Electronic Notes in Theoretical Computer Science 30 No. 2 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume30.html 12 pages

On the need for a different backtracking rule when dealing with late evaluation

Marco Gavanelli, Michela Milano

Dipartimento di Ingegneria
Università di Ferrara
Italy

Abstract

Constraint Satisfaction Problems (CSPs) represent a widely used framework for many real-life problems. Constraint Logic Programming (CLP) languages are an effective tool for modeling problems in terms of CSPs and solving them efficiently.

Lazy domain evaluation is a solving technique that has proven effective for solving CSPs, allowing the minimization of the number of constraint checks. However, exploiting lazy domain evaluation in CLP is not very effective, mainly because of the chronological backtracking rule used in CLP. After each backtracking step, in fact, all the obtained results are lost, even if they had nothing to do with the culprit of the failure. The intelligent backtracking rule widely studied in the past does not solve the problem either.

In this paper, we propose a backtracking rule useful for dealing efficiently and declaratively with lazy domain evaluation in CLP, and we show a simple implementation of a metainterpreter providing the depicted functionality.

1 Introduction

A large class of problems can be modeled as Constraint Satisfaction Problems, or CSPs. A CSP is defined on a set of variables whose possible values are defined by a set of domains and on a set of constraints limiting the possible assignments of values to variables. CSPs are usually solved by means of search techniques, such as Backmarking [10], Forward Checking (FC) and Looking Ahead [12]. All these search techniques assign a tentative value to a variable, then propagate constraints; if propagation detects inconsistency, the algorithm chronologically backtracks, i.e., the previous assignment is undone (as well as the propagation corresponding to the assignment) and another value is tried.

[1] Work partially supported by MURST Project on "Intelligent Agents Interaction and Knowledge Acquisition".

The concept of backtracking is included in some programming languages, such as Logic Programming and Constraint Logic Programming (CLP) languages [13] [14]. Implementing search algorithms that exploit chronological backtracking in these languages is straightforward: resulting programs are very simple and clear. CLP languages are well-suited for CSP solving, as they also include domain managing and a constraint propagation engine, usually based on Arc-Consistency [16].

Some CSP search algorithms use a more sophisticated backtracking rule. For example, Backjumping [11] and Conflict-Directed Backjumping [18] try to avoid unnecessary computation in the unlabeling phase. These algorithms propose a solution for one of the chronological backtracking's drawbacks: when failure is detected, backtracking returns to the last choice, even if it has nothing to do with the reason of failure. This means that all the (unnecessary) work performed between the set of inconsistent assignments and the last performed assignment will be reexecuted again and again. More efficient (and informed) approaches consider the dependency between the various assignments and backtrack directly to the source of the failure instead of the last assignment. To obtain this, more complex data structures have to be maintained.

In order to implement these algorithms, (constraint) logic programming with chronological backtracking is not the most suitable framework. The backtracking rule used in the algorithm is more informed than the rule used in the programming language, so implementation is not straightforward. Implementing these algorithms can be done in various ways. It can be obtained by simply considering Prolog as every other programming language: the information needed by the backtracking rule is kept in some structures and maintained by the algorithm itself. In this way, however, the program will not exploit all the expressive power of logic programming, being forced to extensively use extralogic predicates. A second possibility is building an interpreter or a compiler providing the needed backtracking rule. Prolog itself is a powerful tool for building interpreters and compilers [21] and meta-interpretation is a widely used technique in logic programming.

Many works deal with intelligent backtracking rules implemented at language level, such as [2] [4] [3] [15]. Of course, the new method will probably introduce some overhead, which slows down the system in the worst case analysis; anyway, these methods usually provide on average a good speedup because of the avoidance of useless computation.

The backtracking rule used e.g. in Prolog can be criticized by another viewpoint also. After a backtracking, all the performed computation is lost, nothing is remembered or learnt about the problem. On the other hand, some search algorithms use a form of remembering of results obtained in failing branches [10] [7]. Implementing these algorithms in a language exploiting Prolog backtracking makes both complex and inefficient programs.

For this reason, we propose a backtracking rule providing the possibility to remember information from failures. In particular, this is useful if we postpone some of the calculus and perform it only when required by the search. In other words, it is suitable when we exploit lazy domain evaluation, such as in Minimal Forward Checking (MFC)[7].

The paper is organized as follows. In section 2 we define a new predicate that allows recording dependency information of results on choices. In section 3 we show a CLP implementation of MFC exploiting the new defined predicate. In section 4 we show a simple Prolog metainterpreter providing the discussed predicate as a built-in and some experimental results.

2 A New Predicate

In every CSP solving technique, a variable is associated with a domain, which can be modified during computation. Typically, as propagation is performed, new information is inferred and domains are reduced. In other words, at the beginning every variable domain contains all the possible elements; then values are deleted thanks to constraint propagation and knowledge about the final value the variable will assume is refined.

When propagation can infer no more information about the problem, we are forced to perform a guess. Many search algorithm are based on the idea to assign a trial value to a variable and then propagate the consequences of the choice. If the choice leads to a failure, we have to undo the assignment and all the computation results that depend on it. In many important search algorithms (e.g., Forward Checking (FC) and Looking Ahead [12], Maintaining Arc Consistency (MAC) [8]), every elimination occurs because of the *last* assigned value. In other words, in these algorithms a value cannot be deleted from a domain because of a previously-made assignment and all the computation is performed in chronological order. For this reason, the chronological backtracking of Prolog is perfectly suited to implement these algorithms. However, if we want to implement an algorithm that learns during computation, we are forced to use extralogic predicates.

An idea that has been proven effective for CSP solving is *lazy domain evaluation* [22] [7] [19]. The idea behind it is that every constraint check should be performed as late as possible and only if effectively required by the search. For example, the Minimal Forward Checking (MFC) [7] algorithm is a lazy version of FC that keeps only *one* consistent value in each variable domain. If the only consistent value is deleted from the domain, another value must be selected and checked for consistency with respect to all the past connected variables. This means that a constraint involving a past connected variable can delete a value from a domain; this elimination should not be undone if the past connected variable's assignment is not undone.

For example, consider the CSP in Figure 1. Suppose X_1 is assigned value 1, then propagation starts (step 1). Value 1 is deleted from X_2's domain, value 2 is consistent and becomes the candidate (bold in the figure). Value 1 in the

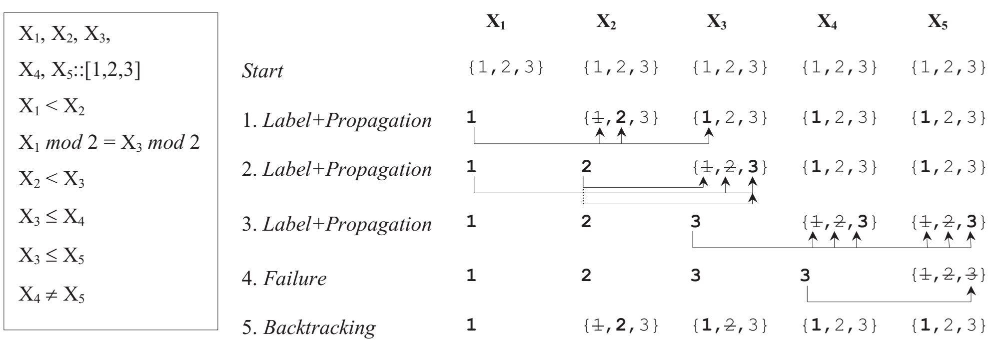

Fig. 1. Example of Minimal Forward Checking computation

domain of X_3 is consistent and another labeling phase starts (step 2). X_2 is assigned value 2; the candidate of X_3 is inconsistent, so it is eliminated and the successive value 2 must be checked against the past variable X_1. This value is inconsistent and is removed; value 3 is consistent with all the constraints and becomes the candidate. Note that value 2 has been eliminated because of the value assigned to X_1 when the current variable (the last assigned variable) was X_2. This means that the value 2 must not be re-inserted in X_3's domain while the variable X_1 (the cause of the elimination) keeps its value. In particular, it must not be re-inserted when variable X_2 is unlabeled.

In the example of Figure 1, we have a failure and we have to unlabel variable X_2. Chronological backtracking suggests to go back to step 1, i.e. to re-insert all the domain values that were removed *after* the labeling of variable X_2. MFC, instead, goes to the status 5, i.e. re-inserts all the domain values that were removed *because of* the labeling of variable X_2: since value 2 in $dom(X_3)$ was removed because of X_1, it is not re-inserted.

What we propose is quite different from the intelligent backtracking widely studied in the past [2] [4] [3] [15]. All those proposals suggest to find a good backtracking point, undo all the computation performed after this point and then jump to it, just as if all the computation performed in this while had not been done. Instead, we do not focus on the backtracking point but on the operations performed after. When we backtrack to a certain point, not all the work have to be undone. If we undo a choice (and we are going to choose another branch), only the computation depending on this choice has to be undone; all the results we got depending on previously made choices have to be kept. For this reason, the system needs to know which result depends on which choice. When dealing with constraint satisfaction, a choice is simply an instantiation of a variable to a domain value; a result is the removal of an element from a domain. We suggest thus to consider a predicate `set_dependency(A,Var,Val)` with the intended meaning that the domain value `Val` must be considered removed from the domain of variable `Var` until the assignment of variable `A` is undone. This predicate can be used to implement constraints and have they delete values. Let us consider now how this predicate can be used to

```
label_mfc([]).
label_mfc([Xa |T]) :-
                     instantiate(Xa),
                     label_mfc(T).

instantiate(Xa) :- Assign the first value in the domain.
instantiate(Xa) :- delete the first element from the domain,
                     instantiate(Xa).
```

Fig. 2. A MFC Prolog Implementation - labeling phase

implement the MFC algorithm.

3 A CLP Implementation of Minimal Forward Checking

In this section we show a simple implementation of the MFC algorithm [7] [1]. The algorithm alternates a labeling phase and a propagation phase. The search starts with a call to `label_mfc`, (figure 2) which takes as input the list of problem variables. This predicate selects a variable (in our example, the first) tries to instantiate it, propagates constraints, then tries to instantiate the following variables. To instantiate a variable, first we try the first domain value, that is the only element checked against all the past connected variables. If we get a failure, the candidate is deleted, triggering propagation, so another candidate is found.

The propagation phase is performed by the successive activation of suspended constraints. As usually happens in CLP, constraints are activated when elements are removed from a domain; in particular, when the candidate is deleted. In Figure 3 the implementation of a lazy constraint is shown. The constraints are suspended waiting for one variable to become instantiated. When a constraint is activated, the first element in each future domain is checked for consistency with respect to the current assignment. If the candidate is consistent, nothing happens: the constraint just suspends in order to check successive values if the candidate is removed. If the candidate is inconsistent, it is removed (thus awaking the past constraints) and the dependency on the current assignment is recorded. Then, another candidate must be found, and the constraint performs a recursive call. When the new candidate is found, the other constraints (involving past variables) can execute and check the new candidate. As explained in [7], a recording of the removed values must be kept in order not to re-execute constraint checks. Thanks to this recording (provided by the `set_dependency` predicate), the domain element will not be re-inserted during backtracking, unless the cause of the elimination is unassigned.

This implementation of MFC has a slightly different behaviour w.r.t. the algorithm described in [7]; in fact, it does not record information on success-

```
lazy_constraint(Xa,Xf) :-
     var(Xa), var(Xf), !, suspend.
lazy_constraint(Xa,Xf) :-                % Suppose Xa is instantiated
     Let E be the first element of Xf's domain
     (is_consistent(Xa,Xf ← E)
     →  % The candidate is consistent
         (var(Xf)
         →  suspend
         ;  true)
     ;   set_dependency(Xa,Xf,E),        % removes the inconsistent value
         lazy_constraint(Xa,Xf)).        % looks for another consistent value
```

Fig. 3. A MFC CLP Implementation - Example of binary constraint

ful constraint checks. Only unsuccessful constraint checks produce a domain modification, so we think that they are more appropriate in a CLP framework. However, information about successful constraint checks can be inserted and checked in the `is_consistent` predicate.

Moreover, in order to minimize the number of constraint checks, the MFC algorithm needs to activate the constraints in the instantiation order. For example, suppose that the variables of a CSP are instantiated in lexicographic order $(X_1, X_2, \ldots, X_n)$. The current variable X_a is assigned a value and removes the candidate for the future variable X_f. The next value (say value v) must be checked for consistency w.r.t. all the past connected variables: suppose that it is inconsistent with the constraints $c(X_1, X_f)$ and $c(X_4, X_f)$. If the constraint $c(X_1, X_f)$ is activated first, the element will not be re-inserted until X_1 is uninstantiated; if $c(X_4, X_f)$ is activated first, then the culprit of the elimination will be X_4, so value v will be re-inserted when X_4 is uninstantiated and will be again checked for consistency. However, the algorithm is correct independently of the activation order, so we consider this issue as a matter of the heuristics used to activate the constraints [9].

Finally, note that lazy constraints can be mixed with eager constraints without affecting correctness, but only efficiency. So, we can exploit laziness for expensive constraints (constraints in which each constraint checking is hard) and eagerness for efficient constraints. For example, the `alldifferent` constraint is usually implemented very efficiently in CLP systems, exploiting OR techniques and allowing intensive pruning of the search space at a reduced computational cost. On the other hand, in some constraints each consistency checking is hard; e.g., in a vision system [6] the perpendicularity test between surfaces is a single constraint check. With this backtracking rule, we allow the user to implement each constraint the most efficient way.

It's worth noting that, thanks to this predicate, we do not have to use extralogic predicates to implement this algorithm. In plain Prolog, we would have need of extralogic predicates, because we need to record information about deleted values after backtracking, but no logic construct of Prolog survives backtracking. In the next section, we show a simple metainterpreter providing the depicted functionality.

4 A simple Metainterpreter

To keep information after a backtracking, the only methods provided by Prolog are the `assert/1` and the `retract/1` predicates. For this reason, the `set_dependency(A,Var,Val)` predicate asserts a fact `dep(A,Var,Val)` stating that the removal of a the element `Val` from the domain of variable `Var` depends on the assignment on the variable `A` [2].

In order to have a backtracking rule exploiting the information recorded in the `dep/3` structures, we developed a simple metainterpreter (Figure 4), based on the idea that after backtracking, we redo all the computation that had not to be undone. In other words, after a backtracking, Prolog undoes an instantiation and all the chronologically successive computation; since we want to undo only the computation that *depends* on the instantiation, the rest has to be redone. This operation is necessary because in a Prolog metainterpreter we do not have direct access to the operations performed during the unlabeling phase. For this reason, the metainterpreter generates two events, called forward and backward event. The forward event is generated just before performing a choice. Since in Prolog choices are made when we select the head of a clause, the metainterpreter triggers this event before selecting a clause for unification. The forward event redoes all the computation which has been unnecessarily undone. The backward event is activated if unification leads to failure. In the backward event we retract all the `dep/3` facts depending on the variables that are no more instantiated.

Of course, this implementation is highly inefficient, due mainly to metainterpretation. However, not every predicate has to be metainterpreted; in fact, only the program parts which use backtracking need to be metainterpreted. For example, in a search algorithm, only the labeling phase has to be metainterpreted, while the propagation phase can be compiled.

Note that the space complexity of the additional data is $O(nd)$ (where n is the number of CSP variables and d is the maximum domain size) as in the most efficient MFC implementation [1]. In fact, the number of asserted `dep` facts cannot be bigger than the number of elements that can be deleted (i.e. the number of variables multiplied by the number of elements in each domain), because each domain element can be present only in one `dep` fact.

If we consider the number of constraint checks (Figure 5.a), we can see that MFC with chronological backtracking performs much worse than our version of MFC. Comparing the classical MAC algorithm usually employed

[2] Actually, we do not assert variables, but references to variables, because in Prolog the link is not maintained in the assertion. We consider to have two predicates: `retrieve_ref/2` and `match_var/3` that provide access to the reference given the variable and viceversa. In our implementation the `match_var/3` predicate needs to have access to the list of CSP variables, but it is system dependent.

```
% solve(Goal List,List of the CSP variables)
solve([],_) :- !.
solve([A|B],L) :- !, solve(A,L), solve(B,L).
solve(H,L) functor(H,Func,Arity), functor(Templ,Func,Arity),
           clause(Templ,Body),     % select a clause for unification
           (fwd_event(L) ; bck_event(L), fail),
           H=Templ,           % perform unification with the selected clause
           solve(Body,L).

fwd_event(L) :-  ( redo_flag          % If we are in a redo phase,
                 →  redo_unback(L),
                    reset redo_flag
                 ;  true).
bck_event(L) :-  % retract all the dep/3 facts which depend on
                 % the instantiation of the variables we are uninstatiating
                 forget_dep(L),
                 set redo_flag.

% delete from the domains of the variables in VarList
% all the values declared in dep/3 facts.
redo_unback(VarList) :-
    findall([Var,Val],dep(_,Var,Val),DeleteList),
    redo_deletions(DeleteList,VarList).
redo_deletions([],_).
redo_deletions([[Ref,Val]|T],VarList) :-
    match_variable(Ref,VarList),
    % get the variable given its reference and the list of CSP variables
    Var ≠ Val, redo_deletions(T,VarList).

forget_dep([]).
forget_dep([Val|T]) :- nonvar(Val), forget_dep(T).
forget_dep([V|T]) :- var(V), retrieve_ref(V,Ref),
    retract_all(dep(Ref,_,_)),
    forget_dep(T).

set_dependency(A,Var,Val) :-
    retrieve_ref(A,RefA), retrieve_ref(Var,RefV),
    assert(dep(RefA,RefV,Val)).
```

Fig. 4. Metainterpreter for the implementation of the `set_dependency` predicate

in CLP systems with MFC (Figure 5.b), we can see that the efficiency gain is even better. In these figures, the area painted in black shows the set of problems that are solved best with chronological backtracking. As the graph goes higher, our backtracking rule outperforms the chronological backtracking.

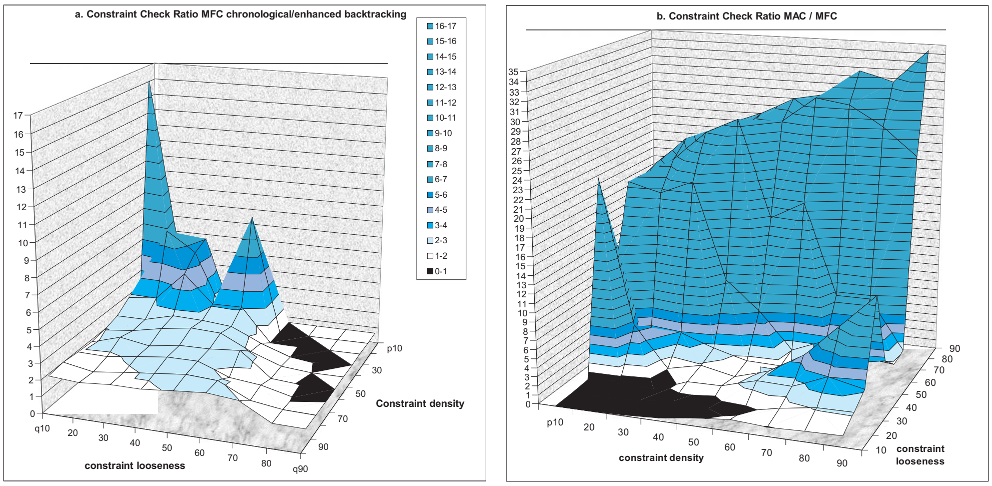

Fig. 5. Constraint check ratios

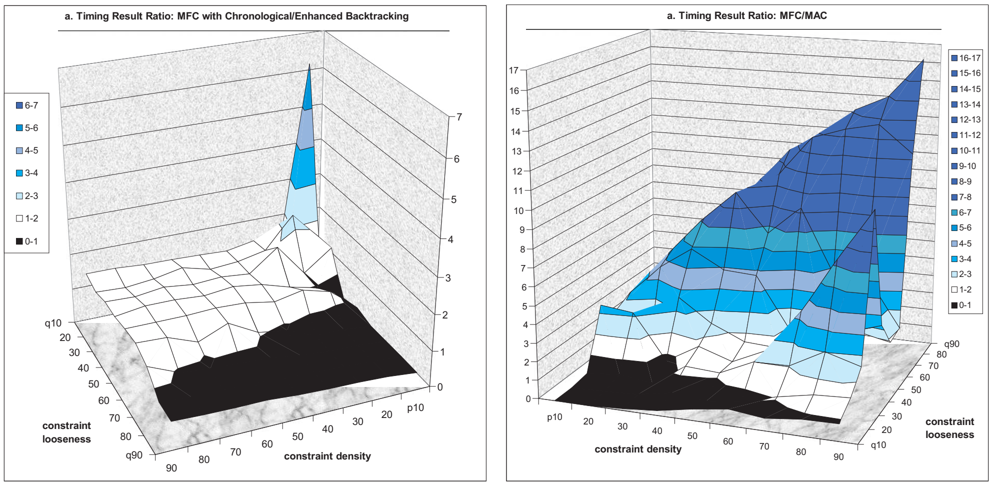

Fig. 6. Timing Results

Timing results (Figure 6)[3] show a smaller improvement, because we wanted mainly to obtain a simple implementation, and it is not optimized, so the introduced overhead is significant. Since this overhead does not affect the number of constraint checks, it cannot be seen in the graphs of Figure 5.

5 Conclusions and Future Work

In this paper, we proposed a non chronological backtracking rule needed when implementing lazy domain evaluation. We presented a metainterpreter providing the required backtracking rule and an implementation of MFC exploiting it. Some experimental results obtained with randomly generated problems

[3] For the sake of fairness, also the algorithms exploiting chronological backtracking were meta-interpreted.

have been shown.

We believe that Interactive CSP [5,6] solving could benefit from it. In every non-toy CSP, the problem parameters come from the outer world and have to be acquired; in some important cases this acquisition is a hard process. The ICSP framework allows to acquire only the necessary information during the CSP solving process. In an ICSP, variables range on *partially known* domains, i.e., some elements can be known and others are unknown. For example, a variable X_i can range on a domain $D_i = [1, 2, 3|U_i]$ where U_i is a new variable representing elements that still have to be acquired. In this case is very important to postpone as late as possible domain value acquisition and it is necessary keeping it in a domain even after backtracking, in order not to request previously acquired values.

CLP systems usually exploit Branch and Bound when dealing with CSPs with a cost function [17]. In this algorithm, after finding a solution, a constraint is stated saying that the next solution needs to have a lower cost than the last found solution. Of course, this could be obtained by building another CSP and solving it from scratch; however, a commonly used technique is stating a constraint that survives backtracking. This is usually an "ad hoc" solution; we believe that a more general backtracking style could consider both Branch and Bound as well as lazy constraint evaluation.

Another issue is implementing a Prolog interpreter in a language not providing backtracking itself (e.g. Lisp). This could be important to show that a Prolog system providing the proposed predicate can be efficiently implemented. We believe that in this case, the introduced overhead would be minimal. In a Lisp implementation, we would not be forced to perform backtracking and redo part of the computation; we simply would not undo the necessary computation. Finally, we consider implementing it in a Warren Abstract Machine.

References

[1] Bacchus, F and Grove, A., *On the Forward Checking Algorithm*, Proceedings of the First International Conference on Constraint Programming", (1995), 292–309

[2] Bruynooghe, M. and Pereira, L.M. *Deduction Revision by Intelligent Backtracking* Implementations of Prolog, Ellis Horwood (1984), 196–215

[3] Cicekli, I. *An intelligent backtracking schema in a logic programming environment*, ACM SIGPLAN Notices. **32(3)** (1997), 42–49

[4] Cox, P.T. and Pietrzykowski, T. *Deduction Plans: A Basis for Intelligent Backtracking* IEEE Transactions on Pattern Analysis and Machine Intelligence **1(3)** (1981) 52–65

[5] Cucchiara, R., Gavanelli, M., Lamma, E., Mello P., Milano, M. and Piccardi, M. *Constraint Propagation and Value Acquisition: why we should do it*

Interactively, Proceedings of the International Joint Conference on Artificial Intelligence, (1999), 468–477

[6] Cucchiara, R., Gavanelli, M., Lamma, E., Mello P., Milano, M. and Piccardi, M. *Extending CLP(FD) with interactive Data Acquisition for 3D Visual Object Recognition*, Proceedings of the First International Conference on the Practical Application of Constraint Technologies and Logic Programming, (1999), 137–155

[7] Dent, M.J., and Mercer, R.E., *Minimal Forward Checking*, Proceedings of the Sixth International Conference on Tools with Artificial Intelligence, (1994), 432–438

[8] Freuder, E.C. and Sabin, D., *Contradicting Conventional Wisdom in Constraint Satisfaction*, Lecture Notes in Computer Science, **874**, (1994).

[9] Freuder, E.C. and Wallace, R.J. *Ordering heuristics for arc consistency algorithms*, Proceedings of the Ninth Canadian Conference on Artificial Intelligence (1992).

[10] Gaschnig, J. *A General Backtrack Algorithm That Eliminates Most Redundant Tests*, Proceedings of the 5th International Joint Conference on Artificial Intelligence, Cambridge, MA, 1977

[11] Gaschnig, J. *Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignments problems*, Proceedings of the Canadian Artificial Intelligence Conference, 1978

[12] Haralick, R.M. and Elliott, G.L. *Increasing tree search efficiency for constraint satisfaction problems*, Artificial Intelligence. **14** (1980), 263–313

[13] Jaffar, J. and Lassez, J.L. *Constraint Logic Programming*, Conference Record 14th Annual ACM Symposium on Principles of Programming Languages, (1987), 111–119

[14] Jaffar, J. and Maher, M.J. *Constraint Logic Programming: A Survey*, The Journal of Logic Programming, **19 & 20**, (1994), 503–582

[15] Kumar, V. and Lin, Y-J. *A data-dependency based intelligent backtracking schema for Prolog*, Journal of Logic Programming. **4** (1988)

[16] Mackworth, A.K., *Consistency in networks of relations*, Artificial Intelligence, **8**, (1977)

[17] Prestwich, S., *Three implementations of Branch-and-Bound in CLP*, Compulog Netmeeting on Parallelism and Implementation Technology, (1996)

[18] Prosser, P. *Hybrid algorithms for the constraint satisfaction problem*, Computational Intelligence. **9(3)** (1993)

[19] Schiex, T., Régin, J.C., Gaspin, C. and Verfaillie, G. *Lazy Arc Consistency*, Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference (1996) 216–221

[20] Waltz, D.L., *Generating Semantic Descriptions From Drawings of Scenes With Shadows*, A.I. Lab., M.I.T., Technical Report AI-TR-271, 1972

[21] Warren, D.H.D., *Logic Programming and Compiler Writing*, Softwareemdash Practice and Experience, **10(2)** (1980) 97–125

[22] Zweben, M. and Eskey, M., *Constraint Satisfaction with Delayed Evaluation*, IJCAI 89, (1989) 875–880.