A Local Search Framework for Semiring-Based Constraint Satisfaction Problems

Stefano Bistarelli[1], Spencer K.L. Fung[2], J.H.M. Lee[2], and Ho-fung Leung[2]

[1] Dipartimento di Scienze
Universita degli studi "G. D'Annunzio" di Chieti-Pescara, Italy
and
Istituto di Informatica e Telematica
C.N.R. Pisa, Italy
bista@sci.unich.it

[2] Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T, Hong Kong SAR, China
{sklfung,jlee,lhf}@cse.cuhk.edu.hk

Abstract. Solving semiring-based constraint satisfaction problem (SCSP) is a task of finding the best solution, which can be viewed as an optimization problem. Current research of SCSP solution methods focus on tree search algorithms, which is computationally intensive. In this paper, we present an efficient local search framework for SCSPs, which adopts problem transformation and soft constraint consistency techniques, and E-GENET local search model as a foundation. Our framework is parameterized by the semiring structure S, resulting in a family of algorithms for various kinds of soft constraint problems. We build a prototype solver that is based on the proposed framework, and test it on both structured and non-structured problems. The benchmarking results show that it is feasible to tackle SCSPs in an efficient manner.

1 Introduction

Classical constraint satisfaction problem (CSP) [15, 16] is an expressive and natural formalism to specify many kinds of problems from scheduling, bin-packing to resource allocation. In practice, however, many problems are over-constrained and involve preferences, going outside the realm of class CSPs and calling for the notion of soft constraints. Semiring-based Constraint Satisfaction Problems (SCSP) is a general framework for specifying soft constraint problems, encompassing classical CSPs [15, 16], fuzzy CSPs [19], partial CSPs [9], probabilistic CSPs [8], weighted CSPs [13], among others. Solving an SCSP amounts to finding valuations that satisfy the soft constraints the "best", where the goodness measure depends on the semiring structures of the problem. Existing SCSP solution methods are based on local consistency techniques and tree search [11, 18], and also dynamic programming [1], which are complete methods with likely an exponential cost. Recently, Yan Georget and Philippe Codognet [11] developed a backtracking-based constraint logic programming system, `clp(FD,S)`, for

solving SCSPs, which is an integration framework for their proposed semiring kernel (SFD) and the constraint logic programming system `clp(FD)`. Bistarelli *et al.* [1] proposed a dynamic programming approach to solve SCSPs, which solves a sub-problem (a subset of constraints) of the original problem repeatedly, and replaces the sub-problem by a new generated constraint. Besides, local consistency techniques [1, 2] are also useful for solving SCSPs, however, the $\times$ operator of the semiring is restricted to be idempotent.

In this paper, we present a local search framework for tackling SCSP problems. Like the systematic search counterpart, our framework is parameterized by the semiring structure S, resulting in a family of algorithms. Basically, our framework consists of three main components: *sinking*, *shrinking* and *local search engine*. The novelty of our approach is that, we do forward and backward transformation between original SCSP and crisp CSP, and perform searching on the solution space of the crisp problem repeatedly. Besides, shrinking techniques are also applied to the search space to further improve the searching performance. Our work is most related to the iterative abstraction method, proposed by Rossi *et. al.* [18], which solves the abstract version of the original SCSP problem via Conflex, and narrow the upper and lower bounds of the desired solution iteratively. The main difference between the two approaches is the searching strategy and the bounds approximation. We use local search approach, while iterative abstraction is based on branch-and-bound tree search. Our implementation is based on that of E-GENET [14], which allows the problem transformation to be done naturally with little overhead. Benchmarking results show that the approach is effective and efficient on tackling both structured and non-structured problems.

The rest of this paper is organized as follows. In Section 2, we recall the idea of semiring-based CSP solving. We present the components and the underlying theory of our local search framework in Section 3 and its experimental results in Section 4. Conclusions and perspectives for future work are addressed in the Section 5.

2 Semiring-based Constraint Solving

A soft constraint may be seen as a constraint where each instantiations of its variables has an associated value from a partially ordered set which can be interpreted as a set of preference values. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints.

Here we give the basic notions about constraint solving over semirings, the details definitions and proofs of properties can be found in [4].

2.1 Semirings

A semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that:

- A is a set and $\mathbf{0}, \mathbf{1} \in A$;
- $+$ is commutative, associative and $\mathbf{0}$ is its unit element;
- $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element.

A c-semiring is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [4]):

- $\leq_S$ is a partial order;
- $+$ and $\times$ are monotone on $\leq_S$;
- $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum;
- $\langle A, \leq_S\rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$ (where *lub* is the *least upper bound*).

Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ its *glb* (*greatest lower bound*). Informally, the relation $\leq_S$ gives us a way to compare semiring values and constraints. In fact, when we have $a \leq_S b$, we will say that b *is better than* a. In the following, when the semiring S will be clear from the context, $a \leq_S b$ will be often indicated by $a \leq b$.

2.2 Constraint Problems

Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered set of variables V over a finite domain D, a *constraint* is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring.

By using this notation we define $\mathcal{C} = \eta \to A$ as the set of all possible constraints that can be built starting from S, D and V. In this *functional* formulation, each constraint is a function (as defined in [5]) and not a pair (as defined in [3, 4]). Such a function involves all the variables in V, but it depends on the assignment of only a finite subset of them. More precisely, for each tuple of values for the involved variables, a corresponding element (semiring value) of the set A is given, which can be interpreted as the tuples' weight, cost, or degree of satisfaction. For instance, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : V \to D \to A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$. We call this subset the *support* of the constraint.

More formally, consider a constraint $c \in \mathcal{C}$. We define its support as $supp(c) = \{v \in V \mid \exists \eta, d_1, d_2.c\eta[v := d_1] \neq c\eta[v := d_2]\}$, where

$$\eta[v := d]v' = \begin{cases} d & \text{if } v = v', \\ \eta v' & \text{otherwise.} \end{cases}$$

Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$ (that is the operator [ ] has precedence over application). Note also that $c\eta$ is the application of a constraint function $c : V \to D \to A$ to a function $\eta : D \to A$; what we obtain, is a semiring value $c\eta = a$.

A *semiring constraint satisfaction problem* is a pair $\langle C, con \rangle$ where $con \subseteq V$ and C is a set of constraints: *con* is the set of variables of interest for the constraint set C, which, however, may concern also variables not in *con*.

Fig. 1 shows the graph representation of a fuzzy CSP. Variables and constraints are represented respectively by nodes and by undirected (unary for c_1 and c_3 and binary for c_2) arcs, and fuzzy membership values are written to the right of the corresponding tuples as semiring values. The variables of interest (that is the set *con*) are represented with a double circle. Here we assume that the domain D of the variables contains only elements a and b and c.

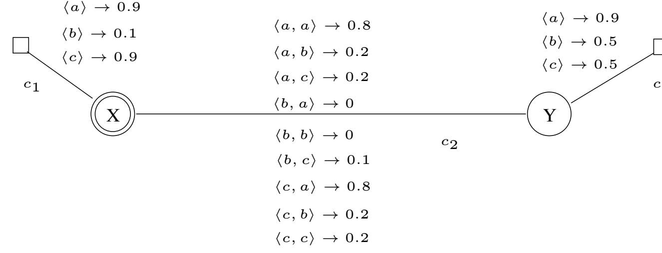

Fig. 1. An example of Fuzzy CSP.

2.3 Combination and Projection of soft constraints

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$.

In other words, combining two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate subtuples. It is easy to verify that $supp(c_1 \otimes c_2) \subseteq supp(c_1) \cup supp(c_2)$. We can easily extend the partial order $\leq_S$ over $\mathcal{C}$ by defining $c_1 \sqsubseteq c_2 \iff c_1\eta \leq c_2\eta$.

Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* of c over $V - \{v\}$, written $c \Downarrow_{(V-\{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$. The projection operator can be easily extended to a set of variable $I \subseteq V$ by defining $c \Downarrow_{(V-I)} = c \Downarrow_{(V-\{v\})} \Downarrow_{(V-\{I-\{v\}\})}$. It is easy to verify that $supp(c \Downarrow_S) \subseteq S$.

Informally, projecting means eliminating some variables from the support. This is done by associating with each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables. In short, combination is performed via the multiplicative operation of the semiring, and projection via the additive one.

2.4 Solutions

The *solution* of an SCSP $P = \langle C, con \rangle$ is the constraint $Sol(P) = (\bigotimes C) \Downarrow_{con}$. That is, we combine all constraints, and then project over the variables in *con*. In this way we get the constraint with support (not greater than) *con* which is "induced" by the entire SCSP. Note that when all the variables are of interest we do not need to perform any projection.

Example 1. The solution of the fuzzy CSP of Fig. 1 associates a semiring element to every domain value of variable x. Such an element is obtained by first combining all the constraints together. For instance, for the tuple $\langle a, a \rangle$ (that is, $x = y = a$), we have to compute the minimum between 0.9 (which is the value assigned to $x = a$ in constraint c_1), 0.8 (which is the value assigned to $\langle x = a, y = a \rangle$ in c_2) and 0.9 (which is the value for $y = a$ in c_3). Hence, the resulting value for this tuple is 0.8. We can do the same work for tuple $\langle a, b \rangle \to 0.2$, $\langle a, c \rangle \to 0.2$, $\langle b, a \rangle \to 0$, $\langle b, b \rangle \to 0$, $\langle b, c \rangle \to 0.1$, $\langle c, a \rangle \to 0.8$, $\langle c, b \rangle \to 0.2$ and $\langle c, c \rangle \to 0.2$. The obtained tuples are then projected over variable x, obtaining the solution $\langle a \rangle \to 0.8$, $\langle b \rangle \to 0.1$ and $\langle c \rangle \to 0.8$.

Note that the solution of an SCSP is a constraint involving all variables in *con*. Each tuple in the solution constraint is associated with a respective semiring value. In practice, however, users are not interested in all tuples in the solution constraint but only certain "good" ones. For the purpose of this paper, we define a *preferred solution assignment* $\eta \in Sol(P)$ with semiring value α to be one such that if $\eta' \in Sol(P)$ has semiring value α', then $\alpha \not\leq \alpha'$. Our local search solver aims to compute preferred solution assignment of an SCSP.

2.5 Local Consistency

The idea of local consistency in classical CSPs is to choose some subproblems of the original problem 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 node-consistency and arc-consistency algorithm. As discussed in [4], local consistency properties also hold in SCSPs, provided that the combination operator is idempotent and the set A of c-semiring is finite.

More precisely, we say that the problem $P = \langle C, con \rangle$ is *locally inconsistent* if there exist $C' \subseteq C$ such that $blevel(C') = 0$, where $blevel(C)$ is defined as *best level of consistency* (i.e. the best semiring value it can produce). An SCSP is said to be α-consistent, if there must be an assignment produces a semiring value not less than α. Consider a set of constraints C and $C' \subseteq C$. If C is α-consistent then C' is β-consistent with $\alpha \leq \beta$. As a consequence, if a problem is locally inconsistent, then it is not consistent.

3 A Local Search Framework

A local search scheme for SCSPs is described in this section. Basically, it consists of three main components: *sinking*, *shrinking* and *local search engine*. The sinking

procedure is used to filter out all the α-*inconsistent* tuples in the problem, which are tuples with semiring values less than that of the current best solution, α. The shrinking procedure is used to eliminate unwanted search space, which can be achieved by local consistency algorithms or interchangeability algorithm [10]. The underlying searching strategy of this framework is based on a GENET style model [20], which is a local search algorithm with min-conflict heuristics for classical CSPs.

We give an abstract formalization of the local search framework similar to the Lagrangian search scheme [6], and followed by the description of each functional component in the framework.

3.1 An Abstract Scheme of the Local Search Framework

In this section, we show a transformation for converting SCSPs into an integer constrained optimization problem. Let $V = \{x_1, ..., x_n\}$, $D = \{D_{x_1}, ..., D_{x_n}\}$ and $C = \{c_1, ..., c_m\}$ be the set of variables, domains, and constraints of the problem to be solved respectively, where $m \leq |D_{x_1}| \times ... \times |D_{x_n}|$. Without loss of generality and for the ease of presentation we assume that all constraints are n-ary `illegal` constraints [20]. Therefore, a constraint $c_i \in C$ is always of the following form:

$$c_i\eta = \begin{cases} a_i & \eta = \eta_i \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

where $\eta_i \in D_{x_1} \times ... \times D_{x_n}$ is an *nogood*, and a is a semiring value associated to constraint c_i with the assignment η.

The solution searching process can be expressed as the following minimization problem,

$$\min L(\eta, \boldsymbol{p}, \boldsymbol{g}) = \sum_{i=1}^{m} p_i g_i(\eta) \tag{2}$$

subject to

$$d_i \in D_{x_i}, \qquad \forall x_i \in V \tag{3}$$

where the objective function L consists of three arguments: η is an assignment, $\boldsymbol{p}$ is a penalty vector $(..., p_i, ...)$, and $\boldsymbol{g}$ is a constraint vector $(..., g_i, ...)$. Let $g_i : D_{x_1} \times ... \times D_{x_n} \rightarrow \{0, 1\}$ be a function mapping from a complete assignment to either 0 or 1. Intuitively, it returns 0 if the semiring value (α) of the best-so-far solution is less than or equal to $c_i\eta$, returns 1 otherwise.

The basic idea of our searching scheme is to descend in the original discrete variable space of η and ascend in the penalty space of $\boldsymbol{p}$, while the landscape depicted by $\boldsymbol{g}$ is modified when the objective function L becomes 0. Therefore, the search process can be described by formula 4 to 7.

$$\eta^{(k+1)} = \eta^{(k)} + \nabla(\eta^{(k)}, \boldsymbol{p}^{(k)}) \tag{4}$$

$$\boldsymbol{p}^{(k+1)} = \boldsymbol{p}^{(k)} + \boldsymbol{g}^{(k)}(\eta^{(k)}) \tag{5}$$

Formula 4 is used to define the moving directions of the search process, where $\nabla(\eta^{(k)}, \boldsymbol{p}^{(k)})$ is a descent direction for updating the current assignment $\eta^{(k)}$, and k is the iteration index. And the procedure for updating penalty vector is governed by formula 5.

$$\langle \alpha^{(k+1)}, \boldsymbol{g}^{(k+1)} \rangle = \begin{cases} \langle \alpha^{(k)}, \boldsymbol{g}^{(k)} \rangle & L(\eta^{(k)}, \boldsymbol{p}^{(k)}, \boldsymbol{g}^{(k)}) > 0 \\ \langle \bigotimes C\eta^{(k)}, \boldsymbol{g'} \rangle & \text{otherwise} \end{cases} \tag{6}$$

where $\boldsymbol{g} = \langle g_1, ..., g_k \rangle$, so that each time we penalize a constraint, the penalty can be an arbitrarily large integer, and

$$g'_i = \lambda\eta. \begin{cases} 1 & \eta = \eta^{(k)} \wedge c_i\eta^{(k)} \leq_S \alpha^{(k)} \\ g_i^{(k)}\eta & \text{otherwise} \end{cases} \tag{7}$$

Formula 6 and 7 is used to facilitate the problem transformation from SCSP to crisp CSP, which is called *Sinking* procedure. The detailed description of sinking operation will be presented in the next section. In addition, the initial setting for $\eta^{(0)}$, $\boldsymbol{g}^{(0)}$, $\alpha^{(0)}$ and $\boldsymbol{p}^{(0)}$ are chosen as follows:

- $\eta^{(0)}$ is chosen randomly.
- $g_i^{(0)} = \lambda\eta.0$ for all i.
- $\alpha^{(0)} = \bigotimes C\eta^{(0)}$.
- $\boldsymbol{p}^{(0)} = \mathbf{1}$, but $\boldsymbol{p}^{(0)}$ can be a vector consisting of any positive integers.

There are several properties of this scheme. We define the execution between the beginning of search process and the first time function $L = 0$ as *phase 1*, and the execution between the first time $L = 0$ to the second time $L = 0$ as *phase 2*, and so on. Theorem 1 states that a saddle point in the $\eta - \boldsymbol{p}$ space is reached at the end of each phase.

Theorem 1. *When $L(\eta^{(k)}, \boldsymbol{p}^{(k)}, \boldsymbol{g}^{(k)}) = 0$, the search is at a saddle point of the $\eta - \boldsymbol{p}$ space. That is,*

$$L(\eta^{(k)}, \boldsymbol{p}, \boldsymbol{g}^{(k)}) \leq L(\eta^{(k+1)}, \boldsymbol{p}^{(k)}, \boldsymbol{g}^{(k)}) \leq L(\eta, \boldsymbol{p}^{(k)}, \boldsymbol{g}^{(k)})$$

for all η that is a neighbor of $\eta^{(k+1)}$, and all possible $\boldsymbol{p}$.

Theorem 2 shows that at the end of each phase, preferred solution assignment will probably be found. And all the preferred solution assignment that have not been found will be preserved.

Theorem 2. *Denote* $S^{(k)} = \{\eta : L(\eta^{(k)}, \boldsymbol{p}^{(k)}, \boldsymbol{g}^{(k)}) = 0\}$ *and* $M^{(k)} = ub\{S^{(k)}\}$ *be the upper bounds of* $S^{(k)}$*, then either* $M^{(k+1)} = M^{(k)}$ *or* $M^{(k+1)} = M^{(k)} \cup \{\eta^{(k+1)}\}$.

Theorem 3 shows that either all preferred solution assignment will be found in a finite number of steps, or the algorithm will not terminate[1].

Theorem 3. *There exists a constant* l_0 *such that* $S^{(l_0)} \neq \emptyset$ *and* $M^{(l_0)} \neq \emptyset$*, and either for all* $l > l_0$*,* $S^{(l)} = \emptyset$ *and* $M^{(l)} = \emptyset$*, or for all* $l > l_0$*,* $S^{(l)} = S^{(l_0)}$ *and* $M^{(l)} = M^{(l_0)}$.

3.2 An Algorithmic Implementation

An implementation of the abstract scheme is described in this section. Basically, the search scheme can be realized by two components: *sinking* and *local search engine*. For the further enhancement, *shrinking* techniques can also be incorporated without violating the completeness of the search scheme. Algorithm 1 gives the pseudo-code of the complete local search framework. Note that an assignment will be collected in each iteration of the `while-loop` by the `collect_soution` function. It stores the preferred solution assignment in the partially order set $\mathcal{S}$.

Algorithm 1 Local Search framework for SCSPs

Let P be a SCSP;
Let α be the current best semiring value;
Let $\mathcal{S}$ be a collection of solutions;

Generate a random assignment η;
while not in termination-condition **do**
  $\alpha := \bigotimes C\eta$;
  sinking(P, α);
  shrinking(P);
  $\eta :=$ *local_search_engine*(P);
  collect_solution($\mathcal{S}$, η);
end while

The Sinking Component Intuitively, equation 6 and 7 is realized as sinking procedure. There is a property in the semiring framework that allows us to modify the discrete variable space η without changing the solution space of problem. Given an SCSP $\langle C, con\rangle$ and an assignment η with the semiring value $\alpha = \bigotimes C\eta$. For each constraint $c \in C$, we have $c\eta \geq \alpha$, where $c\eta$ denotes a semiring value generated by a constraint c with the assignment η.

[1] This is an intrinsic property of local search algorithm.

Theorem 4. [4] *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* $a \times b \leq_S a$*.*

Thus, for any tuple of a constraint $c \in C$ with semiring value a, such that $a \leq_S \alpha$. It is known to be α-inconsistent, since any semiring value combine with a will cause α-inconsistent (by Theorem 4). Therefore the semiring value a can be simply set to $\mathbf{0}$ without changing the solution space. It is referred as *sinking*, and we can say that the tuple is sunk. In other words, sinking is used to mark all the α-inconsistent tuples by setting its semiring value to $\mathbf{0}$. The meaning of this operation is the same as which described by equation 7, that changes the function $g'_i(\eta) = 1$ when the corresponding illegal constraint becomes α-inconsistent. The sinking operation will not discard any solution, since we are looking for a preferred solution and it sinks only the semiring value lower than or equal to α. This approach is similar to the "fill" algorithm which is proposed by Morris [17] and the "Great Deluge" algorithm by Dueck and Scheuer [7].

Note that the complexity of the sinking operation is linear in the size of the problem. We do not need to scan through all the tuples in the problem. Instead, a function s_i is attached to each constraint c_i for semiring value retrieval. The function is defined as follows,

$$s_i = \begin{cases} \mathbf{0} & \text{if } c\eta \leq \alpha, \\ a & \text{otherwise.} \end{cases} \tag{8}$$

where a is the semiring value associated to $c\eta$. And, equation 1 can be rewritten as following,

PSfrag replacements

$$c_i\eta = \begin{cases} s_i & \text{if } \eta = \eta' \\ 1 & \text{otherwise} \end{cases} \tag{9}$$

Example 2. Consider the Fuzzy CSP example in Fig. 1, we have an assignment $\langle$ X $\mapsto c$, Y $\mapsto b \rangle$, and $\alpha = \bigotimes C\eta$[X:=c,Y:=b] $= 0.2$. After executing the sinking procedure, six tuples are sunk. The resulting problem is shown in Fig. 2.

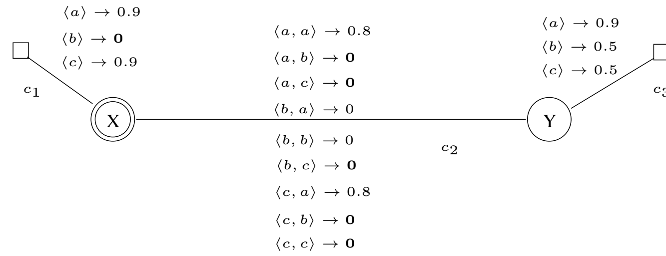

Fig. 2. Sinking with $\alpha = 0.2$.

The Shrinking Component The sinking procedure will generate huge plateau in the local search landscape, since a part of the original landscape will sink to be a flat land. The aim of the shrinking procedure is to eliminate plateaus, which can be done by removing inconsistent domain elements. With the properties of semiring $\times$ operator, for any semiring value $a \in A$, $a \times \mathbf{0} = \mathbf{0}$ (since $\mathbf{0}$ is an absorbing element of $\times$ operator). Therefore, tuples with semiring value $\mathbf{0}$ can be discarded. The domain eliminating task can be efficiently done by the node-consistency algorithm, which examines each domain element within all *unary* constraints and removes it if the tuple has semiring value $\mathbf{0}$. The pseudo-codes of node-consistency algorithm can be found in Algorithm 2. Note that various levels of consistency algorithm can also be applied in this framework, but experimental evident show that it is a trade-off between pruning power and computation overhead. For instance, we have implemented an arc-consistency for binary constraints, however, it does not show to be efficient in the $n \times (n-1)$-queens problem.

Algorithm 2 Node-Consistency

for all $v \in V$ **do**
  Let $\bar{C}_v = \{c \in C : v \in var(c) \wedge |var(c)| = 1\}$;
  Let $\bar{D}_v = \{d \in D_v : c \in \bar{C}_v, c\eta[v := d] = \mathbf{0}\}$;
  Remove all $d \in \bar{D}_v$
end for

PSfrag replacements

Example 3. Consider the Fuzzy example in Fig. 1, after applying sinking and shrinking procedure, domain element b of variable X is removed and all tuples of constraint c_2 that involving $\text{X} \mapsto b$ will no longer exist. The resulting problem is shown in Fig. 3.

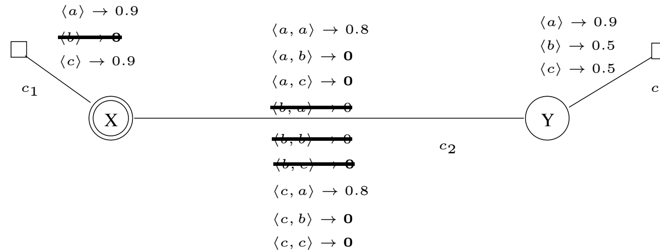

Fig. 3. An example of node consistency.

The Local Search Engine GENET is a neural network model for solving binary CSPs, which has been shown to be efficient and effective in solving hard

and large classical CSPs. In particular, we demonstrate the feasibility of adopting E-GENET [14] as the local search engine in the framework. E-GENET is an extension of GENET algorithm that is able to solve non-binary CSPs. Basically, E-GENET consists of two components: a network architecture and a convergence procedure. We construct a network representation of a classical CSP and update the network state by convergence procedure.

Fig. 4 shows the E-GENET network of the problem in Fig. 3. In our framework, all the α-inconsistent tuples are transformed as *inconsistent connections* of the network. Each connection is in the form of *hyper-arc* connection, which allows modelling of n-ary constraint. In addition, a cluster of *label node* represents a set of domain elements of the corresponding variable. Besides, each label node is associated with an *input* value, which represents the cost of the domain element. Therefore, the solution searching process can be referred as minimization of total cost of the network. Note that higher cost of the network state implies more conflicts among all variables of its assignment.

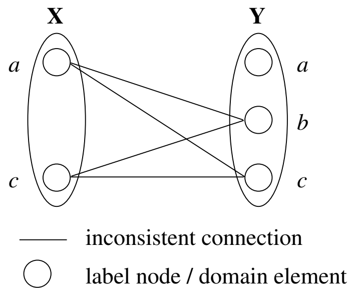

Fig. 4. An example of E-GENET network.

We recall the E-GENET convergence procedure as following. The initial state of the E-GENET network is randomly generated. In each convergence cycle, variable assignments will be updated basing on the min-conflict heuristics. In other words, it minimizes the conflicts among all variables. To avoid cycling, the variables are updated asynchronously in parallel. After a number of cycles, the network will settle in a stable state. If the valuation is conflict-free, a solution is obtained. Otherwise, we penalize the network connections of the current state. The convergence cycle keep going until a solution is found. Algorithm 3 outlines the E-GENET convergence procedure. It is not difficult to see that, the variables update step and penalization step is a realization of equation 4 and 5 respectively.

4 Experimental Results

We have built a SCSP Solver based on the proposed local search framework with E-GENET implementation and test it on three different problems, they

Algorithm 3 Convergence Procedure of E-GENET

```
randomly assign a value for each variable;
repeat
  update all variable nodes in parallel asynchronously until no repair has been done;
  if no conflict with the current assignments then
    terminate and returns solution;
  else
    penalizing inconsistent connections;
  end if
until termination-condition;
```

are $n \times (n-1)$-queens problem, modified latin square problem and random generated problem. The first two problems are structured, while the latter is non-structured. All these problems are modelled as fuzzy CSPs using the semiring structure $S_{FCSP} = \langle [0,1], max, min, 0, 1 \rangle$. The solver is implemented by using C++ language and the benchmarkings are performed on a Sun Blade 1000 workstation with Solaris 8 OS. All timing results presented are mean, and median in brackets, of 50 runs. And all of each are in unit of millisecond.

4.1 The $n \times (n-1)$-queens Problem

In the $n \times (n-1)$-queens problems [12], n queens are placed on an $n \times (n-1)$ chessboard so that there exists at least one pair of queens attacking each other. We define that it is better for any two queens attacking each other to be separated by a greater vertical distance. Formally, the problem can be formulated as follows. There are n variables $\{v_1, ..., v_n\}$, each with a domain of $\{1, ..., n-1\}$. The degree of satisfaction for the fuzzy constraint **noattack**(Q_{i_1}, Q_{i_2}) that prohibits two queens on row i_1 and row i_2 from attacking each other is 1 if these two queens do not attack each other, and $\frac{|i_1 - i_2| - 1}{n-1}$ if they do. Therefore, when the vertical distance between two queens increase, $|i_1 - i_2|$ and the degree of satisfaction also increase.

A user-specified lower bound of the acceptable satisfaction degree, α_o, is used to determine the algorithm termination. It will be terminated when $\alpha_o \leq \alpha$, where α is a semiring value of the latest solution. The timing results are shown in Table 1.

4.2 Modified Latin Square Problem

A Latin Square Problem is a $n \times n$ matrix with $a_{ij} \in \{1, 2, ..., n\}$ elements, such that entries in each row and column are distinct. In order to convert the original problem to be over-constrained, we modify the domain of each element in the matrix to become $\{1, 2, ..., n-1\}$. Thus, at least a pair of elements in each row and column are the same. The problem can be formulated as: n^2 variables $\{v_1, ..., v_{n^2}\}$, each with a domain of $\{1, 2, ..., n-1\}$. The degree of satisfaction for the row and column constraint can be modelled by **AllDiff**$(v_1, ..., v_k)$ constraint,

Problem	α_o 0.5	0.6	0.7	0.8	0.9
10×9-queens	1.67 (<10)	2.67 (<10)	3.67 (<10)	14.5 (<10)	–
20×19-queens	7.33 (10)	10.67 (10)	13.83 (10)	41.67 (45)	148 (115)
30×29-queens	22.83 (20)	30.33 (30)	44.33 (40)	85.83 (80)	136 (110)
40×39-queens	62.67 (60)	74.83 (70)	95.17 (90)	135.17 (120)	243.5 (200)
50×49-queens	130 (130)	154.83 (150)	180.67 (180)	225.67 (215)	357.67 (315)
60×59-queens	232.33 (230)	281.83 (280)	336.83 (330)	372.33 (360)	541 (490)
70×69-queens	390.5 (390)	467.17 (460)	527.67 (530)	624.5 (620)	862.33 (785)
80×79-queens	647.83 (640)	771.5 (770)	870.67 (860)	1047 (1030)	1321.83 (1255)

Table 1. Results on $n \times (n-1)$-queens problems

which is defined by $\frac{|\{(v_i,v_j)\in\{v_1,\ldots,v_k\}:i\neq j\wedge v_i\neq v_j\}|}{|\{(v_i,v_j)\in\{v_1,\ldots,v_k\}:i\neq j\}|}$. **AllDiff** is an n-ary constraint, which determines the degree of satisfaction according to the number of variable pairs are different. That is, more variable pairs are different, higher satisfaction degree. The results are shown in Table 2.

Problem	α_o 0.5	0.6	0.7	0.8	0.9
$n = 10$	<10 (<10)	<10 (<10)	<10 (<10)	0.6 (<10)	49.4 (30)
$n = 20$	0.2 (<10)	0.2 (<10)	0.2 (<10)	<10 (<10)	1.8 (<10)
$n = 30$	0.6 (<10)	0.4 (<10)	0.8 (<10)	0.4 (<10)	<10 (<10)
$n = 40$	1.6 (<10)	1.4 (<10)	1.2 (<10)	2.2 (<10)	1.4 (<10)
$n = 50$	2.4 (<10)	1.2 (<10)	2.4 (<10)	1.6 (<10)	3.4 (<10)
$n = 60$	3.4 (<10)	3.8 (<10)	4 (<10)	4 (<10)	4.6 (<10)
$n = 70$	6.8 (10)	6.2 (10)	6.6 (10)	6.4 (10)	4.8 (<10)
$n = 80$	9.2 (10)	9.4 (10)	8.4 (10)	7.6 (10)	9.2 (10)
$n = 90$	12.6 (10)	12 (10)	11.8 (10)	13.6 (10)	12.6 (10)
$n = 100$	11.8 (10)	17.6 (20)	16 (20)	17 (20)	17.2 (20)

Table 2. Results on the modified Latin Square Problems

4.3 Random Fuzzy CSPs

A set of randomly generated binary FCSPs will be used to test the solver. The problem size varies from 10 to 30 with step 5, and each variable has domain size 10. The network density of each random problem is 1.0 and all tuples has semiring value greater than zero, that means no inconsistent tuple initially. In addition, only one optimal solution exists in each problem, therefore there must be one solution has a semiring value 1.0. The aims of this experiment is to optimize the degree of satisfaction, instead of using α_o. The results are shown in Table 3.

Obviously, the results in Table 3 illustrate that our local search framework is shown to be effective and efficient to obtain an optimal solution (a preferred

	Number of Pruning	Degree of satisfaction	Runtime
10 variables	29	1.0	32.6 (30)
15 variables	42	1.0	43.6 (20)
20 variables	59	1.0	310.6 (230)
25 variables	50	1.0	1201.4 (745)
30 variables	65	1.0	9203.8 (3505)

Table 3. Results on Random FCSPs

solution assignment) in a reasonable computation time. Besides, the shrinking procedure is able to prune certain amount of domain elements that can help to improve the search performance.

5 Conclusion

A local search framework for SCSPs has been investigated. Our framework is parameterized by the semiring structure S, resulting in a family of algorithms. We formalize the local search framework in such a way similar to the Lagrangian search scheme, and give an algorithmic implementation of the scheme. The novelty of our approach transforms a problem forth and back between original SCSP and crisp CSP, and perform searching on the solution space of the crisp problem by a local search algorithm repeatedly. By applying the intensive property of $\times$ operator in semiring, we define the sinking operation, which is used to filter out all the inconsistent tuples in the problem during search. Incorporating the sinking operation and E-GENET model, problem transformation and solution searching can be done naturally with little overhead. Besides, shrinking technique is shown to be useful in our local search framework to enhance the performance. We demonstrate that it can be achieved by node-consistency algorithm effectively. Although it is an incomplete method, benchmarking results show its efficiency and feasibility to tackle structured and non-structured, binary and non-binary fuzzy CSPs.

The direction of our future work is two-fold. The first is to explore other possibilities for the local search engine, such as population based local search algorithm. It will be useful when looking for multiple solutions. The second is to formulate the min-conflict heuristics in semiring framework.

References

1. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and Valued CSPs: Basic Properties and Comparison. *Constraints*, 4:199–240, 1999.
2. S. Bistarelli and F. Rossi. About arc-consistency in semiring-based constraint problems. In *Proceedings of AI and Math Symposium*, 1998.
3. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over semirings. In *Proceedings of the 14th IJCAI*, pages 624–630, 1995.

4. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2), mar 1997.
5. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Soft concurrent constraint programming. In *Proceedings of ESOP*, number 2305 in LNCS, pages 53–67. Springer-Verlag, 2002.
6. Kenneth M. F. Choi, Jimmy Ho-Man Lee, and Peter J. Stuckey. A lagrangian reconstruction of GENET. *Artificial Intelligence*, 123(1-2):1–39, 2000.
7. Gunter Dueck. New optimization heuristics for the great deluge algorithm and the record–to–record travel. *Journal of Computational Physics*, 104, 1993.
8. H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: A probabilistic approach. In *Proceedings of the European Conference on Symbolic and Qualitative Approaches to Reasoning and Uncertainty (ESQUARU)*, volume 747, pages 97–104, New York, 1993.
9. Eugene C. Freuder. Partial constraint satisfaction. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, IJCAI-89, Detroit, Michigan, USA*, pages 278–283, 1989.
10. Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of the 9th National Conference on Artificial Intelligence, AAAI-91*, pages 227–233. AAAI Press/The MIT Press, 1991.
11. Yan Georget and Philippe Codognet. Compiling semiring-based constraints with clp(fd,s). In *Proceedings of Fourth International Conference on Principles and Practice of Constraint Programming, 1998*, 1998.
12. Q. Guan. *Extending Constraint Satisfaction Problem Solving Fuzzy Set Theory.* Ph.d. dissertation, Technical University of Vienna, 1994.
13. Hoong Chuin Lau. A new approach for weighted constraint satisfaction. *Constraints*, 7(2):151–165, 2002.
14. J. Lee, H. Leung, and H. Won. Extending genet for non-binary constraint satisfaction problems. In *Proceedings 7th International Conference on Tools with Articial Intelligence*, pages 338–342, 1995.
15. Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
16. Alan K. Mackworth. Constraint satisfaction. *Encyclopedia of AI (second edition)*, 1:285–293, 1992.
17. Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*, pages 40–45. AAAI Press/The MIT Press, 1993.
18. Francesca Rossi and I. Pilan. Abstracting soft constraints: Some experimental results. In *Proceedings of Joint Annual Workshop of the ERCIM Working Group on Constraints and the CoLogNET area on Constraint and Logic Programming*, Hungary, 2003.
19. Zsofi Ruttkay. Fuzzy constraint satisfaction. In *Proceedings 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994.
20. C.J. Wang and E.P.K. Tsang. Solving constraint satisfaction problems using neural-networks. In *Proceedings of IEE Second International Conference on Artificial Neural Networks*, pages 295–299, 1991.