Experimental Results on Learning Soft Constraints

Alessandro Biso[*], Francesca Rossi[†] and Alessandro Sperduti[*]
[*]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy.
E-mail: `perso@di.unipi.it`
[†]Dipartimento di Matematica Pura ed Applicata,
Universita' di Padova, Via Belzoni 7, 35131 Padova, Italy.
E-mail: `frossi@math.unipd.it`

Abstract

Constraints are a very natural knowledge representation formalism. However, classical constraints (which are either satisfied or not) are not so flexible and cannot describe real-life features like preferences, costs, priorities, and uncertainties. Therefore recently many formalisms for soft constraints (which can be satisfied at a certain level) have been developed.

We address the problem of modeling a real-life problem by using soft constraints. In many real-life situations, one may know his/her preferences over some of the solutions, but have no idea on how to code this knowledge into the constraint problem in terms of local preferences, or also one may be able to give only a rough approximation of the desired levels of satisfaction of the constraints. We therefore suggest to treat the solution preferences as examples, and to employ a learning scheme which learns from such examples (either from scratch or from the available rough model) all the local preferences, so that the global preferences specified in the examples are satisfied. This paper presents experimental results on the use of this technique, presenting both a base version of the learning scheme and also an incremental anytime version, which helps in deciding when the amount of learning is enough.

Keywords: Representation formalisms, machine learning, soft constraints, modeling with constraints.

1 Introduction

By soft constraints we mean constraints where each combination of values for the involved variables has an associated value, taken from a partially ordered set, which can be interpreted as a cost, or a level of preference, or a probability, or other. Soft constraints are much more flexible and expressive than classical "hard" constraints [9, 14] as a knowledge representation formalism, in that many more real-life problems can have a faithful representation as a soft constraint problem. In this paper we consider a formalism for soft constraints, called SCSP [1, 2], which uses a semiring (that is, a set plus two operations satisfying certain properties) to associate with each tuple of values for the variables an appropriate "degree of preference" (taken from the semiring). The two semiring operations define how to combine constraints together and how to compare different solutions. Many existing extensions to the classical constraint framework, such as fuzzy [4, 13], partial [6], probabilistic [5], and hierarchical constraints [3], can all be cast within the SCSP framework.

Although being very expressive, soft constraint problems can make the modeling phase much more difficult, since one has to specify all preferences for every possible combination of values in each constraint. Moreover, sometimes one may know his/her preferences over some of the solutions but have no idea on how to code this knowledge into the constraint problem in terms of local preferences. That is, one has a global idea about the *goodness* of a solution, but does not know the contribution of each single constraint to such a measure. In such a situation, it is difficult both to associate a preference to the other solutions in a compatible way, and to understand the importance of each tuple and/or constraint. Another typical situation occurs when one has just a rough estimate of the local preferences, either for the tuples or for the constraints.

In [11] the first situation is theoretically addressed by proposing to use learning techniques based on gradient descent [12]. More precisely, it is assumed that the level of preference for some solutions (the *examples*) is known, and a suitable learning technique is defined to learn, from these examples, values to be associated with each constraint tuple, in a way that is compatible with the examples.

In this paper we make the technique proposed in [11] concrete, we identify its features, and we show the results of several experiments run by choosing various values of these features. Moreover, we also address the second scenario, by defining and experimenting with a generalized version of the basic learning technique which allows for incremental learning starting form a rough model of the constraint problem.

The SCSP problems on which the experiments are run are randomly generated, in a way which is similar to the random generation of classical CSPs [8]. The results are encouraging, and show that the proposed techniques (especially the incremental version) are both reasonable and promising.

2 Soft constraint problems

Soft constraints do not say if a tuple of values for some variables is allowed or not, but rather *at which level* it is allowed. To describe such problems, in this paper we use the paradigm of semiring-based CSPs (SCSPs) [1, 2]: each tuple in each constraint is assigned a value (taken from the semiring), to be interpreted as the level of preference for that tuple, or its cost, or any other measurable feature. Then, constraints are combined according to the semiring operations, and the result of such a combination is that each assignment for all the variables has a corresponding semiring value too.

A *semiring* S is a tuple $(A, +, \times, \mathbf{0}, \mathbf{1})$ such that A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is a closed, commutative and associative operation such that $a + \mathbf{0} = a = \mathbf{0} + a$; $\times$ is a closed and associative operation such that $\mathbf{1}$ is its unit element and $a \times \mathbf{0} = \mathbf{0} = \mathbf{0} \times a$; $\times$ distributes over $+$ (i.e., $a \times (b + c) = (a \times b) + (a \times c)$). Actually, we need to use a structure called *c-semiring*, which is just a semiring where $+$ is idempotent, $\times$ is commutative, and $\mathbf{1}$ is the absorbing element of $+$.

The idempotence of the $+$ operation is needed in order to define a partial ordering $\leq_S$ over the set A, which enables us to compare different elements of the semiring. Such a partial order is defined as follows: $a \leq_S b$ iff $a+b = a$. Intuitively, $a \leq_S b$ means that a is "better" than b. This can be used to choose the "best" solution in our constraint problems.

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. Given a constraint system where $S = (A, +, \times, \mathbf{0}, \mathbf{1})$, a *constraint* over CS is a pair $\langle def, con \rangle$, where $con \subseteq V$ and $def : D^k \rightarrow A$ (where k is the number of variables in con). A *constraint problem* P over CS is just a set of constraints over CS.

The values specified for the tuples of each constraint are used to compute corresponding values for the tuples of values of all the variables, according to the semiring operations: $\times$ is used to combine the values of the tuples of each constraint to get the value of a tuple for all the variables, and $+$ is used to choose the best among all such tuples. More precisely, we have the operations of *combination* ($\otimes$) and *projection* ($\Downarrow$) over constraints, which are used, respectively, to combine constraints and to project out some variables[1].

Given a constraint problem C over a constraint system CS, the *solution* of P is a constraint defined as $Sol(P) = (\bigotimes C)$: it is the constraint induced on all the variables by the whole problem. Such constraint provides, for each tuple of values of D for all the variables, an associated value of A. More precisely, consider any tuple t with as many elements of D as the number of all variables (in the following such tuples will be called *n-tuples*). Then the corresponding value $val(t)$ can be obtained by (see definition of combination) $val(t) = def_1(t \downarrow^V_{con_1}) \times \ldots \times def_k(t \downarrow^V_{con_k})$, where V is the set of all variables, the set of all constraints is $C = \{c_1, \ldots, c_k\}$, and $c_i = \langle def_i, con_i \rangle$, for $i = 1, \ldots, k$. Given two n-tuples t and t', we say that t is better than t' if $val(t) \leq_S val(t')$. Note that this, by definition of $\leq_S$, means that $val(t) + val(t') = val(t)$. Note also that t and t' could be incomparable, since $\leq_S$ is a partial order. Given an SCSP P, we will call n-tuples(P) the set of all n-tuples of P. Then we will consider the function f :n-tuples(P) $\rightarrow A$ such that, for each n-tuple t, $f(t) = val(t)$. That is, the function which assigns to each n-tuple the corresponding value. We will call this f the *solution-rating function* of the given SCSP.

Many constraint formalisms can be cast in this semiring-based framework: it is enough to choose the appropriate semiring. The instance we consider here is the one describing optimization problems, where each tuple has an associated cost and the goal is to minimize the global cost. The associated semiring is $\langle \Re^+, min, +, +\infty, 0 \rangle$. In the experiments reported in this paper we will use a subset of this semiring.

3 Learning soft constraint problems

Inductive learning [12] can be defined as the ability of a system to induce the correct structure of a map d which is known only for particular inputs. More formally, defining an example as a pair $(x, d(x))$, the computational task is as follows: *given a collection of examples of d, i.e., the training set, return a function h that approximates d.* Function h is called a hypothesis.

A common approach to inductive learning, especially in the context of neural networks, is to evaluate the quality of a hypothesis h (on the training set) through

[1]For the formal definition of these two operations, see [1, 2].

an *error function* [7].

Given a starting hypothesis h_0, the goal of learning is to minimize the error function E by modifying h_0. This can be done by using a definition of h which depends on a set of internal parameters W, i.e., $h \equiv h_W$, and then adjusting these parameters. This adjustment can be formulated in different ways, depending on whether the domain is isomorphic to the reals or not. The usual way to be used over the reals, and if h_W is continuous and derivable, is to follow the negative of the *gradient* of E with respect to W. This technique is called *gradient descent* [7]. Specifically, the set of parameters W is initialized to small random values at time $\tau = 0$ and updated at time $\tau + 1$ according to the following equation: $W(\tau+1) = W(\tau) + \Delta W(\tau)$, where $\Delta W(\tau) = -\eta \frac{\partial E}{\partial W(\tau)}$, and η is the step size used for the gradient descent.

Learning is stopped when a minimum of E is reached. Note that, in general, there is no guarantee that the found minimum is global.

Learning in our context can be used to find a suitable set of values to be assigned to the constraint tuples of an SCSP. More precisely, consider a CSP P, which, we recall, can be seen as an SCSP where the solutions are those n-tuples t such that $val_P(t) = true$.

We assume now that for some solutions of P we are given a desired rating $d(t)$ defining the goodness of t. Let m be the number of such solutions. That is, we have the set of *examples* $TR = \{(t_1, d(t_1)), \ldots, (t_m, d(t_m))\}$. However, giving the examples is not enough, because we must also have an idea of how to combine and compare the values given as examples. Therefore, together with the examples, we must also be given the following two objects: a semiring containing the values in the examples, and a distance function over such a semiring. For our experiments we have chosen the semiring $\langle \Re^+, min, +, +\infty, 0 \rangle$ and, as distance function, the usual one over reals: $dist(val_{P'}(t), d(t)) = \mid val_{P'}(t) - d(t) \mid$. Once we have the above, our goal is to define a corresponding SCSP P' over the same semiring such that P and P' have the same graph topology, and for each n-tuple t such that $(t, d(t))$ is an example, $dist(val_{P'}(t), d(t)) < \epsilon$, where $\epsilon > 0$ and small.

Given the first condition (on the graph topology), the only free parameters that can be arbitrarily chosen in order to satisfy the other conditions are the values to be associated to each constraint tuple. For each constraint $c_i = \langle def_i, con_i \rangle$ in P, consider $S_i = \{t_{ij}$ such that $def_i(t_{ij}) = true\}$ (that is, the set of tuples allowed by that constraint). The idea is to associate, in P', a free parameter w_{ij} (note that w_{ij} must belong to the set of the chosen semiring) to each t_{ij} in S_i. With the other tuples, we associate the constant $\mathbf{0}$, which, we recall, is the worst element of the semiring (w.r.t. $\leq_S$). Given this association, the value assigned to each n-tuple t in P' is

$$val_{P'}(t) = \prod_{i=1}^{k}(\sum_{j=1}^{|S_i|} \text{subtuple}(t_{ij}, t, i) \times w_{ij}), \tag{1}$$

where $\prod$ refers to $\times$, k is the number of constraints in P, $\sum$ refers to $+$, and subtuple$(t_{ij}, t, i) = \mathbf{1}$ if $t \downarrow^V_{con_i} = t_{ij}$ and $\mathbf{0}$ otherwise.

Note that, for each i, there is exactly one j such that subtuple$(t_{ij}, t, i) = \mathbf{1}$. Let l_i be such a j. Thus $\sum_{j=1}^{|S_i|}$ subtuple$(t_{ij}, t, i) \times w_{ij} = w_{il_i}$, and therefore $val_{P'}(t) = w_{1l_1} \times \ldots \times w_{kl_k}$. The values of the free parameters w_{ij} may be obtained via a minimization of the error function, which will be defined according to the distance function of the semiring. When minimizing such a function, one has to assure that, every time the values of the free parameters are changed in order to make the error function smaller, the new values are in the semiring set as well. In other words, no change can be made to a value which is already $\mathbf{1}$ (resp., $\mathbf{0}$) and which the learning technique would like to make even smaller (resp., bigger).

Assume that the desired ratings are real numbers and that the chosen c-semiring is $\langle \Re^+, min, +, +\infty, 0\rangle$. Here $\times$, that is, standard sum, is continuous and derivable. Thus we can use the gradient descent technique.

In this case equation (1) becomes $val_{P'}(t) = \sum_{i=1}^{k}(min_{j=1,\ldots,|S_i|}$ subtuple$(t_{ij}, t, i) + w_{ij})$ and the error function is $E = \frac{1}{2}\sum_{i=1}^{m}(d(t_i) - val_{P'}(t_i))^2$. By deriving the error function w.r.t. one free parameter w_{ij} we get

$$\Delta w_{ij}(\tau) = -\eta \frac{\partial E}{\partial w_{ij}(\tau)} = -\eta \cdot \sum_{i=1}^{m}((d(t_i) - val_{P'}(t_i)) \cdot output(u_{ij})).$$

Thus only those weights which are associated to tuples which are contained in at least one n-tuple in an example will be changed. We recall that one has to make sure that all weights remain positive. This can be done by modifying the learning rule in such a way that weights that would become negative are not modified. See [10] for an example of this technique.

Notice that, for this semiring for constraint optimization problems, the learning problem could also be formulated as a quadratic programming problem. However, the gradient descent technique is in our view preferable, mainly for its locality and for the possibility of developing an incremental learning technique (see following sections). Also, in this case the gradient descent does not have any local minima, so we can be sure that stabilization is achieved only on globally optimal points.

4 Experimental results

To evaluate the behavior of the proposed learning technique, we have considered different learning scenarios. In the first case, we have tried to establish if the technique is able to reconstruct a solution-rating function starting from examples for a soft constraint problem of which we know the parameters. This is the simplest case, since we are sure that the function is representable by the model. To consider different conditions in a structured way, we used a variant of the notion of randomly-generated constraint problems. More difficult cases were analyzed as well. One deals with non-representable functions and the other one simulates a situation in which the user provides imprecise ratings. Both these scenarios are implemented by adding Gaussian noise to the desired ratings. For all these scenarios, we have considered two variants: one in which the system is given all the examples at once, and another, more realistic, one, in which examples are given on demand. Notice that the second variant of the learning technique can be used also to refine an approximate model of the constraint problem, obtained for example by giving preferences to constraints and not to tuples, or by assigning approximate preferences to the tuples.

The randomly-generated SCSP problems we used for our experiments are generated in a way which is very similar to the usual randomly generated CSPs [8], which are characterized by four parameters: $\langle n, m, p_1, p_2 \rangle$, where n is the number of variables, m is the size of the variable domains, p_1 is the density of the problem, that is, the percentage of constraints over the maximum number of constraints, and p_2 is the tightness, that is, the percentage of forbidden pairs in each constraint. To adapt this model to randomly generated SCSPs, we just need to add the semiring to be used and to decide what we mean by "forbidden pairs". In fact, while in standard CSPs pairs are either forbidden or allowed, in SCSPs each pair is allowed, but with a certain level of preference. Then we can decide either that "forbidden" means that its level of preference is not the best semiring value, or that it is the worst value. We have chosen the second interpretation, and thus in our framework p_2 is the percentage of pairs of each constraint which have the worst semiring value[2]. Therefore, once the semiring is fixed, a four-tuple $\langle n, m, p_1, p_2 \rangle$ is used to generate an SCSP with n variables, m values for each variable, p_1% of constraints, and for each constraint p_2% of pairs which have the worst semiring

[2]We also have conducted some experiments with the other notion of tightness, obtaining similar results.

value. All the other pairs are given a semiring value randomly.

Our experiments have been performed on the semiring of constraint optimization over the reals, that is, $\langle \Re^+, min, +, +\infty, 0 \rangle$. For practical reasons, it is usually preferable for the gradient descent technique not to have to deal with $+\infty$. Therefore, one can normalize the semiring $\langle \Re^+, min, +, +\infty, 0 \rangle$ by projecting $\Re^+$ over the interval $[0, x]$, with $x \neq +\infty$ representing $+\infty$, and adjusting the two operations accordingly. In our experiments, we have arbitrarily chosen $x = 11$. Thus p_2 is the percentage of pairs of each constraint which get the semiring value 11.

4.1 Basic version for representable functions

In the experiments reported in this paper, we first focus on the case in which the unknown function can be computed by the model, and we evaluate how well the proposed gradient descent technique can approximate such a function. To model this situation, we randomly generate an SCSP, which will be used as our model: the values of its n-tuples represent the unknown function, to be found by the learning technique. Therefore, some of its solution ratings will constitute the *training set* (that is, the examples), while others will be used as the *test set*, which is the set against which the results of the learning phase will be tested. As usual practice in learning, training set and test set are disjoint.

The n-tuples belonging to the training set are not chosen randomly, but in such a way that each pair in each constraint appears in at least one of the n-tuples taken as examples. This means that, for each parameter, we have at least some data that can help us to learn its value.

Once chosen the training and the test set, we take the SCSP, we forget the semiring values associated to the pairs in the constraints, we assign to each pair a parameter, and we randomly initialize these parameters. Then we start the learning phase, which, at each iteration, checks the value of the error function on the training set, adjusts the step size accordingly (following an adaptive gradient descent scheme), and changes the values of the parameters, until the maximum error on the training set goes below a certain threshold. The threshold we chose for our experiments is 1% of the maximum semiring value associated to an example in the training set.

The results we show are related to several SCSPs, obtained by considering four values of density (20%, 40%, 60%, and 80%) and three values of tightness (20%, 50%, and 80%). The number of variables is always 20 and they have 5 values each. Moreover, for each choice of a density and a tightness, we have generated three

	Mean error			# examples
	t20%	t50%	t80%	
d20%	1.71	1.48	0.60	781
d40%	1.55	1.34	0.60	1545
d60%	1.25	1.17	0.55	2306
d80%	1.32	1.09	0.59	3068

Table 1: Mean error and number of examples.

SCSPs, and we have computed the arithmetic mean of the results over such three SCSPs.

Table 1 shows the mean error on the test set for each density/tightness combination, and also the number of examples for each density. Notice that this is the relative and not the absolute error, since we relate the error to the range of all the correct values on the training set. Therefore, saying that a solution in the test set has an error equal to 1% means that the absolute error is $x/100$, where x is the size of the actual range of correct values. The most important thing to notice here is that in general the error is reasonable. Another point is that, given a certain density, the error decreases as the tightness increases.

4.2 Incremental version for representable functions

Even though the results reported in Table 1 seem reasonable and thus show the feasibility of the proposed technique, one may be worried about the amount of examples needed to achieve such results. In fact, although the order of magnitude of the training set is always much smaller than the domain of the unknown function, still it may be too much for the user to give all such examples at once. For this reason, and also to be able to refine an approximate model, we have developed a variant of the basic technique proposed in the previous section, where the user gives a bunch of examples at a time, on request.

The basic idea is the following. We want to model a realistic situation in which the system uses only a small quantity of examples, and then the user check the results and gives new examples, which represent situations which have not been learnt well by the system. And so on until all situations of interest to the user have been learnt satisfactorily.

The way we model this interactive process consists of performing several phases of learning, where each learning phase uses the same algorithm as in Section 4.1, but has different training and test sets. If the entire training set is called T, we consider its partition into a small part S of n elements, and the rest: $R = T - S$. Moreover, we also consider a subset I of R, again with n elements. The first learning phase will take S as its training set, and I as its test set. At this point, the elements of I which are not learnt well, say $bad(I)$, are then inserted into S to produce a larger training set for the next learning phase. Thus the new training set is $S \cup bad(I)$ and the new test set consists of n elements of $T - (S \cup I)$. This process is repeated until all elements of the current set I have been learnt well. At this point, the current size of the training set tells us how many examples we need in total from the user. One final issue of this technique is how to define the set $bad(I)$: what we do is to use a threshold (1% of the absolute error), which tells us which elements have not been learnt well (those for which the error is above the threshold). Obviously, the final result of the learning phase is then evaluated on a disjoint set of examples, which is the *real* test set for the overall experiment.

In our experiments with this incremental technique, we have chosen just one density (40%) and one tightness (50%), and the usual size for the variable set (20) and the domain size (5). Then we have generated the usual training set, and we have treated it as described above. However, the above description leaves some notions under-defined, and thus we run our experiments with all the possible choices, to check whether these choices influence the results or not. In particular, the value of n, that is, the initial size of the training set and the size of the set I at each time, has been set to 150 in one set of experiments, and to 300 in another set, to check the influence of this parameter on the resulting error. Another source of possible nondeterminism could be the way the training set is "visited" during the whole process. In fact, it could be that the elements considered at the beginning produce an overfitting, that is, a specialization over such elements which could interfere with the correct learning of the elements considered later. Therefore we run our experiments both reading the training set from the beginning to the end, and also from the end to the beginning. The incremental learning technique described above does not say if, after each enlarging of the training set, the next learning phase must start from scratch or from where the previous learning phase has ended. One might think that the final results could be very different, since starting from scratch could mean forgetting what we have learnt before and thus avoiding a possible overfitting over the examples of the training set of the previous phase. Our experiments have shown that the two alternatives produce almost equivalent results, but starting from scratch is less efficient. Thus we will not show the results for this case. Notice that this observation shows the feasibility of the application of our incremental technique when starting from an approximate model.

Summarizing, we have two different choices when running an experiment: $n =$

150 or $n = 300$, and forward or backward visit of the training set. We show the results for the forward visit for both values of n, while the backward visit is shown only for $n = 150$ (results for $n = 300$ are very similar).

Figure 1 shows the number of examples in the training set at each phase (left plot) and the mean error in percentage (right plot). The first thing to notice is that the way the training set is visited is not relevant, since both the number of examples and the mean error stay the same. Also, when $n = 300$, the training set grows faster in size than when $n = 150$. However, at the end of the learning procedure, the training sets have roughly the same size. On the contrary, when considering the mean error, for $n = 300$ the error decreases faster. However, even for this measure, the final values are similar. From this observations, it may seem that the incremental learning procedure is independent from the value of n. However, if one considers the computational cost (see Figure 2), this measure is greater for $n = 150$, since one has to perform a greater number of phases to obtain the same result as for $n = 300$.

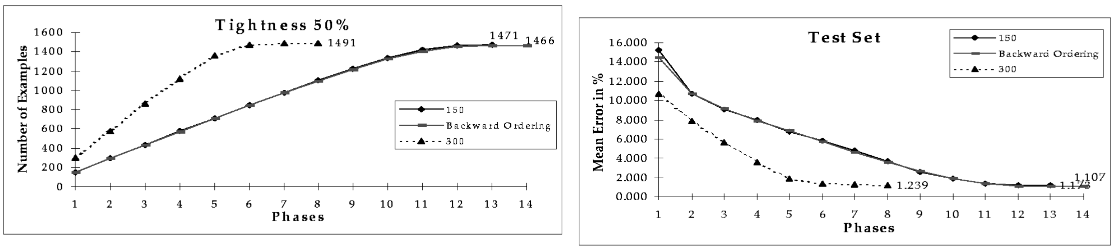

Figure 1: Number of examples (left plot) and mean error (right plot) for each phase.

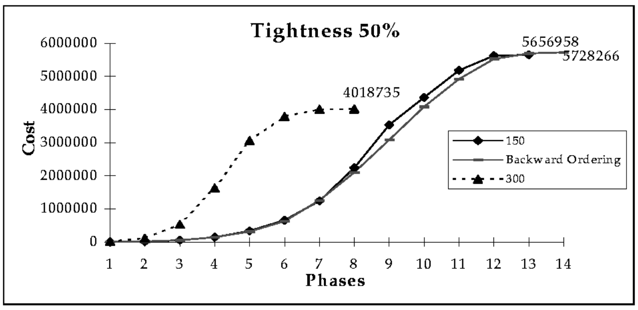

Figure 2: Computational cost up to each phase.

	Mean error					
	Test with noise			Test without noise		
	t20%	t50%	t80%	t20%	t50%	t80%
d20%	4.80	4.84	5.38	4.21	4.05	3.78
d40%	4.79	5.88	5.82	4.07	4.69	3.94
d60%	5.49	6.21	6.95	4.49	4.81	4.46
d80%	6.26	6.54	7.41	5.17	5.06	4.59

Table 2: Mean error for non-representable functions.

4.3 Non-representable functions

We have also simulated the situation in which the desired solution-rating function is not representable in the underlying model. This may happen in our framework, since we consider given a certain constraint graph, which may not be able (whatever the constraint definitions are) to produce the desired solution rating function. For example, the user may give us contradictory ratings for correlated solutions, which means that the constraint graph is not adequate for the user's expectations.

We have modeled this situation by adding noise to the training set and also to the test set. This noise follows a Gaussian model with mean 0 and deviation which is 5% of the highest desired value. Even for this scenario, as we did for representable functions, we experimented with both our basic learning technique and the incremental one. Because of the presence of noise, here it is not possible to use the same stop criterion as before (error smaller than a threshold), thus we continue the learning process until no significant improvement is shown.

The first three columns of Table 2 show the mean error in percentage on the test set for the basic learning procedure. It is easy to see that the error values are much higher than in the representable case. However, the learning process shows a rather robust behavior, since, apart from the higher values, it behaves similarly to the representable case.

Figure 3 shows the number of examples needed for the phases of the incremental technique (left plot) and the mean error in such phases (right plot). These figures do not show what happens when visiting the training set backwards, since we have seen (by running experiments) that the direction of the visit does not influence the result. Again, we may note that these curves are similar in shape to those for the representable case, but show higher error values, as expected.

Further experiments were run to simulate the case of a user which gives imprecise knowledge about the desired solution ratings. This scenario was modeled by the introduction of noise only on the elements of the training set (and not on the test set). The last three columns of Table 2 show the mean error on the test set.

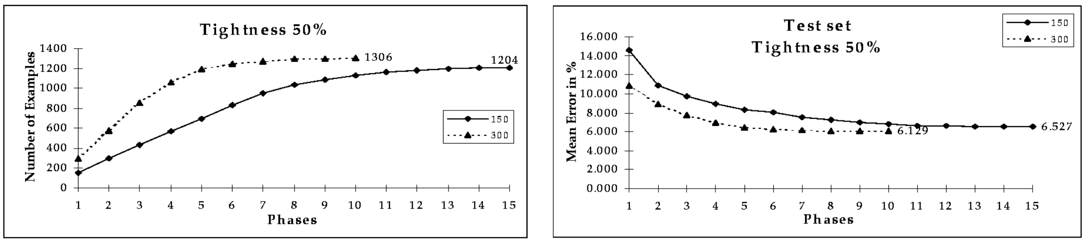

Figure 3: Incremental version: number of examples (left plot) and mean error (right plot) for each phase.

Notice that it is lower than for non-representable functions (due to the absence of noise on the test set), but otherwise has a similar relationship with density and tightness. The curve of the mean error in the various phases (not shown) is similar in shape to that in Figure 3 (right plot) but leads to smaller values: for $n = 150$ the minimum error is 5.514 (instead of 6.527) and for $n = 300$ it is 4.991 (instead of 6.129).

Summarizing, the system seems to have responded well to the introduction of noise, especially in the case which simulates an imprecise user. In fact, the noise introduced was rather relevant (up to 5%), and the error values are very close to this value. This means that the system was able to substantially learn the desired function (except for the noise).

5 Conclusions and future work

We have run several experiments which simulate the task of learning an SCSP, given a fixed constraint graph and some examples of solution ratings. We also developed a variation of the basic learning technique which allows to get a reasonable result without the need to give all the examples at once, thus making the user role in the modeling phase even easier. This new technique is based on an incremental and interactive process, composed of several learning phases in sequence. Finally, we have considered functions which are not representable in the model, simulating this lack of precision by introducing noise in the learning technique.

The results obtained so far are encouraging and show that the proposed technique is promising. We plan to study its behavior on real (that is, not randomly generated) problems, and to extend the learning capability to discrete semirings.

References

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

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

[3] A. Borning, M. Maher, A. Martindale, and M. Wilson. Constraint hierarchies and logic programming. In Martelli M. Levi G., editor, *Proc. 6th International Conference on Logic Programming*, pages 149–164. MIT Press, 1989.

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

[5] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probabilistic approach. In *Proc. European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ECSQARU)*, pages 97–104. Springer-Verlag, LNCS 747, 1993.

[6] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *AI Journal*, 58, 1992.

[7] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[8] T. Hogg, B.A. Hubermann, and C.P. Williams, editors. *Special Volume on Frontiers in Problems Solving: Phase Transition and Complexity.* Artificial Intelligence, vol.81, n. 1-2, 1996.

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

[10] M. Pelillo and M. Refice. Learning compatibility coefficients for relaxation labeling processes. *IEEE Transactions on Patterns Analysis and Machine Intelligence*, 1994. vol.16, n.9.

[11] F. Rossi and A. Sperduti. Learning solution preferences in constraint problems. *Journal of Experimental and Theoretical Computer Science*, 1998. Vol 10.

[12] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

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

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