINEAR ALGORITHM FOR CLASSES WITH CONVENIENT PARSING). *Given an integer N, consider the class of all SCSPs that have an N-bounded parsing tree. Consider now any SCSP P of this class, and let n be the number of its variables. Then in the worst case P can be solved in time $O(n)$.*

PROOF. Just apply the dynamic programming algorithm that follows an N-bounded parsing tree for P, which exists by assumption. Each step of the algorithm applies one of the rules. This is in the worst case $O(|D|^N \times 2^N)$, where D is the domain of each variable and N is an upper bound to the number of variables of the rules. In fact, the number of tuples of values for the variables of the rule is exponential in the number of variables, which by assumption is bounded by N, and for each tuple one has to check a number of constraints equal to the number of locations, which again can be exponential in the number of variables of the rule. Thus, the worst-case time complexity of a rule application is constant, since both N and D are fixed. There are as many steps as rules in the parsing tree. Since each rule by assumption generates at least one variable, the number of rules is bounded by the number of variables n of the whole problem. Thus, in the worst case, the number of steps of the algorithm is $O(n)$. □

Conditions (2) and (3) in Definition 5.4 are not restrictive for Theorem 5.5. In fact, if we have a parsing for some problem that satisfies condition 1 but where some rule, say $r = l \leftarrow L$, does not eliminate any variable, then it is always possible to obtain another parsing for the same problem that still satisfies condition (1) but where each rule eliminates at least a variable: just eliminate rule r and replace every occurrence of l in the other rules with L, and do similarly for all rules which do not eliminate variables. If instead there are rules containing variables that do not appear in the problem, then one can always obtain another parsing where such variables are not there: just remove all locations involving such "ghost" variables. By doing this, we still have the cover property of the parsing, since a location involving one or more ghost variables cannot correspond to any constraint of the given problem.

Theorem 5.5 states that in some cases there is an algorithm which is $O(n)$. One could wonder whether n is a good measure of the size of the given SCSP, and consider instead the number m of its constraints as more significative. In fact, in general the number of constraints may be much larger than the number of variables (e.g., in binary CSPs, m is $O(n^2)$). However, for the classes of SCSPs that have an N-bounded parsing tree, it is possible to show that m is $O(n)$. In fact, consider an N-bounded parsing tree for a problem P of the class. Then, each rule may have at most N variables, thus it will contain at most 2^N locations, and there are at most n rules. Thus the total number of locations in the parsing is $n \times 2^N$. Also, since a parsing tree for P covers P, the number of locations is either the same or greater than the number of constraints in P. Thus $m \leq (n \times 2^N)$. Therefore, we have that m is $O(n)$.

In order to find a parsing tree for a given SCSP, one could adapt the studies on the *secondary problem* in dynamic programming (see, e.g., Bertelè and Brioschi [1972]). However, in some cases the problem is generated in a way that the parsing tree is already explicit, as in the case of constraint logic programming (CLP) languages [Jaffar and Lassez 1987]. In fact, during the execution of a CLP program, the use of the clauses of the program builds a constraint problem with

a parsing tree where each node of the tree directly corresponds to one of the used clauses.

A characterization of what dynamic programming is and when it can be applied that is similar to the one given in this section can be found in Shenoy [1991; 1994]. There, *valuation-based systems* are defined as systems based on variables, valuations, and two operations, called *combination* and *marginalization*. These two operations are very related, respectively, to our notions of *combination* and *projection*. In valuation-based systems, three axioms are required for the correct application of a dynamic programming algorithm. These three axioms are indeed satisfied by our framework as well. In fact, they correspond, respectively, to (1) commutativity and associativity of $\otimes$ (see comment after Definition 3.5), (2) Theorem 3.8, and (3) Theorem 3.9. Note that the proof of Theorem 5.3 relies just on these three properties. In fact, our dynamic programming algorithm, defined in Definition 5.2, can be seen as an extension of the *fusion algorithm* defined in Shenoy [1991; 1994], where the extension consists of the fact that more than one variable at a time can be eliminated. In fact, in our algorithm all variables in the right-hand side of a rule are considered at a time.

Moreover, a related algorithm which solves a constraint problem with a tree-like shape in a bottom-up way, as in Definition 5.2, has been described in Dechter et al. [1990] for optimization problems and in Dechter and Dechter [1988] for belief maintenance. In those papers, the idea is to consider either the constraint graph, if it is acyclic, or the dual graph of a constraint problem (where nodes are constraints and arcs are associated with variables shared among constraints), and to use techniques like cycle-cutset [Dechter and Pearl 1988a] or tree-clustering [Dechter and Pearl 1988b] to provide such a dual graph with a tree-like shape. However, in Dechter et al. [1990] constraints are combined via the usual *and* operator, and the value associated to each tuple is computed in a completely independent way, via a given utility function. Thus, apart from the tree-structure, our approach and that in Dechter et al. [1990] are very different, since we also generalize the way constraints are combined, via a general notion of combination and projection of which *and* and *or* are just instances.

Our notion of parsing tree is more general than that of *hinge-trees* presented in Gyssens et al. [1994]. In fact, we do not assume anything about the structure of each node of the tree, which may be a generic graph. On the contrary, in Gyssens et al. [1994], they consider only nodes which cannot be further decomposed into hinge-trees.

6. *Instances of the Framework*

We will now show how several known, and also new, frameworks for constraint solving may be seen as instances of the SCSP framework. More precisely, each of such frameworks corresponds to the choice of a specific constraint system (and thus of a semiring). This means that we can immediately know whether one can inherit the properties of the general framework by just looking at the properties of the operations of the chosen semiring, and by referring to the theorems in the previous sections. This is interesting for known constraint solving schemes, because it puts them into a single unifying framework and it justifies in a formal way many informally taken choices, but it is especially significant for new

schemes, for which one does not need to prove all the properties that it has (or not) from scratch.

Note that for the purposes of this paper, where we consider only finite domain constraint solving, the constraint systems of different instances differ only in the choice of the semiring. Therefore, in the following we will only specify the semiring that has to be chosen to obtain a particular instance of the SCSP framework.

6.1. CLASSICAL CSPs. A classical CSP problem [Montanari 1974; Mackworth 1992] is just a set of variables and constraints, where each constraint specifies the tuples that are allowed for the involved variables. Assuming the presence of a subset of distinguished variables, the solution of a CSP consists of a set of tuples which represent the assignments of the distinguished variables which can be extended to total assignments (for all the values) while satisfying all the constraints.

Since constraints in CSPs are crisp, that is, they either allow a tuple or not, we can model them via a semiring domain with only two values, say 1 and 0: allowed tuples will have associated the value 1, and not allowed ones the value 0. Moreover, in CSPs, constraint combination is achieved via a join operation among allowed tuple sets. This can be modeled here by assuming the multiplicative operation to be the logical *and* (and interpreting 1 as true and 0 as false). Finally, to model the projection over some of the variables (e.g., the distinguished ones), as the k-tuples (assuming we project over k variables) for which there exists a consistent extension to an n-tuple, it is enough to assume the additive operation to be the logical *or*. Therefore, a CSP is just an SCSP where the semiring is

$$S_{CSP} = \langle \{0, 1\}, \vee, \wedge, 0, 1 \rangle.$$

THEOREM 6.1.1. $S_{CSP} = \langle \{0, 1\}, \vee, \wedge, 0, 1 \rangle$ *is a c-semiring*.

PROOF. It is easy to see that it is a semiring. Thus, we will only prove the additional properties that are needed in a c-semiring:

—the additive operation must be idempotent; it is trivially true since the additive operation is $\vee$, and $a \vee a = a$ for any $a \in \{0, 1\}$;

—the multiplicative operation must be commutative; trivially from the definition of $\wedge$;

—1 is the absorbing element of the additive operation, that is $1 \vee a = 1$, which is trivially true. □

The ordering $\leq_S$ here reduces to $0 \leq_S 1$.

Example. Consider the CSP P in Figure 2(a), where the distinguished variables are marked and the constraints are arcs labeled by the allowed tuples. The solution of such a CSP is the singleton set containing tuple $\langle a, a \rangle$ (which means $x = a$ and $y = a$). This CSP can be seen as an SCSP P' over the semiring S_{CSP} in Figure 2(b). It can then be shown, by using the semiring operations, that the solution of P and that of P' coincide (modulo the semiring domain value). To compute the solution of P', we have to assign a value to each tuple for the distinguished variables. This is done by first considering all 3-tuples and assigning

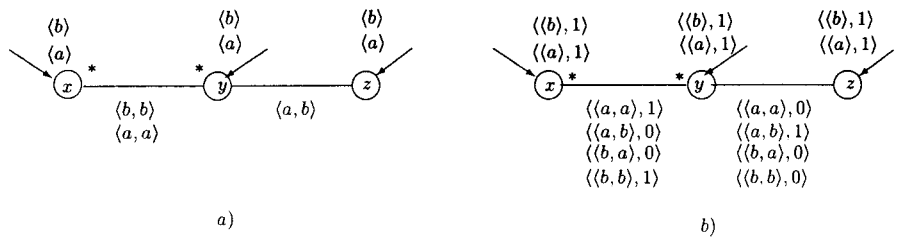

FIG. 2. A CSP and the corresponding SCSP (S_{CSP}).

a value to them, and then by projecting (via the $\vee$ operation) such values over the distinguished variables. To assign a value to a 3-tuple, one has to combine (via the $\wedge$ operation) the values of all the subtuples corresponding to subsets of variables which are connected by some constraint. In Figure 3, for each 3-tuple we have written the values of its subtuples: subtuples are pointed out by underlining them, and subtuples of one element only are not indicated, since they all have value 1. Then the value for the tuple itself can be seen to the right as a combination of the subtuple values (again, for simplicity of the picture one-value subtuples are not considered since they always have value 1), and the value for each 2-tuple which is a projection of some 3-tuples onto the variables x and y can be seen to the left, as a combination of the values of the two tuples which have the same projection over x and y. It is easy to see that a 3-tuple gets the value 1 only when it satisfies all the constraints (due to the definition of $\wedge$), and that a 2-tuple gets a value 1 when there exists a possible extension to a 3-tuple whose value is 1 (due to the definition of $\vee$). From Figure 3, one can see that the only group of tuples that gets value 1 is the first one, thus the solution of the CSP is the projection of this group of tuples onto x and y, that is, $x = a$ and $y = a$.

It is easy to see, also from the example above, that the chosen c-semiring allows us to faithfully represent any given CSP. That is, taken the given CSP P and the SCSP problem P' obtained by applying a suitable transformation to P, the solutions of P and P' are the same modulo, a similar transformation.

Let us now consider the application of a local consistency algorithm to CSPs. As predicted, we will show that all the classical properties hold. First, $\wedge$ is idempotent. Thus, by Theorem 4.13, the problem returned by a local consistency algorithm is equivalent to the given one, and from Theorem 4.16, the used strategy does not matter. Also, since the domain of the semiring is finite, by Corollary 4.15 any local consistency algorithm terminates in a finite number of steps.

Note that in this instance the dynamic programming algorithm described in Section 5 coincides with the perfect relaxation algorithm proposed in Montanari and Rossi [1986].

It is possible to check that an alternative way to represent CSPs in the SCSP framework is by using the following c-semiring: $\langle \wp(\{a\}), \cup, \cap, \emptyset, \{a\}\rangle$, where $\wp(S)$ is the powerset of a set S, and a is any value.

6.2. FUZZY CSPs. Fuzzy CSPs (FCSPs) [Rosenfeld et al. 1976; Dubois et al. 1993; Ruttkay 1994] extend the notion of classical CSPs by allowing non-crisp constraints, that is, constraints which associate a preference level with each tuple of values. Such level is always between 0 and 1, where 1 represents the best value

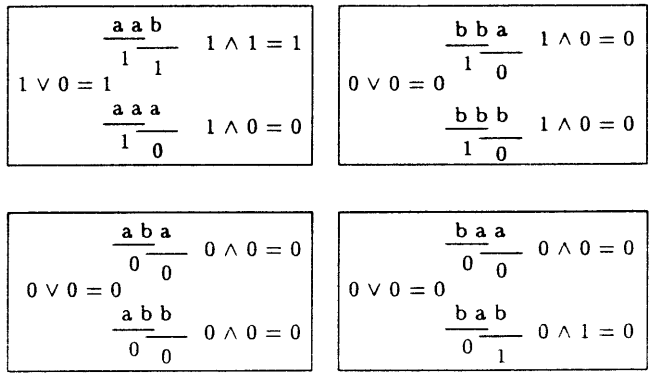

FIG. 3. Combination and projection in classical CSPs.

(i.e., the tuple is allowed) and 0 the worst one (i.e., the tuple is not allowed). The solution of a fuzzy CSP is then defined as the set of tuples of values (for all the variables) which have the maximal value. The way they associate a value with an n-tuple is by minimizing the values of all its subtuples. The reason for such a max-min framework relies on the attempt to maximize the value of the least preferred tuple.

Fuzzy CSPs are already a very significant extension of CSPs. In fact, they are able to model partial constraint satisfaction [Freuder and Wallace 1992], so to get a solution even when the problem is overconstrained, and also prioritized constraints, that is, constraints with different levels of importance [Borning et al. 1989].

It is easy to see that fuzzy CSPs can be modeled in our framework by choosing the semiring

$$S_{FCSP} = \langle \{x | x \in [0, 1]\}, max, min, 0, 1 \rangle.$$

THEOREM 6.2.1. *S_{FCSP} is a c-semiring.*

PROOF. It is easy to see that it is a semiring. Thus, we will only prove the additional properties that are needed in a c-semiring:

—the additive operation must be idempotent; this is trivially true since the additive operation is *max*, and $max(a, a) = a$ for any a;

—the multiplicative operation must be commutative; trivially from the definition of *min*;

—1 is the absorbing element of the additive operation, that is $max(1, a) = 1$, which is true since $0 \leq a \leq 1$. □

The ordering $\leq_S$ here reduces to the $\leq$ ordering on reals. It is also easy to see that any FCSP P can be rewritten as an SCSP P' over the semiring S_{FCSP} such that $sol(P) = sol(P')$ (modulo a suitable transformation between the problems and the solutions).

Fuzzy CSPs are much closer to our framework than classical CSPs, since they already introduce the notion of preference levels. This means that they have to generalize the notions of constraint combination and projection as well, so as to

get the appropriate definition of solution. However, we are obviously much more general in that we do not make any assumption on the semiring operations.

Let us now consider the properties that hold in the FCSP framework. The multiplicative operation (i.e., min) is idempotent. Thus, local consistency algorithms on FCSPs do not change the solution of the given problem (by Theorem 4.13), and do not care about the strategy (by Theorem 4.16). Moreover, min is AD-closed for any finite subset AD of [0, 1]. Thus, by Theorem 4.14, any local consistency algorithm on FCSPs terminates.

Thus, FCSPs, although providing a significant extension to classical CSPs, can exploit the same kind of algorithms. Their complexity will of course be different due to the presence of more than two levels of preference. However, it can be proved that if the actually used levels of preference are p, then the complexity of a local consistency algorithm is just $O(p)$ times greater than that of the corresponding algorithm over CSPs (as discussed also above and in Dubois et al. [1993].

An implementation of arc-consistency, suitably adapted to be used over fuzzy CSPs, is given in Schiex [1992]. However, no formal properties of its behavior are proved there. Thus, our result can be seen as a formal justification of Schiex [1992].

6.3. Probabilistic CSPs. Probabilistic CSPs (Prob-CSPs) [Fargier and Lang 1993] have been introduced to model those situations where each constraint c has a certain probability $p(c)$, independent from the probability of the other constraints, to be part of the given real problem.[10] This allows one to reason also about problems that are only partially known. The probability levels on constraints gives then, to each instantiation of all the variables, a probability that it is a solution of the real problem. This is done by first associating with each subset of constraints the probability that it is in the real problem (by multiplying the probabilities of the involved constraints), and then by summing up all the probabilities of the subsets of constraints where the considered instantiation is a solution. Alternatively, the probability associated with an n-tuple t can also be seen as the probability that all constraints that t violates are indeed in the real problem. This is just the product of all $1 - p(c)$ for all c violated by t. Finally, the aim is to get those instantiations with the maximum probability.

The relationship between Prob-CSPs and SCSPs is complicated by the fact that Prob-CSPs contain crisp constraints with probability levels, while SCSPs contain noncrisp constraints. That is, we associate values with tuples, and not to constraints. However, it is still possible to model Prob-CSPs, by using a transformation which is similar to that proposed in Dubois et al. [1993] to model prioritized constraints via soft constraints in the FCSP framework. More precisely, we assign probabilities to tuples instead of constraints. Consider any constraint c with probability $p(c)$, and let t be any tuple of values for the variables involved in c. Then $p(t) = 1$ if t is allowed by c, otherwise $p(t) = 1 - p(c)$. The reasons for such a choice are as follows: if a tuple is allowed by c and c is in the real problem, then t is allowed in the real problem; this happens with probability $p(c)$; if instead c is not in the real problem, then t is still allowed in

[10] Actually, the probability is not of the constraint, but of the situation which corresponds to the constraint: saying that c has probability p means that the situation corresponding to c has probability p of occurring in the real-life problem.

the real problem, and this happens with probability $1 - p(c)$. Thus, t is allowed in the real problem with probability $p(c) + 1 - p(c) = 1$. Consider instead a tuple t which is not allowed by c. Then it will be allowed in the real problem only if c is not present; this happens with probability $1 - p(c)$.

To give the appropriate value to an n-tuple t, given the values of all the smaller k-tuples, with $k \leq n$ and which are subtuples of t (one for each constraint), we just perform the product of the values of such subtuples. By the way values have been assigned to tuples in constraints, this coincides with the product of all $1 - p(c)$ for all c violated by t. In fact, if a subtuple violates c, then by construction its value is $1 - p(c)$; if instead a subtuple satisfies c, then its value is 1. Since 1 is the unit element of $\times$, we have that $1 \times a = a$ for each a. Thus, we get $\Pi(1 - p(c))$ for all c that t violates.

As a result, the semiring corresponding to the Prob-CSP framework is

$$S_{prob} = \langle \{x | x \in [0, 1]\}, max, \times, 0, 1 \rangle.$$

THEOREM 6.3.1. *S_{prob} is a c-semiring.*

PROOF. It is a semiring. To be also a c-semiring, we need the following:

—the additive operation must be idempotent; this is trivially true since the additive operation is *max*, and $max(a, a) = a$ for any a;

—the multiplicative operation must be commutative; trivially from the definition of $\times$;

—1 is the absorbing element of the additive operation, that is $max(1, a) = 1$, which is true since $0 \leq a \leq 1$. □

The associated ordering $\leq_S$ here reduces to $\leq$ over reals.

It is easy to see that any Prob-CSP P can be rewritten as an SCSP (S_{prob}) P' such that $sol(P) = sol(P')$ (modulo a suitable transformation between the problems and the solutions). Note that the fact that P' is α-consistent means that in P there exists an n-tuple which has probability α to be a solution of the real problem.

The multiplicative operation of S_{prob} (i.e., $\times$) is not idempotent. Thus, the result of Theorem 4.13 cannot be applied to Prob-CSPs. That is, by applying a local consistency algorithm one is not guaranteed to obtain an equivalent problem. Theorem 4.16 cannot be applied as well. Thus, the strategy used by a local consistency algorithm may matter. Also, $\times$ is not closed on any finite superset of any subset of [0, 1]. Thus the result of Theorem 4.14 cannot be applied to Prob-CSPs. That is, we cannot be sure that a local consistency algorithm terminates.

As a result of these observations, local consistency algorithms may make no sense in the Prob-CSP framework. Thus, we are not sure that their application has the desired properties (termination, equivalence, and order-independence). However, the fact that we are dealing with a c-semiring implies that, at least, we can apply Theorem 4.3: if a Prob-CSP problem has a tuple with probability α to be a solution of the real problem, then any subproblem has a tuple with probability at least α to be a solution of a subproblem of the real problem. This can be fruitfully used when searching for the best solution. In fact, if one employs a branch-and-bound search algorithm, and in one branch we find a partial

instantiation with probability smaller than α, then we can be sure that such a branch will never bring to a global instantiation with probability α. Thus, if a global instantiation with such a probability has already been found, we can cut this branch. Similar (and possibly better) versions of branch-and-bound for non-standard CSPs may be found in Freuder and Wallace [1992] and Schiex et al. [1995].

6.4. WEIGHTED CSPs. Whereas fuzzy CSPs associate a level of preference with each tuple in each constraint, in weighted CSPs (WCSPs) tuples come with an associated cost. This allows one to model optimization problems where the goal is to minimize the total cost (time, space, number of resources, . . .) of the proposed solution. Therefore, in WCSPs the cost function is defined by summing up the costs of all constraints (intended as the cost of the chosen tuple for each constraint). Thus, the goal is to find the n-tuples (where n is the number of all the variables) which minimize the total sum of the costs of their subtuples (one for each constraint).

Another way to understand the difference between WCSPs and FCSPs is that FCSPs have an egalitarianistic approach to optimization problems (i.e., they aim at maximizing the overall level of consistency while balancing the levels of all constraints), while WCSPs have an utilitarianistic approach (i.e., they aim at getting the minimum cost globally, even though some constraints may be neglected and thus present a big cost) [Moulin 1988]. We believe that both approaches present advantages and drawbacks, and thus each of them may be preferred to the other one, depending on the real-life situation to be modeled.

According to the informal description of WCSPs given above, the associated semiring is

$$S_{WCSP} = \langle \mathcal{R}^+ U\{+\infty\}, \min, +, +\infty, 0\rangle.$$

THEOREM 6.4.1. *S_{WCSP} is a c-semiring.*

PROOF. It is easy to see that it is a semiring. To be a c-semiring, we need:

—the additive operation must be idempotent; this is trivially true since the additive operation is *min*, and $min(a, a) = a$ for any a;

—the multiplicative operation must be commutative; trivially from the definition of $+$;

—0 is the absorbing element of the additive operation, that is $min(0, a) = 0$, which is true since $a \geq 0$. □

The associated ordering $\leq_S$ reduces here to $\geq$ over the reals. This means that a value is preferred to another one if it is smaller.

The multiplicative operation of S_{WCSP} (i.e., $+$) is not idempotent. Thus, the results of Section 4 on local consistency algorithms cannot be applies to WCSPs. However, as in Prob-CSPs, the fact that we are dealing with a c-semiring implies that, at least, we can apply Theorem 4.3: if a WCSP problem has a best solution with cost α, then the best solution of any subproblem has a cost smaller than α. This can be fruitfully used when searching for the best solution in a branch-and-bound search algorithm.

Note that the same properties hold also for the semirings $\langle \mathcal{Q}^+ U\{+\infty\}, min, +, +\infty, 0\rangle$ and $\langle \mathcal{Z}^+ U\{+\infty\}, min, +, +\infty, 0\rangle$ (which can be proved to be c-semirings).

6.5. EGALITARIANISM AND UTILITARIANISM. As noted above, the FCSP and the WCSP systems can be seen as two different approaches to give a meaning to the notion of optimization. The two models correspond in fact to two definitions of social welfare in utility theory [Moulin 1988]: *egalitarianism*, which maximizes the minimal individual utility, and *utilitarianism*, which maximizes the sum of the individual utilities. FCSPs are based on the egalitarian approach, while WCSPs are based on utilitarianism.

In this section, we will show how our framework allows also for the combination of these two approaches. In fact, we will construct an instance of the SCSP framework, where the two approaches coexist, and allow us to discriminate among solutions which otherwise would result indistinguishable. More precisely, we can first compute the solutions according to egalitarianism (i.e., using a max-min computation as in FCSPs), and then discriminate more among them via utilitarianism (i.e., using a max-sum computation as in WCSPs).

The resulting semiring is the following:

$$S_{ue} = \langle \{\langle l, k\rangle | l, k \in [0, 1]\}, \underline{max}, \underline{min}, \langle 0, 0\rangle, \langle 1, 1\rangle \rangle$$

where $\underline{\text{max}}$ and $\underline{\text{min}}$ are defined as follows:

$$—\langle l_1, k_1\rangle \underline{max} \langle l_2, k_2\rangle = \begin{cases} \langle l_1, max(k_1, k_2)\rangle & \text{if} \quad l_1 = l_2 \\ \langle l_1, k_1\rangle & \text{if} \quad l_1 > l_2 \end{cases}$$

$$—\langle l_1, k_1\rangle \underline{min} \langle l_2, k_2\rangle = \begin{cases} \langle l_1, k_1 + k_2\rangle & \text{if} \quad l_1 = l_2 \\ \langle l_2, k_2\rangle & \text{if} \quad l_1 > l_2 \end{cases}$$

That is, the domain of the semiring contains pairs of values: the first element is used to reason via the max-min approach, while the second one is used to further discriminate via the max-sum approach. The operation $\underline{\text{min}}$ (which is the multiplicative operation of the semiring, and thus is used to perform the constraint combination) takes two elements of the semiring, say $a = \langle a_1, a_2\rangle$ and $b = \langle b_1, b_2\rangle$) and returns the one with the smallest first element if $a_1 \neq b_1$; otherwise, it returns the pair that has the sum of the second elements as its second element (i.e., $\langle a_1, a_2 + b_2\rangle$). The operation $\underline{\text{max}}$ performs a max on the first elements if they are different (thus, returning the pair with the maximum first element); otherwise (if $a_1 = b_1$), it performs a max on the second elements (thus, returning a pair which has such a max as second element, that is, $\langle a_1, max(a_2, b_2)\rangle$). More abstractly, we can say that, if the first elements of the pairs differ, then the $\underline{\text{max}}$-$\underline{\text{min}}$ operations behave like a normal max-min; otherwise, they behave like max-sum. This can be interpreted as the fact that, if the first elements coincide, it means that the max-min criterion cannot discriminate enough, and thus the max-sum criterion is used.

One can show that S_{ue} is a c-semiring. However, since it uses a combination of the operations of the semirings S_{FCSP} and S_{WCSP}, and since the multiplicative operation of S_{WCSP} is not idempotent, also the multiplicative operation of S_{ue}, that is, $\underline{\text{min}}$, is not idempotent. This means that we cannot guarantee that local consistency algorithms can be used meaningfully in this instance of the framework.

Note that also the opposite choice (i.e., first perform a max-sum and then a max-min) can be made, by using a similar c-semiring. However, due to the same reasoning as above, again the multiplicative operation would not be idempotent,

thus we cannot be sure that the local consistency techniques possess the properties of Section 4.

6.6. SET-BASED CSPs. An interesting class of instances of the SCSP framework which are based on set operations like union and intersection is the one which uses the c-semiring

$$S_{set} = \langle \wp(A), \cup, \cap, \emptyset, A \rangle,$$

where A is any set. It is easy to see that S_{set} is a c-semiring. Also, in this case the order $\leq_{S_{set}}$ reduces to set inclusion (in fact, $a \leq b$ iff $a \cup b = b$), and therefore it is partial in general. Furthermore, $\times$ is $\cap$ in this case, and thus it is idempotent. Therefore, the local consistency algorithms possess all the properties stated in Section 4 and therefore can be applied. We recall that when a local consistency algorithm is applied to an SCSP with a partially ordered semiring, its application changes the constraints in a way that the new constraints are smaller than the old ones in the ordering $\sqsubseteq$.

A useful application of such SCSPs based on a set is when the variables of the problem represent processes, the finite domain D is the set of possible states of such processes, and A is the set of all time intervals over reals. In this case, a value t associated to a pair of values $\langle d_1, d_2 \rangle$ for variables x and y can be interpreted as the set of all time intervals in which process x can be in state d_1 and process y in state d_2. Therefore, a solution of any SCSP based on the c-semiring S_{set} consists of a tuple of process states, together with (the set of) the time intervals in which such system configuration can occur. This description of a system can be helpful, for example, to discover if there are time intervals in which a deadlock is possible.

7. *N-Dimensional c-Semirings*

Choosing an instance of the SCSP framework means specifying a particular c-semiring. This, as discussed above, induces a partial order which can be interpreted as a (partial) guideline for choosing the "best" among different solutions. In many real-life situations, however, one guideline is not enough, since, for example, it could be necessary to reason with more than one objective in mind, and thus choose solutions which achieve a good compromise with respect to all such goals.

Consider for example a network of computers, where one would like to both minimize the total computing time (thus the cost) and also to maximize the work of the least used computers. Then, in our framework, we would need to consider two c-semirings, one for cost minimization (see Section 6.4 on weighted CSPs), and another one for work maximization (see Section 6.2 on fuzzy CSPs). Then one could work first with one of these c-semirings and then with the other one, trying to combine the solutions which are the best for each of them. However, a much simpler approach consists of combining the two c-semirings and only then work with the resulting structure. The nice property is that such a structure is a c-semiring itself; thus, all the techniques and properties of the SCSP framework can be used for such a structure as well.

More precisely, the way to combine several c-semirings and getting another c-semiring just consists of vectorizing the domains and operations of the combined c-semirings.

Definition 7.1 (*composition of c-semirings*). Given n c-semirings $S_i = \langle A_i, +_i, \times_i, \mathbf{0_i}, \mathbf{1_i}\rangle$, for $i = 1, \ldots, n$, let us define the structure $Comp(S_1, \ldots, S_n) = \langle\langle A_1, \ldots, A_n\rangle, +, \times, \langle\mathbf{0_1}, \ldots, \mathbf{0_n}\rangle, \langle\mathbf{1_1} \cdots \mathbf{1_n}\rangle\rangle$. Given $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_n\rangle$ such that $a_i, b_i \in A_i$ for $i = 1, \ldots, n$, $\langle a_1, \ldots, a_n\rangle + \langle b_1, \ldots, b_n\rangle = \langle a_1 +_1 b_1, \ldots, a_n +_n b_n\rangle$, and $\langle a_1, \ldots, a_n\rangle \times \langle b_1, \ldots, b_n\rangle = \langle a_1 \times_1 b_1, \ldots, a_n \times_n b_n\rangle$.

We will now prove that by composing c-semirings we get another c-semiring. This means that we all the properties of the framework defined in the previous sections hold.

THEOREM 7.2 (A C-SEMIRING COMPOSITION IS A C-SEMIRING). *Given n c-semirings $S_i = \langle A_i, +_i, \times_i, \mathbf{0_i}, \mathbf{1_i}\rangle$, for $i = 1, \ldots, n$, the structure $Comp(S_1, \ldots, S_n)$ is a c-semiring.*

PROOF. It is easy to see that all the properties required for being a c-semiring hold for $Comp(S_1, \ldots, S_n)$, since they directly follow from the corresponding properties of the component semirings S_i. □

According to the definition of the ordering $\leq_S$ (in Section 2), such an ordering for $S = Comp(S_1, \ldots, S_n)$ is as follows. Given $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1, \ldots, b_n\rangle$ such that $a_i, b_i \in A_i$ for $i = 1, \ldots, n$, we have $\langle a_1, \ldots, a_n\rangle \leq_S \langle b_1, \ldots, b_n\rangle$ if and only if $\langle a_1 +_1 b_1, \ldots, a_n +_n b_n\rangle = \langle b_1, \ldots, b_n\rangle$. Since the tuple elements are completely independent, $\leq_S$ is in general a partial order, even if each of the $\leq_{S_i}$ is a total order. This means (see Section 3) that the abstract solution of a problem over such a semiring in general contains an incomparable set of tuples, none of which has $blevel(P)$ as its associated value. Therefore, if one wants to reduce the number of "best" tuples (or to get just one), one has to specify some priorities among the orderings of the component c-semirings.

Notice that, although the c-semiring discussed in Section 6.4 may seem a composition of two c-semirings, as defined in Definition 7.1, this is not so, since the behavior of each of the two operations on one of the two elements of each pair (we recall that that semiring is a set of pairs) is not independent from the behavior of the same operation on the other element of the pair. Thus it is not possible to define two independent operations (i.e., $+_1$ and $+_2$, or $\times_1$ and $\times_2$), as needed by Definition 7.1. For example, operation <u>*max*</u> returns the maximum of the second elements of two pairs *only when* the first elements of the pairs are equal.

8. *Conclusion and Future Work*

We have proposed a general framework for constraint solving where each tuple is allowed by a constraint with a certain level of confidence (or degree, or cost, etc.). This allows for a more realistic modelization of real-life problems, but requires a new constraint solving engine which has to take such levels into account. To do this, we used the notion of semiring, which provides both the levels and the new constraint combination operations. We also considered the

issue of local consistency in such an extended framework, and we provided sufficient conditions which assure the local consistency algorithms to work in the extended framework. Then, we considered dynamic programming-like algorithms, and we proved that these algorithm can always be applied to SCSPs, and have a linear time complexity when the given SCSPs can be provided with a parsing tree of bounded size. Finally, we provided several instances of SCSPs which show the generality and also the expressive power of the framework.

As it may be difficult to assign each tuple in each constraint with a value from the semiring, but it may be instead easy to rate some solutions, we are currently investigating the possibility of *learning* values for all the tuples from some examples of solution ratings, via the use of classical gradient-descent learning algorithms [Rossi and Sperduti 1996]. This would greatly enhance the practical usefulness of SCSPs.

As classical constraint solving has been embedded into Constraint Logic Programming (CLP) [Jaffar and Lassez 1987] systems, it is reasonable to think that also our more general notion of constraint solving can be handled within a logic language, thus giving rise to new instances of the CLP scheme. This would not only give new meanings to constraint solving in CLP, but it would also allow one to treat in a uniform way optimization problem solving within CLP, without the need to resort to ad hoc methods as in Van Hentenryck [1989]. In fact, it would be possible to generalize the semantics of CLP programs to consider the chosen semiring and thus solve problems according to the semiring operations. This could be done by associating with each ground atom an element of the semiring and by using the two semiring operations to combine goals. These issues have been studied already for the instance of WCSPs in Bistarelli [1994]. We plan to do it in the general case as well.

ACKNOWLEDGMENTS. We would like to thank the Esprit BRA project ACCLAIM (n. 7195) for partially supporting the initial stages of this research, Thomas Schiex for many enlightening discussions on this subject, and the anonymous referees for all the useful comments on previous versions of this paper.

REFERENCES

BELLMAN, R. E. 1957. *Dynamic Programming*. Princeton University Press, Princeton, N.J.

BELLMAN, R. E., AND DREYFUS, S. E. 1962. *Applied Dynamic Programming*. Princeton University Press, Princeton, N.J.

BERTELE, U., AND BRIOSCHI, F. 1972. *Nonserial Dynamic Programming*. Academic Press, Orlando, Fla.

BISTARELLI, S. 1994. Programmazione con vincoli pesati e ottimizzazione (in Italian). Tech. Rep. Dipartmento di Informatica, Universita di Pisa, Pisa, Italy.

BISTARELLI, S., FARGIER, H., MONTANARI, U., ROSSI, F., SCHIEX, T., AND VERFAILLE, G. 1996. Semiring-based CSPs and valued CSPs: Basic properties and comparison. In *Over-Constrained Systems*, M. Jampel, ed. Lecture Notes in Computer Science, Vol. 1106. Springer-Verlag, New York.

BISTARELLI, S., MONTANARI, U., AND ROSSI, F. 1995. Constraint solving over semirings. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI95)*. Morgan-Kaufmann, San Mateo, Calif., pp. 624–630.

BORNING, A., MAHER, M., MARTINDALE, A., AND WILSON, M. 1989. Constraint hierarchies and logic programming. In *Proceeding of the 6th International Conference on Logic Programming*. MIT Press, Cambridge, Mass., pp. 149–164.

COUSOT, P. 1977. Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Tech. Rep. R.R.88. Institut National Polytechnique de Grenoble, Grenoble, Switzerland.

DAVEY, B. A., AND PRIESTLY, H. A. 1990. *Introductions to Lattices and Order*. Cambridge University Press, Cambridge, England. pp. 37–42.

DECHTER R., AND DECHTER, A. 1988. Brief maintenance in dynamic constraint networks. In *Proceedings of the AAAI88*. Morgan-Kaufmann, San Mateo, Calif., pp. 37–42.

DECHTER, R., DECHTER, A., AND PEARL, J. 1990. Optimization in constraint networks. In *Influence Diagrams, Belief Nets and Decision Analysis*, R. M. Oliver and J. Q. Smith, eds. Wiley, New York, pp. 411–425.

DECHTER, R., AND PEARL, J. 1988a. Network-based heuristics for constraint-satisfaction problems. In *Search in Artificial Intelligence*, Kanal and Kumar, eds. Springer-Verlag, New York, pp. 370–425.

DECHTER, R., AND PEARL, J. 1988b. Tree-clustering schemes for constraint processing. In *Proceedings of the AAAI88*. Morgan-Kaufmann, San Mateo, Calif., pp. 150–154.

DUBOIS, D., FARGIER, H., AND PRADE, H. 1993. The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proceedings of the IEEE International Conference on Fuzzy Systems*. IEEE, New York, pp. 1131–1136.

FARGIER, H., AND LANG, J. 1993. Uncertainty in constraint satisfaction problems: A probabilistic approach. In *Proceedings of the European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty* (*ESQUARU*). Lecture Notes in Computer Science, vol. 747. Springer-Verlag, New York, pp. 97–104.

FREUDER, E. C. 1978. Synthesizing constraint expressions. *Commun. ACM 21*, 11 (Nov.), 958–965.

FREUDER, E. C. 1988. Backtrack-free and backtrack-bounded search. In *Search in Artificial Intelligence*, Kanal and Kumar, eds. Springer-Verlag, New York, pp. 343–369.

FREUDER, E. C., AND WALLACE, R. J. 1992. Partial constraint satisfaction. *AI J. 58*, 21–70.

GIACOBAZZI, R., DEBRAY, S. K., AND LEVI, G. 1992. A generalized semantics for constraint logic programs. In *Proceedings of Fifth Generation Computer Systems (FGCS92)*. ICOT, Tokyo, Japan, pp. 581–591.

GNESI, S., MONTANARI, U., AND MARTELLI, A. 1981. Dynamic programming as graph searching: An algebraic approach. *J. ACM 28*, 4 (Oct.), 737–751.

GYSSENS, M., JEAVONS, P. G., AND COHEN, D. A. 1994. Decomposing constraint satisfaction problems using database techniques. *Artif. Int. 66*, 1, 57–89.

JAFFAR, J., AND LASSEZ, J. L. 1987. Constraint logic programming. In *Proceedings of Principles of Programming Languages* (Munich, West Germany, Jan. 21–23). ACM, New York, pp. 111–119.

KUMAR, V. 1992. Algorithms for constraint satisfaction problems: A survey. *AI Mag. 13*, 1, 32–44.

MACKWORTH, A. K. 1977. Consistency in networks of relations. *Artif. Int. 8*, 1, 99–118.

MACKWORTH, A. K. 1992. Constraint satisfaction. In *Encyclopedia of AI*, 2nd ed., vol. 1, S. C. Shapiro, ed. Wiley, New York, pp. 285–293.

MACKWORTH, A. K., AND FREUDER, E. C. 1984. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Int. 25*, 65–74.

MARTELLI, A., AND MONTANARI, U. 1972. Nonserial dynamic programming: On the optimal strategy of variable elimination for the rectangular lattice. *J. Math. Anal. Appl. 40*, 226–242.

MONTANARI, U. 1974. Networks of constraints: Fundamental properties and application to picture processing. *Inf. Sci. 7*, 95–132.

MONTANARI, U., AND ROSSI, F. 1986. An efficient algorithm for the solution of hierarchical networks of constraints. In *Proceedings of the International Workshop on Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science, vol. 291. Springer-Verlag, New York, pp. 440–457.

MONTANARI, U., AND ROSSI, F. 1991. Constraint relaxation may be perfect. *AI J. 48*, 143–170.

MOULIN, H. 1988. *Axioms for Cooperative Decision Making*. Cambridge University Press, Cambridge, Mass.

ROSENFELD, A., HUMMEL, R. A., AND ZUCKER, S. W. 1976. Scene labelling by relaxation operations. *IEEE Trans. Syst., Man, Cyber 6*, 6, 420–433.

RUTTKAY, ZS. 1994. Fuzzy constraint satisfaction. In *Proceedings of the IEEE 3rd International Conference on Fuzzy Systems*. IEEE, New York, pp. 1263–1268.

ROSSI, F., AND SPERDUTI, A. 1996. Learning solution preferences in constraint problems. In *Proceedings of the International Workshop on Non-Standard Constraint Processing at ECAI96*. Taylor and Francis publishers. (A longer version will appear in *J. Exp. Theor. Artif. Int.*)

SCHIEX, T. 1992. Possibilistic constraint satisfaction problems or "how to handle soft constraints?" In *Proceedings of the 8th Conference of Uncertainty in Artificial Intelligence*. pp. 269–275.

SCHIEX, T., FARGIER, H., AND VERFAILLE, G. 1995. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI95)*. Morgan-Kaufmann, San Mateo, Calif., pp. 631–637.

SHENOY, P. 1991. Valuation-based systems for discrete optimization. In *Uncertainty in Artificial Intelligence*, L. N. Kanal, P. P. Bonissone, M. Henrion, and J. F. Lemayer, eds. Elsevier Science, Amsterdam, The Netherlands.

SHENOY, P. 1994. Valuation-based systems: A framework for managing uncertainty in expert systems. In *Fuzzy Logic for the Management of Uncertainty*, L. Zadeh and J. Kacprzyk, eds. Wiley, New York, pp. 83–104.

VAN HENTENRYCK, P. 1989. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Mass.

ZADEH, L. A. 1965. Fuzzy sets. *Inf. Control* 8, 338–353.

RECEIVED SEPTEMBER 1995; REVISED FEBRUARY 1997; ACCEPTED DECEMBER 1996