ELSEVIER

Theoretical Computer Science 259 (2001) 287–305

Theoretical Computer Science

www.elsevier.com/locate/tcs

Dynamic algorithms for classes of constraint satisfaction problems ☆

Daniele Frigioni[a,b,*], Alberto Marchetti-Spaccamela[b], Umberto Nanni[b]

[a] *Dipartimento di Ingegneria Elettrica, Università di L'Aquila, Monteluco di Roio, I-67040 L'Aquila, Italy*

[b] *Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, I-00198 Roma, Italy*

Received January 1998
Communicated by G. Ausiello

Abstract

Many fundamental tasks in artificial intelligence and in combinatorial optimization can be formulated as a *Constraint Satisfaction Problem* (CSP). It is the problem of finding an assignment of values for a set of variables, each defined on a finite domain of feasible values, subject to a given collection of *constraints*. Each constraint is defined over a set of variables and specifies the allowed combinations of values as a collection of tuples. In general, the problem of finding a solution to a CSP is NP-complete, but in some cases it has shown to be polynomially solvable. We consider the dynamic version of some polynomially solvable constraint satisfaction problems, and present solutions that are better than recomputing everything from scratch after each update. The updates we consider are either *restrictions*, i.e., deletions of values from existing constraints and introduction of new constraints, or *relaxations*, i.e., insertions of values or deletions of constraints.

Keywords: Constraint Satisfaction Problem; Network of constraints; Dynamic algorithms; Consistency; Backtrack-free search; Amortized complexity

1. Introduction

Networks of constraints are a simple and powerful model which has been extensively used in the literature to represent the knowledge about problems whose solution

☆ Work partially supported by EC ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244. D. Frigioni is partially supported by fellowships program n. 201.15.11 of the Italian National Research Council (CNR).

* Corresponding author.

E-mail addresses: frigioni@dis.uniroma1.it (D. Frigioni), alberto@dis.uniroma1.it (A. Marchetti-Spaccamela), nanni@dis.uniroma1.it (U. Nanni).

0304-3975/01/$ - see front matter
PII: S0304-3975(00)00013-X

must satisfy simultaneously a certain number of constraints. More precisely, a *network of constraints* is defined by a set $V=\{v_1,v_2,\ldots,v_n\}$ of *variables*, where each v_i has a finite domain D_i of feasible values, and by a set $E=\{e_1,e_2,\ldots,e_m\}$ of *constraints*, each defining the collection of allowed values for a given set of variables, i.e., $e_i(v_{i_1},v_{i_2},\ldots,v_{i_k})\subseteq D_{i_1}\times D_{i_2}\times\cdots\times D_{i_k}$. The *Constraint Satisfaction Problem* (CSP) consists of determining a solution for a *network of constraints*, i.e., an assignment of values $\Upsilon=(v_1,v_2,\ldots,v_n)$ to the n variables of the problem such that all the constraints are satisfied, that is, for any constraint e_i, $(v_{i_1},v_{i_2},\ldots,v_{i_k})\in e_i$.

Several problems in artificial intelligence and in combinatorial optimization can be formulated as constraint satisfaction problems, as for example: image processing and image recognition, specification of software systems, satisfiability, graph coloring, scene labeling. The constraint satisfaction problem and the related solution procedures find also interesting applications in the field of constraint logic programming (see, e.g., [22]).

The constraint satisfaction problem has been first considered by Montanari [18], and has been widely studied in the literature due to its theoretical and practical importance in several fields (see, e.g., [3, 9, 13, 17–22]). It is easily shown to be NP-complete, also restricting the attention to binary CSPs (where any constraint is defined over a *pair* of variables). As an example, the graph coloring problem [10] can be formulated as a binary CSP, where each edge in the graph is associated to a constraint consisting of the collection of C^2–C pairs of allowed different colorings of the two endpoints with C colors: the resulting CSP is solvable if and only if the graph is colorable with C colors.

It is straightforward to describe a binary CSP in terms of a labeled *graph of constraints*, where the nodes coincide with the variables of the problem, and any constraint between two variables is represented by an undirected arc between the corresponding nodes in the graph of constraints. Each arc is labeled by the set of allowed pairs for the corresponding variables.

The standard technique used for finding a solution for a CSP is based on backtracking and in general it is exponential in the number of variables. For this reason a lot of efforts have been done in the past in order to find classes of problems with polynomial solutions. Most of these efforts have been devoted to find procedures whose purpose is to improve the backtracking efficiency. The main idea of this kind of approach is to remove local inconsistencies that cannot contribute to yield any global solution for the problem. These techniques, often known as *consistency techniques*, are usually used as preprocessing steps to make the subsequent backtrack search more efficient (see, e.g., [7–9, 16, 22]). Using *consistency techniques* some classes of problems have been characterized to have polynomial solutions. In the binary case, for instance, when the constraint graph is a tree, the corresponding constraint satisfaction problem becomes tractable, and polynomial solutions, based on consistency techniques, have been provided by Mackworth [16], Freuder [7], and Dechter and Pearl [4]. In particular, Dechter and Pearl [4] propose an optimal algorithm with a time bound of $O(nk^2)$, that corresponds to the size of the description of a binary CSP with n variables, and $m<n$

constraints, when each constraint is explicitly represented as an $k \times k$ matrix. Furthermore, in [13] sufficient conditions are provided under which a CSP can be solved by a fast parallel algorithm. On the other hand, some tentatives have been done to find approximate solutions for the constraint satisfaction problem in the general case (see, e.g., [14, 15]).

In some cases the knowledge of the problem defined by the set of constraints may vary along the time giving the problem dynamic features. For example, many problems in *image processing* and in *constraint logic programming* can be modeled as dynamic CSPs [17, 22]. Other applications of dynamic CSPs are known in the design of practical systems (see, e.g., [12]). During the computation, as the knowledge of the problem increases, it may be the case that the universe of allowed solutions shrinks more and more. In other cases, it might be the structure of the constraints themselves that is changing with time. In these situations it is interesting to handle efficiently *restrictions*, i.e., either deletion of tuples from existing constraints, or introduction of new constraints, or *relaxations*, i.e., insertions of tuples or deletions of constraints, while maintaining information on the satisfiability of the constraints and a current solution of the problem. The idea is to design algorithms that, after a change to the input of the problem, are able to update the current solution of the problem more efficiently than recomputing everything from scratch after each modification. If we can deal efficiently only with sequences of restrictions, or sequences of relaxations the problem is said to be *partially dynamic*; if sequences of both the operations can be handled efficiently we call the problem *fully dynamic*. Note that, if we deal only with sequences of relaxations, a non-trivial work is required only when we start from a CSP which is not satisfiable. On the other side, if we perform fully dynamic sequences of relaxations and restrictions, a single relaxation always requires a non-trivial work, because it changes the space of allowed solutions for further restrictions.

Previous results on the dynamic CSP have been proposed in [5, 11, 12, 17]. For instance, in [5, 11] algorithms are given to handle CSPs in which introductions and removals of variables are allowed during the search for a solution. Most of the previous dynamic solutions are not evaluated from a computational point of view, and are not compared with their static counterparts in the usual cost models (*worst case* and *amortized* [24]).

In this paper we deal with binary CSPs whose constraint graph is a tree (*tree-like* CSPs from now on), and present the following partially dynamic algorithms to maintain a solution of the CSP:

- An algorithm that requires $O(nk^2)$ total time over any sequence of pair deletions, thus achieving an optimal $O(1)$ amortized time bound on sequences of $\Theta(nk^2)$ operations. In this solution we follow an approach based on the *directional arc-consistency* property introduced by Dechter and Pearl [4], and in particular we implicitly maintain the set of all the admissible solutions for a CSP of the above kind.
- An algorithm that requires $O(nk(k + \log n))$ total time over sequences of general restrictions, including insertions of new constraints and pair deletions.

We also show how a sequence of relaxations, that is performed starting from a satisfiable CSP, can be trivially handled in O(1) worst case time per operation. Finally, we address a class of (non-binary) CSPs, that we call the *acyclic CSP*s, showing that the approach of Dechter and Pearl and our dynamization are valid also for this larger class.

All the proposed solutions require space bounds proportional to the size of the considered problem. On request, a solution (having size n) is returned in $O(n)$ worst case time.

The remainder of the paper is organized as follows. In Section 2 we introduce a more formal definition of CSP and a description of properties and methods exploited to deal with this problem. In Section 3 we present our approach to the dynamic binary tree-like CSP. In Section 4, we extend our results to a class of non-binary CSPs. Finally, in Section 5 we provide some concluding remarks and discuss some open problems.

2. Basic definitions and previous results

The CSP has been first introduced by Montanari [18] to capture and study in a single formal framework a wide set of problems arising in various fields of computer science and combinatorics. CSP can be regarded in terms of *hypergraphs*, a well studied model (see, e.g., [2]).

Definition 2.1. A *labeled hypergraph* H_L is a triple $\langle N, A, l\rangle$ where
- N is a finite set of *nodes*;
- A is a finite set of *hyperarcs* such that $A \subseteq \mathscr{P}^+(N)$, where $\mathscr{P}^+(N) = \{X \mid X \subseteq N \text{ and } X \neq \emptyset\}$;
- $l : A \to L$ is a *labeling function* with values in the set of *labels* L.

Sometimes it is convenient to partition the set of hyperarcs A according to their cardinality, defining $A_k = \{h \mid h \in A \text{ and } |h| = k\}$. In this case it is possible to consider the restrictions $l_k : A_k \to L_k$ of the labeling function l, with the resulting set of labels given by $L = \bigcup_{k=1}^{n} L_k$.

In the previous section we have given a first possible definition of a CSP as the problem of finding an assignment $\Upsilon = (v_1, v_2, \ldots, v_n)$ of values for a set of variables V satisfying a given set of constraints E. A CSP or, more precisely, the *network of constraints* (V, E) which it is based upon, can be modeled by means of a *constraint hypergraph*.

Definition 2.2. The *constraint hypergraph* of a *network of constraints* (V, E) is a labeled hypergraph $H_L = \langle N, A, l\rangle$ where
- $N \equiv V$;
- $A = \{(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \mid e_i(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \text{ is a constraint in } E\}$;
- any hyperarc in A is labeled by l with the corresponding constraint in E. In particular, $l_k : A_k \to \mathscr{P}(U^k)$, where U is a finite domain for all the variables in V.

A *solution* for a CSP is an assignment of values $\Upsilon = (v_1, v_2, \ldots, v_n)$ to the n variables of the network that satisfies all the constraints, i.e., such that, for any hyperarc $h = (v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \in A$, $(v_{i_1}, v_{i_2}, \ldots, v_{i_k}) \in l(h)$. A CSP is satisfiable if it admits at least a solution.

In this paper we will be primarily concerned with binary CSPs, i.e., CSPs where each constraint involves at most two variables. The corresponding network is an (ordinary) undirected graph, and will be called the *constraint graph*. A binary constraint $e(v_i, v_j)$ between v_i and v_j is a subset of the cartesian product of their domains, i.e., $e(v_i, v_j) \subseteq D_i \times D_j$.

A binary constraint $e(v_i, v_j)$ can be represented using a $k \times k$ matrix denoted as $R_{i,j}$, where $k = \max\{|D_i|, |D_j|\}$, and the entries 0 and 1 indicate forbidden and permitted pairs of values, respectively. In what follows we refer both to a constraint and to the corresponding binary matrix as $R_{i,j}$, when the context is not ambiguous. A constraint $R_{i,j}$ is called *universal* if $R_{i,j} = D_i \times D_j$. If there is no constraint between two variables v_i and v_j of a binary CSP, then all the pairs of possible values for v_i and v_j are allowed, i.e., $R_{i,j}$ is universal.

A trivial (exponential time) algorithm to find a solution for a CSP is based on *backtracking* (see, e.g., [7]). Backtracking occurs when an instantiation chosen during a backtrack search, consistent with all previous choices, must be discarded later in the search when no consistent instantiation can be made for a variable at a lower level in the backtrack tree.

In [7, 8, 16, 18] it is shown that, if the CSP satisfies certain conditions on the structure of the constraint graph and on the values within the constraints, the problem turns out to be polynomially solvable. Following their approach, which deals with directed graphs, we need to consider any binary constraint $R_{i,j}$ as labeling both the two directed arcs (v_i, v_j) and (v_j, v_i), whose matrix representations satisfy $R_{i,j} = R_{j,i}^{\mathrm{T}}$.

Definition 2.3. Given a constraint graph $G = (N, A)$, a directed arc $(v_i, v_j) \in A$ is *consistent* iff, for any value $x \in D_i$, there is a value $y \in D_j$ such that $(x, y) \in R_{i,j}$. A constraint graph is *arc-consistent* if all its directed arcs are consistent.

In [6] Freuder defined the property of *k-consistency* as a generalization of arc-consistency. This property specifies that if we choose values for any $k-1$ variables that satisfy all the constraints defined only on these variables, together with an arbitrary kth variable, it is possible to find a value for the kth variable such that all the constraints on the k variables will be satisfied by the k values taken together. *Strong k-consistency* is k'-consistency for all $k' \leqslant k$. If a constraint graph is k (strong) consistent then k is called the *level* of (strong) consistency of the graph. Note that, arc consistency corresponds to 2-consistency.

An *ordered constraint graph* (G, d) is a constraint graph G in which the nodes are given a total order $d = \langle v_1, v_2, \ldots, v_n \rangle$. The ordering d corresponds to the order in which the variables are chosen for instantiation in the backtrack search. An arc (v_i, v_j)

of G is *directed along* d if $i<j$ with respect to d. In what follows, without loss of generality, we assume that the constraint graph G is connected. In the case in which this assumption does not hold, we can deal with each connected component of G as a separate problem.

Definition 2.4. The *width of a node* v_j in an ordered constraint graph (G,d) is the number of links (v_i,v_j) such that $i<j$ with respect to d. The *width of an ordering* is the maximum width of all nodes. The *width of a constraint graph* is the minimum width of all the orderings of the graph.

Freuder in [7] provided an $O(n^2)$ algorithm for finding both the width of a graph and the ordering corresponding to that width. He further showed that a constraint graph is a tree if and only if it has width 1.

A lot of attention has been dedicated in the literature to *backtrack-free* searches for finding a solution of a CSP, i.e., searches that terminate without making backtracking steps. The obvious interest in such an approach is due to the fact that a backtrack-free search can provide a solution in linear time. Freuder [7] showed the following relation between width and consistency that guarantees a backtrack-free search for the solution.

Theorem 2.1 (Freuder [7]). *There exists a backtrack-free search for a binary CSP if the level of strong consistency is greater than the width of the constraint graph.*

Subsequently, Dechter and Pearl noticed that full arc-consistency, i.e., arc consistency along every direction, is more than what is actually required to achieve a backtrack-free search for a width-1 CSP [4], i.e., for a tree-like CSP. The arc-consistency is required only with respect to a single direction, i.e., the one in which a backtrack search selects variables for instantiation. With this simple observation they motivate the following definition.

Definition 2.5. An ordered constraint graph (G,d) is *d-arc consistent* if all the arcs directed along d are consistent. A constraint graph is *directional arc-consistent* if it is arc-consistent along a fixed direction.

They also proposed the algorithm called `Dac` and shown in Fig. 1, for achieving directional arc-consistency for any ordered constraint graph (the order $d=\langle v_1,v_2,\ldots,v_n\rangle$ is assumed). Procedure `Revise`(v_i,v_j), given in [16], deletes values from the domain D_i in $O(k^2)$ steps in the worst case, where $k=\max\{|D_i|,|D_j|\}$, until the directed arc (v_i,v_j) is made consistent. Dechter and Pearl in [4] proved the following result.

Theorem 2.2 (Dechter and Pearl [4]). *Algorithm* `Dac` *finds a solution for a tree-like CSP in* $O(nk^2)$ *steps, and this is optimal.*

```
    procedure Dac(G, d)
1.  begin
2.    for i = n to 1 do
3.      for each arc (v_i, v_j) in G with i ⩽ j w.r.t. d do
4.        Revise(v_i, v_j)
5.  end
```

Fig. 1. An algorithm for achieving directional arc-consistency.

3. Dynamic binary CSP

In this section we describe partially dynamic algorithms for maintaining a solution to a tree-like CSP, handling either sequences of *restrictions*, or sequences of *relaxations*. The idea behind this new approach is to maintain the property of d-arc consistency, which guarantees a backtrack-free search for the solution for tree-like CSPs. Hence, we assume that, before any sequence of update operations, the CSP is d-arc consistent for some given order d. In what follows we will first analyze the case of restrictions in Section 3.1, then we will describe briefly how to deal with the simpler case of sequences of relaxations in Section 3.2, and finally we will point out the difficuties of handling fully dynamic sequences of restrictions and relaxations in Section 3.3.

3.1. Restrictions

Given a tree-like CSP that is d-arc consistent, our goal is to maintain a solution for the CSP, or equivalently the property of d-arc consistency that guarantees a backtrack-free search for the solution, while performing arbitrary sequences of restrictions of the following kind:

1. pair deletion from preexisting constraints;
2. arc (constraint) insertion.

We remark that the two operations above are homogeneous. In fact, if we insert an arc (v_i, v_j) then, before the insertion, $R_{i,j}$ is universal, i.e., there is no constraint between v_i and v_j; as a consequence of the insertion of (v_i, v_j) a *new* constraint $R'_{i,j}$ between v_i and v_j is added to the CSP. $R'_{i,j}$ can be computed by simply deleting from $R_{i,j}$ all the pairs not admitted by the new constraint. On the other hand, to ensure a backtrack-free search for the solution for a tree-like CSP after an arc insertion, it is not sufficient to maintain only the property of d-arc consistency, but, by Theorem 2.1, it is necessary to maintain also the property of having width equal to 1. Hence, we are required to verify that the constraint graph obtained after each insertion is still a forest.

In what follows, we will first describe an algorithm for handling pair deletions, and then we will use this algorithm as a subprogram of the algorithm for handling arc insertions.

3.1.1. Pair deletions

A trivial solution of the pair deletion problem consists in applying after each deletion the $O(k^2)$ off-line algorithm `Revise`, given in [16], to all the arcs involved in the modification, obtaining an $O(nk^2)$ worst-case time bound for each deletion. For simplicity, we express bounds in terms of the parameter k, denoting the maximum size of any domain for the variables in V, i.e., $k = \max_{v_i \in V}\{|D_i|\}$.

In this section we propose a data structure and a dynamic algorithm that allow us to maintain the d-arc consistency for a tree-like CSP in $O(nk^2)$ total time, under an arbitrary sequence of pair deletions, i.e., in $O(1)$ amortized time per operation over sequences of $\Theta(nk^2)$ pair deletions.

The only case in which a pair deletion can modify the directional arc-consistency of an ordered tree-like network is when we delete a pair (a,b) from a constraint $R_{i,j}$ and the pair (a,b) is the only one that guarantees consistency of the arc (v_i,v_j) for the value $a \in D_i$. In other words when, before the deletion of (a,b), for any $b' \in D_j$ such that $b' \neq b$, $R_{i,j}(a,b') = 0$.

Lemma 3.1. *Let $(a,b) \in R_{i,j}$ be a pair such that for each value $b' \in D_j$, with $b' \neq b$, $R_{i,j}(a,b') = 0$. The deletion of (a,b) from $R_{i,j}$ requires the deletion of value a from the domain D_i, in order to restore arc consistency for arc (v_i,v_j).*

Proof. It is sufficient to observe that, after the deletion of (a,b), the value $a \in D_i$ satisfies the following condition: for any value $x \in D_j$, $R_{i,j}(a,x) = 0$, i.e., the arc (v_i,v_j) is not consistent according to Definition 2.3. Thus, removing the value a from D_i restores the consistency of arc (v_i,v_j). □

Let us call *critical* both a pair complying the hypothesis of Lemma 3.1, and its deletion. After the deletion of a critical pair $(a,b) \in R_{i,j}$, we have that restoring arc consistency implies the "deletion" of value a from D_i, which means that no solution for the considered CSP can assign $v_i = a$. This, in turn, implies that a pair (z,a) in a constraint $R_{h,i}$, for every z and h, cannot contribute to build up a solution for the CSP, or to support consistency of arcs. These considerations lead to the ideas behind our algorithms: to support *explicit* pair deletions, requested by a user, and *implicit* pair deletions, performed by the algorithms in order to restore consistency.

In the rest of this section we first describe our data structures, and then we provide the details of the algorithms.

The data structure As previously remarked, after a pair deletion the domains of variables can only shrink. Let us denote as $D_i(0)$ $(i = 1,2,\ldots,n)$ the *initial domain* of variable v_i. For any i, the following inequalities trivially hold: $|D_i| \leqslant |D_i(0)| \leqslant k$.

For each variable v_i we maintain the current content of D_i in a binary vector R_i, indexed by the element of the initial domain $D_i(0)$, and defined as follows:

$$R_i(x) = \begin{cases} 1 & \text{if the value } x \text{ is still in } D_i, \\ 0 & \text{if the value } x \text{ has been deleted from } D_i. \end{cases}$$

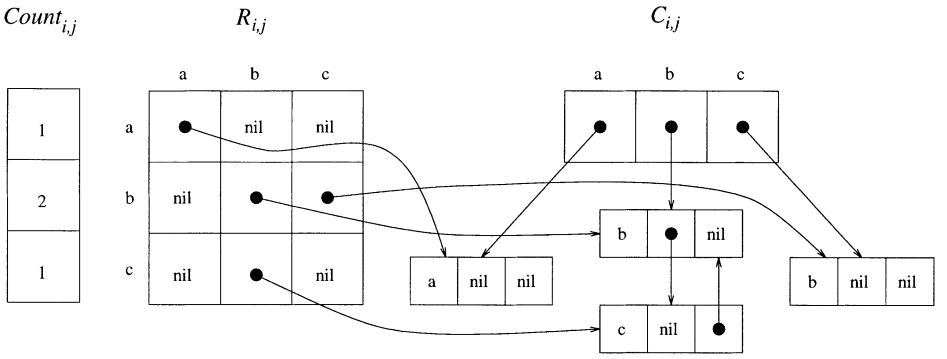

Fig. 2. An example of the data structure used to represent a generic constraint $R_{i,j}$. Here, $D=\{a,b,c\}$, $k=3$, and the illustrated constraint is $R_{i,j}=\{(a,a),(b,b),(b,c),(c,b)\}$.

For each v_i, we maintain also an integer d_i representing the current cardinality of D_i.

A binary constraint $R_{i,j}$ is represented as an array of columns, any column being handled both as an array and as a set. In other words, for each value $y\in D_j$, we maintain the set of values $x\in D_i$ such that $(x,y)\in R_{i,j}$. The sets will be handled by a simple technique shown in [23], that allows us to perform pair deletions in constant amortized time, as we will in the proof of Theorem 3.3.

The data structure required for representing the generic constraint $R_{i,j}$ includes a $k\times k$ matrix of pointers, denoted $R_{i,j}$ itself, and a k-cardinality array $C_{i,j}$ of pointers. The generic component $C_{i,j}(y)$ is a pointer to a double-linked list containing all the values $x\in D_i$ such that $(x,y)\in R_{i,j}$. The definition of matrix $R_{i,j}$ is the following:

$$R_{i,j}(x,y)\begin{cases}\text{points to the item } x \text{ in the list } C_{i,j}(y) & \text{if the pair } (x,y)\in R_{i,j},\\ \textbf{nil} & \text{otherwise.}\end{cases}$$

Finally, we maintain another array of size k, denoted $Count_{i,j}$, where the generic component $Count_{i,j}(x)$ is the number of elements that are not **nil** in the xth row of $R_{i,j}$. In Fig. 2 we provide an example to illustrate the data structure described above.

The algorithm. The data structure described above can be initialized in $O(nk^2)$ time, because we have $O(n)$ constraints, each of size k^2. As we previously remarked, when we delete a critical pair (a,b) from $R_{i,j}$, by Lemma 3.1, we must delete the value a from D_i; as a consequence, all the pairs of the kind $(*,a)$ must be deleted from every constraint $R_{h,i}$. In fact, these pairs are now not necessary for maintaining the directional consistency of both arc (v_h,v_j) and the whole network. This corresponds to empty the whole list pointed by $C_{h,i}(a)$ in the constraint $R_{h,i}$.

The algorithm that we propose for handling pair deletions is implemented by two procedures, whose behavior is summarized in the following:

1. Procedure `Explicit_Deletion`$(a,b,R_{i,j})$ performs explicit pair deletions; in addition it handles in constant time the deletion of a pair that has been previously

```
    procedure Explicit_Deletion (a, b, R_{i,j})
1.  begin
2.    if R_{i,j}(a, b) ≠ nil {i.e., if the pair (a, b) has never been deleted}
3.      then Implicit_Deletion (a, b, R_{i,j})
4.  end
```

Fig. 3. Procedure Explicit_Deletion.

```
    procedure Implicit_Deletion (a, b, R_{i,j})
1.  begin
2.    delete a from C_{i,j}(b)
3.    R_{i,j}(a, b) := nil
4.    decrement Count_{i,j}(a)
5.    if Count_{i,j}(a) = 0
6.      then begin {a critical deletion occurs}
7.             R_i(a) := 0 {i.e., remove a from D_i}
8.             if D_i = ∅
9.           then report ("CSP unsatisfiable")
10.          else
11.            for each arc (v_h, v_i)
12.              for each z ∈ C_{h,i}(a) do
13.                Implicit_Deletion (z, a, R_{h,i})
14.        end {d-arc consistency restored}
15. end
```

Fig. 4. Procedure Implicit_Deletion.

deleted, i.e., a pair (a,b) such that $R_{i,j}(a,b) = \textbf{nil}$. Otherwise, i.e., if $R_{i,j}(a,b) \neq \textbf{nil}$, it calls Procedure Implicit_Deletion with the same parameters.

2. Procedure Implicit_Deletion$(a,b,R_{i,j})$ is in charge to delete pairs from the data structure. The deletions of non-critical pairs is performed in constant time, while a critical deletion determines recursive calls to Procedure Implicit_Deletion in order to restore arc consistency.

The two procedures are shown in Figs. 3 and 4, respectively, and are described in the following.

The deletion of a pair (a,b) from a binary constraint $R_{i,j}$ is obtained by calling Procedure Explicit_Deletion$(a,b,R_{i,j})$ which, in turn, makes a call to Implicit_Deletion$(a,b,R_{i,j})$ in the case that pair (a,b) was not previously deleted (explicitly or implicitly). Procedure Implicit_Deletion initially deletes the item a pointed to by $R_{i,j}(a,b)$ in the list appended to $C_{i,j}(b)$, sets $R_{i,j}(a,b)$ to **nil**, and then decreases $Count_{i,j}(a)$ in order to record the deletion.

Now, the algorithm must verify if the deletion modifies d-arc consistency for the arc (v_i,v_j), i.e., if the pair (a,b) was critical for constraint $R_{i,j}$. This can be easily done by testing whether $Count_{i,j}(a)$ is 0 or not. If $Count_{i,j}(a) \neq 0$ the property is maintained,

otherwise from Lemma 3.1 the value a is deleted from D_i setting $R_i(a) = 0$. Each of the operations listed above requires O(1) worst-case time. If the domain D_i becomes empty ($d_i = 0$ after the deletion), the problem becomes unsatisfiable, otherwise the algorithm must try to restore d-arc consistency for any arc (v_h, v_i), if possible, by recursive calls to Procedure `Implicit_Deletion`.

We observe that the algorithm terminates, because the only case in which this could not be true is when it enters in the nested loop to propagate the previous updates (lines 11–13). In this case the total number of recursive calls to Procedure `Implicit_Deletion` is $O(nk^2)$ (i.e., the total number of pairs in the data structure), that is finite.

The following theorem proves the correctness of the described procedures.

Theorem 3.2. *Given a binary CSP with width* 1 *which is d-arc consistent, then after the execution of* `Explicit_Deletion`$(a, b, R_{i,j})$ *the CSP is d-arc consistent.*

Proof. Only a critical deletion may possibly destroy directional consistency of the network. The directional consistency of arc (v_i, v_j) in guaranteed in one of the following ways:

1. If the pair (a, b) is not critical for $R_{i,j}$ with respect to the value $a \in D_i$, then its deletion cannot modify the property.
2. Otherwise, if the pair (a, b) is critical for $R_{i,j}$, then by Lemma 3.1 its deletion determines the elimination of the value a from D_i, to restore the property.

In the second case the deletion could modify the consistency of some arc (v_h, v_i). The consistency of this arc is however restored by the deletion of all the pairs of the kind $(*, a)$ from the matrix $R_{h,i}$, by recursive calls to Procedure `Implicit_Deletion`. For each of these pairs we can reapply the reasoning used above.

In this way we always guarantee the consistency of each arc in the tree-like network, and the theorem is proved. □

We have already given a worst-case analysis for the execution of a single-pair deletion; now we want to show that the data structure and the algorithm proposed are efficient in amortized sense [24]. In particular, we consider an arbitrary sequence of explicit deletions, i.e., calls to Procedure `Explicit_Deletion`, and evaluate the time required to handle the whole sequence.

Theorem 3.3. *The total time required to maintain d-arc consistency of a width-*1 *CSP, under an arbitrary sequence of pair deletions, is* $O(nk^2)$, *which gives* O(1) *amortized time per deletion over any sequence of* $\Theta(nk^2)$ *operations.*

Proof. Constant worst-case time is required to perform each call to `Explicit_Deletion`, without considering the cost needed for the execution of `Implicit_Deletion`. The data structure contains at most nk^2 non-nil pairs. In each

execution of Procedure `Implicit_Deletion` at least one pair becomes nil (line 3), and the procedure never reconsiders a pair which is nil, i.e., that has been previously deleted (see line 2 of `Explicit_Deletion` and lines 10–13 of `Implicit_Deletion`). In fact, `Implicit_Deletion` is called in line 2 of `Explicit_Deletion` only when the deleted pair is not nil; in lines 10–13, for each considered arc (v_h, v_i), `Implicit_Deletion` deletes only the pairs (z, a) in $R_{h,i}$ such that z belongs to the list $C_{i,j}(a)$, i.e., only the pairs that are not nil. This implies that over any sequence of pair deletions at most nk^2 pairs can be deleted, and $O(nk^2)$ total time is required to perform the deletions. If the length of the sequence is $\Theta(nk^2)$ then we obtain $O(1)$ amortized time per deletion. □

3.1.2. Arc (constraint) insertions

In this section we propose an extension of the data structure used in the previous section, and a dynamic algorithm that allow us to maintain a solution to a tree-like CSP in $O(nk(k + \log n))$ total time, under an arbitrary sequence of arc insertions and pair deletions.

A generic insertion of an arc (x_i, x_j) is treated here as a sequence of deletions from the universal constraint $R_{i,j}$, of all the pairs that are not allowed after the insertion.

To ensure a backtrack-free search for the solution for a tree-like network under a sequence of arc insertions, it is not sufficient to maintain only the property of d-arc consistency, but, by Theorem 2.1, it is necessary to maintain also the property of having width equal to 1. Hence, we are required to verify that the constraint graph obtained after each insertion is a forest. This can be easily done by using the fact (shown in [7]) that a graph has width 1 if and only if it is a forest. It follows that, given a forest F, the insertion of an arc (x_i, x_j) maintains the property of having width 1 for F if and only if the nodes x_i and x_j belong to different trees in the forest.

The data structure: In order to maintain the properties of d-arc consistency and width 1 for a tree-like CSP subject to an arbitrary sequence of arc insertions, we use an extension of the data structure described in Section 3.1.1. Precisely, we use the following additional data structures:

- A double-linked list L of pointers to the roots of the trees in the forest.
- A record for each node in the forest, containing the following fields: a pointer p to its parent in the forest; a pointer *children* to the list of its children in the corresponding tree; a pointer *list*, which is significant only for the root nodes, to the item in L that points to it.

Using this extended data structure we can maintain efficiently the property of d-arc consistency in the same way described in Section 3.1.1. The property of width 1 is maintained as described in the following. We denote as $r(x)$ the root of the tree in the forest containing node x, as $T(x)$ the tree itself, and as $|T(x)|$ its size, i.e., the number of nodes it contains. Furthermore, we assume that each node in the forest knows which is the root of the tree in which it is contained, and that the root of each tree in the forest knows the size of the corresponding tree.

```
    procedure Insert(x_i, x_j: node; R_{i,j}: constraint)
1.  begin
2.    if r(x_i) ≠ r(x_j) then
3.      begin
4.        Update(x_i, x_j, R_{i,j})
5.        delete the item in L which points to x_i
6.      end
7.      else report ("width(G) > 1")
8.  end
```

Fig. 5. Procedure Insert.

The algorithm: After the insertion of an arbitrary arc (x_i, x_j), our algorithm performs the following operations:

1. If $r(x_i) = r(x_j)$ then the algorithm halts because the width of the actual constraint graph becomes 2;
2. otherwise, i.e., if $r(x_i) \neq r(x_j)$, the smallest tree between $T(x_i)$ and $T(x_j)$, say $T(x_i)$, is "reversed"; that is, the direction of all arcs in the path from x_i to $r(x_i)$ is reversed, thus making x_i the new root of $T(x_i)$, and the additional structures involved in this operation are properly updated;
3. the directional consistency of the arcs involved in this process is suitably restored, in order to maintain the property on the whole network; in addition, for every node in $T(x_i)$, the information on the root of the tree in which it is contained is properly updated;
4. $p(x_i)$ is set to point to x_j and the item in the double-linked list L which points to x_i is deleted;
5. the universal constraint $R_{i,j}$ is created, and a sequence of deletions of the pairs that are not allowed by arc (v_i, v_j) is performed;
6. the performed updates are propagated to all the arcs in the path from x_j to $r(x_j)$.

Our algorithm is structured in the form of three procedures called Insert, Update and Reverse. Procedure Insert$(x_i, x_j, R_{i,j})$ in Fig. 5 first checks whether $r(x_i) \neq r(x_j)$. In such a case the insertion is allowed and the algorithm properly call Procedure Update described in Fig. 6. In the other case the algorithm halts because the insertion of (x_i, x_j) introduces a cycle in the current constraint graph. This implies that the property of having width 1 for the considered CSP, is maintained after an arc insertion.

In the proposed procedures the union of the domains of all the variables is denoted as D. Procedure Update(x_i, x_j, R) calls Procedure Reverse only if the node x_i is not a root. Arc (x_i, x_j) is inserted in constant time by setting $p(x_i)$ to x_j, and inserting x_i in the list of children of x_j. Then, the procedure creates in $O(k^2)$ time the universal matrix $R_{j,i}$, deletes from $R_{j,i}$ all the pairs not allowed after the insertion, and possibly propagates these updates to the arcs on the path between x_j and $r(x_j)$. This is done in $O(k^2|T(x_j)|)$ total time by using Procedure Implicit_Deletion to restore the directional consistency of the encountered arcs.

```
     procedure Update(x_i, x_j: node; R: constraint)
1.   begin
2.     rename x_i and x_j in such a way that |T(x_i)| ≤ |T(x_j)|
3.     if x_i ≠ r(x_i) then
4.       Reverse(x_i, p(x_i), p(p(x_i)))
5.     p(x_i) := x_j
6.     R_{j,i} := R
7.     insert the item x_i in the list children(x_j)
8.     for each a ∈ D do
9.       if a ∉ D_i then
10.        for each z ∈ C_{j,i}(a) do
11.            Implicit_Deletion(z, a, R_{j,i})
12.  end
```

Fig. 6. Procedure Update.

```
     procedure Reverse(x_i, x_j, x_h: node)
1.   begin
2.     if x_j ≠ x_h then Reverse(x_j, x_h, p(x_h))
3.     p(x_j) := x_i
4.     p(x_i) := x_i
5.     update the lists children(x_i) e children(x_j)
6.     for each a ∈ D do
7.       if a ∉ D_j do
8.         for each z ∈ C_{i,j} do
9.             Implicit_Deletion(z, a, R_{i,j})
10.  end
```

Fig. 7. Procedure Reverse.

Procedure Reverse, shown in Fig. 7, reverses the direction of all the arcs on the path from x_i to $r(x_i)$ and, for each of them, ensures the consistency in the new direction in $O(k^2|T(x_i)|)$ worst-case total time, by using Procedure Implicit_Deletion.

By the considerations above we have that the total time required for inserting an arc (i, j) between two trees T_i and T_j in a forest is $O(k^2 \cdot (|T_i| + |T_j|))$. Furthermore, the correctness of the algorithm for arc insertions is a straightforward consequence of the above discussion, and of the correctness of Procedure Implicit_Deletion.

Now, we will prove that the proposed data structures and algorithms are efficient in amortized sense [24]. In particular, we will prove that the total time required to perform an arbitrary sequence of arc insertions, each of size $O(k^2)$, is $O(nk(k + \log n))$. This is better than the $O(n^2k^2)$ bound which we would obtain by the above worst-case analysis.

Theorem 3.4. *The total time required to maintain the properties of d-arc consistency and width* 1 *for a binary CSP, under an arbitrary sequence of* $\Theta(n)$ *constraint insertions, each of size* $O(k^2)$*, starting from an empty constraint graph and leaving the graph acyclic, is* $O(nk(k+\log n))$.

Proof. Since arc insertions delete pairs (by calling `Implicit_Deletion`), any sequence of arc insertions will accumulate a cost of $O(nk^2)$ from pair deletions. This is the time required by the algorithm to maintain the property of d-arc consistency of the whole network under the sequence of arc insertions.

To enforce width 1 for the constraint graph, our algorithm must perform in the worst-case one `Reverse` operation, for each arc insertion. During a sequence of $\Theta(n)$ arc insertions, since we reverse always the tree with minimum size, each arc can be reversed at most $O(\log n)$ times. In fact, any time that an arc (x, y) is reversed, it belongs to a tree whose size is at least twice the size of the tree that contained (x, y) the last time it was reversed. During a sequence of $\Theta(n)$ arc insertions the size of a tree can double at most $O(\log n)$ times, that leads to a total of $O(n \log n)$ arc reversals during the sequence.

During each arc traversal Procedure `Reverse` performs a subset of the $O(nk^2)$ possible pair deletions. Furthermore, for each arc traversal Procedure `Reverse` executes the loop starting at line 6 which requires $O(k)$ time to be executed, without considering the cost of the calls to `Implicit_Deletion` at lines 8–9. So the total time required for maintaining the width 1 property of the constraint graph is $O(kn \log n)$.

Finally, the total time required to perform a sequence of $\Theta(n)$ arc insertions, each of size $O(k^2)$, is $O(nk^2 + kn \log n) = O(nk(k+\log n))$. This gives $O(\max\{\log n/k, 1\})$ amortized time for every arc insertion in the sequence. □

We know that inserting an arc and deleting a pair from a tree-like CSP are homogeneous operations, in the sense that these operations can only reduce the set of solutions of the current CSP. For this reason, we can perform sequences of arc insertions and pair deletions on a constraint graph by using the same data structure described in Section 3.1.2. The next theorem follows directly from Theorems 3.3 and 3.4.

Theorem 3.5. *The total time required for maintaining the properties of d-arc consistency and width* 1 *for a binary CSP under a sequence of arc insertions and pair deletions is* $O(nk(k+\log n))$.

3.2. Relaxations

Given a tree-like CSP that is d-arc consistent, we want to maintain a solution for the CSP, or equivalently the property of d-arc consistency that guarantees a backtrack-free search for the solution, while performing arbitrary sequences of relaxations of the following kind:

1. pair insertion in preexisting constraints;
2. arc (constraint) deletion.

We remark that deleting an arc (v_i, v_j) is equivalent to inserting in $R_{i,j}$ all the pairs currently not contained in that constraint; in this way $R_{i,j}$ becomes a universal constraint. Both the operations cannot modify the d-arc consistency of the constraint graph; in fact, by Definition 2.3 the insertion of a pair in a preexisting constraint does not violate that property. Furthermore, the deletion of an arc cannot destroy the property of the constraint graph of having width 1. For the above reasons, both the operations (inserting a pair in a preexisting constraint or deleting an arc) can be trivially handled in constant worst-case time.

Problems arise if we add any pair in a preexisting constraint of a network which is not satisfiable (or if we perform an intermixed sequence of restrictions and relaxations). In fact, in this case it is necessary to consider the newly introduced values that could yield a satisfiable network. As it will be shown in the next section, this makes the fully dynamic problem harder.

3.3. The fully dynamic problem

In this section we show that our data structure can accommodate a mixed sequence of insertions and deletions of tuples from constraints. Each pair insertion can be handled in constant time. The time required to process any subsequence of pair deletions can be $O(nk^2)$. This can be done by performing *lazy* pair insertions. This approach requires an operation that we call *restore*, consisting in applying the off-line $O(nk^2)$ algorithm to recompute from scratch the contents of the data structures. Some details follow:

- when an explicit deletion occurs, the corresponding entry in the binary matrix representation is *marked*: in this way both the original constraints (the unmarked items) and the pruned version updated by the algorithm (the non-null items) are represented in the data structures;
- when the problem becomes unsatisfiable, and at least one insertion has been performed since the last *restore* occurred, a new *restore* is performed.

In this way the newly introduced values are considered only when the old ones are not sufficient to build a solution for the considered CSP. Since the time required by deletions between two subsequent *restore* operations is at most $O(nk^2)$, computing a *restore* does not modify the asymptotic performance of the algorithm.

4. An extension to a class of non-binary CSP

In this section we extend the results described in the previous sections to a class of non-binary CSPs, that we call *acyclic CSP*s. In particular, this is the class of those CSPs for which the constraint hypergraph is *strongly acyclic*. This property is formally defined below, by using Definition 2.1.

Definition 4.1 (Ausiello [1]). Given an hypergraph $H = \langle N, A \rangle$, its FD-*graph* is a bipartite undirected graph $G(H) = \langle N_H, A_H \rangle$, where:

- $N_H = N \cup A$,
- $A_H = \{(x, y) \mid x \in N, y \in A, \text{ and } x \in y\}$.

Definition 4.2. An hypergraph $H = \langle N, A \rangle$ is *strongly acyclic* if the corresponding FD-graph $G(H) = \langle N_H, A_H \rangle$ is acyclic, i.e., it is a tree.

Definition 4.3. A CSP is *acyclic* if the corresponding labeled constraint hypergraph is strongly acyclic.

After having specified the class of CSPs we deal with, we now describe how to use the dynamic approach proposed in the previous sections to this larger class of problems.

As seen in Section 2, in a generic CSP each k-cardinality hyperarc a_i of the corresponding constraint hypergraph H is labeled by a table T_i which contains all the k-tuples of values allowed for the variables involved. Given a generic labeled constraint hypergraph $H_l = \langle N, A, l \rangle$, the corresponding labeled FD-graph $G(H_l) = \langle N_{H_l}, A_{H_l}, l' \rangle$ can be built as follows:

1. for any node in N there is a corresponding node in N_{H_l};
2. if $a_j = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$ is a k-cardinality hyperarc in A, then there is a corresponding node F_j in N_{H_l}, and the arcs $(v_{i_1}, F_j), (v_{i_2}, F_j), \ldots, (v_{i_k}, F_j)$ in A_{H_l};
3. for each hyperarc a_j, if the label of a_j is $e_j(v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, then generate a new label $e_j^*(v_{i_1}, v_{i_2}, \ldots, v_{i_k}, \texttt{key})$ for a_j, as a collection of $(k+1)$-tuples obtained from the k-tuples in e_j adding a field, denoted `key`, which takes a different value for each of the k-tuples;
4. for each of the arcs (v_i, F_j) introduced in step 2, create the corresponding binary constraint $e(v_{i_h}, F_j)$, such that

$$l'((v_{i_h}, F_j)) = e(v_{i_h}, F_j) = \{(x_{i_h}, \texttt{key}) \mid (x_{i_1}, x_{i_2}, \ldots, x_{i_k}, \texttt{key}) \in e_j^*\}.$$

This relationship also defines the labeling function l' for the resulting graph $G(H_l)$.

If the FD-graph $G(H_l)$ obtained from H_l by applying the previous steps is acyclic, all the algorithms described in the previous sections can be simply extended to $G(H_l)$ and so to H_l. In fact, this only requires to create, for each arc (x, y) in $G(H_l)$, the matrix $R_{x,y}$ starting from the available label $e_{x,y}$.

Now, if $G(H_l) = \langle N_H, A_H, l' \rangle$ is the FD-graph corresponding to a *strongly acyclic* constraint hypergraph $H_l = \langle N, A, l \rangle$ of an acyclic CSP, let us reconsider the parameters:

- $n = |N|$, i.e., n is the number of variables in the considered CSP. We now show that $|N_H| \leqslant 2n - 1$, that is $|A| \leqslant n - 1$. First of all note that the following properties hold: (i) for each $h_i \in A$, $|h_i| \geqslant 2$ (by definition of constraint hypergraph of a CSP); (ii) for each $h_i, h_j \in A$, $|h_i \cap h_j| \leqslant 1$ (by the fact that G_{H_l} is a tree). Let us suppose that $|A| \geqslant n - 1$, and let us consider an arbitrary subset A' of A of size $n - 1$. Let S be the subhypergraph of H_l induced by the hyperarcs of A'. Since $|A'| \leqslant n - 1$,

and the FD-graph of S is a tree, if we apply inequalities (i) and (ii) above to the hyperarcs of A', then they become equalities. This implies that S is a tree. If we add to A' a new hyperarc of A (i.e., we consider an arbitrary subset A' of A of size n), then the subhypergraph of H_l induced by the hyperarcs of A' has a cycle, and hence also G_{H_l}, contradicting the fact that G_{H_l} is a tree.

- $k = \max_{a_j \in A}\{|e_j|\}$, i.e., k is the maximum cardinality of the constraints in the original CSP (note that for any variable v_i, $|D_i| \leqslant k$).

The above discussion and the results of Section 3 imply the following theorem.

Theorem 4.1. *Let us consider an arbitrary CSP defined on n variables and using arbitrary constraints with maximum size k. If the considered CSP is acyclic, then there exist data structures and algorithms supporting arbitrary sequences of operations of the following kinds*:

1. *deletion of k-tuples from any constraint*;
2. *insertion of new constraints that leave the CSP acyclic.*

The total time required to maintain a solution for such a dynamic CSP is $O(nk(k+\log n))$. *If the allowed operations are restricted to be tuple deletions, the required total time is* $O(nk^2)$. *In both cases, reporting a solution for the current CSP, i.e., an assignment for the n variables, requires* $O(n)$ *time.*

5. Conclusions

In this paper we propose dynamic algorithms for the constraint satisfaction problem. In particular, we consider a simple case, where the constraints are binary and the network is acyclic. Following an approach proposed by Dechter [4] for the static case, we propose algorithms that support deletion of allowed tuples from constraints, and insertion of new constraints, while maintaining a data structure which implicitly represents all possible solutions for the considered CSP. We also extend the class of constraints suitable to be tackled by these algorithm to include what we defined acyclic CSPs.

The arguments treated in this paper deserve further considerations. For instance, it would be interesting to design efficient algorithms for the fully dynamic problem. Another field of investigation is to study the possibility of applying the proposed technique to other classes of binary CSPs, and the extension to the general case. Finally, it seems to be very interesting to study the dynamic CSP in the framework proposed in [11, 17], where the possibility of dynamically adding variables to the current problem is considered.

Acknowledgements

We are grateful to the anonymous referees for providing us with many useful comments that significantly improved the presentation of the paper.

References

[1] G. Ausiello, A. D'Atri, D. Saccà, Minimal representation of directed hypergraphs, SIAM J. Comput. 15 (1986) 418–431.
[2] C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, 1973.
[3] R. Dechter, Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition, Artif. Intell. 41 (1989) 273–312.
[4] R. Dechter, J. Pearl, Network based heuristic for constraint satisfaction problems, Artif. Intell. 34 (1988) 1–38.
[5] B.N. Freeman-Benson, J. Maloney, A. Borning, An incremental constraint solver, Comm. ACM 33 (1) (1990) 54–63.
[6] E.C. Freuder, Synthesizing constraint expression, Comm. ACM 21 (11) (1978) 958–966.
[7] E.C. Freuder, A sufficient condition for backtrack-free search, J. ACM 29 (1) (1982) 24–32.
[8] E.C. Freuder, A sufficient condition for backtrack-bounded search, J. ACM 32 (4) (1985) 755–761.
[9] E.C. Freuder, A.K. Mackworth, The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, Artif. Intell. 25 (1985) 65–74.
[10] M.R. Garey, D.S. Johnson, Computers and Intractability, W.H. Freeman, New York (1979).
[11] E. Gelle, I. Smith, Dynamic constraint satisfaction with conflict management in design, OCS'95 Workshop on Over-Constrained Systems at CP'95, Lecture Notes in Computer Science, vol. 1106, Springer, Berlin, 1996, pp. 237–247.
[12] K. Hua, B. Faltings, D. Haroud, G. Kimberly, I. Smith, Dynamic constraint satisfaction in a bridge design system. Proc. International Workshop on Expert Systems in Engineering: Principles and Applications, Lecture Notes in Artificial Intelligence, vol. 462, 1990, pp. 217–233.
[13] L.M. Kirousis, Fast parallel constraint satisfaction, Artif. Intell. 64 (1) (1993) 147–160.
[14] H.C. Lau, Approximation of the constraint satisfaction problem via local search, Proc. 4th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science, vol. 955, Springer, Berlin, 1995, pp. 461–472.
[15] H.C. Lau, O. Watanabe, Randomized approximation of the constraint satisfaction problem, Nordic J. Comput. 3 (4) (1996) 405–424.
[16] A.K. Mackworth, Consistency in networks of relations, Artif. Intell. 8 (1977) 99–118.
[17] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problem, Proc. 8th National Conf. on Artificial Intelligence (AAAI'90), 1990, pp. 25–32.
[18] U. Montanari, Network of constraints: fundamental properties and application to picture processing, Inform. Sci. 7 (1974) 95–132.
[19] U. Montanari, F. Rossi, An efficient algorithm for the solution of hierarchical networks of constraints, Proc. 3rd International Workshop on Graph-Grammars and their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, Springer, Berlin, 1986, pp. 440–457.
[20] U. Montanari, F. Rossi, Fundamental properties of networks of constraints: a new formulation, in: L. Kanal, V. Kumar (Eds.), Search in Artificial Intelligence, Springer, Berlin, 1988, pp. 426–449.
[21] U. Montanari, F. Rossi, Constraint relaxation may be perfect, Artif. Intell. 48 (1991) 143–170.
[22] U. Montanari, F. Rossi, Perfect relaxation in constraint logic programming, Proc. 8th International Conf. on Logic Programming, MIT Press, Cambridge, MA, 1991, pp. 223–237.
[23] G. Ausiello, A. Marchetti-Spaccamela, U. Nanni, Dynamic maintenance of paths and path expressions in graphs, Proc. 13th International Symposium on Symbolic and Algebraic Computation, Lecture Notes in Computer Science, Vol. 358, Springer, Berlin, 1988, pp. 1–12.
[24] R.E. Tarjan, Amortized computational complexity, SIAM J. Algebraic Discrete Methods 6 (1985) 306–318.